Navigating the Floating-Point Seas: From Bitwise Reproducibility to Reduced Precision Computing

By:

Bálint Siklósi

Supervisor:

Dr. István Zoltán Reguly, PhD

A dissertation submitted for the degree of

Doctor of Philosophy

Pázmány Péter Catholic University

Faculty of Information Technology and Bionics

Roska Tamás Doctoral School of Sciences and Technology

Budapest, 2025

In memory of my father,
whose legacy lives on in who I've become.

Acknowledgements

First of all, I want to thank my supervisor, István Reguly. He supported me in every part of this journey: from tracking down the tiniest code bugs to helping me shape the writing and structure of this dissertation. He always stayed calm and patient, even when I had long periods without real progress. Beyond research, he supported me in life too. He never judged when things got tough, and always understood when personal events set me back. And when I faced practical difficulties, he helped out where he could, without hesitation.

During my time abroad, I had the privilege of working with two amazing researchers: Gihan R. Mudalige at the University of Warwick and Neil Sandham at the University of Southampton. Both of them welcomed me warmly and treated me as a colleague from the first moment, not just as a visitor or a student, but as someone whose work mattered. Their trust and support meant a lot.

I'd also like to thank Pushpender K. Sharma at the University of Southampton for his patient help in understanding the physics and CFD background, and David Lusher for his support with OpenSBLI-related coding questions.

During my research stay at the University of Southampton, I had the privilege of staying with the Robinson family, who welcomed me into their home for a month. Their kindness and joyful presence made a big difference – they created a calm and supportive environment that helped me focus on my work in a much healthier way.

To my fellow PhD students and friends: thank you. Especially Dani and Attila, for all the "duck debugging" sessions, the countless technical discussions, and just being there when things didn't make sense (which happened often). Also thanks to everyone in the "Ebéd délben?" group for sharing your advice – on research, but also on how to survive the rest of life during a PhD. I also want to thank my flatmates, past and present: Dani, Anna, Tomi, and Dalma. For the everyday support, laughs, and shared chaos that made this whole period more bearable (and often, more fun).

I'm incredibly thankful to my family: my mother, my sisters, my brother-in-law, and my nieces. Their support was always there in the background, even when I didn't always show how much I needed it. I also want to mention my father, who sadly passed away midway through my PhD. I wish he could have seen me finish – I know he would have been proud.

There are some people who were a big part of my life during this journey, even if not all of them are still here in the same way. To the people I shared life with more deeply during this time: your presence helped carry me through. Especially Ágota who is with me now: thank you for your patience, support, and everything else.

I'm also grateful to my friends – the ones who listened, distracted, encouraged, or just spent time with me when I needed it most.

Thanks to the administrative staff at the university. Tivadarné Vida, from the doctoral office, thank you for always handling the paperwork and deadlines with a smile (and flexibility). I also want to thank the international relations office for their help with arranging and financing my visits abroad, and all other staff members who contributed in any way during these years.

Finally, I want to thank all the grants and programs that made this research possible: multiple EKÖP, ÚNKP fellowships, Erasmus+, OTKA projects, and – especially – Tamás Zsedrovits who invited me to join his grant project when I truly needed that opportunity.

Thank you all.

Abstract

Floating-point arithmetic is a cornerstone of scientific and high-performance computing, enabling the numerical approximation of complex real-world phenomena. However, its inherent limitations – such as rounding errors, limited precision, and non-associativity – pose significant challenges to reproducibility and computational efficiency. This dissertation addresses two critical and interrelated areas in floating-point computing: bitwise reproducibility and reduced or mixed-precision computing. These challenges are explored in the context of large-scale scientific simulations, with a focus on enhancing both numerical robustness and performance portability across heterogeneous computing platforms.

The first half of the research concentrates on bitwise reproducibility on unstructured mesh computations, which refers to the ability to obtain identical binary results across multiple executions of a program given the same input. This property is essential for debugging, validation, cross-platform consistency, and scientific integrity, especially in high-performance environments that involve parallelism through MPI or GPU acceleration. The primary causes of non-reproducibility are identified as parallel reductions and indirect memory access patterns that introduce non-deterministic execution orders. To address these challenges, the dissertation proposes and implements a set of novel techniques within the OP2 domain-specific language for unstructured mesh computations. These include temporary array-based accumulation to impose deterministic update orders, deterministic graph coloring to avoid race conditions during parallel execution, and the integration of the ReproBLAS library to ensure reproducibility of global reductions. The resulting methods are platform-agnostic and require minimal changes to application source code, thus promoting portability and ease of use.

These techniques were evaluated on a range of real-world applications, including industry-grade computational fluid dynamics (CFD) solvers such as Rolls-Royce Hydra, as well as benchmark applications like Airfoil, Aero, and MG-CFD. The results demonstrate that full bitwise reproducibility can be achieved with varying levels of performance overhead, depending on the application, hardware platform, and chosen strategy. While the overhead is generally reasonably low on CPUs, it can be significantly higher on GPUs, especially for complex workloads. Despite these costs, the benefits of deterministic behavior and consistent results across executions make these methods suitable for both research and industrial use cases where reproducibility is essential.

The second part of the dissertation explores reduced and mixed-precision computing as a means to improve performance and reduce energy and memory consumption. By using lower precision representations – such as single or half precision – scientific applications can achieve significant speedups and reduced memory bandwidth usage. However, reduced precision can compromise numerical stability and accuracy. To navigate this trade-off, this work implements mixed-precision strategies within the OpenSBLI framework, a code generation system for CFD applications that targets OPS as its backend. The strategy

involves maintaining critical variables in higher precision while performing secondary calculations in reduced precision. Several precision combinations were tested using simulations of the Taylor-Green vortex problem.

Results show that carefully designed mixed-precision schemes can retain sufficient accuracy while achieving notable runtime and memory improvements. For example, the use of single and half precision in specific subroutines led to up to 2.37× speedup on GPU platforms and up to 3.7× on CPU platforms, with acceptable loss in accuracy. The dissertation also highlights how algorithmic formulations, such as split-forms of the Navier-Stokes equations, influence numerical stability under low-precision conditions. These insights are essential for developing robust reduced-precision solvers for compressible and turbulent flows.

Together, the work presented in this dissertation delivers significant contributions to the field of high-performance scientific computing. It offers a reproducibility framework that is practical for use in industrial CFD applications and proposes strategies for leveraging low-precision arithmetic without compromising scientific validity. The combination of reproducibility and mixed-precision optimization enables scientists and engineers to develop simulations that are both accurate and efficient, marking a meaningful step toward the challenges posed by future exascale computing systems. The techniques introduced here are broadly applicable and can be extended to other scientific domains where floating-point operations play a crucial role.

Kivonat

A lebegőpontos aritmetika a tudományos és nagy teljesítményű számítástechnika egyik alappillére, amely lehetővé teszi a valós világ összetett jelenségeinek numerikus közelítését. Ugyanakkor a lebegőpontos számábrázolás velejáró korlátai – például a kerekítési hibák, a korlátozott pontosság és a nem-asszociativitás – komoly kihívásokat jelentenek a reprodukálhatóság és a számítási hatékonyság szempontjából. Ez a disszertáció két, egymással szorosan összefüggő területet vizsgál a lebegőpontos számítások világában: a bitpontosan reprodukálható számításokat és a csökkentett vagy kevert pontosságú számítást. A vizsgálat középpontjában nagyléptékű tudományos szimulációk állnak, azzal a céllal, hogy növeljék a numerikus stabilitást és biztosítsák a teljesítmény hordozhatóságát heterogén számítási architektúrák között.

A kutatás első része a bitpontosságú reprodukálhatóságra fókuszál, amely azt jelenti, hogy egy program többszöri futtatása azonos bemeneti adatokkal minden alkalommal pontosan azonos bináris eredményt produkál. Ez a tulajdonság kulcsfontosságú a hibakeresés, a validálás, a platformfüggetlen viselkedés és a tudományos megbízhatóság szempontjából, különösen az olyan nagy teljesítményű környezetekben, ahol MPI-alapú vagy GPU-gyorsított párhuzamosítás történik. A nem reprodukálható viselkedés fő okai között szerepelnek a párhuzamos redukciós műveletek, valamint az indirekt memóriahozzáférési minták, amelyek nem-determinisztikus végrehajtási sorrendeket eredményeznek. A disszertáció ennek kezelésére új módszereket javasol és valósít meg az OP2 domén-specifikus nyelv keretein belül, amelyet strukturálatlan térhálókon végzett számításokra terveztek. A bemutatott technikák között szerepel ideiglenes tömbök használata a determinisztikus frissítési sorrendek biztosítására, determinisztikus gráfszínezés az adatversenyek elkerülésére, valamint a ReproBLAS könyvtár integrálása a globális redukciók bitpontosságú végrehajtásához. Ezek a megoldások platformfüggetlenek, és csak minimális módosítást igényelnek az alkalmazáskódban, ezáltal elősegítve a technológiák közti átjárhatóságot és a könnyű használatot.

A bemutatott módszerek valós alkalmazások széles körében kerültek kiértékelésre, beleértve ipari szintű áramlástani szimulátorokat, például a Rolls-Royce Hydra-t, valamint benchmark célú alkalmazásokat, mint az Airfoil, Aero és MG-CFD. Az eredmények azt mutatják, hogy a teljes bitpontosságú reprodukálhatóság elérhető eltérő mértékű teljesítménybeli többletteher mellett, amely függ az alkalmazás típusától, a hardverplatformtól és a választott módszertől. Míg CPU-n ez az lassulás jellemzően mérsékelt, GPU-n, különösen összetett problémák esetén, jelentősebb lehet. Ennek ellenére a determinisztikus viselkedés és a konzisztens eredmények előnyei különösen értékessé teszik a módszereket olyan kutatási és ipari szimulációs környezetekben, ahol elengedhetetlen a reprodukálhatóság.

A dolgozat második része a csökkentett és kevert pontosságú számításokat vizsgálja, mint lehetséges eszközt a teljesítmény növelésére, illetve az energia- és memóriafelhasználás csökkentésére. Az alacsonyabb pontosságú számábrázolások – mint például a single vagy half pontosság – használata révén tudományos alkalmazások jelentős gyorsulást és kisebb memória-sávszélesség igényt érhetnek el. Ugyanakkor a csökkentett pontosság numerikus instabilitáshoz és pontosságvesztéshez vezethet. E kompromisszum kezelésére a disszertáció kevert pontosságú stratégiákat valósít meg az OpenSBLI keretrendszerben, amely egy CFD-alkalmazásokhoz tervezett kódgeneráló rendszer, az OPS könyvtárra épülve. A megközelítés lényege, hogy a kritikus változók magasabb pontosságban maradnak, míg az alacsonyabb prioritású számítások csökkentett pontossággal történnek. Számos pontossági kombináció került tesztelésre a Taylor-Green örvényléstani problémán keresztül.

Az eredmények azt mutatják, hogy a gondosan megtervezett kevert pontosságú eljárások képesek megtartani a szükséges pontosságot, miközben számottevő futásidő- és memória-megtakarítást biztosítanak. Például egyes alprogramokban a single és half pontosság alkalmazása akár 2.37-szeres gyorsulást eredményezett GPU-kon és akár 3.7-szerest CPU-kon, elfogadható pontosságveszteség mellett. A dolgozat továbbá kiemeli, hogy az olyan algoritmikus megfogalmazások, mint a Navier – Stokes egyenletek szétválasztott alakjai, jelentősen befolyásolják a numerikus stabilitást alacsony pontosságú környezetekben. Ezek az eredmények kulcsfontosságúak a összenyomható és turbulens áramlásokhoz alkalmazható robusztus, csökkentett pontosságú szimulátorok fejlesztéséhez.

Összességében a disszertációban bemutatott munka jelentős hozzájárulást nyújt a nagy teljesítményű tudományos számítástechnika területéhez. Olyan reprodukálhatósági keretrendszert kínál, amely ipari CFD-alkalmazásokban is hatékonyan használható, és olyan stratégiákat javasol, amelyek révén kihasználható az alacsonyabb pontosság nyújtotta teljesítménynövekedés anélkül, hogy ez veszélyeztetné a tudományos érvényességet. A reprodukálhatóság és a kevert pontosság kombinációja lehetővé teszi olyan szimulációk létrehozását, amelyek egyszerre pontosak és hatékonyak, ezzel megalapozva a jövő exaszintű számítógépes rendszereinek alkalmazásait. Az itt bemutatott technikák széles körben alkalmazhatók, és kiterjeszthetők más tudományterületekre is, ahol a lebegőpontos műveletek központi szerepet játszanak.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Scientific Computing

Scientific computing is the application of computational methods to solve complex problems across a wide array of scientific and engineering disciplines. At its core, it involves the development and implementation of numerical models and algorithms that approximate the behavior of real-world systems. Many problems – whether they arise in fluid dynamics, materials science, or biological systems – are analytically intractable; computational methods allow researchers to explore these systems through simulation, analysis, and visualization. Over the decades, advances in algorithm design, software engineering, and computer architecture have dramatically expanded the scope and depth of problems that can be addressed [1].

A significant factor in this evolution has been the ongoing advancement of computational hardware. Earlier computational efforts were often constrained by limited processing power and memory, but modern systems now offer immense capabilities that enable the simulation of systems with millions, or even billions, of degrees of freedom. This progress has not only enhanced the precision and scale of simulations but also broadened their applicability. Researchers can now conduct comprehensive computational experiments that complement and, at times, guide traditional laboratory and theoretical studies.

Moreover, scientific computing today is not just about simulation. It also involves sophisticated data analysis and visualization techniques, which help in interpreting the often massive volumes of data produced by computational experiments. This holistic approach – combining model development, simulation, and data exploration – has become essential for advancing our understanding of complex phenomena. As computational methods continue to evolve, they are increasingly integrated into the fabric of research, policy-making, and industry, underscoring their critical role in modern scientific inquiry.

1.2 High Performance Computing

High performance computing (HPC) represents a specialized branch of scientific computing dedicated to solving large-scale and computationally intensive problems. By harnessing the capabilities of supercomputers and computer clusters, HPC systems perform vast numbers of calculations in parallel, enabling breakthroughs in fields where high-speed data processing and complex simulations are essential [2]. These systems are built upon advanced hardware architectures that include multi-core processors, high-speed interconnects, and expansive memory configurations.

The evolution of HPC is deeply intertwined with historical trends in semiconductor technology. Moore's Law, which observes that the number of transistors on a chip doubles approximately every two years [3], has driven exponential growth in computing power over the past several decades. Dennard scaling, which posited that as transistors shrink, their power density remains constant [4], further enabled performance gains while managing energy consumption. However, as Dennard scaling has slowed due to physical and practical limits, the focus has increasingly shifted toward exploiting parallelism and innovative architectural designs to sustain performance improvements.

Parallel programming models – whether using distributed memory, shared memory, or hybrid approaches – are central to efficiently utilizing these evolving hardware capabilities. The challenge is not only to design algorithms that can be effectively partitioned and executed concurrently but also to ensure that the underlying hardware is fully exploited while minimizing overhead from communication and synchronization. This balance between hardware and software innovation continues to drive the evolution of HPC systems.

Beyond raw computational power, the success of HPC relies on a robust software ecosystem. Specialized libraries and frameworks have been developed to abstract the complexities of parallel architectures, thereby allowing domain scientists to focus on problem-solving rather than the intricacies of hardware management. These tools are indispensable for managing inter-process communication, load balancing, and fault tolerance – critical aspects when operating at large scales.

Looking ahead, emerging technologies such as exascale computing and quantum computing promise to further transform the HPC landscape. These advances aim to extend the boundaries of computational science, enabling simulations and analyses that are currently beyond our reach. The continued collaboration among computer scientists, mathematicians, and domain experts will be key to overcoming the challenges posed by these new frontiers, ensuring that HPC remains at the cutting edge of scientific discovery.

1.2.1 GPU Acceleration and Partitioned Architectures in Modern HPC

In recent years, Graphics Processing Units (GPUs) have become a cornerstone of high performance computing, significantly reshaping the architecture and performance capabilities of modern supercomputers. Unlike traditional CPUs, which offer a relatively small number of powerful cores optimized for sequential execution, GPUs consist of thousands of simpler cores capable of executing massively parallel workloads with high throughput. This architectural difference makes GPUs particularly well-suited for numerically intensive tasks such as matrix operations, particle simulations, and finite-difference computations that dominate scientific applications.

Many of the world's leading HPC systems – including Leonardo (Italy), LUMI (Finland), and Frontier (USA) – derive a substantial portion of their computational power from large-scale GPU partitions. These GPU-enabled partitions are not mere accelerators; they often act as the primary computational resource, with CPUs coordinating high-level control and data orchestration. For example, the Frontier supercomputer uses AMD Instinct MI250X GPUs in combination with EPYC CPUs to achieve exascale performance, with the vast majority of FLOPs delivered by the GPUs.

Effective utilization of GPU-based partitions requires careful consideration of data locality, memory hierarchy (e.g., shared vs. global memory on GPUs), and parallel granularity. Programming models such as CUDA, HIP, OpenACC, and OpenMP offloading are commonly used to harness GPU capabilities, often through abstraction layers or domain-specific libraries (like OP2 or OPS) that shield application developers from low-level complexity.

Partitioning plays an equally critical role in the design and performance of HPC systems. At the cluster level, workloads are distributed across nodes using partition-aware job schedulers and MPI-based communication. Within a node, further partitioning occurs across NUMA domains and GPU devices, which can be independently managed for compute or memory tasks. Modern resource managers (e.g., SLURM, PBS) allow users to specify job partitions explicitly, targeting specific node types (e.g., GPU vs. CPU-only) or even GPU models (e.g., A100, V100). This allows for hardware-aware scheduling, energy optimization, and better resource utilization.

Partitioning is not only a hardware-level concern but also a software design principle. Algorithms must be structured to accommodate domain decomposition, load balancing, and overlapping communication with computation. Libraries such as ParMETIS and Scotch help in graph partitioning, ensuring minimal communication overhead across partitions. In large-scale simulations, especially those using unstructured meshes, partition-aware computation is essential for both scalability and reproducibility.

1.3 Floating-Point precision

Floating-point arithmetic (FP) is a cornerstone of modern scientific computing, enabling the representation and manipulation of real numbers in a format suitable for digital computation [5]. Unlike integer arithmetic, which deals with discrete values, floating-point arithmetic approximates real numbers using a finite number of bits. This approximation introduces inherent limitations, but it also allows for the representation of a wide range of magnitudes, from the extremely small to the extremely large. This capability is essential for tackling complex problems in fields such as physics, engineering, finance, and machine learning.

A floating-point number is represented in the form:

$$(-1)^{sign} \times significand \times base^{exponent}$$

where:

- *sign* is a single bit indicating the number's polarity (positive or negative).
- *significand* (also known as mantissa or coefficient) is a digit string of a given length in a given radix (or base). It determines the precision to which numbers can be represented.

- *base* is the base of the number system used (typically 2 for binary floating-point numbers).
- *exponent* is a signed integer that modifies the magnitude of the number.

The IEEE 754 standard[6] defines several floating-point formats, including single-precision (32-bit), double-precision (64-bit), and half-precision (16-bit). Each format allocates a specific number of bits to the sign, exponent, and significand, thereby determining the range and precision of representable numbers. For example, a double-precision number has a wider exponent range and a larger significand than a single-precision number, allowing it to represent a broader range of values with higher accuracy.

Floating-point operations, such as addition, subtraction, multiplication, and division, approximate the corresponding real number arithmetic operations [5]. Because the result of an operation may not be exactly representable as a floating-point number, it must be rounded to the nearest representable value. The IEEE 754 standard specifies several rounding modes, including round to nearest, round up, round down, and round toward zero [6]. The choice of rounding mode can affect the accuracy and stability of numerical algorithms [5]. Rounding error is a characteristic feature of floating-point computation [7].

The range of floating-point numbers is determined by the number of bits allocated to the exponent. A double-precision (64-bit) binary floating-point number has an exponent of 11 bits, which means the complete range of positive normal floating-point numbers in this format is from approximately 2×10^{-308} to approximately 2×10^{308} [5].

This dissertation focuses on two critical aspects of floating-point arithmetic: *bitwise reproducibility* and *reduced/mixed-precision computing*. Bitwise reproducibility aims to ensure that, given the same inputs, a floating-point computation produces identical results across executions on the same platform, and across different platforms that conform to the same floating-point standard and maintain consistent execution behavior. Reduced-precision computing explores the use of lower-precision floating-point formats to improve performance and energy efficiency, while mixed-precision computing combines different precision levels within a single computation to optimize accuracy and performance.

1.3.1 Non-Associativity and Round-off Error

A fundamental challenge in floating-point arithmetic is its non-associativity. In real number arithmetic, the order in which operations are performed does not affect the result (e.g., $(a + b) + c = a + (b + c)$). However, due to the limited precision of floating-point representation and the need for rounding, this property does not generally hold for floating-point arithmetic [8].

Before presenting the concrete associativity example, I introduce two key concepts for quantifying numerical errors:

- *Absolute error*: $\epsilon_{\text{abs}} = |\hat{x} - x|$ measures how far the computed value $\hat{x}$ is from the true value x.

- *Relative error*: $\epsilon_{\text{rel}} = \frac{|\hat{x}-x|}{|x|}$ indicates how significant the error is with respect to the magnitude of x.

In scientific computing, relative error is usually the more relevant measure, as it reflects the proportional accuracy of a computation.

Consider the following example:

$$a = 1.0 \times 10^4, \quad b = -1.0 \times 10^4, \quad c = 1.0$$

In real number arithmetic, $(a + b) + c = a + (b + c) = 1.0$. However, in floating-point arithmetic with limited precision, we might have:

$$(a + b) + c = (1.0 \times 10^4 + (-1.0 \times 10^4)) + 1.0 = 0.0 + 1.0 = 1.0$$

but

$$a+(b+c) = 1.0\times10^4+(-1.0\times10^4+1.0) = 1.0\times10^4+(-9999.0) = 1.0\times10^4+(-1.0\times10^4) = 0.0$$

This result emerges from the limitations of the floating-point representation format – in this case, IEEE 754 binary16 (half precision). The key issue is that the mantissa (or significand) has finite length. During the computation $-1.0 \times 10^4 + 1.0$, the smaller number (1.0) is shifted right to align exponents, causing its least significant bits to be discarded – effectively rounding it to zero. This phenomenon is known as *loss of significance* or *catastrophic cancellation.*

This non-associativity can lead to significant problems, especially in large-scale high-performance computing (HPC) simulations[8]. In such scenarios, a vast number of floating-point operations are performed, and round-off errors can accumulate and propagate, leading to inaccurate or even unstable results. The order in which these operations are performed can significantly impact the final outcome.

The effects of catastrophic cancellation can be quantified more precisely through error propagation analysis. For example, in the subtraction of two nearly equal values a and b, the relative error of the result can be bounded as follows:

$$\delta(a-b) \leq \frac{|a| \cdot \delta a + |b| \cdot \delta b}{|a-b|}$$

This inequality shows that as a approaches b, the denominator becomes small and the relative error increases significantly, even if δa and δb are small. This is why subtraction is particularly vulnerable to amplification of error in floating-point systems.

It has to be noted here that the differently accumulated roundoff error in most cases should not change the validity of an application [9]. Changing the execution order of an algorithm may still produce valid results, independently of being reproducible.

A related example is the summation of a convergent series such as the harmonic series $\sum_{n=1}^{\infty} \frac{1}{n}$. In floating-point arithmetic, adding terms in decreasing order may lead to inaccurate results, because the smaller terms may be truncated entirely. This motivates summing values in increasing order of magnitude – a well-known numerical strategy to reduce rounding error.

These considerations are critical when designing reliable and reproducible HPC simulations, especially in parallel environments.

We can observe this effect in Figure 1.1, using a more realistic finite element method example with a conjugate-gradient solver, with calculations in double precision (Aero [10]—detailed in Section 2.2.2). On this histogram, we counted the number of different values of the end results in relative differences for several magnitudes between running the application using eight processes and 16 processes. From the 6.5M elements, there were only 3599, which had a bitwise identical result, the rest had a difference between 10^{-12} and 10^{-4}, and most of them were around the magnitude of 10^{-8}.

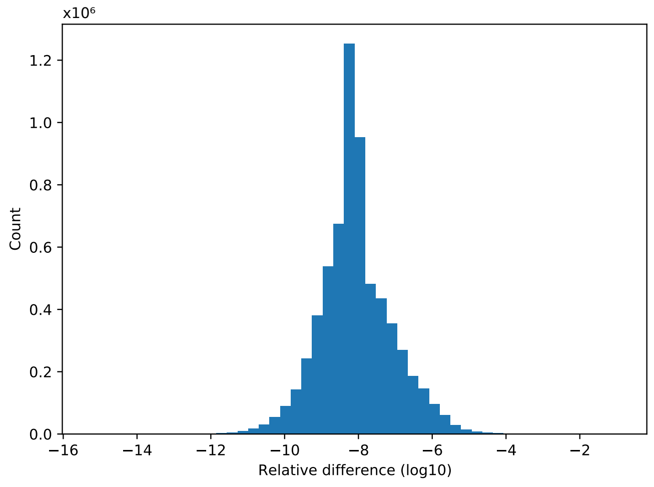


Figure 1.1: Histogram, showing the relative differences in a conjugate-gradient solver (Aero) between runs with eight processes and 16. The result converges to a numerically stable state, but on average there is a 4.05×10^{-7} difference.

Reproducibility

Reproducibility is often understood as experimental reproducibility. This is also a widely researched topic [11]–[15], but my aim is to obtain bitwise identical results of an application run with the same input parameters regardless of the level of parallelism, be it the number of threads or processes executed simultaneously. Non-reproducibility is not caused by the roundoff error but by the non-determinism of accumulative roundoff error. Due to the non-associativity of floating-point addition, accumulative roundoff errors depend on the order of evaluation, which is almost always relaxed in parallel and distributed environments. In a distributed MPI environment, there are multiple possible sources of non-associativity: number of MPI nodes, MPI reduction tree shape, number of cores per node, and data ordering. The histogram in Figure 1.1, which runs the Aero benchmark of the OP2 library, shows the relative differences $(\frac{(a-b)}{a} \mid a > b)$ of a non-reproducible application run with different numbers of MPI processes. In general, some of the causes might be efficiently addressed, such as the reduction tree shape, which can be defined by network interface cards [16], but changing the number of processes can cause issues that are not as easily addressed. A general solution might be to fix the order of evaluation but that is, in many cases, incompatible with parallelization, and running sequentially is prohibitively costly. Another solution is to eliminate rounding errors. We can use exact arithmetics [17], but that will substantially increase the memory usage and the cost of the computations, as well as the amount of communication when applied to more complicated operations such as matrix multiplication. Higher precision can be used, but it will be reproducible only with higher probability [18].

Reproducible Reductions

One of the most common sources of non-reproducibility comes from reductions, where we add up the elements of an array into a single result. When carrying this out with a parallel execution, the rounding errors can accumulate rapidly. There are multiple solutions for this problem [19]–[21], but the underlying observation is common to all approaches; adding up numbers with similar magnitudes is going to be exact. Demmel et al. [19] use pre-roundings to a well-calculated magnitude with an extra sweep through the array, add the values together, and then apply the same method on the remainders of the roundings. Arteaga et al. [20] extended their work by calculating the magnitudes without the additional sweep. The ReproBLAS library [21] creates bins for the magnitudes in advance and uses them in parallel for the summations. In our project, we use ReproBLAS, due to its user-friendly implementations; though the necessary reductions can be calculated by using other techniques as well.

ReproBLAS

Reproducible Basic Linear Algebra Subprograms [21] (ReproBLAS), intends to provide users with a set of parallel and sequential linear algebra routines that guarantee bitwise reproducibility independent of the number of processors, data partitioning, reduction scheduling, or the sequence in which the sums are computed in general. It assumes that floating-point values are binary and conform to IEEE Floating-Point Standard 754-2008, floating-point operations are conducted in ROUND-TO-NEAREST mode (ties may be broken at will) and that underflow happens gradually. Summing n floating-point values with their default settings costs around $9n$ floating-point operations (arithmetic, comparison and absolute value). The new "augmented addition" and "maximum magnitude" instructions in their proposed IEEE Floating-Point Standard 754-2019 [6] can theoretically reduce this count to $5n$. On a single Intel Sandy Bridge core, for example, the ReproBLAS slowdown compared to a performance-optimized non-reproducible dot product is $4\times$ [22]. Here, the output is reproducible regardless of how the input vector is permuted. For the summing of 1,000,000 double-precision floating-point (FP64) values, the slowdown on a large-scale system with more than 512 Intel "Ivy Bridge" CPUs (the Edison machine at NERSC) is less than $1.2\times$. The result is also reproducible regardless of how the input vector is partitioned across nodes or how the local input vector is stored within a node.

Bitwise Reproducibility in Parallel HPC Applications

While roundoff errors are expected and often tolerable, non-determinism in their accumulation poses a unique challenge in HPC. This is especially true in parallel or heterogeneous systems, where factors such as the number of MPI processes, thread scheduling, or network-level reduction trees influence the order of floating-point operations. As a result, repeated executions with identical inputs may diverge at the bit level. These discrepancies are not inherently errors, but they complicate debugging, verification, and scientific reproducibility.

The degree of reproducibility required varies across domains. In some cases, relative agreement within machine epsilon suffices. However, there are many applications – for example, in regulatory simulations, cross-architecture validation, or industrial workflows – where bitwise identical output is required. Examples include structural wind vulnerability prediction [23] or nonlinear aeroelastic analysis [24], where repeatability is crucial for safety or engineering certification.

To address this, multiple algorithmic techniques have been proposed. A widely known approach is Kahan's compensated summation [25], which stores low-order errors during accumulation to preserve precision. More sophisticated strategies, such as the binned summation method by Demmel et al. [19], separate values by magnitude before summation, effectively controlling rounding error propagation. These approaches improve determinism but often come with overhead, intrusive changes to code, or limitations to specific architectures.

The reproducibility challenge becomes especially pronounced in unstructured mesh applications, where computation frequently involves indirect memory access patterns and data races. In such scenarios, the cost of enforcing strict operation ordering can outweigh the gains from parallelization. Emerging methods aim to strike a balance: maintaining acceptable performance while enforcing reproducibility. Notably, the ReproBLAS library [21] provides drop-in reproducible versions of BLAS routines, with overheads as low as 1.2× in massively parallel environments, such as NERSC's Edison system [22].

1.3.2 Precision Trade-offs in HPC

The choice of floating-point precision involves a trade-off between accuracy, performance, and memory usage [26]. Higher precision formats, such as double-precision, provide greater accuracy and a wider dynamic range but require more memory and computational resources. Conversely, lower precision formats, such as single-precision and half-precision, offer improved performance and reduced memory footprint at the cost of reduced accuracy and dynamic range [26].

To better understand these trade-offs, it is useful to examine the numerical properties of standard floating-point formats as defined by the IEEE 754 standard. Table 1.1. summarizes the key characteristics of single (32-bit) and half (16-bit) precision:

Table 1.1: Comparison of IEEE 754 Half and Single Precision Formats

Property	**Half Precision (FP16)**	**Single Precision (FP32)**
Total bits	16	32
Significand (mantissa) bits	10 (plus implicit 1)	23 (plus implicit 1)
Exponent bits	5	8
Exponent bias	15	127
Approximate decimal precision	$\sim$3.3 digits	$\sim$7.2 digits
Smallest positive normal value	6.1×10^{-5}	1.2×10^{-38}
Largest finite value	6.5×10^{4}	3.4×10^{38}
Machine epsilon (ε)	9.8×10^{-4}	1.2×10^{-7}

As the table shows, half precision offers significantly reduced representable range and decimal precision compared to single precision. For example, FP16 can represent values roughly between 10^{-5} and 10^{4} with only around three decimal digits of precision, while FP32 spans over 30 orders of magnitude and supports over seven digits of precision. This has direct implications in scientific computing, where algorithms must be analyzed for sensitivity to rounding errors and underflow/overflow risk.

In many high-performance computing (HPC) applications, performance is often limited by memory bandwidth [26]. The time required to transfer data between memory and the processor can be a bottleneck, rather than the floating-point operations themselves. Using lower precision formats can significantly improve performance by reducing the amount of transferred data. For example, replacing double-precision with single-precision can halve the memory footprint and potentially double the memory bandwidth, leading to significant speedups, provided that the reduced precision does not compromise the accuracy of results [26].

Modern hardware accelerators, such as GPUs and specialized AI accelerators, offer significantly higher throughput for lower precision floating-point operations [26]. This makes reduced-precision computing an attractive option for accelerating a wide range of applications, including machine learning, signal processing, and scientific simulations. However, numerical stability and accuracy requirements must be carefully analyzed before adopting reduced-precision formats, as inappropriate usage may lead to unacceptable errors or instability [26]. Mixed-precision computing provides a potential solution by allowing different parts of the computation to be performed at different precision levels, optimizing both accuracy and performance [26].

The advent of artificial intelligence has further encouraged the adoption of reduced precision formats in hardware, such as half-precision (16-bit) floating-point arithmetic. While this transition presents opportunities for significant performance gains, it also raises concerns regarding numerical stability, particularly in HPC applications sensitive to precision. The challenge lies in balancing precision and performance to meet the growing demand for computational efficiency.

Existing mixed precision techniques have demonstrated promise in various applications.

For instance, many linear solvers have successfully employed mixed precision to improve performance without sacrificing accuracy [27]–[31]. Abdelfattah et al. [32] reported advancements in mixed precision algorithms, including speedups in dense and sparse LU factorization, eigenvalue solvers, and GMRES implementations. Computational fluid dynamics (CFD) applications, such as FluidX3D [33] and OpenFOAM [34], have also integrated mixed precision strategies to enhance computational efficiency while maintaining accuracy.

Many finite difference methods (FDMs) exhibit a structural pattern where a smaller value is iteratively added to a larger one. This structure enables strategic precision allocation: the larger value, which accumulates multiple iterations and is prone to roundoff errors, is typically maintained in full precision, while the smaller update values, which are discarded after each step, can be represented in reduced precision formats. This controlled trade-off enhances computational efficiency without excessive loss of accuracy.

Challenges of Using Insufficient Precision

Employing inadequate floating-point precision can lead to significant computational errors, particularly in CFD applications where numerical stability is critical. Key issues include:

- **Round-off Errors:** Limited mantissa length causes rounding errors, which can accumulate in iterative methods and lead to divergence [35].
- **Overflow and Underflow:** Floating-point arithmetic constraints can result in catastrophic failures when numbers exceed the representable range.
- **Loss of Significance:** Subtracting nearly equal numbers can lead to catastrophic cancellation, significantly affecting CFD simulations [36].

These challenges highlight the importance of selecting an appropriate precision level that balances computational efficiency with accuracy requirements.

Half-Precision Floating-Point Arithmetic

Half-precision floating-point arithmetic (FP16) has gained traction in HPC due to its balance between computational efficiency and memory usage. Recent NVIDIA GPU architectures, such as Volta and Ampere, have introduced native FP16 support, enabling substantial performance improvements in scientific computations. Studies have shown that employing mixed precision techniques, which integrate FP16 with higher precision formats, can achieve speedups of up to four times in iterative refinement solvers [37]. This acceleration is primarily due to enhanced computational throughput enabled by Tensor Cores optimized for FP16 arithmetic. Additionally, FP16 reduces memory bandwidth requirements, facilitating faster data transfers and improved overall performance in large-scale simulations [38].

Despite its advantages, transitioning to FP16 is not without challenges. Its limited range and precision can introduce numerical instability, particularly in operations requiring high accuracy. Catastrophic cancellation is a notable concern, especially in time-marching methods where small perturbations can accumulate over iterations. To mitigate these issues, researchers have proposed techniques such as iterative refinement and modified LU factorization, which enhance numerical stability while leveraging FP16's performance benefits [39].

Beyond FP16, alternative reduced precision formats are being explored. Bfloat16 [40], which retains FP32's exponent size while reducing the significand to 7 bits, has proven useful in deep learning due to its stable training properties. However, despite its advantages, bfloat16 has precision limitations that can lead to numerical instability in certain computational scenarios. Even smaller floating-point formats, such as float8 [41], have been proposed, though their accuracy challenges make them less suitable for general scientific computing. Given the difficulties encountered with FP16, using even lower precision formats is generally discouraged.

In the following chapters, this dissertation will dive into the challenges and opportunities presented by bitwise reproducibility and reduced/mixed-precision computing, exploring novel algorithms and techniques to address the limitations of floating-point arithmetic and harness its full potential for scientific discovery.

1.4 The OPS and OP2 Domain Specific Libraries

1.4.1 The Unstructured Mesh Computational Motif

Computations defined on unstructured meshes form an important basis for many engineering calculations commonly used in PDE discretizations, such as finite elements of finite volumes. An unstructured mesh is characterized by a number of sets (vertices, edges, cells, etc.) with explicit connectivity information between them (e.g., edges to vertices). Computations are commonly expressed as a parallel loop over a set, with computations accessing data either directly on the iteration set or through an indirection. For example, a common operation in computational fluid dynamics is to compute fluxes across faces (edges), and then increment/decrement state variables defined on connected cells. The key motif here is the edge-centered computations indirectly incrementing cell data, which then gives rise to non-determinism when the order of execution of the edges is relaxed for the sake of parallelism. Another common pattern is the global reduction, often conducted in a non-deterministic order, where the result is then used in subsequent computations. For example, in the conjugate gradient algorithm, the results of dot products are used as weights in the next step.

The distributed and parallel execution of unstructured mesh algorithms is a well-established field [42]–[45]. For distributed memory execution, the mesh is partitioned using one of many established libraries, such as PT-Scotch or ParMetis [46], [47]. It is important to note here that an unstructured mesh is a hypergraph, consisting of multiple "vertex" types, whereas most partitioners only partition a simple graph, and the rest of the hypergraph is usually partitioned in a greedy way through connections to the simple graph. This is then related to how computations are executed, an "owner-compute" approach is commonly utilized, where all computations associated with a given element are performed on the process that owns that element. So, for instance, in the earlier example, the process that owns a given cell will execute all the edges that increment that cell, even if some of those edges are not owned by it. This requires communicating all the data needed to execute those edges as well. This often leads to redundant computations around partition boundaries. Depending on the exact implementation the deterministic order of execution for elements is often relaxed at this point to allow shared memory parallelization and powerful optimizations such as overlapping computations and communications.

To enable shared-memory parallel execution of unstructured mesh computations, one needs to address the issue of race conditions when indirectly incrementing/updating data. Virtually all execution schemes used in the literature rely on the associativity of these operations, for example, by using atomic updates, a staging of increments in an auxiliary array and their separate sum, or a coloring scheme [48], [49]. We are not aware of related works that explicitly aim to maintain an ordering of operations whilst enabling shared memory parallel execution.

1.4.2 OP2 Domain-Specific Library

OP2 is a domain-specific library (DSL) designed to facilitate the development of portable and efficient applications operating on unstructured meshes [45]. It provides a high-level abstraction for expressing mesh-based computations, separating the specification of numerical algorithms from their parallel implementation across diverse hardware platforms, including CPUs, GPUs, and clusters.

The OP2 framework is centered around a small set of key abstractions:

- **Sets**: Represent collections of topological entities in the mesh, such as edges, cells, or vertices.
- **Maps**: Define connectivity between sets, by listing connections from one set to another. For example, a map from edges to the two adjacent cells. A map in OP2 must have the same number of elements pointed to from each element of the first set.
- **Datasets** (`op_dat`): Store data associated with a set, such as pressure or velocity fields.
- **Parallel loops** (`op_par_loop`): Describe computations over elements of a set, using user-defined kernel functions.

Listing 1.1 demonstrates how these abstractions are used in practice. The example code shows a parallel loop over the `edges` set, applying a kernel function `res` to perform updates on data defined on adjacent `cells`, via an indirection map.

Listing 1.1: Specification of an OP2 parallel loop

```
/* ----- elemental kernel function in res.h ------*/
void res(const double *edge,
         double *cell0, double *cell1 ){
```

```
  //Computations, such as:
  *cell0 += *edge; *cell1 += *edge;
}
/* ———————— in the main program file ————————*/
// Declaring the mesh with OP2 sets
op_set edges = op_decl_set(numedge, "edges");
op_set cells = op_decl_set(numcell, "cells");
// mappings - connectivity between sets
op_map edge2cell = op_decl_map(edges, cells,
                    2, etoc_mapdata,"edge2cell");
// data on sets
op_dat p_edge = op_decl_dat(edges,
                1,"double",edata,"p_edge");
op_dat p_cell = op_decl_dat(cells,
                4,"double",cdata,"p_cell");
// OP2 parallel loop declaration
op_par_loop(res,"res", edges,
  op_arg_dat(p_edge,-1,OP_ID      ,4,"double",OP_READ),
  op_arg_dat(p_cell, 0,edge2cell,4,"double",OP_INC ),
  op_arg_dat(p_cell, 1,edge2cell,4,"double",OP_INC));
```

The key mechanism in OP2 is the use of the `op_par_loop` construct, which applies a kernel function across elements of a set. The arguments to this loop specify:

- the dataset involved (`op_dat`),
- the indirection (if any) via an `op_map`,
- the access mode (`OP_READ`, `OP_WRITE`, `OP_INC`, etc.), and
- the type and arity of the data.

In the example, `p_edge` is accessed directly with `OP_READ`, while `p_cell` is updated via an indirection map, using `OP_INC` (increment), reflecting a common pattern in unstructured mesh computations.

OP2 distinguishes between two types of loops:

- **Direct loops**: All data is accessed directly on the iteration set. These are free from data races and inherently parallelizable.
- **Indirect loops**: At least one `op_dat` is accessed indirectly through a mapping to another set. These are the general case in unstructured mesh computations and may involve data hazards, requiring careful parallelization strategies (e.g., coloring, atomics, or staging).

An important constraint imposed by the OP2 API is that the execution order of iterations in an `op_par_loop` must be semantically interchangeable—that is, the operations performed within the loop must be mathematically associative, so that reordering them does not affect the final result beyond variations within machine epsilon. This contract enables OP2 to apply aggressive parallel execution and optimization strategies, such as dynamic scheduling, coloring, or hardware-specific kernel transformations. Consequently, applications must be written to respect this constraint, relinquishing control over loop execution order and avoiding order-dependent side effects in kernel functions. This design choice is crucial not only for performance portability, but also for enabling reproducibility features, as it permits the library to re-structure computations in deterministic ways when required.

This access-execute model allows OP2 to automatically generate platform-optimized code, targeting backends such as OpenMP, CUDA, OpenCL, and MPI. By abstracting away the parallelization logic, OP2 ensures performance portability and maintainability, enabling users to write a single high-level description of their numerical algorithm.

Several real-world and production-scale applications have successfully adopted OP2 to achieve performance portability and maintainability across heterogeneous platforms. The VOLNA-OP2 tsunami simulation code demonstrates the applicability of OP2 to geophysical modeling, achieving high performance on modern parallel architectures while maintaining numerical fidelity [50]. In the aerospace domain, Hydra, a full-scale industrial CFD application developed by Rolls-Royce, was ported to OP2 and accelerated across CPU and GPU platforms with significant performance gains, highlighting OP2's ability to handle complex legacy codebases [51]. Additionally, the MG-CFD mini-application has been developed as a compact, research-oriented code that leverages OP2 (and later SYCL) to explore strategies for achieving performance portability in multi-grid CFD solvers [52]. These applications illustrate the practical maturity and flexibility of the OP2 model for unstructured mesh computations in both industrial and academic contexts.

In this dissertation, OP2 serves as the foundation for implementing bitwise reproducibility techniques in unstructured mesh applications. Because OP2 has full control over execution order and data access patterns, it is well-suited to introducing deterministic variants of indirect operations – such as accumulation using temporary arrays or coloring – and integrating external libraries like ReproBLAS for reproducible reductions.

Overall, OP2 enables concise, readable code that can scale efficiently across modern heterogeneous architectures, while also supporting advanced capabilities like reproducibility, debugging, and algorithmic experimentation.

1.4.3 OPS Domain-Specific Library

The Oxford Parallel library for Structured mesh solvers (OPS) [53] is a domain-specific library aimed at simplifying the development of stencil-based applications operating over structured Cartesian meshes. OPS provides a high-level abstraction that allows developers to describe numerical algorithms independently of the underlying hardware or parallelization strategy. This enables performance portability across diverse computing architectures, including multi-core CPUs and GPUs.

OPS applications are structured around a set of well-defined abstractions:

- **Blocks**: Represent topological regions of the computational domain (e.g., a structured grid).
- **Datasets** (`ops_dat`): Multi-dimensional arrays (fields) defined on blocks, storing quantities like pressure or velocity.
- **Stencils**: Define relative access patterns to neighboring data points in a dataset, used within a computational kernel.
- **Parallel loops** (`ops_par_loop`): Apply user-defined kernels across elements of a block, using specified stencils and data access modes.

Listing 1.2 shows a minimal example of a 5-point stencil kernel applied to a 2D dataset. The user kernel `stencil` updates a point in `u2` based on neighboring values in `u` and a forcing term `f`, using discrete approximations to the Laplace operator.

Listing 1.2: Example of an OPS parallel loop for a 5-point stencil update in 2D.

```
// User kernel
void stencil(const ACC<double> &u, const ACC<double> &f,
             ACC<double> &u2) {
  u2(0, 0) = ((u(-1, 0) + u(1, 0)) * dy * dy +
              (u(0, -1) + u(0, 1)) * dx * dx -
              f(0, 0) * dx * dx * dy * dy) /
             (2.0 * (dx * dx + dy * dy));
}
// ...
// Declaring a dataset on a block
int size[2] = {size_x, size_y};
int base[2] = {0, 0};
int d_p[2] = {1, 1};  // positive halo depth
int d_m[2] = {-1, -1}; // negative halo depth
ops_dat u = ops_decl_dat(block, 1, size, base, d_m, d_p,
                         nullptr, "double", "u");
// Execute a given loop on the block
int iter_range[] = {0, size_x, 0, size_y};
ops_par_loop(stencil, "stencil", block, 2, iter_range,
  ops_arg_dat(u,  1, S2D_4PT, "double", OPS_READ),
  ops_arg_dat(f,  1, S2D_00, "double", OPS_READ),
  ops_arg_dat(u2, 1, S2D_00, "double", OPS_WRITE));
```

The kernel function is expressed in terms of the `ACC` wrapper, which provides indexed access to data in the stencil neighborhood. In this example, `u(-1, 0)` accesses the value of `u` one cell to the left of the current grid point.

OPS manages all parallel execution and halo exchange logic. Similar to OP2, OPS assumes that the order of execution of iterations in an `ops_par_loop` is interchangeable within machine epsilon. This constraint allows OPS to exploit a wide range of optimization strategies, such as loop tiling, or parallel scheduling across distributed memory, while ensuring the correctness of the results under relaxed ordering. Consequently, OPS users must avoid introducing order-dependent side effects in kernel code and should write stencil kernels that are agnostic to iteration ordering. Each `ops_dat` is defined with its halo depths (`d_m` and `d_p`), enabling efficient communication of ghost cells in distributed memory environments. The execution region is defined by the iteration range, and stencil patterns such as `S2D_4PT` specify which neighboring points are required during computation.

Data access modes (e.g., `OPS_READ`, `OPS_WRITE`, `OPS_INC`) inform OPS of dependencies, allowing it to perform data movement, synchronization, and loop fusion optimizations automatically. Importantly, OPS restricts write access in stencil computations to only the center of the stencil – the element at offset `(0,0)`. This design simplifies dependency analysis and guarantees data race freedom, but also imposes a discipline on kernel authors: any update to a dataset must occur only at the current iteration point, not at neighboring locations. This constraint aligns with OPS's goal of maintaining safe and portable parallel execution across diverse hardware backends.

OPS applications are backend-agnostic: the same high-level code can be compiled into parallel implementations using OpenMP, MPI, CUDA, or OpenCL, depending on the target hardware. Source-to-source translation is used to generate optimized implementations for each backend, abstracting away the details of low-level parallel programming.

OPS also supports:

- **Per-dataset precision control**: Each `ops_dat` can be declared with its own floating-point type (e.g., float, double, or half), enabling memory-efficient storage tailored to the accuracy requirements of each field.
- **Multi-block domains**: Supporting more complex geometries or decompositions across structured blocks.
- **Backend portability**: Achieved through code generation and scheduling optimizations across OpenMP, MPI, CUDA, and other targets.

A notable feature of OPS is its ability to declare datasets with different precisions, allowing developers to express varying numerical accuracy requirements across different fields in a simulation. This design enables precision-aware memory optimization and data management. However, prior to the work presented in this dissertation, OPS lacked support for *true mixed-precision computations* – that is, stencil kernels combining operands of different precisions within a single operation or across interdependent datasets.

As part of this research, support for such mixed-precision arithmetic was introduced into OPS, enabling computations that combine datasets of different types directly within user-defined kernels. This advancement extends OPS's flexibility beyond memory layout and storage into the computational domain, making it possible to systematically explore trade-offs between performance, accuracy, and energy efficiency in real-world stencil-based applications.

OPS has been successfully employed in a range of structured-mesh applications to demonstrate both performance portability and scalability on modern high-performance computing systems. One such example is CloverLeaf, a hydrodynamics mini-application representative of many production-level shock physics codes. By porting CloverLeaf to OPS, researchers introduced features such as dynamic loop tiling and multi-target backend support without altering the high-level algorithmic structure [54]. Another notable example is OpenSBLI, an automated code generation framework for finite-difference solvers, which uses OPS as its parallel back end. OpenSBLI is used extensively in this research and will be introduced in detail in the following section [55].

1.4.4 OpenSBLI

OpenSBLI [55] is a complete code generation system for computational fluid dynamics that automatically generates OPS code starting from a compact description of the governing equations using subscript notation. It uses symbolic Python to expand the governing equations and carry out the subsequent discretisation, using high-order finite differences. The frontend is used to define and expand the governing equations and constituent relations. It also defines all the boundary and initial conditions and sets all the run time parameters. This provides an effective implementation of the 'separation of concerns' workflow [56], whereby users focusing on the fluid dynamics can carry out most tasks from the python frontend. Alternatively, numerical methods developers can work either in the OPSC code or in the OpenSBLI code generation, while computer-science-based performance optimisations can be carried out in the OPS translator. The main applications of OpenSBLI are direct numerical simulation (DNS) and large eddy simulation (LES) of compressible-flow problems involving transition to turbulence or fully-developed turbulence on structured grids.

Finite difference methods (FDMs) form the mathematical foundation underlying most of OpenSBLI's numerical strategies. In essence, FDMs approximate derivatives of continuous functions by discrete differences over a structured grid. This approach is particularly efficient for problems defined on regular domains, and supports high-order accurate schemes with straightforward stencil implementations. OpenSBLI leverages explicit high-order central differencing for both convective and diffusive terms, allowing efficient exploitation of structured memory access patterns. Compared to other discretization techniques such as finite-volume or finite-element methods, FDMs offer a favorable balance between simplicity, performance, and accuracy in structured mesh simulations, making them especially attractive in the context of GPU-accelerated large-scale DNS and LES workflows.

Definition of the Blaisdell quadratic split-form: for convective terms of the form

$$\mathcal{C} = \frac{\partial(\rho u_j \varphi)}{\partial x_j},$$

the Blaisdell form rewrites this as a skew-symmetric or quadratic average to reduce aliasing:

$$\mathcal{C}_{\text{Blaisdell}} = \tfrac{1}{2}\left[\rho u_j \frac{\partial \varphi}{\partial x_j} + \varphi \frac{\partial(\rho u_j)}{\partial x_j}\right],$$

which preserves kinetic energy in the inviscid limit and improves numerical robustness by mitigating aliasing errors [57].

An initial version (V1) of the software [58] demonstrated the code-generation concept for simple low-speed periodic flows and the benefits of the OPS code translation in being able to run efficiently on heterogeneous computing architectures. The project was restarted from a clean code base in [59], which led to the subsequent public code release in [55]. This version (V2) included shock capturing using Weighted Essentially Non-Oscillatory (WENO) and less dissipative Targeted Essentially Non-Oscillatory (TENO) schemes along with a greatly expanded set of boundary-conditions and application demonstrations, including shock-wave/boundary-layer interactions [59] and channel flow validation test cases [60]. This release also included generalised curvilinear geometries. The current release version of OpenSBLI is V3 [61]. Version 3 added multi-block mesh support, airfoil simulations, new numerical methods and filters, and various performance and efficiency improvements. Reduced-dimension I/O, partial reductions for spanwise averaging, and mixed-precision support were also added, via upgrades to both OpenSBLI and the OPS library. The combination of multi-block capability and new filter-based shock-capturing schemes [61] has led to new applications of implicit LES simulations of flow over airfoils, including state-of-the-art studies of airfoil buffet in the transonic flow regime [62], [63] that exploit large GPU-based machines to perform high-fidelity simulations on the order of $N \sim 10^{10}$ mesh points.

Flow Simulations and Discretization of Models

In computational fluid dynamics (CFD), simulating fluid flow involves solving the governing equations – e.g. the compressible Navier–Stokes equations – on a discrete computational mesh. These equations encapsulate the conservation of mass, momentum, and energy within the fluid domain. To translate them into a solvable numerical format, discretization techniques are applied. In the context of this work, high-order finite difference methods on structured grids are used for this purpose, as facilitated by the OpenSBLI framework. The resulting discrete equations are advanced in time using explicit Runge–Kutta schemes, while spatial derivatives are evaluated via central differencing. Additional stabilization is achieved through split formulations such as the Blaisdell quadratic form, which mitigate aliasing errors and improve robustness at high Reynolds numbers. These methods enable accurate and scalable simulations of turbulent flow phenomena, particularly in GPU-based exascale environments.

Numerical Methods and Split Formulations of the Equations

The accuracy of low- and mixed-precision algorithms is assessed in this work in the context of the unsteady full-3D Navier-Stokes equations for a compressible Taylor-Green vortex problem described in Section 3.3.1. All simulations are performed using explicit 4th-order accurate non-dissipative central-differencing. Both the convective and diffusive parts of the Navier-Stokes equations are solved at 4th-order to maintain consistent spatial order throughout. Time-stepping is performed by a low-storage explicit 3rd-order Runge-Kutta scheme. Details of the implementation have previously been given in [55].

Depending partly on the magnitude of the Reynolds number, direct application of standard central derivative approximations to the Navier-Stokes equations can lead to numerical instabilities [64]. This is due to an accumulation of aliasing errors that occurs from discrete evaluation of the product between two or more terms within the non-linear convective derivatives. The lack of numerical robustness can also be linked to the failure of standard formulations to discretely preserve quadratic invariants such as global kinetic energy in the inviscid limit [65]. To alleviate these discretisation issues, convective terms of the base Navier-Stokes equations are routinely reformulated in modern CFD codes in what are known as split formulations. These alternative formulations of the governing equations have been reported to improve numerical robustness via reduced aliasing errors and preservation of certain invariant quantities [66].

Various split-forms are available in the literature [66]. They typically focus on the reformulation of non-linear convective derivative terms that take the general form

$$\mathcal{C} = \frac{\partial \rho u_j \varphi}{\partial x_j}, \tag{1.1}$$

where φ takes the value of $(1, u_i, E)$, for the continuity, momentum, and energy components of the Navier-Stokes equations, respectively. As an example of one of the split-forms available in OpenSBLI, the Feiereisen quadratic split-form [67], expands these terms quadratically as

$$\frac{\partial \rho u_j \varphi}{\partial x_j} \to \frac{1}{2}\frac{\partial \rho u_j \varphi}{\partial x_j} + \frac{1}{2}\left(\varphi \frac{\partial \rho u_j}{\partial x_j} + \rho u_j \frac{\partial \varphi}{\partial x_j}\right). \tag{1.2}$$

While the underlying physical equations are mathematically equivalent in an algebraic sense, the split-form is considerably more robust numerically when the equations are evaluated on a discrete mesh with finite-difference approximations [66]. The split-forms are computationally more expensive due to the increased number of floating-point operations required, however, as CFD codes are typically memory-bound, they are an efficient way of improving numerical stability for large-scale calculations at high Reynolds numbers. In this work, the quadratic Blaisdell split-form [57] of the equations is applied to improve numerical robustness throughout. As we are interested in the effect of low- and mixed-precision and the propagation of numerical errors in finite-difference approximations, other quadratic and cubic split-forms are also tested in the inviscid limit with varying numerical precision in Section 3.3.2.

2 Bitwise reproducibility

2.1 Introduction

Floating-point operations are the foundation of modern scientific computing. However, their finite-precision representation means that operations are non-associative [7]. In parallel computing environments, where execution order is not guaranteed, this leads to run-to-run variability even for the same inputs.

This motivates the concept of *bitwise reproducibility* – producing identical binary outputs across repeated executions. While tolerances are sometimes acceptable, exact reproducibility is often required for debugging, port validation, and testing frameworks [68]. Such reproducibility is challenging to maintain efficiently on modern hardware, especially in large-scale or GPU-based simulations.

This chapter presents a reproducibility solution for unstructured mesh computations – an HPC domain prone to indirect memory access and data races. The approach integrates into the OP2 DSL and demonstrates reproducibility in industrial-scale applications such as Rolls-Royce Hydra, while preserving performance across CPU and GPU architectures.

2.2 Backround

2.2.1 Related Works

Bitwise reproducibility is a widely researched problem, usually investigated in a specific application.

Mascagni et al. [69] list the main sources of non-reproducibility in a neuroscience application: (i) the introduction of floating-point errors in an inner product; (ii) the introduction of floating-point errors at each an increasing number of time steps during temporal refinement (ii) and (iii) differences in the output of library mathematical functions at the level of round-off error. They highlight the importance of numerical reproducibility without providing a general solution.

Liyang et al. created a special method [70] for molecular dynamics applications in the LAMPPS Molecular Dynamics Simulator [71]. From each particle, the potentials are calculated first and then stored temporarily. Then they loop over every particle again, sort the components for one element, and accumulate them in ascending order. This way, they were able to eliminate the effect of non-associative accumulation.

Langlois et al. [72] tested multiple techniques for reproducible execution on an industrial free-surface flow application: the 2D simulation of the Malpasset dam break. All methods passed, but their main purpose is to determine how easy it is to use them.

Kahan's compensated solution method [25] appeared to be the easiest to apply and provided accurate results for low computing overhead. The integer conversion provided in Tomawac [73] was also easy to derive and introduced a low overhead. The solution that uses reproducible sums [19] was efficient, but was applied less easily in their case and introduced a significant communication overhead.

He et al. [74] experimented on a dynamical weather science application. They tested several methods, such as Kahan's [25], or the double-double number technique [75] which is an unevaluated sum of two IEEE double precision numbers. They also provide an MPI operator for reductions.

Taufer et al. [76] were looking into a molecular dynamics application, whereby reproducibility meant that results of the same simulation running on GPU and CPU lead to the same scientific conclusions; in their case, bitwise reproducibility was not necessary. They tried double precision arithmetic, which partially corrected the drifting, but was significantly slower than single precision, comparable to CPU performance. They created a library of float-float composite type, which is comparable in accuracy to double, but the performance loss is only 7%, versus a loss of 182% of normal double precision.

Robey et al. also experimented with a dynamical fluid application [77]. They tried to sort their data first and then sum, but that was too slow. They applied Ozawa's pair-wise summation [78], which produced less truncation, but not bitwise reproducibility, although this method is quick and can run in parallel. The double-double technique used too much memory, so finally they used Kahan's [25] and Knuts's [79] approach due to their simplicity, low additional cost and their added precision.

Apostal et al. [80] developed a source code scanner to recognize reductions over MPI in C or C++ codes and automatically modify them to use Kahan's summation [25] or an algorithm developed by Demmel and Nguyen [19].

Olsson et al. [81] defined some transformation techniques to describe concurrent applications written in the SR programming language to achieve reproducibility. They can transform an arbitrary SR program into two parts: one for recording a sequence of events and one for replaying those events.

Reproducible Basic Linear Algebra Subprograms [21] (ReproBLAS), intends to provide users with a set of parallel and sequential linear algebra routines that guarantee bitwise reproducibility independent of the number of processors, data partitioning, reduction scheduling, or the sequence in which the sums are computed in general. The BLAS are commonly used in scientific programs, and the reproducible versions provided in the ReproBLAS will provide high performance while reducing user effort for debugging, correctness checking, and understanding the reliability of programs.

Graph coloring is a widely used method in HPC to maximize parallel efficiency, without facing any race conditions. We can see a detailed example of using coloring techniques in the work of Zhang et al. [82] . Their paper addresses challenges in parallelizing unstructured CFD on GPUs, employing graph coloring for data locality optimization and parallelization, resulting in substantial speed-up with GPU codes outperforming serial CPU versions by 127 times and parallel CPU versions by more than thirty times in the same MPI ranks.

2.2.2 Test Applications

The following applications are implemented in OP2 to evaluate and assess the efficacy and performance of the proposed algorithms.

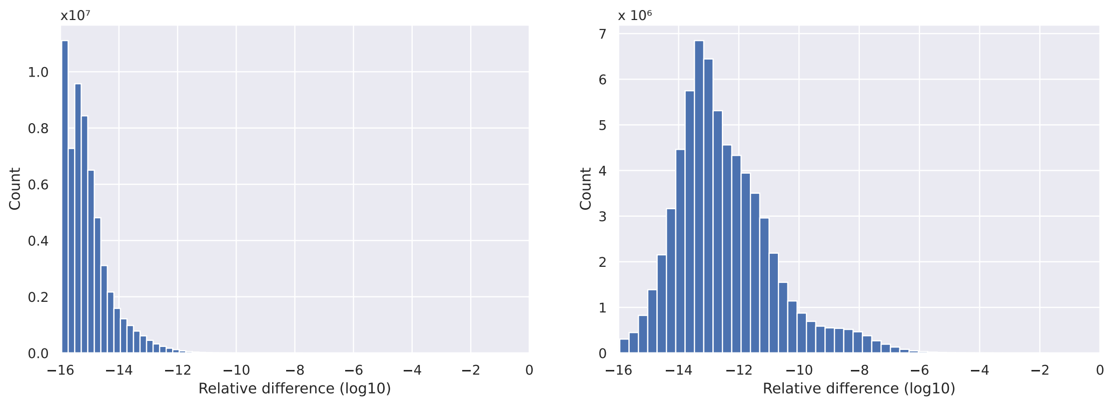
Airfoil [83] is a representative CFD code, written using OP2's C/C++ API. It is a non-linear 2D inviscid airfoil code that uses an unstructured grid. Airfoil uses a finite volume method to solve the steady-flow 2D Euler equations using scalar numerical dissipation. Airfoil is available as part of the OP2 framework.

Aero [10] is a 2D non-linear steady potential flow simulation of air moving around an airfoil, developed based on standard finite element methods. It uses a quadrilateral grid similar to that used by the Airfoil application but uses a Newton iteration to solve the non-linear equations defined by a finite element approximation. Each Newton iteration requires the solution of a linear system of equations. The assembly algorithm is based on quadrilateral elements and uses transformations from the reference square to calculate the derivatives of the first-order basis functions. Dirichlet-type boundary conditions are applied on the far-field, and the symmetric sparse linear system is solved with the standard conjugate-gradient (CG) algorithm. Aero is also available as part of the OP2 framework.

MG-CFD is a 3D unstructured multigrid, finite-volume computational fluid dynamics (CFD) mini-app for solving an inviscid flow problem. It performs a three-dimensional finite-volume discretization of the Euler equations for inviscid, compressible flow across an unstructured grid by extending the CFD solver in the Rodinia benchmark suite [84], [85]. It accumulates fluxes by performing a sweep across edges, which is implemented as a loop over all edges. Multigrid support is achieved by supplementing the Euler solver's architecture in the work of Corrigan et al. [84] with crude operators that transport the simulation's state between multigrid levels. MG-CFD was originally created as a CPU-only implementation[86], but it has since been implemented with OP2 as well. It can be downloaded as open-source software [87].

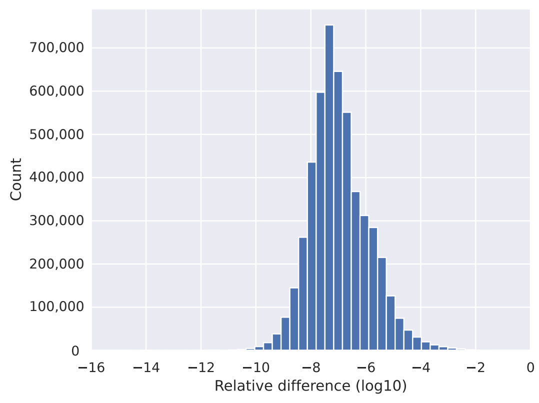
Hydra [88] is a full-scale industrial CFD application for the design of turbomachine components of aircraft engines at Rolls-Royce. Hydra is a complex and configurable application that can perform various simulations on highly detailed unstructured meshes. Its development originally started 23 years ago [89], and it is still actively developed and optimized to this day. The simulations implemented in Hydra are typically applied to large meshes, which can contain tens to hundreds of millions of edges and can run from a few minutes to weeks. It consists of several components that simulate various aspects of the design, including the steady and unsteady flows that occur in the engine around adjacent rows of rotating and stationary blades, the operation of compressors, turbines and exhaust, and the simulation of behavior such as ingestion of ground vortices. The guiding equations to be solved are the Reynolds-Averaged Navier–Stokes (RANS) equations, which are second-order PDEs. By default, Hydra uses a 5-step Runge–Kutta method for the time-marching, which is accelerated by multigrid and block-Jacobi preconditioning [89], [90].

This work uses the Hydra setup with several configurations: an unsteady simulation of two blades of DLR's Rig250 mesh and a steady simulation of NASA's Rotor37 mesh with different turbulence models: the Spalart–Allmaras wall function model, which is a one-equation model that solves a modeled transport equation for the kinematic eddy turbulent viscosity and a k-ω, which is a two-equation model that is used as an approximation for the Reynolds-averaged Navier–Stokes equations (RANS equations). Once again, the effect of non-reproducibility is highlighted through several examples with Hydra. In Figure 2.1 we can observe how the relative difference accumulates when increasing the number of time-steps from 10 to 100 while using the same unsteady numerical method on the same mesh. For a full revolution of two blade rows, 2000 time-steps are needed, where one time-step contains 10 iterations. Figure 2.2 illustrates how different turbulence models are affected by the relaxation of execution order, tested over 100 iterations in a steady simulation on the NASA Rotor37 benchmark. The k-ω is more susceptible to rounding error than the Spalart–Allmaras. The variable ω is used to avoid singularity near the wall, but it also becomes more sensitive to precision than the Spalart variable. This has a knock-on effect on the whole boundary layer, and hence the flow field. All four histograms present the magnitude of differences between two runs with the same setup, just running with different numbers of MPI processes.

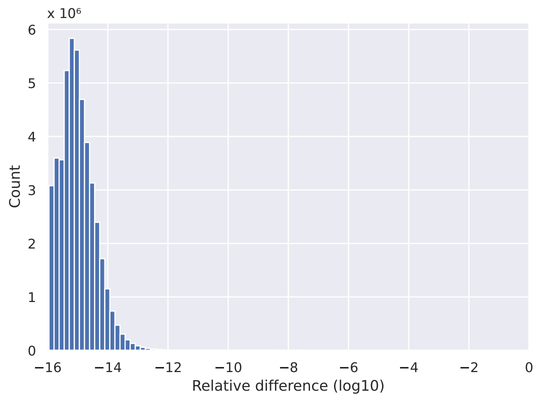


(a) (b)

Figure 2.1: Histograms, generated by using Hydra. The relative difference increases with more timesteps on an unsteady numerical solver. (**a**) Rig250 mesh with 20M nodes, 10 timesteps, Spalart–Allmaras model; (**b**) Rig250 mesh with 20M nodes, 100 timesteps, Spalart–Allmaras model.



(a)



(b)

Figure 2.2: Histograms, generated by using Hydra. The two models are not directly comparable, but they illustrate how the relative difference depends on the numerical properties of the applied model. (**a**) Rot37 mesh with 700k nodes, 100 iterations, k-ω model; (**b**) Rot37 mesh with 8M nodes, 100 iterations, Spalart–Allmaras model.

2.3 Theory and Calculation

Altering an already existing nonreproducible code to be reproducible might be tedious and laborious. Fortunately, in some ways, this process can be automated.

OP2 has an already established workflow to generate platform-specific optimized applications [91], Figure 2.3 summarizes the main mechanisms. If an application is implemented using OP2's API, then a source-to-source translator can generate platform-specific application files, which later can be compiled and linked with the backend libraries of OP2. In this work, I modified three stages of the workflow. I added API calls to the application description, so the user can choose which reproducible strategy should be applied. In order to use these strategies, the source-to-source translator had to be updated to generate such application files that use the reproducible backend libraries with MPI or CUDA.

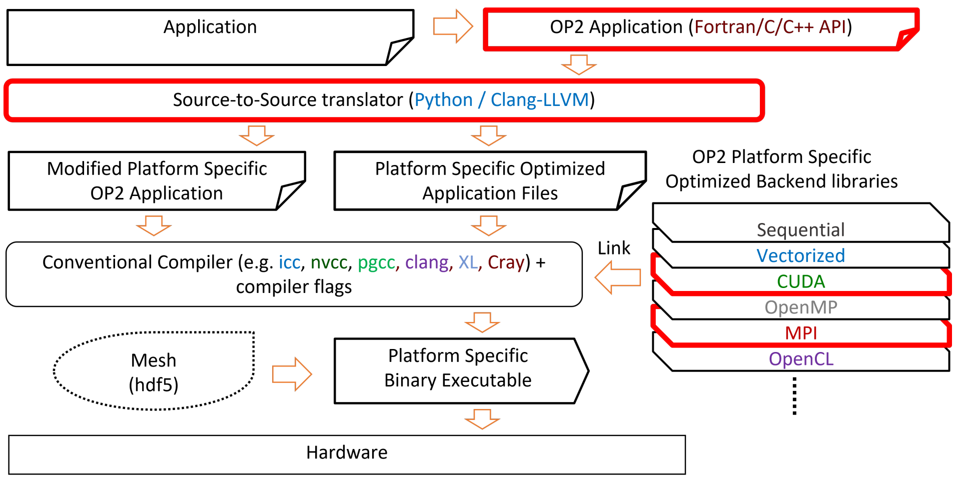

Application
OP2 Application (Fortran/C/C++ API)
Source-to-Source translator (Python / Clang-LLVM)
Modified Platform Specific OP2 Application
Platform Specific Optimized Application Files
OP2 Platform Specific Optimized Backend libraries
Sequential
Vectorized
CUDA
OpenMP
MPI
OpenCL
Link
Conventional Compiler (e.g. icc, nvcc, pgcc, clang, XL, Cray) + compiler flags
Mesh (hdf5)
Platform Specific Binary Executable
Hardware

Figure 2.3: Flow diagram of the mechanism of OP2. The bold, red frames represent the updated steps of OP2's workflow from my work.

This section presents techniques for addressing the two primary sources of non-reproducibility: local element-wise reductions and global reductions. The focus is primarily on local reductions, while global reductions are handled using ReproBLAS. Although most examples in this section employ an edges→cells mapping, all algorithms are implemented generically based on the specific mapping dimension.

To solve the issue of ordering in local (element-wise) reductions, I provide two separate approaches: (1) a method storing increments temporarily and applying them later in a fixed order and (2) different reproducible coloring techniques, which later can be used as colored execution, maintaining deterministic ordering. For all of these techniques, one must provide a common deterministic seed that will always be the same, even with different numbers of MPI processes. That common seed is the global ID of all elements in the whole mesh. If there are multiple MPI processes, then the global IDs of each element must be communicated between the processes. If an element is owned by the given process, then its global ID can be looked up from an internal data array of OP2. If an element is not owned, then its global ID must be imported from the MPI process that owns it. All of my techniques use two main parts: (1) the OP2 backend must calculate the execution order and (2) the generated code must execute the computations in this order. The reproducible execution method is applied only to kernels where the order of summation matters. These are loops with global reductions, indirect incrementing operations (`OP_INC`), or operations with an indirect read and write access pattern (`OP_RW`).

2.3.1 Temporary Array Method

A temporary array-based technique can be used to ensure reproducibility for incrementing operations. Consider using an edge→cells mapping and an incrementing operation. Here, we would iterate through all the edges, calculate values, and add them to a variable defined on a neighboring cell. To achieve reproducibility, I modify this structure by storing the calculated increments in a temporary array defined on the edges, and after all the increments are calculated, I iterate through all the cells and apply these increments in a fixed order defined by the global_IDs of the edges. In Figure 2.4 we can see an example of this method, where `edge2`'s global_ID is the smallest, so the value from `edge2` is applied first on the cell, then `edge0`, etc.

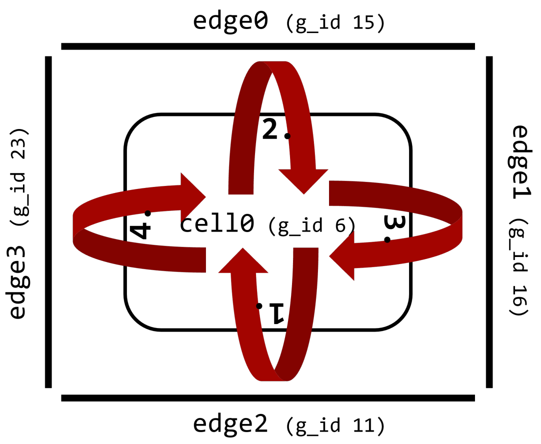


Figure 2.4: Example execution order of edges around a cell. Due to local id renumbering, the global ids must be used for a reproducible execution order.

To achieve this modified execution, a few extra preparations must be conducted in the backend, which are shown in Algorithm 1. After the global_IDs are shared, the next step is to create a reversed mapping for every map. The reversed mapping is needed so we can iterate through the cells and in each iteration we can access the edges connected to the given cell. This reversed map uses local indices which might be in different order in different MPI ranks. That is why we need to reorder them by using their previously shared global indices. Another modification conducted on the reversed map is that it actually stores indices of a temporary array where the increments from the edges are stored for a cell. In other words, if the k^{th} element in line n (k^{th} edge connection of cell n) of the reversed map is x, then it means that in the temporary increments array at location x the increment for cell n from edge k can be found.

The main disadvantage of this method is the need for significant additional memory to store the reversed mapping, and to store the increments. The reversed map uses a Compressed sparse row (CSR) format, which consists of a main array of increment indexes (integers), with the size of `set_from_size` $*$ `original_map_dimension`, and another array indexing the previous array with a size of `set_to_size`+1. The temporary arrays themselves can use much more memory: `set_from_size` $*$ `map_dimension` $*$ `data_element_size`.

After creating the reversed map with the correct order, OP2 generates a new `op_par_loop` implementation code to use this modified method. The main changes can be seen in Algorithm 2. After the initialization phase, it is imperative to set all elements in a temporary array to zero to accommodate individual increments. This step is crucial as the user kernel performs the increments, and proper initialization is required beforehand. Moreover, this approach ensures that the data remain in the cache, enhancing overall performance. Then we can call the kernel function for all edges to access the elements

Algorithm 1 Algorithm of generating incrementing order

```
exchange global IDs
$OP\_map\_index$ = number of maps
for $m = 0$ to $OP\_map\_index$ do
    create reversed mapping for map $m$
    $set\_to\_size$ = target set's size of map $m$
    for $i = 0$ to $set\_to\_size$ do
        sort the reversed connections of $i$ by global IDs
    end for
end for
```

defined on the cells. If a parameter is accessed through an `OP_READ` or `OP_WRITE` method, then the execution order does not matter, so we can use the original method of directly storing the new state in the data. If the parameter is incremented (`OP_INC`), then we need to store each increment value in the `tmp_incs` array instead of adding it to the actual data. After the iteration on edges is completed and all increments are calculated, we need to apply those to the actual data on cells. For that, OP2 starts a new cell-based loop on the cells and by using the reversed mapping with the fixed ordering, for each cell, it can gather and apply the increments. This method is generally applicable to other types of mappings as well.

Algorithm 2 Algorithm for applying the order of increments

```
$set\_from\_size$ = source set's size of the original map
$original\_map\_dim$ = the dimension of the original map
$set\_to\_size$ = target set's size of the original map
for $n = 0$ to $set\_from\_size * original\_map\_dim$ do
    $tmp\_incs[n] \leftarrow 0$
end for
for $n = 0$ to $set\_from\_size$ do
    prepare regular access indices for $OP\_READ$ and $OP\_WRITE$ parameters
    call kernel function, using the $tmp\_incs$ array for $OP\_INC$ parameters
end for
for $n = 0$ to $set\_to\_size$ do
    for all connection $i$ of $n$ do
        apply the temporary increment from connection $i$ on the final location of the data
    end for
end for
```

2.3.2 Reproducible Coloring

The temporary arrays method only works for increment-type operations, where increments can be stored separately. If a kernel not only increments a variable but also reads and rewrites it (`OP_RW`), then the kernel call from one edge must be executed, storing its result in the cell before another edge accessing that cell can be executed. Although OP2 still requires that the computation be associative, we cannot store the increments separately. This problem needs a solution to be able to really execute the kernel calls in a predefined fixed order and achieve reproducibility. To solve this issue, we can apply a regular coloring scheme with the following restriction, we are looking for an equivalence class of colorings where if the color of one element is smaller than that of another connected element in the case of one coloring, then it should also be the same in the case of any other coloring.

I have three main approaches to solve this problem. An initial trivial solution is to choose the global index of the edge as the color. With this, we have as many colors as edges in any given subgraph, but we do not have multiple edges with the same color. This is useful for MPI-only parallelization, but not for a shared memory method. The advantage is that this trivial method can be solved without actually coloring the elements. We can just use the numbering from the `global_ids` for ordering sequential execution. This trivial execution schedule can be considered as a special case of colored execution and in fact they use the same generated code. Therefore, I refer to it as a coloring method. The second method is a non-distributed method. OP2 applies a greedy coloring algorithm on the whole mesh in a single process as a pre-processing step and save the assigned colors in a file. When we rerun the application on multiple processes, we load and distribute the saved colors the same way as we distribute the mesh elements between the processes. With greedy coloring, we can generate a near-optimal number of colors, thus we have a high degree of parallelism. The drawback of this option is that we have to execute the pre-processing part in a single process. This carries the restriction that the whole mesh must be able to fit into the memory on a single node. The third method is a novel distributed coloring scheme, which does not suffer from this restriction.

Distributed Reproducible Coloring Method

I base my method on an algorithm developed by Osama et al. [92]. This original non-reproducible parallel method can be seen in Algorithm 3 between lines 7 and 40. OP2 iterates through each element, calculates a local hash value and then compares it to its (as yet uncolored) neighbors' hash values. If the examined hash value is a local minimum or maximum in its neighborhood in a given iteration, then OP2 can assign it a color. In my implementation I use Robert Jenkins' 32-bit integer hash function [93]. This hash function is a custom, non-cryptographic function that operates on unsigned integers. It uses a combination of bitwise operations and arithmetic with specific constants to compute the hash of an input.

While the core idea of local extrema-based coloring is shared with Osama et al., there are several important differences and challenges that arise in my distributed, reproducible variant.

Most importantly, the original method by Osama is limited to a single shared-memory context – it assumes all graph elements and their neighbors are immediately accessible in memory. This is not valid in distributed-memory environments, such as those using MPI-based partitioned meshes. To address this, I introduce a second halo layer, which includes not only the directly connected elements across partitions but also additional neighbors required to fully resolve adjacency across process boundaries. Without this second ghost layer, the coloring decisions would be based on incomplete information,

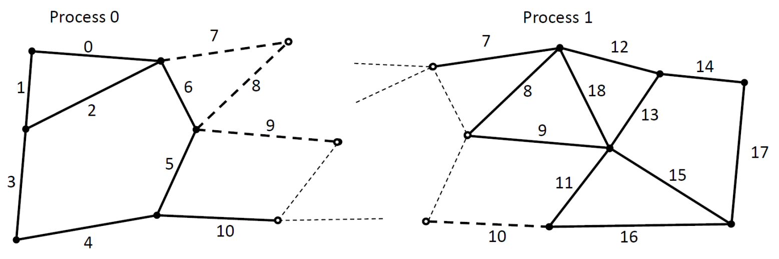


Figure 2.5: An example of a second ghost layer to determine the edge→edge neighbors on the partition borders. The numbers on the edges indicate their unique ID.

breaking correctness and reproducibility.

Figure 2.5 illustrates this key challenge. Elements like edge 0, 2, 5, and 6 on Process 1 do not directly belong to the process's ownership, but their presence is essential for determining whether a local hash value is minimal or maximal. The second ghost layer ensures these indirect neighbors are accessible and integrated into the neighborhood evaluation. Creating and synchronizing this extended halo involves a nontrivial neighbor discovery phase, including communication of both the topological structure and the coloring states of remote neighbors.

The difficulty of applying this algorithm in a distributed graph comes from two sources. First, in each iteration of the previously described algorithm, we must know if the neighbor element already received a color, or not. Thus, we need to synchronize the assigned colored values on the borders of each subgraph (MPI partition). Secondly, it is difficult to figure out all the neighbors of an element on the border of a subgraph in a standard owner-computed model. We can see an example of this problem in Figure 2.5. Solid dots and continuous lines are the owned elements. In this example, I use an edge $\rightarrow$ nodes mapping, thus we import one layer of halo elements (e.g., edge 7, 8, 9 on Process 0) so we can update the owned nodes from all attached edges (so far it is a standard owner compute model). However, to calculate the smallest hash value in a neighborhood, we also need to communicate edges even around the non-owned nodes (e.g., edge 0, 2, 5, 6 on Process 1).

A second key difference is that while Osama's method does not guarantee deterministic behavior across runs (due to non-deterministic hash seed and tie-breaking), my implementation ensures determinism by using a consistent hash seed based on element global IDs and structure, independent of execution order or partitioning. This guarantees reproducibility across distributed runs.

Moreover, my implementation handles color synchronization explicitly: after each iteration, newly assigned color values must be communicated across partition boundaries to ensure globally consistent state for subsequent decisions. This requirement is absent in Osama's original shared-memory variant.

My extension to distributed execution can also be applied to other iterative coloring techniques that use only local information (the algorithm is not sequential) and are deterministic even with different graph partitioning. The number of colors is not explicitly minimized.

Algorithm 3 Algorithm for reproducible coloring in a distributed graph

```
1: create neighbor lists
2: global_done = 0
3: local_done = false
4: if set_size == 0 then
5:     local_done = true
6: end if
7: iteration = 0
8: low_color = 0
9: high_color = 1
10: while global_done < number of subgraphs do
11:     if not local_done then
12:         for all element e in from_set do
13:             if e has no color then
14:                 calculate hash value of e in iteration i
15:                 is_min = true
16:                 is_max = true
17:                 for all neighbors n of e do
18:                     if n has no color then
19:                         calculate hash value of n in iteration i
20:                         if n's hash < = e's hash then
21:                             is_min = false
22:                         else if n's hash > = e's hash then
23:                             is_max = false
24:                         end if
25:                     end if
26:                 end for
27:                 if is_min then
28:                     give low_color as color of e
29:                     number of noncolored elements − = 1
30:                 end if
31:                 if is_max then
32:                     give high_color as color of e
33:                     number of noncolored elements − = 1
34:                 end if
35:             end if
36:         end for
37:         if number of noncolored elements == 0 then
38:             local_done = true
39:         end if
40:     end if
41:     exchange halo color values
42:     reduce local_done values into global_done
43:     low_color += 2
44:     high_color += 2
45:     iteration += 1
46: end while
```

2.3.3 Parallel Global Reduction

Global reductions are another source of non-reproducibility in MPI applications. This operation is commonly conducted by performing a local sum on each process, then calling `MPI_Reduce`, however, this assumes associativity. If we use different numbers of MPI processes, then we would sum different elements and even a different number of elements locally, which again can produce different results. To solve this issue, I introduced another temporary storage. If a kernel performs an increment reduction, then OP2 gives a temporary storage point to store the increment for the result of each element. Then, in each MPI process, it reduces these increments reproducibly by using the ReproBLAS library. First, OP2 creates a local ReproBLAS's `double_binned` variable for every MPI process, then uses `binnedBLAS_dbdsum` to collect those into the `local_sum`. After that, it uses reproBLAS's method to call an `MPI_Allreduce` with the `binnedMPI_DBDBADD` operator. Finally, OP2 converts the result back to a regular double precision variable and return it.

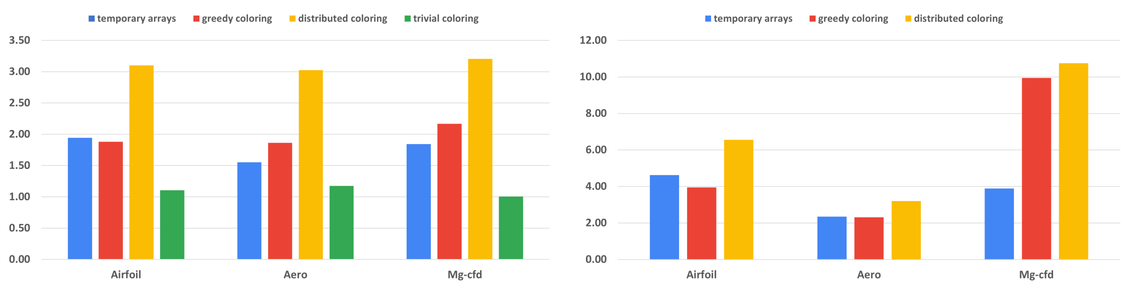
2.3.4 Reproducible Codegeneration with OP2

Using OP2's source-to-source translator, a user can easily generate reproducible code from an app that already has an implementation using OP2. A few flags are responsible for controlling the mechanisms that allow reproducible code to be generated. In the translator scripts these are: `reproducible`—needed for all methods, `repr_temp_array`—for using temporary arrays, `repr_coloring`—for using reproducible coloring method and `trivial_coloring` which will produce the trivial coloring version. To enable the greedy coloring technique, the `-op_repro_greedy_coloring` command line flag must be used with the application.

2.4 Performance Results

I measured my techniques with four test applications, introduced in Section 2.2.2. All results are the average of 10 measurements. The Cirrus-CPU machine is equipped with an Intel Xeon E5-2695 (Broadwell) processor running at 2.1 GHz. It does not feature a GPU and provides 18 cores per node, with 2 threads per core. The code on this system was compiled using the Intel compiler icc (version 19.0.0.117), and the operating system is Red Hat Enterprise Linux 8.1 (Ootpa). The Cirrus-GPU machine, on the other hand, features an Intel Xeon Gold 6248 processor running at 2.5 GHz, along with NVIDIA Tesla V100-SXM2-16GN GPUs. Each node includes 20 cores, with 2 threads per core, and is equipped with 4 GPUs. Compilation on this system was performed using the NVIDIA HPC compiler nvc++ (version 21.9-0). Like Cirrus-CPU, it also runs Red Hat Enterprise Linux 8.1 (Ootpa).

All of the introduced methods provide full reproducibility at the expense of additional computations, suboptimal scheduling, or redundant memory usage. The overall cost of these techniques is visualized in Figures 2.6 and 2.7 and in Table 2.1. Each run is compared with its original, non-reproducible version. On CPU systems, slowdowns are



(a) (b)

Figure 2.6: Slowdown effect of the different methods compared to the non reproducible version. (**a**) Using 40 MPI-only processes on the Cirrus machine; (**b**) Using one MPI+CUDA GPU process on the Cirrus-GPU machine.

Table 2.1: Memory usage of the reference run and with using the proposed methods in GB.

App	**Non Reproducible**	**Temporary Arrays**	**Coloring Method**
Airfoil	0.92	1.6	1.3
Aero	2.6	3.4	2.8
MG-CFD	7.5	14.4	9

between 1 and 3.21 times. The difference between the greedy and distributed coloring methods comes down to data reuse and cache line utilization, because of the different number of colors used. The main reason for that is that the data for neighboring elements are located close in memory, but when using coloring, adjacent elements will have different colors, leading to poor utilization. A few examples of the number of colors used are shown in Table 2.2. While the greedy scheme leads to near-optimal color counts, the parallel scheme yields much higher color counts particularly in 3D. The performance of the trivial coloring scheme is close to the reference, since it uses a similar order of execution to the nonreproducible version, with the only differences around the borders of MPI partitions. Since with the trivial scheme we still require sequential execution within a process, we cannot use additional parallelization techniques, such as CUDA or OpenMP. In contrast, the slowdown on GPUs is more significant, because they are even more sensitive to data access patterns and cache locality than CPUs. In particular, with the usage of the temporary arrays, we have to iterate through the increment data twice, once when populating it and once when gathering the results, each time with a different access pattern. If we optimize for one stage, then the other will suffer from the non-coalesced data accesses. This is even true for the coloring methods. If we reorganize the data in a set according to one map, then later, using another map to the same set, we again obtain inefficient access patterns.

The runtime overhead of the preprocessing preparations of the temporary array and coloring methods against the number of MPI processes are detailed in Figure 2.8 and using only one process in Table 2.3.

Figure 2.9 shows how well the test applications scale with the different methods using

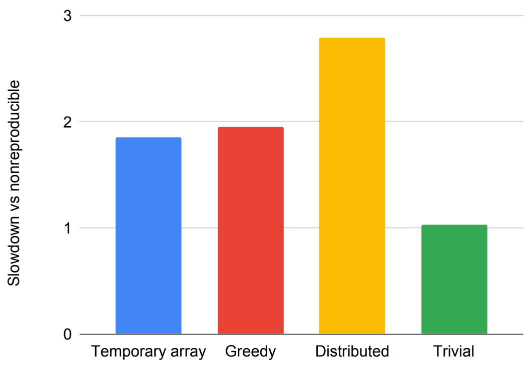
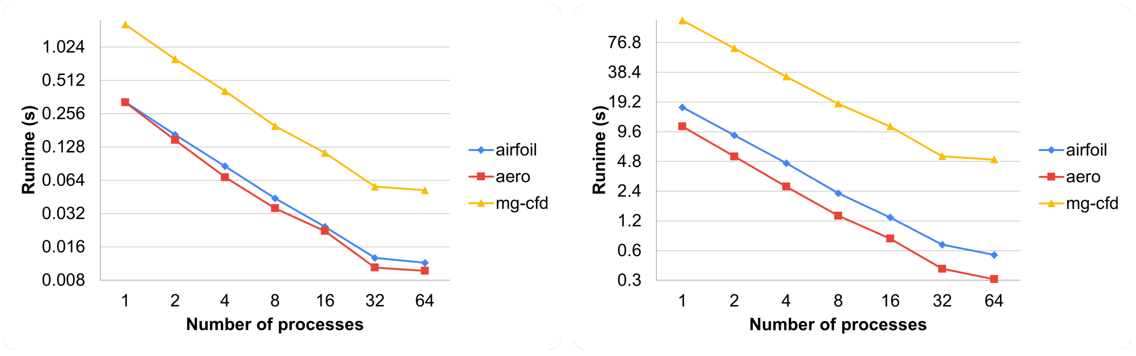


Figure 2.7: Slowdown of Hydra measured on an 8M mesh, 20 iterations, using the Cirrus-CPU machine.

Table 2.2: Number of colors with the different methods on the applications main map.

App (Map)	**Greedy**	**Distributed**
Airfoil (pecell1)	4	14
Aero (pcell1)	5	17
MG-CFD (edge→node0)	7	19



(a) (b)

Figure 2.8: Scaling of preprocessing overhead. (**a**) reversed map and temporary array creation time for the temporary array method; (**b**) reversed map creation and distributed coloring time.

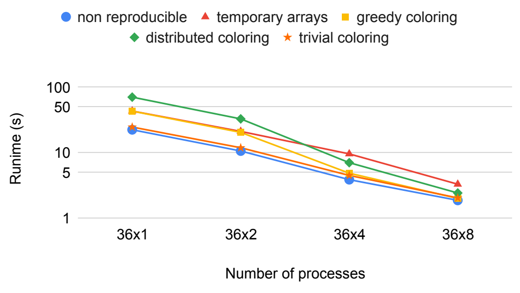
Table 2.3: Reversed map creation and greedy coloring time.

App	**Runtime**
Airfoil	4.65 s
Aero	1.79 s
MG-CFD	128.88 s

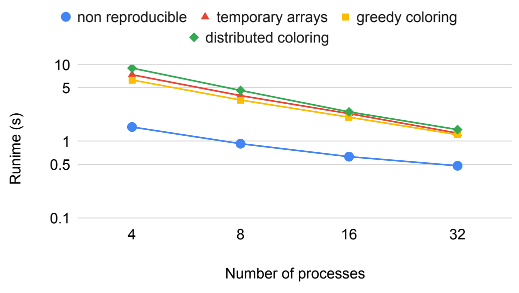
one, two, four, and eight nodes on the Cirrus cluster. On the CPU side, all methods have the same parallel efficiency on each application, except the distributed and greedy coloring methods on Airfoil, where we can observe superlinear scaling (Figure 2.9a) since much of the data used can fit into the cache if they are divided between at least four nodes. We cannot observe this on the temporary array method, because it uses extra memory to store increments separately. Apart from the reductions (discussed in detail below), MPI communications and communication times do not differ between reproducible and non-reproducible. For non-reproducible execution, the communication overhead (as a fraction of total runtime) will become higher using multiple nodes. In the case of reproducible execution, because we spend more time in the colored execution, we spend a smaller fraction of the total time in communications. Therefore, the relative difference is decreasing and the slowdown effect with any method compared to the non-reproducible is less when more nodes are used.

We can observe the strong scaling of a reduction kernel in Figure 2.10. Since all reproducible methods use the same reduction technique, there is no separate measurement for them. Again on CPUs, we can see the superlinear effect as the application fits more and more into the cache. We can also observe that there is an additional cost of the reduction caused by the reproBlas functions. The most significant factor in the cost of reproducible reduction is that we must write all the values to be reduced into a separate array and perform a reduction on it within a process. This leads to extra memory movement compared to the reference version. This is particularly expensive on GPUs because this array must be copied to the host to perform the local summation. `MPI_reduce` is not significantly more expensive.

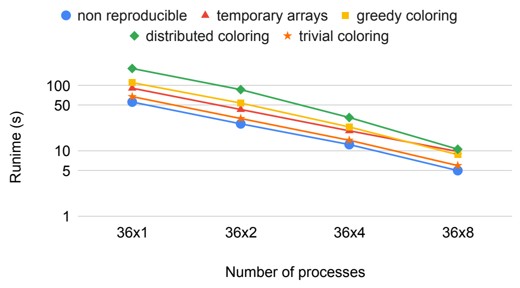
Using only MPI parallelization, the overhead is quite small (between 1 and 1.12 times). Using shared memory parallelism, it is a bit greater due to the bad cache locality. In some extreme cases, we can even lose the speedup gain from GPUs, my reproducible methods work better on CPU-only systems.



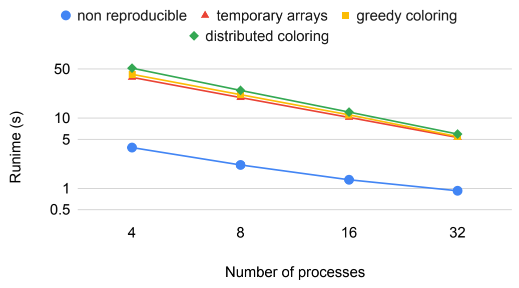
(a)



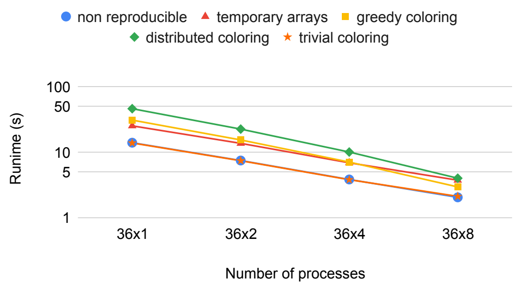
(b)



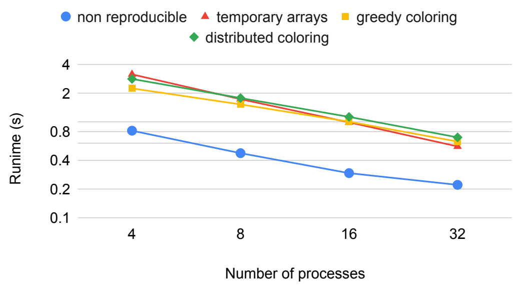
(c)



(d)

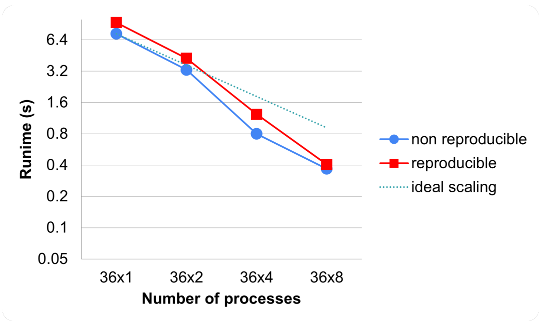


(e)

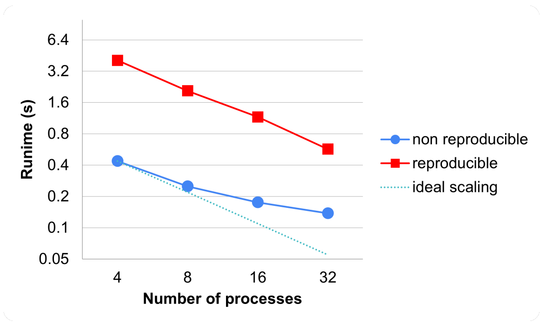


(f)

Figure 2.9: Strong scaling measurement of the different methods, using 1,2,4,8 nodes; (**a**) Airfoil, using 36 MPI Intel Xeon CPU processes per node; (**b**) Airfoil, using four Nvidia V100 GPU processes per node; (**c**) Aero, using 36 MPI Intel Xeon CPU processes per node; (**d**) Aero, using four Nvidia V100 GPU processes per node; (**e**) Mg-cfd, using 36 MPI Intel Xeon CPU processes per node; (**f**) Mg-cfd, using four Nvidia V100 GPU processes per node.



(a)



(b)

Figure 2.10: Strong scaling measurement of a reduction kernel; (**a**) Airfoil_update on the Cirrus-CPU machine; (**b**) Airfoil_update on the Cirrus-GPU machine.

2.5 Conclusions

In this chapter, I examined the non-reproducibility phenomenon that occurs due to the non-associative property of the floating-point number representation on applications defined on unstructured meshes. I compared the differences in results without reproducibility across a range of applications, including Rolls-Royce's production application, Hydra. While non-reproducibility is a widely studied problem; I did not find an effective solution for distributed systems in the literature that could also be applied to arbitrarily partitioned meshes. In this work, I developed a collection of parallel and distributed algorithms to create a plan and then execute it, guaranteeing the reproducibility of the results. Of these, I highlight a graph coloring scheme that gives the same colors regardless of how many parts the graph was partitioned into. I implemented all of my methods in the OP2 DSL and then I showed how they can be automatically applied without user intervention to any application that is already using OP2. I demonstrated that on CPU systems, my methods can achieve bitwise reproducible results with a slowdown between 1.0 and 3.21 times in various applications, and on GPU systems with a slowdown between 2.31 and 10.7 times due to the modified data access patterns.

While there are alternative methods addressing the issue of reproducible reduction, their complexity is akin to mine and from the perspective of OP2, the choice of method is non-critical. This is why I do not draw comparisons on this aspect, as the time spent on reductions is relatively short. My work stands out in the development of a generalized method ensuring reproducible execution, applicable to various applications. This is in contrast to other solutions that are application-specific. There are several general methods available. Kahan's method, although popular, does not guarantee reproducibility, just higher accuracy. The most straightforward method involves sorting the elements before adding them. The most general method, perhaps, is the binned method, like in the ReproBLAS library. However, all these methods are more complex and mainly more expensive in computing and/or in memory usage. By leveraging the properties of the unstructured mesh, we can keep the costs low, thus presenting a more efficient solution.

3 Reduced precision computing

3.1 Introduction

The advent of artificial intelligence has led to the widespread adoption of reduced precision formats in hardware, such as half-precision (16-bit) floating-point arithmetic. While this transition presents opportunities for significant performance gains, it also raises concerns regarding the accuracy and stability of numerical results, particularly in high-performance computing (HPC) applications that are sensitive to precision. The challenge lies in determining the appropriate balance between precision and performance, as the demand for computational efficiency continues to grow.

Existing mixed precision techniques have shown promise in various applications. For instance, many linear solvers have been successfully implemented using mixed precision, allowing for improved performance without sacrificing accuracy [27]–[31]. Abdelfattah et al. [32] reported advancements in mixed precision algorithms, including speedups in dense and sparse LU factorization, eigenvalue solvers, and GMRES implementations. FluidX3D [33], a lattice Boltzmann CFD software, has explored mixed precision techniques, although it is limited to specific models and lacks flexibility in accommodating other simulation types. OpenFOAM [34], another popular CFD framework, with its recent update, compiles its code in single precision while employing double precision exclusively for its linear algebra solvers. This approach allows OpenFOAM to leverage the efficiency of single precision across a wide range of simulations, although it primarily focuses on finite volume solvers.

Many finite difference methods (FDMs) are characterized by a common structural pattern: the iterative addition of a smaller value to a larger one at each computational step. This inherent structure presents an opportunity for strategic precision allocation that is exploited in the present contribution. In this approach, the larger value, which accumulates the results of multiple iterations and is more susceptible to accumulating roundoff errors, is typically maintained in a full precision format to minimize the loss of precision. Conversely, the smaller update values, which are discarded after each step and do not contribute to long-term error accumulation, can be represented in reduced precision formats, allowing for a controlled trade-off between precision and computational efficiency. By employing the OpenSBLI [55] framework, which automates the derivation of finite difference solvers, in conjunction with the OPS [53] library—designed for structured mesh solvers—I demonstrate the capability to generate optimized codes for multi-GPU and multi-CPU environments. OpenSBLI facilitates the generation of optimized reduced/mixed precision implementations of OPS codes, enabling efficient computations across various hardware configurations.

The flexibility afforded by OpenSBLI allows users to customize the precision of datasets utilized in their simulations. This capability is particularly advantageous in scenarios where computational efficiency is paramount, necessitating a careful balance between precision and performance. In this study, I present a series of tests conducted on a compressible Taylor-Green vortex simulation, which serves as a benchmark for evaluating the performance and accuracy of reduced precision computing. My findings reveal that employing half, single, and double precision formats, as well as mixtures such as half-single and single-double, yields significant speedups without compromising the numerical accuracy of the results. Notably, only the half-precision runs exhibited numerically unsatisfactory outcomes, highlighting the critical importance of precision selection in computational simulations.

The chapter is structured as follows: Section 3.2 focuses on the algorithmic changes required to implement mixed precision strategies effectively within finite difference methods. Section 3.3 presents my testing and evaluation, starting with an introduction to the Taylor-Green vortex test case. Then I assess the accuracy of various precision configurations and subsequently analyze the performance, highlighting both the computational speedups and memory savings achieved with mixed precision. Finally, in Section 3.4, I discuss conclusions and future work, highlighting the potential of mixed precision techniques in larger and more complex applications, as well as improvements in half-precision performance and flexibility in the OPS framework.

3.2 Enabling mixed precision in OpenSBLI

3.2.1 Mixed precision using explicit finite difference methods

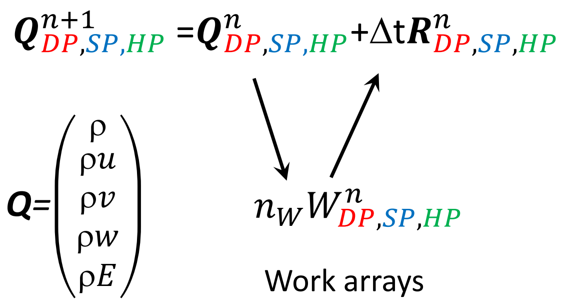
For simulation of transitional and turbulent compressible flows, in which a wide range of spatial and temporal scales are present, explicit finite difference schemes are commonly used. High-order finite differences are used for evaluating spatial derivatives, while low-storage variants of Runge-Kutta time advance schemes are adopted. The conservation form of the governing Navier-Stokes equations is used, with the vector of flow variables given by $\mathbf{Q} = (\rho, \rho u, \rho v, \rho w, \rho E)^T$, where ρ is the density, u, v, and w are the velocity components and $E = e + (u^2 + v^2 + w^2)/2$ is the total energy per unit mass, adding the internal energy per unit mass e to the kinetic energy per unit mass. The complete governing equations are given in [94]. Here, the focus is on the time advance step to explain how reduced precision schemes may be implemented.

The conservative flow variables are advanced in time using compact Runge-Kutta methods based on storage of $\mathbf{Q}$ and a change denoted $\tilde{\mathbf{Q}}$. At each substep i these storage locations are updated according to

$$\tilde{\mathbf{Q}}^i = A_i \tilde{\mathbf{Q}}^{i-1} + \Delta t \mathbf{R}^{i-1} \tag{3.1}$$

and

$$\mathbf{Q}^i = \mathbf{Q}^{i-1} + B_i \tilde{\mathbf{Q}}^i, \tag{3.2}$$



$Q^{n+1}_{DP,SP,HP} = Q^{n}_{DP,SP,HP} + \Delta t R^{n}_{DP,SP,HP}$
$Q = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho E \end{pmatrix}$
$n_W W^{n}_{DP,SP,HP}$
Work arrays

Figure 3.1: Schematic of the use of n_W variable precision work arrays W to form residuals R for update of conservative variables Q during a typical Runge-Kutta substep (DP=double precision, SP=single precision, HP=half precision).

where A_i and B_i are scalar coefficients of the scheme, Δt is the time step and $\mathbf{R}$ is the residual, containing all the remaining terms from the governing equations. After m substeps the solution at the next time level $n+1$ is given by

$$\mathbf{Q}^{n+1} = \mathbf{Q}^{m}. \tag{3.3}$$

The update procedure is shown schematically in Figure 3.1 based on a simple Euler update, but containing the essential features of equation 3.1. The residual $\mathbf{R}$ needs to be computed from the solution $\mathbf{Q}$ at the previous step. This step contains most of the computational cost of the algorithm and is typically accomplished using a number n_W of work arrays W, where n_W may be of the order of 20, but can be much more if curvilinear co-ordinates are used. The work arrays typically are the datasets containing first and second derivatives of various quantities which are eventually used to evaluate the residual, $\mathbf{R}$.

To discuss the use of mixed precision algorithms, it is first noted that each of the array types $\mathbf{Q}$ (flow variables), $\mathbf{R}$ (residuals), and W (work arrays) could, in principle, be represented using different numerical precision. A standard treatment would be to store all in double precision (denoted DP and shown in red in figure 3.1). One could in principle also do all the operations in single precision (SP, blue) or even half precision (HP, green). In this work, mixed precision cases are also considered; for example, a single/double mixed precision case (SPDP) can be defined where $\mathbf{Q}$ is retained in double precision, while single precision is used for $\mathbf{R}$ and all the W. Similarly, a half/single mixed precision case (HPSP) stores $\mathbf{Q}$ in single precision and $\mathbf{R}$ and W in half precision. Clearly, there are other options that will be discussed later. One could also split the residual $\mathbf{R}$ into different components that could be treated with different precision, as proposed in [95].

Two approaches to the work array usage are applied in this study. The first one is called the “default" method, and the second is called the “Storesome" method. The default method in OpenSBLI creates various 3D work arrays to calculate and store the residual terms in memory before passing them to the main kernel that does the final evaluation of $\mathbf{R}$ involving the inviscid and viscous fluxes. This method is memory intensive, but helps with code structure and readability, mimicking what was previously done with manually-written code. The second method, called the “Storesome" method, only uses a few of the 3D work arrays and computes most of the derivatives directly within the kernel functions, as and when required. Jammy et al [96] demonstrated significant improvements with the Storesome approach, in terms of memory usage and run time.

3.2.2 Generation of mixed-precision source codes using OpenSBLI

The most recent version of OpenSBLI [61] was extended for this work to implement the different low- and mixed-precision algorithm strategies. This required the following changes to be made to the internal code-generation engine:

- A new user interface option in the high-level Python script for specification of the precision to be used globally within the simulation (double/single/half).
- Modifications to the array declaration types and input/output arguments to OPS parallel loops based on the selection made by the user.
- Explicit C/C++ casting of quantities appearing on the right-hand-side of equations in the simulation code, as required.
- A Python interface for specifying a lower/higher precision for certain array quantities (e.g. **Q**, **R**, **W** discussed in Section 3.2.1) to enable mixed-precision strategies.

The new interface for specifying lower- and mixed-precision is shown in the example code snippet below:

Listing 3.1: Selecting mixed-precision inputs in OpenSBLI V3 [61]

```
# Set the global simulation precision to single
SimulationDataType.set_datatype(FloatC)
# Define custom mixed-precision options using presets
mixed_precision_config = {
'q_vector' : ([], Double),
'RK_arrays' : ([], Double),
'residuals' : ([], FloatC),
'wk_arrays' : ([], FloatC),
'casting' : 'explicit'}
# Optionally, select custom arrays to set the precision of
custom_arrays = [
    block.location_dataset(dset)
    for dset in ['T', 'p']
]
mixed_precision_config['custom'] = (custom_arrays, Half)
# Call the OPS C code-generator
OPSC(alg, mixed_precision_config)
```

In line 2, the default global precision of the simulation is specified by the user as single (`FloatC`). A configuration dictionary is then created in lines 4-9 to specify mixed-precision alterations to the global precision depending on the storage quantity in question. Four presets are shown between lines 5-8, which, in this example, raise the precision of the conservative $\mathbf{Q}$ and temporary Runge-Kutta $\tilde{\mathbf{Q}}$ storage arrays to double precision. Other available presets for residual $\mathbf{R}$ and work-arrays $\mathbf{W}$ are also shown. For completeness, a custom array option is also shown in lines 11-15. Here, the arrays for temperature and pressure (T, p) are set as half precision, to provide additional flexibility to explore more complex mixed-precision strategies. Finally, OpenSBLI's OPS C code-writer is called in line 17 with the requested mixed-precision configuration. Internally, the code-generation engine will define the precision of these quantities and their API calls within the OPS library automatically based on the user input.

3.3 Results

3.3.1 Taylor Green test case

The Taylor-Green vortex problem has become a standard test case for assessing the accuracy and efficiency of schemes to solve the unsteady Navier-Stokes equations, particularly of high-order schemes [97]. It is a straightforward case to set up, consisting of a triply-periodic domain with a uniformly spaced grid and a prescribed analytic initial condition. From this starting condition, the flow evolves into discrete vortices which subsequently break down to a turbulent flow state. The kinetic energy of the initial condition eventually dissipates into heat and the accuracy of the method is assessed by monitoring the time evolution of the turbulence kinetic energy and the dissipation rate. Both incompressible and compressible forms of the problem can be specified. The effect of Mach number was demonstrated by Lusher & Sandham [94] and one of their cases was used for a 7-way multi-code comparison with a variety of numerical methodologies in [98].

The starting equations for the Taylor-Green vortex problem are given by

$$u(x,y,z,t=0) = \sin(x)\cos(y)\cos(z), \tag{3.4}$$

$$v(x,y,z,t=0) = -\cos(x)\sin(y)\cos(z), \tag{3.5}$$

$$w(x,y,z,t=0) = 0, \tag{3.6}$$

$$p(x,y,z,t=0) = \frac{1}{\gamma M^2} + \frac{1}{16}\left[\cos(2x)+\cos(2y)\right]\left[2+\cos(2z)\right]. \tag{3.7}$$

Here, M is the reference Mach number, while $\gamma = 1.4$ for air. The equations are solved in a non-dimensional form and all the quantities are non-dimensionalised with corresponding reference values. Also, the initial density ρ at the start of the simulations is evaluated based on the non-dimensional form of the equation of state, i.e.,

$$\rho(x,y,z,t=0) = \gamma M^2 p. \tag{3.8}$$

In the TGV problem, a constant initial reference temperature is assumed across the entire domain at the beginning at $t = 0$, hence the non-dimensional temperature is equal to one and doesn't feature in Eq. 3.8.

All the simulations are carried out with fourth order central differencing. The solution domain size can be either a $(2\pi)^3$ or (exploiting symmetries in the initial condition) π^3. Key parameters are the Reynolds and Mach numbers. For the tests of reduced precision, the triply symmetric case (denoted TGSym) is adopted, that was part of the 2021 OpenSBLI release [55] with a Reynolds number set to $Re = 800$ to make the case better resolved on modest grids (compared to the $Re = 1600$ used in [98] requiring grids between 512^3 and 2048^3 for a supersonic case). A default resolution of 256^3 is adopted here, which is enough for these cases to be fully resolved (equivalent to 512^3 in a $(2\pi)^3$ domain). Using an intermediate Mach number of $M = 0.5$ allows larger time steps and is more representative of compressible flow applications than the choice of $M = 0.1$ that is commonly used to compare with incompressible simulations; however, the effect of Mach number will also be considered, since we wish to check for any issues in using reduced precision towards the incompressible limit. Also later, an inviscid version of the problem is considered to study the stability of various splitting schemes in the context of reduced precision algorithms.

Figure 3.2 shows a typical evolution of the flow field at a few different time instances. Contours of ρE are shown for a $M = 0.5$ TGV simulation run in double precision. The first frame (Fig. 3.2a) shows the smooth state of the flow at time $t = 0$, as per the description in Eqs. 3.4-3.7. As time evolves, the evolution of the flow leads to the formation of smaller and smaller scale vortical structures (Figs. 3.2b and 3.2c), illustrating the cascade process of turbulent flow. With no production of turbulence, the flow (Fig. 3.2d) slowly decays.

Before I move to quantitative results, the physical quantities of interest must be outlined in the TGV problem to assess the efficacy of various precision calculations presented in this research. The volume-averaged kinetic energy at each instance of time is defined as

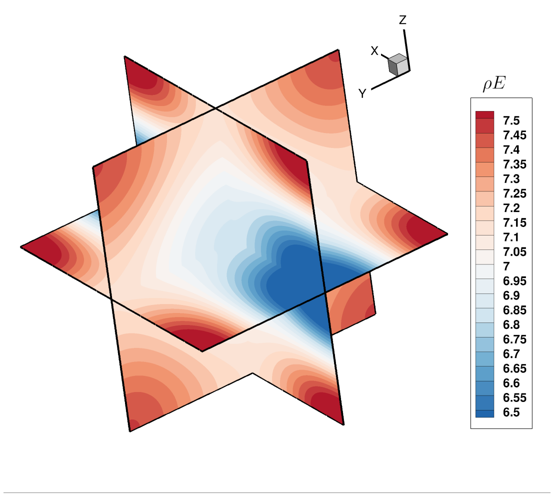
$$K = \frac{1}{V}\int_V \frac{1}{2} u_i u_i dV. \tag{3.9}$$

Here, V is the volume of the computational box and is equal to π^3 for the symmetric case. The evolution of solenoidal dissipation, which represents the enstrophy contribution to dissipation, is also examined. The simulations performed in the present work are all for subsonic Mach numbers where the dilatation part of dissipation is negligible. The equation for solenoidal dissipation is

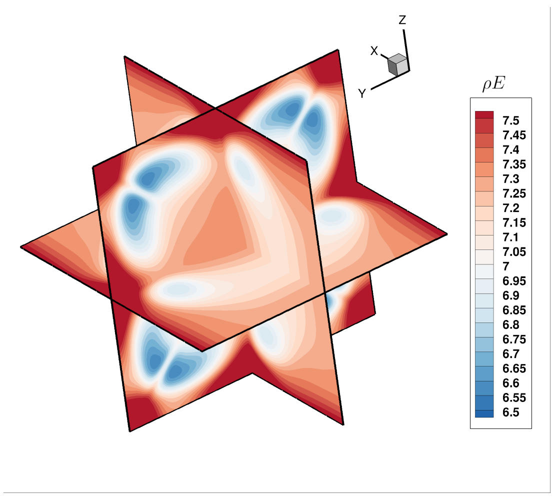
$$\epsilon^S = \frac{1}{Re}\int_V \left(\epsilon_{ijk}\frac{\partial u_k}{\partial x_j}\right)^2 dV. \tag{3.10}$$

Here, Re is the simulation Reynolds number, while the integrand represents the inner product of the vorticity vector, also known as enstrophy.

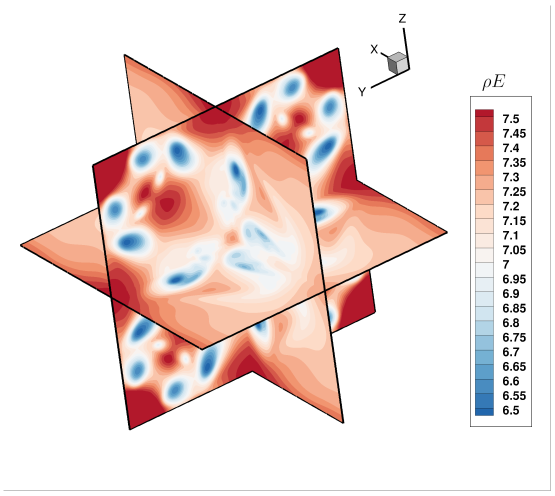
Figure 3.3 shows the behavior of K (left hand scale) and ϵ^S (right hand scale) over a



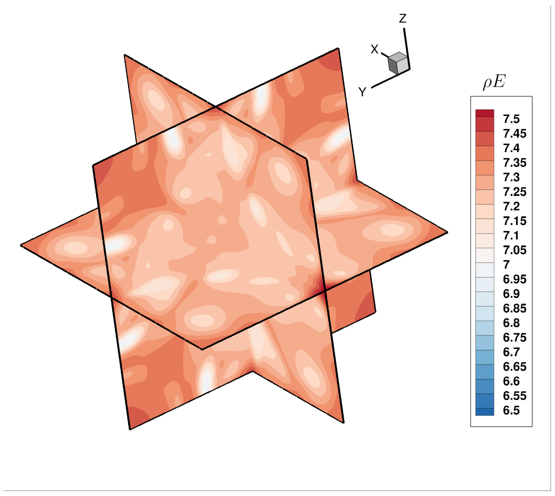
(a) $t = 0$



(b) $t = 5$



(c) $t = 10$



(d) $t = 15$

Figure 3.2: Contours of ρE in three mutualy perpendicular slices at the mid locations in x, y and z-directions, demonstrating the evolution of TGV state at different times: (a) t=0, (b) t=5, (c) t=10 and (d) t=15.

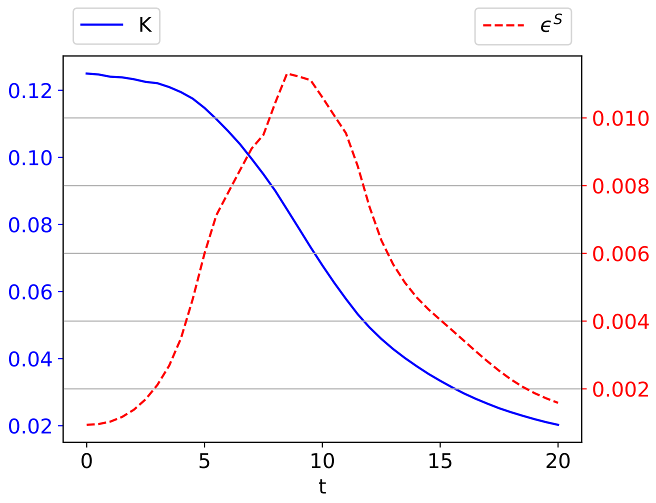
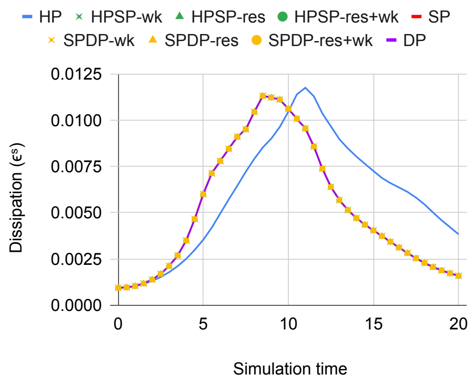
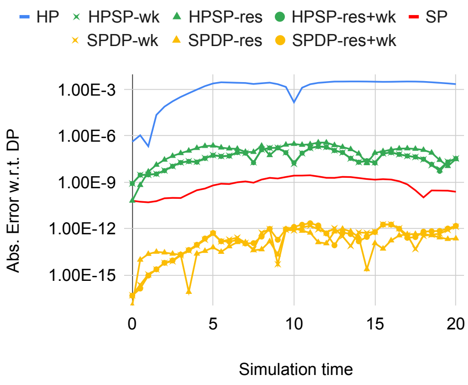


Figure 3.3: Kinetic energy (K) and dissipation (ϵ^S) relative to the time.



(a) Dissipation



(b) Absolute difference of dissipation, compared to double precision

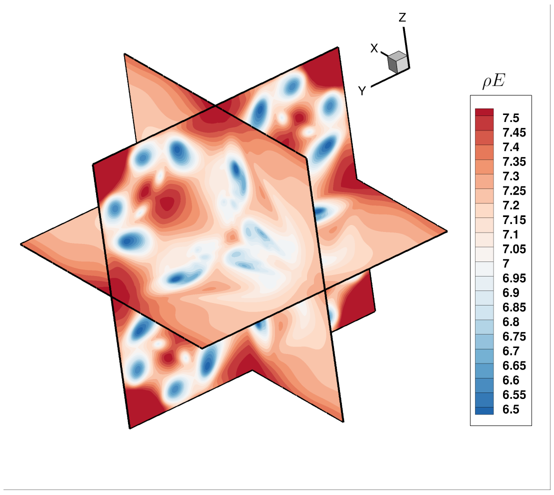
Figure 3.4: Numerical accuracy of TGsym app using different precision levels. Mesh size $= 256^3$, $M = 0.5$, $Re = 800$. The simulations were run for 8000 iterations using the default method.

simulation time from $t = 0$ to $t = 20$. For spatially homogeneous decaying turbulence, the rate of change of K is proportional to $-\epsilon$, so the kinetic energy decays monotonically once small scale structures have formed and the dissipation, which is positive definite, becomes significant. Given the derivative nature of the relation between K and ϵ, the solenoidal dissipation rate ϵ^S, representing almost all of ϵ, is a more sensitive measure of the numerical accuracy that is used to compare the different techniques employed in this study.

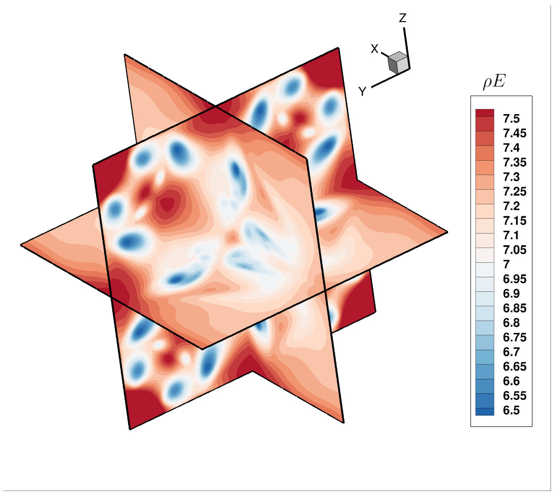
3.3.2 Accuracy

Figure 3.4a shows the effect of numerical precision on the behavior of the solenoidal dissipation rate as a function of time. Three standard versions are shown: HP using FP16 exclusively, SP using FP32, and DP using FP64. In addition, three kinds of mixed precision cases, namely '-wk', '-res', and '-res+wk', are presented between each pair of pure precisions. The '-wk' cases store only the work arrays in the lower precision, the '-res' cases store only the residual arrays in the lower precision, and the '-res+wk' cases store both the residual and work arrays in the lower precision. All test cases demonstrate overlapping results, with the exception of the FP16 version. The consistency observed among the other cases, with overlapping dissipation profiles, suggests comparable physical outcomes across these precision levels. Figure 3.4b presents a more detailed comparison and magnifies the differences by showing the absolute differences between each case and the highest precision run (FP64). The mixed-precision versions exhibit closer alignment with the higher precision of the two component precisions. Given the similarity in the observed error among the three mixed precision strategies, subsequent analyses will focus solely on the '-res+wk' strategy, as it offers the most significant computational speedup.

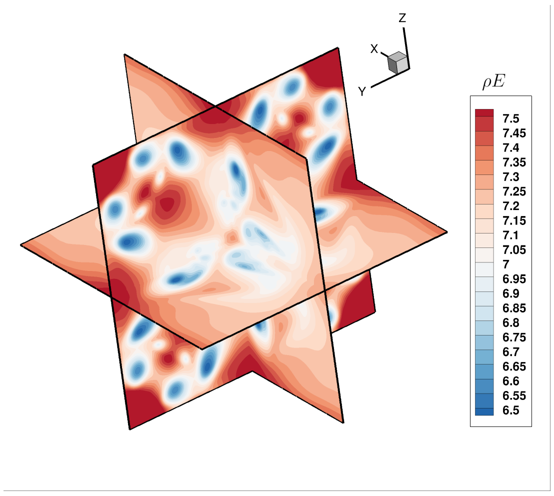
Figure 3.5 shows the state of TGV at $t = 10$ for various types of precision computations. It is clear that the flow state is qualitatively the same for lower precision and mixed precision computations all the way to the HPSP mixed setup, when compared to the



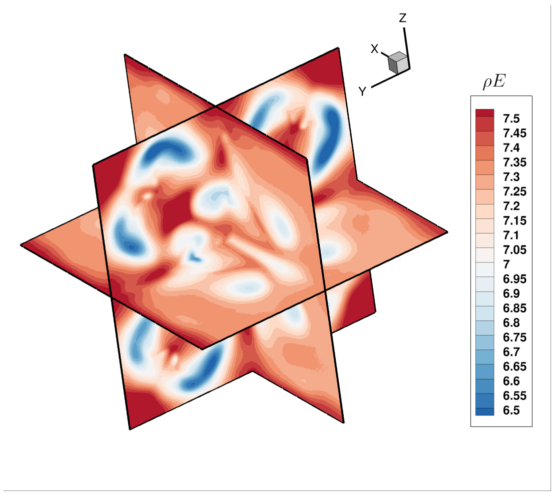
(a) **SPDP mixed** TGV state at $t = 10$.



(b) **SP** TGV state at $t = 10$.



(c) **HPSP mixed** TGV state at $t = 10$.



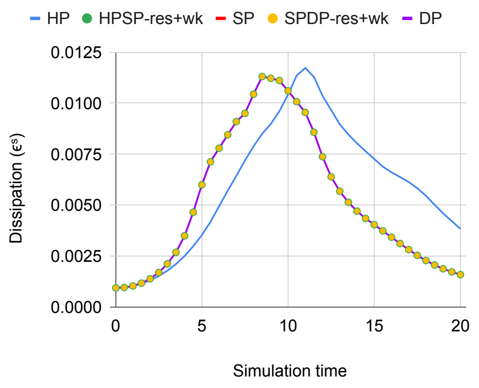
(d) **HP** TGV state at $t = 10$.

Figure 3.5: Contours of ρE showing the TGV state at $t = 10$, close to the peak of dissipation: (a) SPDP, (b) SP, (c) HPSP and (d) HP.

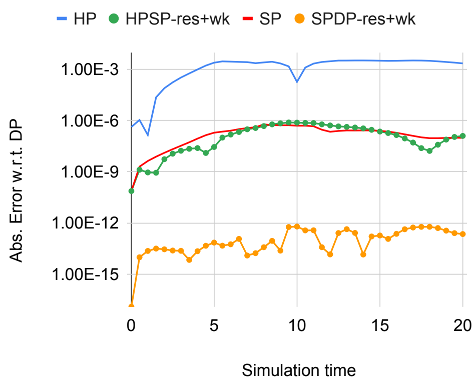
most accurate DP results presented in Fig. 3.2c. The results only deviate for the pure half-precision computations (HP), as can be noted from the last frame in the figure, where the vortex structures are more diffuse and sometimes in different locations.

The accuracy of the Storesome method is presented in Figure 3.6 in the same format as Figure 3.4. This method also demonstrates a high level of accuracy across different configurations, with the exception of the half-precision (FP16) run, which yielded inaccurate results. The overlapping results for all the other precision configurations indicate comparable physical outcomes. From Figure 3.6b, we can see how the levels of error varied among the mixed-precision runs. Notably, as the number of work arrays decreases significantly in the Storesome approach, a larger portion of the data is retained in higher precision. Consequently, a mixed half-single precision run exhibits a similar level of accuracy to that of a pure single precision run.

Figure 3.7 illustrates the impact of varying the timestep and Mach number M on the precision of the results. Small timesteps are required for low Mach number simulations and imply the additions of progressively smaller updates to the flow variables that are

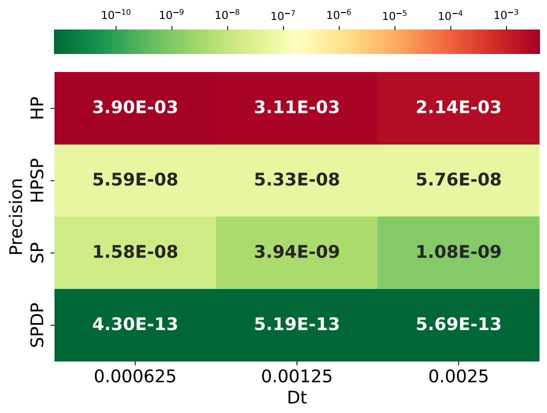


(a) Dissipation

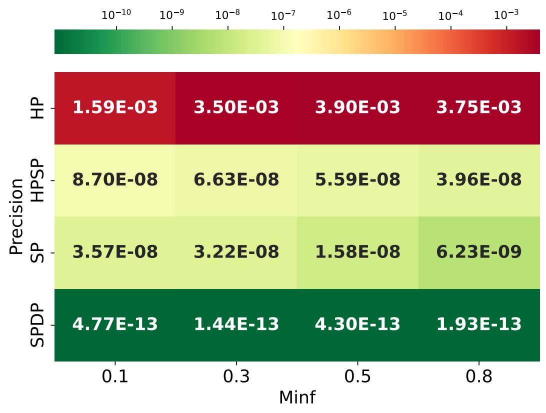


(b) Absolute difference of dissipation, compared to double precision

Figure 3.6: Numerical accuracy of TGsym app using different precision levels. Mesh size $= 256^3$, $M = 0.5$, $Re = 800$. The simulations were run for 8000 iterations using the Storesome method.



(a) Effect of changing timestep. $M = 0.5$, $Re = 800$, $20/dt$ iterations.



(b) Effect of changing Mach number M. $Re = 800$, 32000 iterations, $dt = 0.000625$

Figure 3.7: The effect of changing parameters of the model on the numerical accuracy of TGsym app using different precision levels. Size=256^3, default method. Values are the average of the absolute differences against the DP run of the dissipation at every 0.5 stepsize of simulation time.

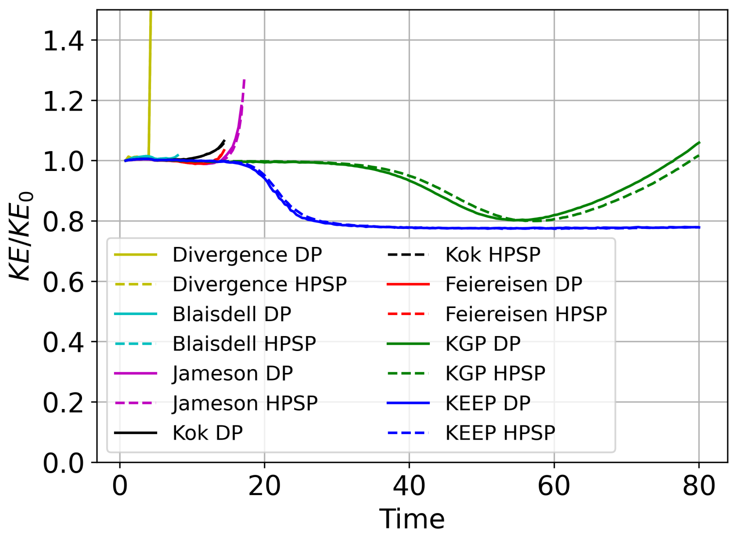


Figure 3.8: The effect of different split-forms on numerical accuracy of the inviscid Taylor-Green vortex application using DP and HPSP precision levels. $N = 64^3$ grid points, $M_\infty = 0.4$, $dt = 0.004$, inviscid calculation, StoreSome method.

potentially sensitive to reduced precision. Here I separate out the effects of time step from those of Mach number. In Figure 3.7a, where the time step is varied while keeping the Mach number constant, it is evident that the reduced precision runs, where all variables are maintained at the same precision level (HP or SP), yield slightly less accurate results with smaller timesteps. However, when residuals and work arrays are set to lower precision, changes in timestep do not affect the results. This is because the Q array is retained in higher precision, allowing for more accurate result storage. All observed errors remain within a factor of 10 of the smallest representable number for each specific precision. Figure 3.7b shows the effect of Mach number, while maintaining the same time step. While the HP case is incorrect for all Mach numbers, only a small increase in error is seen for the HPSP, SP, and SPDP cases as the Mach number is reduced. Overall, both subplots indicate that mixed/reduced precision runs can be effectively utilized.

Next a selection of the convective split-forms are considered that are available in the OpenSBLI solver [61] to improve simulation robustness. As we have already observed that half precision does not produce consistent results for the TGV application (Figure 3.6), I limit the discussion here only to the HPSP algorithm compared to the full double precision (DP) result. A full list of the equations for each of the split forms is given in the appendix of [99]. In order to isolate the relationship between precision and split-formulation of the equations, this comparison is performed as an inviscid TGV calculation with the diffusive terms omitted. A coarsened $N = 64^3$ grid is used at $M_\infty = 0.4$ [99] to test the robustness of the split-forms at different numerical precisions.

Figure 3.8 shows the time evolution of global kinetic energy normalised by its initial value. For a robust numerical scheme, the normalised kinetic energy for the inviscid case should stay at a value of one for as long as possible. The first thing to note in the figure is that, in the absence of physical viscosity in the inviscid limit considered here, several of the formulations (used for the convective terms of Navier-Stokes equations) diverge by showing exponential growth and eventually produce `NaN` when evaluated on a discrete mesh as non-linearities amplify aliasing errors. In particular, it can be observed that direct application of central differencing in the divergence form is unsuitable and the simulation rapidly diverges. Quadratic-split formulations by Blaisdell, Jameson, Kok, and Feiereisen (all listed in [99]) perform better, but still diverge within the standard simulation time used for Taylor-Green vortex problem (i.e. $t = 20$). In contrast, the cubic-split Kennedy-Gruber-Pirozzoli (KGP) [66] and Kinetic Energy and Entropy Preserving (KEEP) [99] schemes are robust and are able to maintain numerical stability even in the inviscid limit on a coarse grid. The physical behaviour we observe between the different split formulations is in good agreement with previous studies of this problem [99].

The effect of numerical precision is also shown on Figure 3.8, comparing double precision (solid line) with half-single precision (dashed line). Only very minor differences are observed between the full and mixed precision algorithms, with the largest differences being observed for the KGP scheme [66]. In each case, the trends between the split forms are consistent between double and half-single precision. This demonstrates that the improved numerical robustness of split formulations for the convective terms of the Navier-Stokes equations persists even when applying reduced/mixed numerical precision. The benefits of the mixed precision strategies are maintained when using other formulations of the equations, and are therefore not limited to only the baseline split-form.

To conclude this section, I highlight the findings regarding the accuracy of mixed and half precision formats in my simulations. My evaluations demonstrate that while mixed precision configurations, such as half-single precision (HPSP), maintain acceptable accuracy levels, pure half precision (HP) fails to meet the required standards across various conditions. Importantly, these results are consistent regardless of the simulation parameters, such as Reynolds number and Mach number, as well as the choice of numerical schemes. This reinforces the notion that while mixed precision techniques can be effectively employed in computational fluid dynamics (CFD) applications, careful selection of precision is crucial to ensure reliable outcomes. As I move forward in my research, understanding these accuracy implications will be essential for optimizing performance without compromising the integrity of simulation results.

3.3.3 Performance measurements

In the previous sections, I demonstrated how the usage of lower precision affects accuracy at the application level. In this subsection, I will examine the changes in performance and quantify the trade-offs involved. Under normal circumstances, a complete TGV simulation (mesh size = 256^3 $M = 0.5$. $Re = 800$, 8000 iterations, $\Delta t = 0.0025$) using FP64 precision on a GPU takes approximately 825.68 seconds. In contrast, I observed that the simulation rearranged according to the Storesome method is significantly faster, completing in just 402.48 seconds while utilizing nearly half as much memory. This efficiency allows us to conduct more or even larger simulations within given hardware constraints while ensuring that the final results remain qualitatively comparable. For

	Default		Storesome	
	runtime	speedup	runtime	speedup
HP	42.43 ms	2.43 ×	18.67 ms	2.69 ×
HPSP	47.71 ms	2.16 ×	22.13 ms	2.27 ×
SP	56.95 ms	1.81 ×	25.00 ms	2.01 ×
SPDP	74.02 ms	1.39 ×	37.60 ms	1.34 ×
DP	103.21 ms	1.00 ×	50.31 ms	1.00 ×

Table 3.1: Runtime per iteration and speedup compred to the double precision run time of the TGsym app are shown for the default and storesome generation methods. Mesh size=256^3, $M = 0.5$, Re=800, 800 iterations. The measurements are performed on a single NVIDIA A100-SXM4-40GB GPU with an AMD EPYC™ 7763 (Milan) CPU.

the performance measurements, I retained the mesh size of 256^3 but ran the simulations for only 10% of the total time steps (i.e. to time $t = 2$), executing each case five times and reporting the average of these runs. Then plotted the compute time per iteration, as this metric remains consistent throughout a simulation. Through my analysis of the representative TGV application, I illustrate what performance improvements users can expect in terms of both time and scale. For context, JAXA's aerofoil buffet applications run for three to four weeks using 120 Nvidia V100 GPUs [61], [63]. This highlights that running a substantial industrial application is not cheap; thus, any performance enhancement can be highly beneficial.

The performance evaluation of the TGsym application, shown in Table 3.1, reveals significant differences in runtime and speedup across various precision techniques. It is important to note that the Storesome method, by design, uses significantly fewer data arrays compared to the default method. This inherent efficiency of the Storesome approach contributes to its faster runtime performance by default.

As we transition to lower precision methods, we can observe a consistent increase in speedup. However, the approximate doubling of speedup ceases when using half precision (FP16). Currently, OPS does not support packed half precision types, such as `half2`, meaning that instruction throughput is still as if we were using FP32, and only the memory movement is reduced. The mixed precision configurations, specifically HPSP and SPDP, strike a balance between performance and accuracy, yielding speedups that fall between their respective pure precision counterparts (HP/SP and SP/DP).

Table 3.2 presents the performance of various methods on a CPU platform. Unlike the GPU, this CPU utilizes fixed 512-bit wide vectors, allowing it to fully utilize these vectors with smaller data sizes thanks to compiler auto-vectorization. As a result, we observe the expected doubling of speedup even when employing FP16 precision. Additionally, the data indicates superlinear scaling, which occurs as more data fits into the cache memory.

Table 3.3 presents the memory consumption associated with each configuration. All mixed precision cases fall between the memory requirements of their respective pure precision counterparts. The degree to which data arrays are cast to lower precision directly influences the overall memory savings. For instance, when fewer arrays are converted, as seen with the Storesome method, the resulting memory savings are less pronounced,

	Default		Storesome	
	runtime	speedup	runtime	speedup
HP	68.48 ms	5.71 ×	21.39 ms	8.12 ×
HPSP	105.22 ms	3.71 ×	47.31 ms	3.67 ×
SP	185.23 ms	2.11 ×	65.05 ms	2.67 ×
SPDP	249.11 ms	1.57 ×	123.22 ms	1.41 ×
DP	390.71 ms	1.00 ×	173.68 ms	1.00 ×

Table 3.2: Runtime per iteration and speedup compared to the double precision run of the TGsym app using the default and Storesome generation methods. Mesh size=256^3, Minf=0.5, Re=800, 800 iterations. The measurements are performed on Intel Xeon Platinum 8592+ CPU

	Default		Storesome	
	Memory	Gain	Memory	gain
HP	2.21 GB	4.00 ×	1.09 GB	4.00 ×
HPSP	2.72 GB	3.25 ×	1.60 GB	2.72 ×
SP	4.41 GB	2.00 ×	2.17 GB	2.00 ×
SPDP	5.43 GB	1.63 ×	3.19 GB	1.36 ×
DP	8.83 GB	1.00 ×	4.35 GB	1.00 ×

Table 3.3: Memory used with the TGsym app and memory gain compared to the double precision run. Size=256^3.

as are the resulting speedups. Overall, reducing an application's memory footprint enables the accommodation of larger problems within the same hardware constraints. For example, by halving the data size in an application that fully utilizes a GPU's memory, we can effectively double the overall mesh size. This capability is crucial for improving computational efficiency and expanding the scope of solvable problems in high-performance computing environments.

Figure 3.9 illustrates the communication requirements associated with utilizing multiple MPI processes. It is important to note that inter-node communication (usually through Infiniband) can be significantly slower than intra-node communication (usually through NVLink). Consequently, the reduction in data sizes has a pronounced effect on speedup when scaling across multiple devices. This figure also emphasizes the impact of selecting arrays to use in lower precision during mixed configurations. Since work arrays are communicated less frequently than other state arrays and the Storesome method utilizes fewer work arrays, the result is that mixed setups do not substantially reduce the MPI communication volume when employing the Storesome approach.

Figure 3.10 illustrates both the strong and weak scaling performance of the TGsym application across different precision levels. As anticipated, all configurations demonstrate effective scaling behavior. The parallel efficiency results from the strong scaling measurements indicate effective scalability across all precision configurations, with efficiencies ranging from 90% to 25% as the number of processes increases. It is important to note that at higher process counts, message latency has a more significant impact on performance than message size. As the number of processes increases, the number of messages per process remains constant (and even slightly increases); however, the sizes

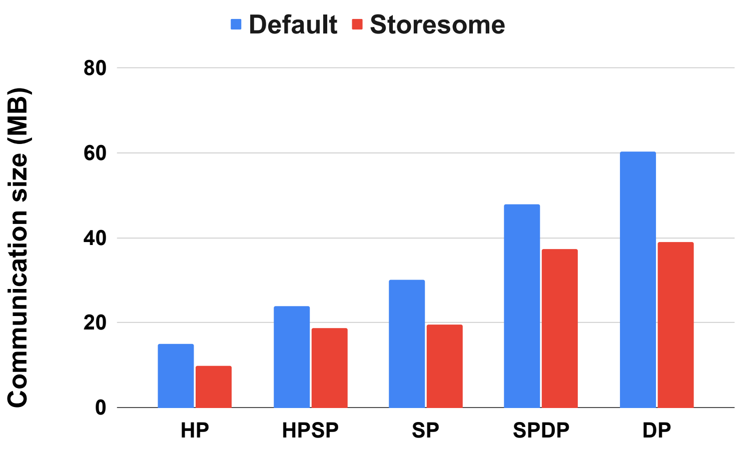
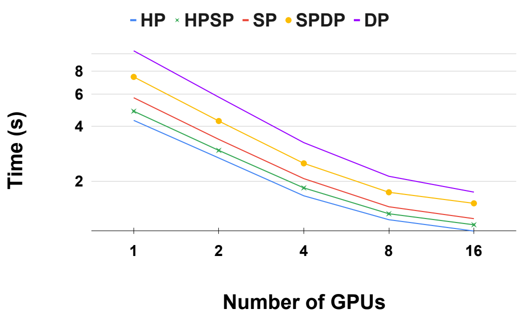
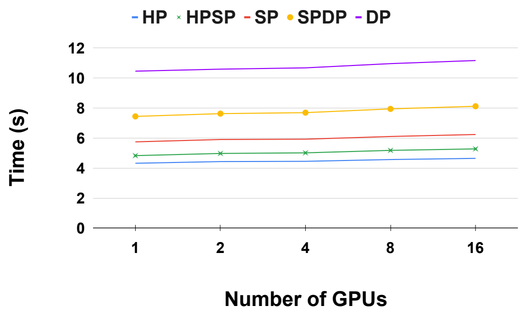


Figure 3.9: Average volume of MPI communications per process per iteration on the TGsym app, using 4 MPI processes. Size=256^3



(a) Strongscaling. Mesh size= 256^3



(b) Weakscaling, Mesh size= 256^3, multiplied by the number of processes in X direction.

Figure 3.10: Strong- and weakscaling of the TGsym app with GPUdirect using different precision levels. Minf=0.5, Re=800, 8000 iterations.

of these messages decrease. This reduction in message size helps explain why double precision (DP) performance approaches that of single precision (SP) in these scenarios. In contrast, the weak scaling measurements reveal consistently high efficiencies across all configurations, exceeding 92%. This indicates that the TGsym application scales exceptionally well, irrespective of the precision level used.

3.4 Conclusions and Future Work

In this work, I have explored the implementation and effectiveness of mixed precision techniques in compressible turbulent flow simulations using explicit finite difference schemes. By extending both the OPS library and the OpenSBLI framework to support mixed precision arithmetic, I demonstrated that significant performance gains can be achieved without sacrificing numerical accuracy, provided that the precision is selected carefully. My experiments using the Taylor-Green vortex benchmark revealed that while reduced precision formats such as single and mixed half-single precision yield acceptable results, pure half-precision computations suffer from unacceptable numerical inaccuracies.

The proposed mixed precision algorithm effectively balances performance and precision, allowing for improved computational efficiency, particularly in memory usage and communication overheads, which are crucial in multi-CPU and multi-GPU environments. My analysis indicates that mixed precision approaches can provide substantial speedups, especially in larger-scale simulations, without compromising the integrity of the results.

Several avenues remain for further exploration:

- Larger and More Complex Applications: To fully understand the potential of mixed precision techniques, future work will experiment with larger and more complex computational fluid dynamics (CFD) applications. These experiments will help assess the scalability and generalizability of my methods in real-world, high-performance computing (HPC) environments.

- Improving Half-Precision Performance: Since pure half-precision (FP16) simulations did not yield acceptable accuracy in my current tests, I did not prioritize optimizing FP16 performance in this work. However, there may be specific simulations where FP16 can produce satisfactory results, particularly on GPU architectures. To support these scenarios, future work will focus on enhancing OPS to define global constants in half precision and to utilize packed types, allowing multiple half-precision calculations within a single warp. These optimizations could unlock further performance gains on hardware optimized for FP16 computations, such as modern GPUs.

By addressing these areas, I aim to further optimize the trade-offs between computational efficiency and numerical accuracy, thereby pushing the boundaries of what is possible with mixed precision in large-scale turbulent simulations.

4 Summary of the Dissertation

As high-performance computing continues to evolve, the pressure to improve computational performance is matched by a growing need for numerical reliability. Scientific codes today are executed across a wide variety of architectures and parallel environments, yet even small changes in configuration – such as the number of MPI processes or the precision level of a variable – can result in differing outputs. For fields where simulation results underpin critical design or research decisions, this lack of determinism is increasingly problematic.

This dissertation explores two distinct, and often opposing, strategies for addressing these challenges. On one hand, it investigates bitwise reproducibility: ensuring that floating-point computations yield identical results regardless of the platform or level of parallelism. On the other, it evaluates reduced and mixed-precision computing, where parts of a simulation are executed with lower numerical precision to improve performance and reduce memory usage. While reproducibility prioritizes consistency at the cost of speed, mixed-precision aims for speed with carefully managed compromises in accuracy. Both directions respond to real needs in high-performance scientific computing, and both require new techniques to be effectively integrated into modern simulation workflows.

The core contributions of this work lie in the development and extension of OPS and OP2: two domain-specific libraries for structured and unstructured mesh applications. These tools provide the foundation for implementing and evaluating the methods described in the following sections. The remainder of this chapter begins by summarizing the computational techniques and frameworks used (Section 4.1), followed by a detailed presentation of the new scientific results and thesis points (Section 4.2), and a discussion of potential applications and benefits (Section 4.3).

4.1 Methods and tools

The research described in this dissertation was implemented primarily within the OP2 and OPS libraries: high-level code generation frameworks designed to support portable, scalable, and performance-efficient simulation codes on modern computing platforms.

OP2 is a domain-specific language and runtime system for unstructured mesh computations, commonly used in finite volume and finite element methods. It allows users to express computation over mesh elements (e.g., edges, cells, vertices) while abstracting away details of data dependencies and parallelization. OP2 supports automatic code generation for multiple backends, including OpenMP, MPI, and CUDA, enabling efficient execution on both CPU and GPU systems. In this work, OP2 was extended to support new mechanisms for enforcing bitwise reproducibility. These included:

- a temporary array-based scheme for deterministic accumulation of indirect increments,
- reproducible graph coloring algorithms, including a distributed version that functions independently of mesh partitioning,
- and the integration of ReproBLAS routines to support deterministic global reductions in parallel MPI environments.

OPS is the structured-mesh equivalent of OP2, providing similar abstractions for stencil-based computations. It serves as the target backend for a number of higher-level code generation tools – including OpenSBLI, which was used in this work to test and evaluate reduced and mixed-precision computing strategies. OpenSBLI allows users to define partial differential equations using symbolic notation, and automatically generates OPS-based C++ code tailored to the desired simulation and hardware environment.

To explore the trade-offs of reduced precision, modifications were made to the OPS and OpenSBLI pipelines to support different precision levels for specific variables and operations. These changes enabled experiments where, for example, residual calculations could be performed in FP16 or FP32, while conserved state variables remained in FP64. The goal was to identify combinations of precision that preserved simulation accuracy while improving runtime performance – especially on bandwidth-limited GPU architectures.

The methodologies developed in this work were validated using several benchmark and industrial-scale applications. On the unstructured side, reproducibility techniques were tested on Airfoil, Aero, MG-CFD, and the Rolls-Royce Hydra application, all implemented in OP2. On the structured side, reduced-precision experiments were conducted using OpenSBLI on the Taylor-Green vortex benchmark – a widely used test case for evaluating numerical schemes in compressible turbulence.

Together, the tools and frameworks developed in this dissertation provide a flexible and extensible platform for exploring critical aspects of numerical correctness and performance optimization in large-scale scientific codes. The following section presents the new scientific results derived from these methodologies.

4.2 New scientific results

This section presents the main scientific contributions of the dissertation in the form of thesis points. Each thesis group highlights a major area of research, while individual points describe specific results supported by the methods and tools introduced in earlier chapters.

Thesis group I. *Algorithms for reproducible floating-point operations defined on unstructured mesh applications.*

Thesis I.1. *Starting from the OP2 DSL abstraction, I showed which floating-point operations – such as parallel reductions, indirect memory updates, and non-deterministic execution order – are responsible for the potential violation of the reproducibility property.*

I showed what steps – temporary array-based accumulation, deterministic graph coloring, and the use of ReproBLAS – can be taken to ensure that these operations still produce reproducible results.

In this part of the work, I began by analyzing the computational patterns inherent in unstructured mesh applications written using the OP2 DSL. These patterns typically involve indirect memory accesses and accumulation operations – particularly in reductions and read-write kernels – which are highly susceptible to non-determinism in parallel environments. I systematically identified the categories of floating-point operations in OP2 that can lead to violations of bitwise reproducibility, with a focus on those involving reductions (e.g., OP_INC and OP_RW) over shared data.

To address these issues, I proposed two main algorithmic solutions. First, a temporary array-based accumulation scheme was introduced, which ensures that increments are applied in a fixed, deterministic order based on globally unique element IDs. This is illustrated in Figure 2.4, where the consistent ordering of edge contributions to a cell guarantees reproducibility across runs. Second, the reproducibility of global reductions was addressed by integrating the ReproBLAS library into OP2, enabling deterministic summation of floating-point arrays regardless of thread count, process layout, or reduction tree structure. These methods were implemented with minimal intrusion into OP2's abstraction, preserving the productivity benefits of the DSL while ensuring deterministic output.

The effectiveness of these methods was demonstrated through test cases such as Aero and MG-CFD, where non-determinism in the results was visibly reduced or eliminated under varying process counts. For instance, Figure 1.1 presents a histogram of differences in a conjugate-gradient solver where bitwise reproducibility was not enforced – highlighting the numerical drift caused by changes in execution configuration.

Thesis I.2. *I introduced a new algorithm that ensures reproducible coloring on distributed, partitioned graphs, independent of the number of partitions. The algorithm builds on M. Osama's graph coloring method originally developed for GPUs, and extends it to ensure full determinism in a distributed setting. I demonstrated how these algorithms – temporary array-based accumulation, deterministic graph coloring, and ReproBLAS-based reductions – can be efficiently mapped to diverse parallelization strategies. I demonstrated that these methods achieve near-optimal performance on multi-core processors, distributed systems, and GPUs, and exhibit practical scalability across industrially representative applications.*

Graph coloring is a widely used technique in parallel computing to avoid race conditions during updates to shared data. Traditional coloring methods, however, often depend on the mesh partitioning strategy, introducing variability when the number or configuration of partitions changes. To address this, I developed a novel distributed coloring algorithm that guarantees reproducible color assignments regardless of partitioning. The algorithm extends existing parallel coloring techniques with a deterministic ordering based on global element identifiers and hash-based tie-breaking. It is detailed in Algorithm 3 and illustrated through the use of an additional communication layer (ghost elements) in

Figure 2.5. This design ensures that color assignments remain consistent even when the mesh is redistributed across varying numbers of MPI processes or affected by load balancing.

The reproducibility of this coloring method was validated across several OP2 applications, including the industrial-grade Hydra CFD code, where consistent parallel execution is critical for debugging and validation. Moreover, the algorithm is generic and applicable beyond OP2, making it a robust tool for deterministic parallelism in unstructured domains.

One of the key challenges in enabling reproducible execution is minimizing its performance overhead. To this end, the reproducibility techniques – including the new coloring algorithm, temporary array accumulation, and deterministic global reductions – were designed to integrate efficiently with OP2's code generation system and parallel backends. I demonstrated how these algorithms can be mapped onto different parallelization strategies with near-optimal performance across diverse architectures, including multi-core CPUs, distributed-memory clusters, and CUDA-enabled GPUs.

Extensive benchmarking was conducted on representative OP2 applications. Figures 2.6 through 2.10 present detailed performance evaluations, including slowdowns compared to non-reproducible baselines and scaling behavior across increasing core counts. In the Hydra case study, the reproducibility infrastructure was tested at scale, and results showed that overheads remained within acceptable bounds – especially in light of the benefits offered by deterministic outputs.

On GPU systems, where controlling race conditions is particularly challenging, the proposed methods were successfully deployed using OP2's CUDA backend. Reproducible execution was achieved on Nvidia V100 accelerators without significant restructuring of user code, demonstrating the practical feasibility and portability of the proposed solutions across a broad range of real-world computing environments.

Publications related to this thesis group are:[J1], [C1], [C2], [C3], [C4] .

Thesis II.

I introduced a methodology for systematically reducing numerical precision in structured grid applications and measuring its impact on both performance and numerical accuracy. Applying this to a representative turbulent simulation (Taylor-Green vortex), I demonstrated that using 32-bit floating-point representation yields performance gains while maintaining acceptable accuracy, whereas 16-bit usage significantly alters the physical conclusions.

Building on this, I extended the OpenSBLI framework to support a mixed-precision strategy, enabling specific state variables and temporary storage to be computed and stored at lower precision. This enables the evaluation of mixed precision configurations in a controlled, quantitative manner, and I showed that carefully selected combinations of 16- and 32-bit precision can preserve accuracy and lead to substantial runtime improvements on both modern CPU and GPU architectures.

This part of the dissertation addresses the emerging opportunity – driven largely by hardware developments – to use reduced precision arithmetic in scientific simulations. Motivated by the increasing support for 16- and 32-bit floating-point operations on modern GPUs and CPUs, I developed a methodology to systematically evaluate how such precision reductions impact both simulation accuracy and runtime performance.

The methodology was applied to a canonical fluid dynamics problem: the Taylor-Green vortex, a 3D unsteady turbulence benchmark. This problem is well-suited for sensitivity analysis, as it exhibits complex, nonlinear dynamics that can amplify numerical errors and highlight instability in lower precision settings.

Using OpenSBLI as the frontend for code generation and OPS as the backend for parallel execution, I implemented double-precision (FP64), single-precision (FP32), and half-precision (FP16) versions of the simulation. Accuracy was assessed using global quantities such as kinetic energy dissipation and numerical error norms, while performance was measured in terms of runtime per iteration and memory usage.

The results, shown in Figures 3.3–3.7, confirm that FP32 precision offers a good balance, delivering significant speedups (especially on GPU platforms) while preserving physically accurate behavior. However, simulations run entirely in FP16 precision showed large deviations in key quantities, leading to altered or unphysical results, particularly near peak dissipation. This reinforces the conclusion that while 16-bit arithmetic may be viable in localized contexts, it cannot be blindly applied across an entire CFD code without compromising scientific validity.

Importantly, the infrastructure developed for this analysis is not specific to the Taylor-Green vortex. Thanks to OpenSBLI's symbolic and backend-agnostic design, the same methodology can be reused to analyze other CFD models using finite difference discretizations on structured grids. As such, this work lays the groundwork for future investigations into precision-aware modeling practices in high-performance simulation workflows.

Building on the findings from full reduced-precision simulations, this thesis point presents a more granular and flexible approach: mixed-precision computing. Rather than applying the same numerical format across all variables and operations, the strategy here was to assign precision levels selectively, based on the numerical role and stability sensitivity of each component.

To enable this, I extended OpenSBLI's code generation infrastructure to support variable-specific precision control. This required modifications to the symbolic frontend, variable type declarations, and the generated OPS kernel code, ensuring that temporary arrays, work buffers, and conservative state variables could all be defined with differing levels of floating-point precision. The implementation supports combinations such as FP64 for conserved variables and FP32 or FP16 for intermediate operations.

Using the Taylor-Green vortex problem again as a testbed, I evaluated several mixed-precision configurations, such as FP64 state variables with FP32 residuals, or FP32 state with FP16 temporaries. Performance benchmarks (Figures 3.9 and 3.10) show that these configurations offer meaningful speedups without the loss of simulation accuracy observed in the pure FP16 runs. The runtime-per-iteration and memory usage improvements are summarized in Tables 3.1–3.3, highlighting gains of more than 2× speedup with negligible loss of fidelity in key physical quantities.

Publications related to this thesis group are: [J2], [C5], [C6].

Publication related to this thesis group under review at the time of the submission of this dissertation: [J3],

4.3 Potential applications and benefits

The techniques and methodologies developed in this dissertation address two pressing concerns in computational science – numerical reproducibility and efficient use of floating-point precision – both of which have practical implications for current and future simulation workflows across a wide range of disciplines.

The reproducibility work in unstructured mesh applications has immediate applications in industrial CFD codes, such as Rolls-Royce Hydra, and similar engineering simulation platforms used in aerospace, automotive, and energy sectors. For these domains, consistency of results across executions is not only desirable – it is often required for validation, certification, and regulatory compliance. The ability to run large simulations on many different hardware platforms, or with different levels of parallelism, and still produce bitwise identical results improves debugging, testing, and verification workflows significantly. In particular, the reproducible coloring and reduction strategies integrated into OP2 provide a foundation for deterministic behavior in highly parallel environments, without requiring developers to abandon performance or scalability.

The reduced- and mixed-precision computing strategies, tested using OpenSBLI and OPS, are highly relevant for modern heterogeneous computing systems, especially those built around GPUs and AI accelerators. As hardware trends continue to favor lower-precision execution units (e.g., tensor cores, float16 pipelines), simulation codes that can exploit these capabilities without compromising accuracy will have a significant performance advantage.

These techniques are particularly beneficial in scenarios where simulation time is a limiting factor – such as real-time decision support, design optimization, or large parameter sweeps in uncertainty quantification. For example, in aerodynamics or combustion modeling, being able to run more simulations in less time can dramatically accelerate design cycles. Similarly, in academic research, the ability to run high-resolution simulations on modest hardware using mixed precision opens new doors for smaller research groups and under-resourced institutions.

The methodology introduced here – along with the precision-aware infrastructure integrated into OPS – enables future work on adaptive precision, where the simulation dynamically adjusts numerical accuracy based on local error estimates or physics-driven criteria. This points to longer-term potential for even more intelligent and efficient simulation strategies.

Finally, by embedding all enhancements within OP2 and OPS – rather than developing standalone prototypes – the work ensures accessibility and long-term maintainability. Developers using these libraries benefit from reproducibility and mixed precision features without needing to reimplement their core simulation logic. This lowers the barrier to entry for high-performance, precision-aware computing and supports the broader goal of sustainable software in computational science.

In summary, the contributions of this dissertation offer robust, production-ready solutions for two major concerns in scientific simulation. Whether the goal is consistency of results or speed and efficiency, the methods described here are applicable, portable, and scalable – ready to meet the evolving demands of high-performance computing.

Use of AI Assistance

Parts of this dissertation benefited from the assistance of AI tools. In particular, ChatGPT (OpenAI), Perplexity.ai, and GitHub Copilot were used for refining the phrasing of technical descriptions, organizing text, checking grammar, and accelerating code development workflows. All results, insights, and critical reasoning remain my own, and the use of these tools was limited to supporting clarity and productivity.

List of author publications

List of journal publications

[J1] **B. Siklósi**, G. R. Mudalige, and I. Z. Reguly, "Enabling bitwise reproducibility for the unstructured computational motif", *Applied Sciences*, vol. 14, no. 2, 2024, ISSN: 2076-3417. DOI: `10.3390/app14020639`. [Online]. Available: `https://www.mdpi.com/2076-3417/14/2/639` (cit. on p. 69).

[J2] D. J. Lusher, A. Sansica, N. D. Sandham, J. Meng, **B. Siklósi**, and A. Hashimoto, "Opensbli v3.0: High-fidelity multi-block transonic aerofoil cfd simulations using domain specific languages on gpus", *Computer Physics Communications*, vol. 307, p. 109 406, 2025, ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2024.109406`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0010465524003291` (cit. on p. 71).

[J3] **B. Siklosi**, P. K. Sharma, D. J. Lusher, I. Z. Reguly, and N. D. Sandham, "Reduced and mixed precision turbulent flow simulations using explicit finite difference schemes", *Future Generation Computer Systems*, 2025, Under review (cit. on p. 71).

List of conference publications

[C1] **B. Siklósi**, I. Z. Reguly, and G. R. Mudalige, "Bitwise reproducible task execution on unstructured mesh applications", in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 889–892. DOI: `10.1109/CCGrid49817.2020.00015` (cit. on p. 69).

[C2] **B. Siklósi**, "Bitwise reproducible execution of unstructured mesh applications", in *PhD Proceedings Annual Issues of the Doctoral School, Faculty of Information Technology and Bionics*, vol. 16, 2021, pp. 165–168 (cit. on p. 69).

[C3] **B. Siklósi**, "Bitwise reproducible execution of unstructured mesh applications", in *PhD Proceedings Annual Issues of the Doctoral School, Faculty of Information Technology and Bionics*, vol. 15, 2020, pp. 161–164 (cit. on p. 69).

[C4] **B. Siklósi**, I. Z. Reguly, and G. R. Mudalige, "Bitwise reproducible execution of unstructured mesh applications", *Jedlik Laboratories Reports*, vol. 9, no. 2, pp. 13–19, 2020 (cit. on p. 69).

[C5] **B. Siklósi**, "Achieving mixed precision computing with the help of domain specific libraries", in *PhD Proceedings Annual Issues of the Doctoral School, Faculty of Information Technology and Bionics*, vol. 17, 2022, pp. 187–189 (cit. on p. 71).

[C6] **B. Siklósi**, “Utilizing the op2 domain specific library for adaptive multi-precision computing”, in *PhD Proceedings Annual Issues of the Doctoral School, Faculty of Information Technology and Bionics*, vol. 18, 2023, pp. 141–144 (cit. on p. 71).

List of publications not related to the dissertation

[O1] **B. Siklosi**, I. Z. Reguly, and G. R. Mudalige, “Heterogeneous cpu-gpu execution of stencil applications”, in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 71–80. DOI: `10.1109/P3HPC.2018.00010`.

[O2] B. Keömley-Horváth, G. Horváth, P. Polcz, *et al.*, “The design and utilisation of pansim, a portable pandemic simulator”, in *2022 First Combined International Workshop on Interactive Urgent Supercomputing (CIW-IUS)*, 2022, pp. 1–9. DOI: `10.1109/CIW-IUS56691.2022.00006`.

List of references related to the dissertation

[1] I. Foster and J. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering* (Literature and Philosophy). Addison-Wesley, 1995, ISBN: 9780201575941. [Online]. Available: `https://books.google.hu/books?id=r5JsQgAACAAJ` (cit. on p. 14).

[2] J. Dongarra, P. Beckman, P. Aerts, *et al.*, "The international exascale software project: A call to cooperative action by the global high-performance community", *The International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 309–322, 2009. DOI: `10.1177/1094342009347714`. eprint: `https://doi.org/10.1177/1094342009347714`. [Online]. Available: `https://doi.org/10.1177/1094342009347714` (cit. on p. 14).

[3] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff.", *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006. DOI: `10.1109/N-SSC.2006.4785860` (cit. on p. 15).

[4] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions", *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974. DOI: `10.1109/JSSC.1974.1050511` (cit. on p. 15).

[5] S. Boldo, C.-P. Jeannerod, G. Melquiond, and J.-M. Muller, "Floating-point arithmetic", *Acta Numerica*, vol. 32, pp. 203–290, 2023. DOI: `10.1017/S0962492922000101` (cit. on pp. 16, 17).

[6] "Ieee standard for floating-point arithmetic", *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: `10.1109/IEEESTD.2019.8766229` (cit. on pp. 17, 20).

[7] D. Goldberg, "What every computer scientist should know about floating-point arithmetic", *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991, ISSN: 0360-0300. DOI: `10.1145/103162.103163`. [Online]. Available: `https://doi.org/10.1145/103162.103163` (cit. on pp. 17, 33).

[8] O. Villa, D. Chavarría-Miranda, V. Gurumoorthi, A. Marquez, and S. Krishamoorthy, "Effects of floating-point non-associativity on numerical computations on massively multithreaded systems", in *CUG Proceedings*, May 2009 (cit. on pp. 17, 18).

[9] B. L. Massingill, T. G. Mattson, and B. A. Sanders, “Reengineering for parallelism: An entry point into plpp for legacy applications”, *Concurrency and Computation: Practice and Experience*, vol. 19, no. 4, pp. 503–529, 2007. DOI: https://doi.org/10.1002/cpe.1147. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1147. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1147 (cit. on p. 18).

[10] O. Zienkiewicz, R. Taylor, and J. Zhu, *The Finite Element Method: its Basis and Fundamentals (Seventh Edition)*, Seventh Edition. Oxford: Butterworth-Heinemann, 2013, ISBN: 978-1-85617-633-0. DOI: https://doi.org/10.1016/B978-1-85617-633-0.00001-0 (cit. on pp. 19, 35).

[11] J. D. Zechar, D. Schorlemmer, M. Liukis, *et al.*, “The collaboratory for the study of earthquake predictability perspective on computational earthquake science”, *Concurrency and Computation: Practice and Experience*, vol. 22, no. 12, pp. 1836–1847, 2010. DOI: https://doi.org/10.1002/cpe.1519. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1519. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1519 (cit. on p. 19).

[12] L. Teodosio, L. Marchitto, C. Tornatore, F. Bozza, and G. Valentino, “Effect of cylinder-by-cylinder variation on performance and gaseous emissions of a pfi spark ignition engine: Experimental and 1d numerical study”, *Applied Sciences*, vol. 11, no. 13, 2021, ISSN: 2076-3417. DOI: 10.3390/app11136035. [Online]. Available: https://www.mdpi.com/2076-3417/11/13/6035 (cit. on p. 19).

[13] J. Ren, Y. Zeng, S. Zhou, and Y. Zhang, “An experimental study on state representation extraction for vision-based deep reinforcement learning”, *Applied Sciences*, vol. 11, no. 21, 2021, ISSN: 2076-3417. DOI: 10.3390/app112110337. [Online]. Available: https://www.mdpi.com/2076-3417/11/21/10337 (cit. on p. 19).

[14] P. N. Sergi, N. De la Oliva, J. del Valle, X. Navarro, and S. Micera, “Physically consistent scar tissue dynamics from scattered set of data: A novel computational approach to avoid the onset of the runge phenomenon”, *Applied Sciences*, vol. 11, no. 18, 2021, ISSN: 2076-3417. DOI: 10.3390/app11188568. [Online]. Available: https://www.mdpi.com/2076-3417/11/18/8568 (cit. on p. 19).

[15] L. Elster, J. P. Staab, and S. Peters, “Making automotive radar sensor validation measurements comparable”, *Applied Sciences*, vol. 13, no. 20, 2023, ISSN: 2076-3417. DOI: 10.3390/app132011405. [Online]. Available: https://www.mdpi.com/2076-3417/13/20/11405 (cit. on p. 19).

[16] F. Petrini, A. Moody, J. Peinador, E. Frachtenberg, and D. Panda, “Nic-based reduction algorithms for large-scale clusters”, *IJHPCN*, vol. 4, pp. 122–136, Jan. 2006. DOI: 10.1504/IJHPCN.2006.010635 (cit. on p. 20).

[17] S. Siegel and J. Wolff von Gudenberg, "A long accumulator like a carry-save adder", *Computing*, vol. 94, no. 2, pp. 203–213, Mar. 2012, ISSN: 1436-5057. DOI: 10.1007/s00607-011-0164-x. [Online]. Available: https://doi.org/10.1007/s00607-011-0164-x (cit. on p. 20).

[18] H. Atmanspacher and S. Maasen, *Reproducibility: principles, problems, practices, and prospects*. John Wiley & Sons, 2016 (cit. on p. 20).

[19] J. Demmel and H. D. Nguyen, "Fast reproducible floating-point summation", ser. 2013 IEEE 21st Symposium on Computer Arithmetic, 2013, pp. 163–172. DOI: 10.1109/ARITH.2013.9 (cit. on pp. 20, 21, 34).

[20] A. Arteaga, O. Fuhrer, and T. Hoefler, "Designing bit-reproducible portable high-performance applications", ser. 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 1235–1244. DOI: 10.1109/IPDPS.2014.127 (cit. on p. 20).

[21] J. Demmel, P. Ahrens, and H. D. Nguyen, "Efficient reproducible floating point summation and blas", EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-121, Jun. 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-121.html (cit. on pp. 20, 21, 34).

[22] P. Ahrens, H. D. Nguyen, and J. Demmel, "Efficient reproducible floating point summation and blas", *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-229*, 2015 (cit. on pp. 20, 21).

[23] R. Salgado-Estrada, A. Galván, J. Y. Moreno-Martínez, *et al.*, "Wind vulnerability of flexible outdoor single-post billboards", *Applied Sciences*, vol. 13, no. 10, 2023, ISSN: 2076-3417. DOI: 10.3390/app13106197. [Online]. Available: https://www.mdpi.com/2076-3417/13/10/6197 (cit. on p. 21).

[24] X. An, S. Li, and T. Wu, "Modeling nonlinear aeroelastic forces for bridge decks with various leading edges using lstm networks", *Applied Sciences*, vol. 13, no. 10, 2023, ISSN: 2076-3417. DOI: 10.3390/app13106005. [Online]. Available: https://www.mdpi.com/2076-3417/13/10/6005 (cit. on p. 21).

[25] W. Kahan, "Pracniques: Further remarks on reducing truncation errors", *Commun. ACM*, vol. 8, no. 1, p. 40, Jan. 1965, ISSN: 0001-0782. DOI: 10.1145/363707.363723. [Online]. Available: https://doi.org/10.1145/363707.363723 (cit. on pp. 21, 34).

[26] S. Cherubin, G. Agosta, I. Lasri, E. Rohou, and O. Sentieys, "Implications of reduced-precision computations in hpc: Performance, energy and error", in *International Conference on Parallel Computing*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:3762567 (cit. on pp. 21, 22).

[27] J. Sun, G. D. Peterson, and O. O. Storaasli, "High-performance mixed-precision linear solver for fpgas", *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1614–1623, 2008. DOI: 10.1109/TC.2008.89 (cit. on pp. 23, 50).

[28] A. Abdelfattah, S. Tomov, and J. Dongarra, “Towards half-precision computation for complex matrices: A case study for mixed precision solvers on gpus”, in *2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, 2019, pp. 17–24. DOI: `10.1109/ScalA49573.2019.00008` (cit. on pp. 23, 50).

[29] J. D. Hogg and J. A. Scott, “A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems”, *ACM Trans. Math. Softw.*, vol. 37, no. 2, Apr. 2010, ISSN: 0098-3500. DOI: `10.1145/1731022.1731027`. [Online]. Available: `https://doi.org/10.1145/1731022.1731027` (cit. on pp. 23, 50).

[30] N. J. Higham and T. Mary, “Mixed precision algorithms in numerical linear algebra”, *Acta Numerica*, vol. 31, pp. 347–414, 2022. DOI: `10.1017/S0962492922000022` (cit. on pp. 23, 50).

[31] P. Luszczek, A. Abdelfattah, H. Anzt, A. Suzuki, and S. Tomov, “Batched sparse and mixed-precision linear algebra interface for efficient use of gpu hardware accelerators in scientific applications”, *Future Generation Computer Systems*, vol. 160, pp. 359–374, 2024, ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2024.06.004`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0167739X24003017` (cit. on pp. 23, 50).

[32] A. Abdelfattah, H. Anzt, A. Ayala, *et al.*, “Advances in mixed precision algorithms: 2021 edition”, Aug. 2021. DOI: `10.2172/1814447`. [Online]. Available: `https://www.osti.gov/biblio/1814447` (cit. on pp. 23, 50).

[33] M. Lehmann, M. J. Krause, G. Amati, M. Sega, J. Harting, and S. Gekle, “Accuracy and performance of the lattice boltzmann method with 64-bit, 32-bit, and customized 16-bit number formats”, *Phys. Rev. E*, vol. 106, p. 015 308, 1 Jul. 2022. DOI: `10.1103/PhysRevE.106.015308`. [Online]. Available: `https://link.aps.org/doi/10.1103/PhysRevE.106.015308` (cit. on pp. 23, 50).

[34] F. Brogi, S. Bnà, G. Boga, G. Amati, T. Esposti Ongaro, and M. Cerminara, “On floating point precision in computational fluid dynamics using openfoam”, *Future Generation Computer Systems*, vol. 152, pp. 1–16, 2024, ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2023.10.006`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0167739X23003813` (cit. on pp. 23, 50).

[35] E. Goubault, “Static analyses of the precision of floating-point operations”, in *Static Analysis*, P. Cousot, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 234–259, ISBN: 978-3-540-47764-8 (cit. on p. 23).

[36] F. Benz, A. Hildebrandt, and S. Hack, “A dynamic program analysis to find floating-point accuracy problems”, *SIGPLAN Not.*, vol. 47, no. 6, pp. 453–462, Jun. 2012, ISSN: 0362-1340. DOI: `10.1145/2345156.2254118`. [Online]. Available: `https://doi.org/10.1145/2345156.2254118` (cit. on p. 23).

[37] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, “Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers”, in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 603–613. DOI: 10.1109/SC.2018.00050 (cit. on p. 23).

[38] J. Wan, W. Wang, and Z. Zhang, “Enhancing computational efficiency in 3-d seismic modelling with half-precision floating-point numbers based on the curvilinear grid finite-difference method”, *Geophysical Journal International*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:271042153 (cit. on p. 23).

[39] P. Luszczek, I. Yamazaki, and J. Dongarra, “Increasing accuracy of iterative refinement in limited floating-point arithmetic on half-precision accelerators”, in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–6. DOI: 10.1109/HPEC.2019.8916392 (cit. on p. 24).

[40] Intel, *Bfloat16 – hardware numerics definition*, https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf, Nov. 2018 (cit. on p. 24).

[41] Y. Tortorella, L. Bertaccini, L. Benini, D. Rossi, and F. Conti, “Redmule: A mixed-precision matrix–matrix operation engine for flexible and energy-efficient on-chip linear algebra and tinyml training acceleration”, *Future Generation Computer Systems*, vol. 149, pp. 122–135, 2023, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.07.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23002546 (cit. on p. 24).

[42] F. Rathgeber, D. A. Ham, L. Mitchell, *et al.*, “Firedrake: Automating the finite element method by composing abstractions”, vol. 43, no. 3, Dec. 2016, ISSN: 0098-3500. DOI: 10.1145/2998441. [Online]. Available: https://doi.org/10.1145/2998441 (cit. on p. 24).

[43] G. N. W. e. a. A. Logg K.-A. Mardal, *Automated Solution of Differential Equations by the Finite Element Method.* Springer, 2012. DOI: 10.1007/978-3-642-23099-8 (cit. on p. 24).

[44] R. Biedron, J.-R. Carlson, J. Derlaga, P. Gnoffo, D. Hammond, and K. J. et al., *Fun3d manual 13.7 nasa/tm-2020-5010139*, 2020 (cit. on p. 24).

[45] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, “Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures”, ser. 2012 Innovative Parallel Computing (InPar), 2012, pp. 1–12. DOI: 10.1109/InPar.2012.6339594 (cit. on pp. 24, 25).

[46] C. Chevalier and F. Pellegrini, “Pt-scotch: A tool for efficient parallel graph ordering”, *Parallel Computing*, vol. 34, no. 6–8, pp. 318–331, Jun. 2008, ISSN: 0167-8191. DOI: 10.1016/j.parco.2007.12.001. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2007.12.001 (cit. on p. 24).

[47] G. Karypis, “Metis and parmetis”, in *Encyclopedia of Parallel Computing.* Boston, MA: Springer US, 2011, pp. 1117–1124, ISBN: 978-0-387-09766-4. DOI: `10.1007/978-0-387-09766-4_500`. [Online]. Available: `https://doi.org/10.1007/978-0-387-09766-4_500` (cit. on p. 24).

[48] X. Zhang, X. Sun, X. Guo, Y. Du, Y. Lu, and Y. Liu, “Re-evaluation of atomic operations and graph coloring for unstructured finite volume gpu simulations”, ser. 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2020, pp. 297–304. DOI: `10.1109/SBAC-PAD49847.2020.00048` (cit. on p. 25).

[49] A. Sulyok, G. Balogh, I. Reguly, and G. Mudalige, “Locality optimized unstructured mesh algorithms on gpus”, *Journal of Parallel and Distributed Computing*, vol. 134, Aug. 2019. DOI: `10.1016/j.jpdc.2019.07.011` (cit. on p. 25).

[50] I. Reguly, D. Giles, D. Gopinathan, *et al.*, “The volna-op2 tsunami code (version 1.5)”, *Geoscientific Model Development*, vol. 11, pp. 4621–4635, Nov. 2018. DOI: `10.5194/gmd-11-4621-2018` (cit. on p. 27).

[51] I. Z. Reguly, G. R. Mudalige, C. Bertolli, *et al.*, “Acceleration of a full-scale industrial cfd application with op2”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1265–1278, 2016. DOI: `10.1109/TPDS.2015.2453972` (cit. on p. 27).

[52] B. Szilniczky-Erőss and I. Z. Reguly, “Performance portability of the mg-cfd mini-app with sycl”, in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL ’20, Munich, Germany: Association for Computing Machinery, 2020, ISBN: 9781450375313. DOI: `10.1145/3388333.3388659`. [Online]. Available: `https://doi.org/10.1145/3388333.3388659` (cit. on p. 27).

[53] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, “Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 873–886, 2018. DOI: `10.1109/TPDS.2017.2778161` (cit. on pp. 27, 50).

[54] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, “Loop tiling in large-scale stencil codes at run-time with ops”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 873–886, 2018. DOI: `10.1109/TPDS.2017.2778161` (cit. on p. 30).

[55] D. J. Lusher, S. P. Jammy, and N. D. Sandham, “OpenSBLI: Automated code-generation for heterogeneous computing architectures applied to compressible fluid dynamics on structured grids”, *Computer Physics Communications*, vol. 267, p. 108 063, 2021, ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2021.108063` (cit. on pp. 30–32, 50, 55).

[56] I. Ober and I. Ober, “On Patterns of Multi-domain Interaction for Scientific Software Development focused on Separation of Concerns”, *Procedia Computer Science*, vol. 108, pp. 2298–2302, 2017, ISSN: 1877-0509 (cit. on p. 30).

[57] G. Blaisdell, E. Spyropoulos, and J. Qin, "The effect of the formulation of nonlinear terms on aliasing errors in spectral methods", *Applied Numerical Mathematics*, vol. 21, no. 3, pp. 207–219, 1996, ISSN: 0168-9274. DOI: `https://doi.org/10.1016/0168-9274(96)00005-0` (cit. on pp. 31, 32).

[58] C. T. Jacobs, S. P. Jammy, and N. D. Sandham, "OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures", *Journal of Computational Science*, vol. 18, pp. 12–23, 2017 (cit. on p. 31).

[59] D. J. Lusher, S. P. Jammy, and N. D. Sandham, "Shock-wave/boundary-layer interactions in the automatic source-code generation framework opensbli", *Computers & Fluids*, vol. 173, pp. 17–21, 2018, ISSN: 0045-7930 (cit. on p. 31).

[60] D. J. Lusher and G. N. Coleman, "Numerical study of compressible wall-bounded turbulence – the effect of thermal wall conditions on the turbulent Prandtl number in the low-supersonic regime", *International Journal of Computational Fluid Dynamics*, vol. 36, no. 9, pp. 797–815, 2022 (cit. on p. 31).

[61] D. J. Lusher, A. Sansica, N. D. Sandham, J. Meng, B. Siklósi, and A. Hashimoto, "OpenSBLI v3.0: High-Fidelity Multi-Block Transonic Aerofoil CFD Simulations using Domain Specific Languages on GPUs", *Computer Physics Communications*, p. 109 406, 2024, ISSN: 0010-4655 (cit. on pp. 31, 53, 60, 62).

[62] D. J. Lusher, A. Sansica, and A. Hashimoto, "Effect of Tripping and Domain Width on Transonic Buffet on Periodic NASA-CRM Airfoils", *AIAA Journal*, pp. 1–20, 2024 (cit. on p. 31).

[63] D. J. Lusher, A. Sansica, M. Zauner, and A. Hashimoto, "High-fidelity study of three-dimensional turbulent transonic buffet on wide-span infinite wings", *arXiv*, p. 2401.14793, 2024. [Online]. Available: `https://arxiv.org/abs/2406.01232` (cit. on pp. 31, 62).

[64] S. Pirozzoli, "Numerical methods for high-speed flows", *Annual Review of Fluid Mechanics*, vol. 43, pp. 163–194, Jan. 2011. DOI: `10.1146/annurev-fluid-122109-160718` (cit. on p. 32).

[65] G. Coppola, F. Capuano, and L. de Luca, "Discrete Energy-Conservation Properties in the Numerical Simulation of the Navier–Stokes Equations", *Applied Mechanics Reviews*, vol. 71, no. 1, Mar. 2019, 010803, ISSN: 0003-6900 (cit. on p. 32).

[66] G. Coppola, F. Capuano, S. Pirozzoli, and L. de Luca, "Numerically stable formulations of convective terms for turbulent compressible flows", *Journal of Computational Physics*, vol. 382, pp. 86–104, 2019, ISSN: 0021-9991 (cit. on pp. 32, 61).

[67] W. J. Feiereisen, W. C. Reynolds, and J. H. Ferziger, "Numerical simulation of a compressible homogeneous, turbulent shear flow. ph.d. thesis", 1981. [Online]. Available: `https://ntrs.nasa.gov/citations/19820003523` (cit. on p. 32).

[68] W. M. Ahmad K., “Automatic testing of openacc applications”, *Chandrasekaran S., Juckeland G. (eds) Accelerator Programming Using Directives. WACCPD 2017. Lecture Notes in Computer Science*, vol. 10732, 2018. DOI: 10.1007/978-3-319-74896-2_8 (cit. on p. 33).

[69] M. Mascagni, “The white rat of numerical reproducibility”, *AIP Conference Proceedings*, vol. 2365, no. 1, p. 020 018, 2021. DOI: 10.1063/5.0057176. eprint: https://aip.scitation.org/doi/pdf/10.1063/5.0057176. [Online]. Available: https://aip.scitation.org/doi/abs/10.1063/5.0057176 (cit. on p. 33).

[70] L. Xu, X. Ren, Q. Wang, X. Xu, and X. Yang, “Full-neighbor-list based numerical reproducibility method for parallel molecular dynamics simulations”, *Parallel Computing*, vol. 85, pp. 109–118, 2019, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2019.04.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819119300754 (cit. on p. 33).

[71] A. P. Thompson, H. M. Aktulga, R. Berger, *et al.*, “LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales”, *Comp. Phys. Comm.*, vol. 271, p. 108 171, 2022. DOI: 10.1016/j.cpc.2021.108171 (cit. on p. 33).

[72] P. Langlois, R. Nheili, and C. Denis, “Numerical reproducibility: Feasibility issues”, ser. 2015 7th International Conference on New Technologies, Mobility and Security (NTMS), 2015, pp. 1–5. DOI: 10.1109/NTMS.2015.7266509 (cit. on p. 33).

[73] *Open TELEMAC-MASCARET. v.7.0, Release notes*, www.opentelemac.org, 2014 (cit. on p. 34).

[74] Y. He and C. H. Q. Ding, “Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications”, *The Journal of Supercomputing*, vol. 18, no. 3, pp. 259–277, Mar. 2001 (cit. on p. 34).

[75] Y. Hida, S. Li, and D. Bailey, “Library for double-double and quad-double arithmetic”, Jan. 2008 (cit. on p. 34).

[76] M. Taufer, O. Padron, P. Saponaro, and S. Patel, “Improving numerical reproducibility and stability in large-scale numerical simulations on gpus”, ser. 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010, pp. 1–9. DOI: 10.1109/IPDPS.2010.5470481 (cit. on p. 34).

[77] R. W. Robey, J. M. Robey, and R. Aulwes, “In search of numerical consistency in parallel programming”, *Parallel Comput.*, vol. 37, no. 4–5, pp. 217–229, Apr. 2011, ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.02.009. [Online]. Available: https://doi.org/10.1016/j.parco.2011.02.009 (cit. on p. 34).

[78] K. Ozawa and M. Miyazaki, “A summation algorithm with error correction for parallel computers”, *Systems and Computers in Japan*, vol. 24, no. 7, pp. 62–68, 1993. DOI: https://doi.org/10.1002/scj.4690240706. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/scj.4690240706. [Online]. Avail-

able: https://onlinelibrary.wiley.com/doi/abs/10.1002/scj.4690240706 (cit. on p. 34).

[79] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997, ISBN: 0201896834 (cit. on p. 34).

[80] S. F. Jalal Apostal, D. Apostal, and R. Marsh, "Improving numerical reproducibility of scientific software in parallel systems", ser. 2020 IEEE International Conference on Electro Information Technology (EIT), 2020, pp. 066–074. DOI: 10.1109/EIT48999.2020.9208338 (cit. on p. 34).

[81] R. A. Olsson, "Reproducible execution of sr programs", *Concurrency: Practice and Experience*, vol. 11, no. 9, pp. 479–507, 1999. DOI: https://doi.org/10.1002/(SICI)1096-9128(19990810)11:9<479::AID-CPE441>3.0.CO;2-S. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/\%28SICI\%291096-9128\%2819990810\%2911\%3A9\%3C479\%3A\%3AAID-CPE441\%3E3.0.CO\%3B2-S. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/%5C%28SICI%5C%291096-9128%5C%2819990810%5C%2911%5C%3A9%5C%3C479%5C%3A%5C%3AAID-CPE441%5C%3E3.0.CO%5C%3B2-S (cit. on p. 34).

[82] J. Zhang, Z. Dai, R. Li, L. Deng, J. Liu, and N. Zhou, "Acceleration of a production-level unstructured grid finite volume cfd code on gpu", *Applied Sciences*, vol. 13, no. 10, pp. 479–507, 2023, ISSN: 2076-3417. DOI: 10.3390/app13106193. [Online]. Available: https://www.mdpi.com/2076-3417/13/10/6193 (cit. on p. 34).

[83] M. Giles, D. Ghate, and M. Duta, "Using automatic difierentiation for adjoint cfd code development", *Computational Fluid Dynamics Journal*, vol. 16, Jan. 2008 (cit. on p. 35).

[84] A. Corrigan, F. Camelli, R.Löhner, and J. Wallin, "Running unstructured grid cfd solvers on modern graphics hardware", ser. 19th AIAA Computational Fluid Dynamics Conference AIAA 2009-4001, Jun. 2009 (cit. on p. 35).

[85] *Rodinia: Accelerating Compute-Intensive Applications with Accelerators*, https://rodinia.cs.virginia.edu/, accessed 2019 (cit. on p. 35).

[86] A. Owenson, S. Wright, R. Bunt, Y. Ho, M. Street, and S. Jarvis, "An Unstructured CFD Mini-Application for the Performance Prediction of a Production CFD Code", *Concurrency Computat: Pract Exper*, 2019. DOI: 10.1002/cpe.5443. [Online]. Available: https://doi.org/10.1002/cpe.5443 (cit. on p. 35).

[87] *MG-CFD-OP2 GitHub Repository*, https://github.com/warwick-hpsc/MG-CFD-app-OP2, accessed 2019 (cit. on p. 35).

[88] L. Lapworth, "Hydra-cfd: A framework for collaborative cfd development", in *International conference on scientific and engineering computation (IC-SEC)*, vol. 30, 2004 (cit. on p. 35).

[89] P. Moinier, J.-d. Muller, and M. Giles, “Edge-based multigrid and preconditioning for hybrid grids”, *AIAA Journal*, vol. 40, Apr. 2000. DOI: 10.2514/2.1556 (cit. on p. 35).

[90] M. B. Giles, M. C. Duta, J.-D. Muller, and N. A. Pierce, “Algorithm developments for discrete adjoint methods”, *AIAA Journal*, vol. 41, no. 2, pp. 198–205, 2003. DOI: 10.2514/2.1961. eprint: https://doi.org/10.2514/2.1961. [Online]. Available: https://doi.org/10.2514/2.1961 (cit. on p. 35).

[91] C. Bertolli, A. Betts, G. Mudalige, M. Giles, and P. Kelly, “Design and performance of the op2 library for unstructured mesh applications”, M. Alexander, P. D’Ambra, A. Belloum, *et al.*, Eds., ser. Euro-Par 2011: Parallel Processing Workshops, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 191–200, ISBN: 978-3-642-29737-3 (cit. on p. 37).

[92] M. Osama, M. Truong, C. Yang, A. Buluç, and J. Owens, “Graph coloring on the gpu”, ser. 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 231–240. DOI: 10.1109/IPDPSW.2019.00046 (cit. on p. 41).

[93] B. Jenkins, “Algorithm alley: Hash functions”, vol. 22, no. 9, pp. 107–109, 115–116, Sep. 1997, ISSN: 1044-789X (cit. on p. 41).

[94] D. J. Lusher and N. D. Sandham, “Assessment of Low-Dissipative Shock-Capturing Schemes for the Compressible Taylor–Green Vortex”, *AIAA Journal*, vol. 59, no. 2, pp. 533–545, Dec. 2020, ISSN: 0001-1452. DOI: 10.2514/1.J059672 (cit. on pp. 51, 54).

[95] P. Schlatter, M. Karp, R. Stanly, *et al.*, “Impact of low floating-point precision on high-fidelity simulations of turbulence”, in *77th Annual Meeting of the Division of Fluid Dynamics, American Physical Society*, Presented at 77th Annual Meeting of the Division of Fluid Dynamics, APS, Chair: Adrian Lozano-Duran, Caltech / MIT, Salt Lake City, Utah, Nov. 2024. [Online]. Available: https://meetings.aps.org/Meeting/DFD24/Session/R37.6 (cit. on p. 52).

[96] S. P. Jammy, C. T. Jacobs, and N. D. Sandham, “Performance evaluation of explicit finite difference algorithms with varying amounts of computational and memory intensity”, *Journal of Computational Science*, vol. 36, p. 100 565, 2019, ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2016.10.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750316302708 (cit. on p. 53).

[97] J. DeBonis, “Solutions of the Taylor-Green Vortex Problem Using High-Resolution Explicit Finite Difference Methods”, in *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, ser. Aerospace Sciences Meetings, American Institute of Aeronautics and Astronautics, Jan. 2013 (cit. on p. 54).

[98] J.-B. Chapelier, D. J. Lusher, W. Van Noordt, *et al.*, "Comparison of high-order numerical methodologies for the simulation of the supersonic Taylor–Green vortex flow", *Physics of Fluids*, vol. 36, no. 5, p. 055 146, May 2024, ISSN: 1070-6631. DOI: `10.1063/5.0206359` (cit. on pp. 54, 55).

[99] Y. Kuya, K. Totani, and S. Kawai, "Kinetic energy and entropy preserving schemes for compressible flows by split convective forms", *Journal of Computational Physics*, vol. 375, pp. 823–853, 2018, ISSN: 0021-9991. DOI: `https://doi.org/10.1016/j.jcp.2018.08.058` (cit. on pp. 60, 61).