Paris-Saclay University

End of Studies Internship Report
Master of High Performance Computing and Simulation

Numerical Optimization Targeting Sustainable Scientific Computing

A Case Study on LULESH and Reactor Simulator Benchmarks Using Verificarlo

Supervisors:
Roman IAKYMCHUK
Pablo DE OLIVEIRA CASTRO

Author:
Gülçin GEDİK

September 4th, 2024

Contents

List of Figures

List of Tables

Abstract

The computational energy consumption required to accurately model exascale problems is extraordinary. As simulations become more realistic, the demand for higher accuracy increases, leading to prohibitively long computation times and high power draw. Moreover, as hardware architecture components reach the physical limits of their efficiency, there has been a growing interest in designing energy efficient scientific applications. Thus, this internship aims to investigate the potential of mixed-precision computing as a strategy to target these needs with the help of computer arithmetic tools.

Through case studies on the Reactor Simulator and LULESH benchmarks, we demonstrated that applying mixed-precision strategies can lead to significant reductions in both time-to-solution and energy-to-solution, sometimes without sacrificing accuracy. Our findings show a 15% reduction in both metrics without any compromise on accuracy, for the Reactor Simulator and up to a 20% improvement in time-to-solution and a 10% reduction in energy-to-solution for the LULESH benchmark. These results highlight the importance of adopting 'just enough precision' to optimize performance and energy use in scientific computing.

Abstract

La consommation d'énergie informatique nécessaire pour modéliser avec précision les problèmes à l'échelle exascale est extraordinaire. À mesure que les simulations deviennent plus réalistes, la demande de précision accrue augmente, entraînant des temps de calcul prohibitivement longs et une consommation d'énergie élevée. De plus, alors que les composants de l'architecture matérielle atteignent les limites physiques de leur efficacité, il y a un intérêt croissant pour la conception d'applications scientifiques à faible consommation d'énergie. Ainsi, ce stage vise à explorer le potentiel du calcul en précision mixte comme stratégie pour répondre à ces besoins.

À travers des études de cas sur le simulateur de réacteur et les benchmarks LULESH, nous avons démontré que l'application de stratégies de précision mixte peut entraîner des réductions significatives à la fois du temps de calcul et de la consommation d'énergie, parfois sans sacrifier la précision. Nos résultats montrent une réduction de 15% de ces deux métriques, sans aucune perte de précision, pour le simulateur de réacteur, et jusqu'à 20% d'amélioration du temps de calcul et 10% de réduction de la consommation d'énergie pour le benchmark LULESH. Ces résultats soulignent l'importance d'adopter une "précision juste suffisante" pour optimiser les performances et l'utilisation de l'énergie dans le calcul scientifique.

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Roman Iakymchuk, who provided unwavering support throughout this internship. For navigating the challenges of my journey to Umeå, and offering invaluable advice on staying focused, his patience and feedback were truly invaluable. I would also like to extend my gratitude to my co-supervisor, Pablo de Oliveira Castro, for always welcoming my questions and generously sharing his knowledge and expertise.

I am also grateful to the friends I made during this journey, who lightened the burden of being a foreigner and greeted me with smiles. A special thanks to Saloni and Divya in Umeå, and Sebastien and Sanja in Uppsala. I am thankful to all the staff members at Umeå University and Uppsala University. My thanks also go to Yianxiang for all the joyful fika breaks we shared and for his immense help when I felt stuck.

A special thanks to Dilan, Dorian, and Luna, who took care of me during the writing process, offering endless coffee and emotional support. Thanks to all my other friends who supported me throughout this journey, providing encouragement, friendship, and understanding.

Lastly, I want to express my heartfelt gratitude to my family, who always listened to me and believed in me. I will do my best to pay it all forward.

I would like to acknowledge financial support provided by the Center of Excellence in Exascale Computational fluid dynamics (CEEC) grant No 101093393 from European Union via EuroHPC Joint Undertaking and Sweden, Germany, Spain, Greece, and Denmark.

Umeå University

Founded in 1965, Umeå University is Sweden's fifth oldest university and is recognized for its strong international and multicultural environment. The university strives to establish itself as one of Scandinavia's leading centers for education and research, addressing the challenges of an increasingly globalized society. Umeå University is among Sweden's most comprehensive institutions, conducting extensive research across various scientific fields to maintain a competitive edge internationally. The Department of Computing Science at Umeå University has over 150 employees from more than 20 different countries and engages in diverse research areas, including high-performance computing, numerical methods, social and ethical AI, robotics, and distributed systems. The department is actively involved in six Horizon Europe projects, contributing significantly to the field of exascale algorithms, particularly in scalable methods and mixed-precision algorithms. Leveraging its expertise in numerical methods and accuracy-ensuring strategies, especially in computational fluid dynamics (CFD), Umeå University leads advancements in this domain.

HPC2N

The High Performance Computing Center North (HPC2N) at Umeå University is a national hub for scientific and parallel computing and a part of the National Academic Infrastructure for Supercomputing in Sweden (NAISS). HPC2N offers advanced computational resources and support for research across various disciplines, including physics, chemistry, and life sciences. The center is equipped with the Kebnekaise system, a high-performance heterogeneous cluster designed to manage both compute-intensive and data-intensive applications, providing vital infrastructure for scientific discovery and innovation.

CEEC

The EuroHPC JU Center of Excellence in Exascale CFD (CEEC), funded by the European Commission, brings together partner institutions from Sweden, Denmark, Germany, Greece, and Spain, integrating a wide range of expertise in mathematics, physics, and software engineering. Specializing in numerical simulations for turbulence, CEEC addresses the complex and computationally demanding task of modeling turbulent air currents — a phenomenon familiar to anyone who has experienced turbulence during flight. While current methods exist for predicting these unpredictable air flows, they require significant computational resources. Exascale computing presents an opportunity to perform these simulations with unprecedented realism, yet existing CFD codes are not yet fully optimized for such powerful systems. Moreover, even with exascale capabilities, running realistic flow simulations without optimizing granularity and accuracy would be infeasible due to excessively long computation times. CEEC aims to prepare key computational fluid dynamics applications for exascale computing, showcasing their potential through flagship case studies that highlight the advanced capabilities of these next-generation supercomputers.

1 Introduction

As hardware architecture components approach the limits imposed by physics, the responsibility has shifted to the software community to develop new, energy efficient, high-performance numerical methods. Power efficient chips are only as efficient as the software those chips run on. Therefore, in recent years, there has been a rising awareness in designing energy efficient scientific applications among high performance computing (HPC) developers[21][7][25].

Achieving favorable results in energy efficiency requires a comprehensive understanding of hardware architecture, mathematics behind the simulations and software engineering choices while also simultaneously prioritizing the accuracy of the software. In that sense, we understand sustainable scientific computing as a trade-off between time-to-solution, energy-to-solution, and accuracy. There exists two main approaches in order to achieve better energy-to-solution metrics. First one is through leveraging the hardware capabilities, for example, the Green500 [26] list highlights the most energy efficient supercomputers, demonstrating how advancements in hardware design can lead to significant improvements in energy efficiency. The second approach focuses on finding the right algorithmic strategies, which involves selecting and designing algorithms that minimize computational complexity and energy consumption while delivering accurate results without making a compromise on time-to-solution. With this regard, mixed-precision algorithms/ computing offers great potential considering its direct impact on lowering data movement, increased vectorized computation speed and smaller register size in use comparing to the full (double) precision computing. From this perspective, we believe that every digit comes with a cost, and we advocate for computing with just the necessary level of accuracy rather than striving for the highest possible accuracy. However, it is not always trivial to pinpoint when and how to introduce low precision computation without causing significant loss in accuracy or drop in performance. Therefore, the challenge relies on identifying the routines that can be computed in reduced precision.

The EuroHPC JU Center of Excellence in Exascale CFD (CEEC) specializes in numerical simulations for turbulence, aiming to prepare Computational Fluid Dynamics (CFD) applications for the era of exascale computing. As simulations become more realistic, the demand for higher accuracy increases, which could result in prohibitively long computation times if not managed properly [17]. At exascale levels, the challenge lies in balancing granularity and accuracy to ensure practical run times. Hence, leveraging mixed-precision computations becomes essential, as it offers significant opportunities to optimize performance and efficiency while addressing the increasing demands of realistic simulations.

All these design decisions aimed at reducing both time-to-solution and energy-to-solution contribute to the creation of an optimization space. The key metrics within this optimization space include hardware considerations, algorithmic choices, the selection of instances where mixed-precision can be applied.

During this internship, we explored this optimization space through the lens of mixed-precision implementation concerns in scientific applications, while keeping other parameters constant. We particularly focused our research on answering the following questions :

1. How can mixed-precision computing strategies be effectively implemented to balance energy efficiency and computational accuracy in high-performance scientific applications?

2. What methodologies can be developed to identify routines within scientific applications that can benefit from reduced precision without significant loss of accuracy or performance?

3. How can performance profiling and computer arithmetic tools be used to identify scenarios where 'just enough precision' can be applied to reduce energy consumption while resolving problems related to floating-point errors and maintaining or improving computational performance in scientific applications?

We aim to demonstrate that using *just enough amount of precision* can reduce the energy footprint and enhance the performance in scientific applications, sometimes even without settling for lower accuracy.

This report is structured to provide a comprehensive view of our investigation into energy efficiency and mixed-precision computing in the context of scientific computing. Chapter 2 discusses floating-point operations and introduces Verificarlo as computer arithmetic tool of choice, along with a review of the current state-of-the-art. Chapter 3 reflects on our efforts to contribute to the scientific computing community by presenting insights into energy measurement on European HPC systems, highlighting the challenges, appealing to facilitate such measurements, and showcasing few successful stories from CEEC. Chapter 4 introduces the methodology employed in our study, providing its high-level overview. Chapters 5 and 6 explore two distinct case studies, respectively Reactor Simulator and LULESH Benchmarks, providing a detailed analysis of the mixed-precision strategies implemented for each application. Finally, Chapter 7 concludes the thesis with our contributions and findings from the case studies.

2 Background

2.1 Floating Point Arithmetic

Many highly accurate scientific applications rely on computation in double precision, which is natively supported by most modern hardware. In some instances, even higher precision types are employed to ensure a reliable level of accuracy, though this often results in increased execution time. Floating-point arithmetic, which defines these precision types, allows for the representation of a vast range of numbers with a finite number of bits on the computer. However, this approach can introduce various types of errors due to its inexact nature. These include representation errors, roundoff errors that can accumulate over time, and catastrophic cancellation, which occurs when subtracting two nearly equal small numbers, leading to significant error accumulation during subsequent computations. Even though being more complex to manipulate compared to fixed-point representations, floating-point arithmetic is well-suited for representing a wide range of values. Modern hardware adopts this approach, and its implementation is standardized by the IEEE-754 standard, keeping consistency across different machines performing identical operations. Inspired by the usual scientific notation, they represent a number as a mantissa multiplied by a scaling factor [5].

More precisely, while the choice of base $\beta = 10$ is arbitrary in the real representation of numbers with an infinite number of digits, the definition holds for any integer $\beta > 1$. In this case, each digit d_n belongs to the set $0, 1, \ldots, \beta - 1$. Applying this concept to a finite number of bits, a floating-point number can be represented as $\pm d.dd\ldots d \times \beta^e$, where $d.dd\ldots d$ is known as the significand (or mantissa) and consists of p digits, representing the working precision. The exponent e is an integer within the range $e_{min} < e < e_{max}$ [8].

Definition 2.1.1 ***Floating Point Representation***

$$x \approx \pm\left(\sum_{n=1}^{p} d_n \times \beta^{-n}\right) \times \beta^e \qquad (2.1)$$

Therefore a system of floating point numbers F form a subset of the real numbers and is characterized by $F = F(\beta, p, e_{min}, e_{max})$ [11].

Muller et al. [22] offer an alternative definition that is more concise in its mathematical form and can be more practical in certain contexts. In this version, the significand is treated as an integer M, and the real number it represents is given by $M \cdot \beta^{e-p+1}$. Essentially, this definition is equivalent to the previous one, with the key difference being that the radix point is effectively shifted to the rightmost position.

Floating point operations are performed as if the result was first computed with an infinite precision and then rounded to the floating point format. The classical floating point error model [11] links the error committed as a result of elementary operations with the real computations :

$$fl(a \operatorname{op} b) = (a \operatorname{op} b)(1+\delta), \quad |\delta| \le u, \quad \operatorname{op} \in \{+,-,\times,/,\surd\}$$

where the relative error is defined as $\delta = \frac{\operatorname{fl}(a \operatorname{op} b)-(a \operatorname{op} b)}{(a \operatorname{op} b)}$ and the unit round-off u is given by $u = \frac{1}{2}\beta^{1-p}$, which represents the unit round off, i.e machine epsilon/ sensitivity. As there are more than one way to round a number, in order to achieve a deterministic and well-defined answer across the various architectures, the IEEE standard requires the result of the operation to be correctly rounded to the nearest value of the real computation, keeping the δ minimal.

While having these strict error bounds provides a deterministic framework for error analysis, they often overestimate the potential errors resulting from precision reduction. This overestimation arises because these methods are based on worst-case scenarios, which can become unrealistic as the input size grows. Consequently, they may offer limited practical insight into error behavior in large-scale computations.

To address this limitation, probabilistic methods have been developed to provide more nuanced estimates of errors introduced by low precision computations. These methods typically model rounding errors as independent random variables, allowing for the estimation of both upper and lower limits of error.

Higham et al. [10] make use of Hoeffding's inequality to derive probabilistic error bounds for numerical computations, which applies to any number of inputs n. They base their analysis on two assumptions, the rounding erros have mean zero and are independent. Given these assumptions, and using concentration inequalities, they highlight three important features: (1) backward error bounds, which can be used to estimate forward error by multiplying with the application's condition number; (2) these bounds are exact; and (3) they are valid for any n.

Figure 2.1: Figure describing the relation between the backward and the forward error.

$$\begin{tikzcd}
x \arrow[r] \arrow[d, "Rep(x)"'] & y = f(x) \arrow[d] \\
x+\delta x \arrow[r, "App(x)"'] & \hat{y} = f(x+\delta x)
\end{tikzcd}$$

δx : backward error $\qquad \delta y = y - \hat{y}$: forward error

2.2 Verificarlo

Verificarlo is the primary tool for computer arithmetic manipulation used throughout this project. It operates at the compiler's optimized Intermediate Representation (IR) level, therefore no need to make a change in the source code, replacing each floating-point operation with a custom call. This approach allows it to capture all front-end and middle-end optimization passes. Through these instrumentations, Verificarlo enables users to explore the error sensitivity of specific regions within an application, additionally helping to identify parts of the code that can effectively use reduced precision formats. This capability not only helps to numerically debug the code but also prevents the additional cost associated with porting through virtual precision instrumentation. Verificarlo has a diverse set of backends but in our study we focused on VPREC and MCA backends.

The Variable Precision (VPREC) backend [2] can emulate any precision format that fits within the binary64 format. It supports a range of exponent values, $e \in [1,11]$, and virtual precision, $p \in [3,52]$. During each instrumented floating-point operation, VPREC rounds the operands to the specified virtual precision and exponent, converts them to `binary64`, and then performs the operation in double precision. The result is subsequently rounded to the defined precision and exponent, and stored in the standard `binary64` double representation.

Monte Carlo Arithmetic (MCA), introduced by Stott Parker [23], is a method that builds on standard floating-point arithmetic by adding randomness to arithmetic operations. In normal floating-point arithmetic, rounding errors are predictable. However, in MCA, these errors are treated as random variables, meaning the arithmetic operations and their numbers can be randomized. This causes the same calculation to give different results if they are run multiple times. For example, instead of following a fixed rule for adding two numbers, MCA allows some variation in the result due to randomness.

Rather than relying on assumptions about error independence or mean, as in probabilistic error analysis methods, MCA uses randomization to simulate various perturbations in computations, making it possible to study the behaviour of floating point errors in an empirical way. MCA uses random rounding to spread roundoff errors in a truly random way, without any predictable pattern or bias. When a program is run multiple times with the same inputs under MCA, it produces a range of results that can be analyzed statistically. This analysis helps estimate the true answer, evaluate how sensitive the output is to small changes, and understand how errors build up.

Verificarlo's MCA backend [6] introduces randomness to standard floating-point operations by replacing each floating-point call with a custom call that incorporates a noise function, simulating the loss of accuracy through perturbed computation. The backend offers three rounding modes. The Random Rounding (RR) mode focuses on capturing the impact of rounding errors by applying perturbations only to the output of the operation. The Full-MCA mode extends this by introducing perturbations to both the operands and the result, providing a comprehensive view of the effects of rounding throughout the computation. Lastly, the Precision Bounding (PB) mode applies perturbations solely to the inputs, which helps in analyzing issues such as catastrophic cancellations.

2.3 Related Work

Chatelain et al. [2] explores empirical applications of mixed precision by introducing a piecewise search heuristic to identify optimal code regions for precision reduction. They demonstrate this approach with a program that calculates the inverse of π using the iterative Newton–Raphson method. In the initial iterations, the majority of digits are inaccurate, suggesting that lower precision is adequate. As the iterations progress, a suitable reduced precision is chosen for each iteration to maintain the desired convergence.

Several mixed precision algorithms have been developed to leverage the high performance of lower precision formats while maintaining the accuracy benefits of higher precision. The surveys from [11] and [1] on mixed precision algorithms in numerical linear algebra covers a wide range of both direct and iterative methods for problems such as matrix multiplication, matrix factorization, solving linear systems, least squares problems, eigenvalue decomposition, and singular value decomposition.

With all these accuracy concerns and performance tradeoffs in mind, determining the most effective approach to the problem can be challenging. Chen et al. [3] propose a methodology for implementing mixed-precision arithmetic using computer arithmetic tools such as Verificarlo and the Roofline Model. Their methodology focuses on identifying candidate code sections and significant variables to guide the adaptation process. Details of their approach are discussed in Chapter 4. Additionally, they provide a case study involving Nekbone, a mini-application for the CFD solver Nek5000. Their study highlights both the challenges and benefits of precision reduction. Specifically, they demonstrate that applying mixed-precision to Nekbone can reduce the time-to-solution by 40.7% and the energy-to-solution by 47% across 128 MPI ranks.

Williams et al. [28] introduced the Roofline Model, which visualizes application performance by comparing its computational intensity with system peak performance and memory bandwidth, helping to identify whether performance issues stem from compute or memory bounds. However, it does not account for complex cache hierarchies and assumes a single memory bandwidth level. Ilic et al. [14] enhanced this model with the Cache-Aware Roofline Model, adding boundaries for different cache levels (L1, L2, L3) to provide more detailed insights into cache and memory usage, and optimization opportunities. For our study, we created roofline models using Intel Advisor [16], which utilizes the Cache-Aware Roofline Model [15]. The models were generated with version 2023.2.0 of Intel Advisor, installed on the Kebnekaise system, running on an Intel Xeon E5-2690 processor (single core).

3 Measuring Energy on EuroHPC

During our efforts to produce energy efficient scientific codes, we contributed to the CEEC Best Practice Guide on Harvesting Energy Consumption on European HPC Systems [13]. CEEC aims to reduce the energy footprint of its consortium codes using novel algorithmic solutions. Measuring energy, unlike performance, is more complex and difficult than measuring time-to-solution with basic tools. Additionally, energy measurement on European HPC systems is challenging due to the lack of a universal tool and the need for special privileges on many systems. Optimizing for energy consumption can be counterintuitive; for some programs, better performance means higher energy use, while for others, slower computation or lower precision can save energy. The main goal is to achieve energy efficiency (sustainability) while applying the 'lagom' principle, a Swedish ethos of balance or 'just enough', particularly regarding precision in scientific computations. However, integrating energy-to-solution as a standard HPC metric, alongside performance, scalability, and resource efficiency, is still in its infancy.

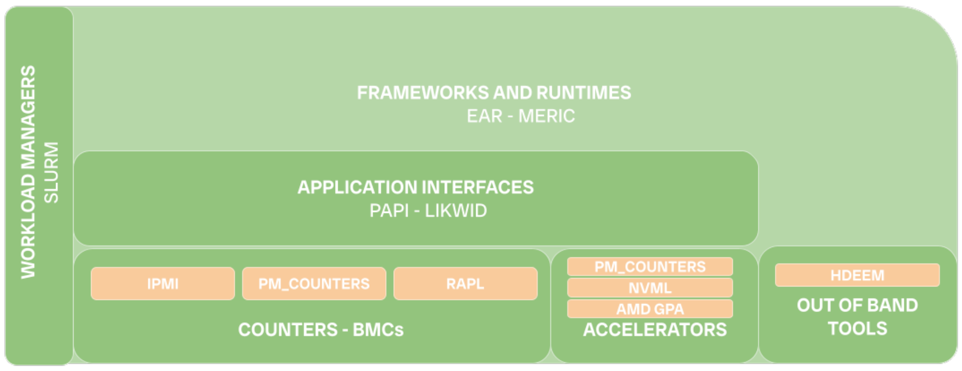


Figure 3.1: Hierarchical relation between energy measurement methods.

Therefore with this Best Practice Guide, we have provided a best practices guide to measure energy, and highlighted its importance. We share experiences from the CEEC project involving different codes and systems, along with an extensive collection of external references for researchers to take advantage.

We started by describing the underlying structure of each measurement method, the hardware performance counters used to estimate or, in some cases, directly measure energy consumption. There is a variety of them and each one offers different type of opportunities and challenges. We then proceeded to the tools and frameworks built upon these counters. Figure 3.1 shows the complexity of the software stack and the relationships between the tools explored in our document. Then, we explored different metrics in use in research articles and our internal deliverables, such as Energy Normalized Performance Index, Energy Delay Product and of course FLOPs per Watt. And lastly, we finalized with the achievements from the CEEC projects, such as the 47 % reduction we obtained with Nekbone mentioned in Section 2.3, and 56 % saving achieved with a GPU based high order accurate flow solver based on the discontinuous Galerkin (DG) spectral element method comparing to its CPU counterpart.

As seen from the examples, the members of the CEEC project have already started to take advantage of this diversity of tools. For this reason, we examined available tools with respect to their ease of use, coarse and fine granularity over temporal and spatial aspects of the application. We observed that SLURM is the go-to way for starters who want to measure how much energy their application consumes over the job's runtime with minimal overhead. It doesn't require any root privileges and it handles the underlying measurement infrastructure itself, leaving nothing to users but just the exclusive allocation of the resources for the application and reduced system noise in order to be able to retrieve accurate results. During our research, we found that the field lacks a convenient method for presenting metrics, making it harder for readers to grasp implementation details. We recommend that researchers provide more detailed descriptions of their measurement techniques, including the scope (e.g., node, core, cooling equipment) and the intervals between measurements. Given the variety of tools and their interdependence, as shown in Figure 3.1, researchers should also specify the tools they used and their underlying methodologies. Providing such information is essential for ensuring research verifiability and reproducibility.

One of the main takeaways we present on this document that the community/ data centers need to facilitate the access to energy measurements on HPC systems. As a result of this challenge, we had to explore what mixed-precision computation offers in terms of energy-to-solution on our Laptop, instead of Kebnekaise at HPC2N [1]. This system offers us energy measurements via Intel RAPL (Running Average Power Limit), which is also detailed in our best practice guide, and it offers multiple domains, such as Cores, Package, System and Integrated GPU as shown in Figure 3.2.

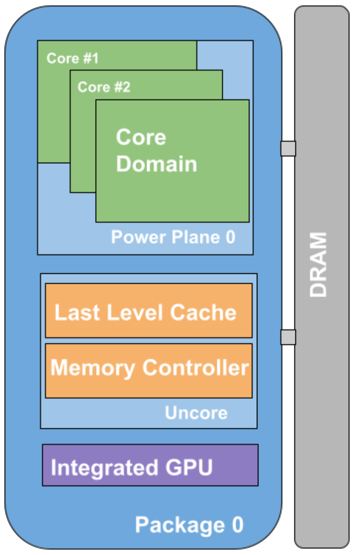


Figure 3.2: RAPL Power Domains

RAPL operates on the socket level and exposes the measured metrics with the help of cumulative MSR registers that has a wrapround time. It provides hierarchical control over different power domains, and it can also function as a power measurement tool, as suggested by Hackenberg et al. [9], transitioning from modeling to actual measurements.

During our research into the impact of precision reduction on energy-to-solution, we used a custom tool to measure energy counters. This tool recorded energy data every 5 milliseconds by reading them with the help of `perf_event_open()` system call to read the RAPL registers, calculating the difference between consecutive readings to determine energy consumption for each interval. We then summed these values to obtain the total energy used. While the time interval could be adjusted to cover the entire application runtime, doing so would complicate detecting and correcting the register wraparound. Therefore, selecting an appropriate interval is necessary to guarantee measurement accuracy.

[1]Detailed description of the system can be found in the Appendix.

4 Methodology

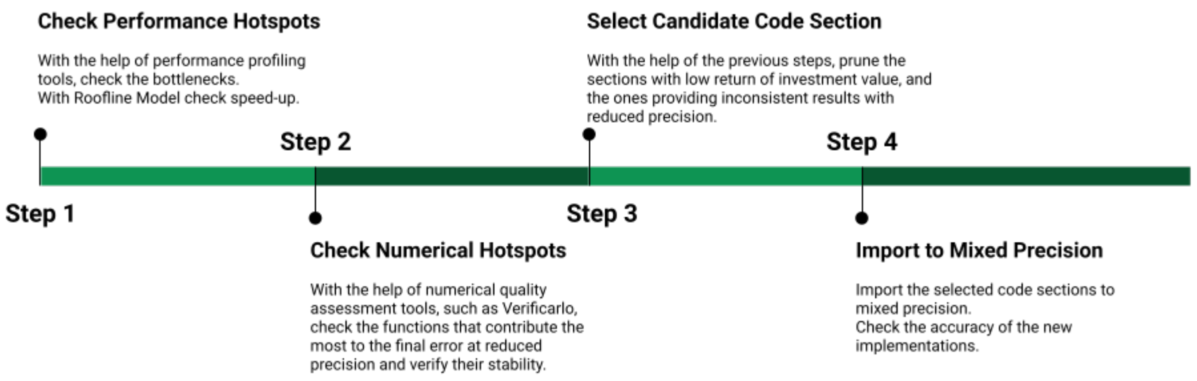

Check Performance Hotspots
With the help of performance profiling tools, check the bottlenecks.
With Roofline Model check speed-up.
Step 1
Step 2
Check Numerical Hotspots
With the help of numerical quality assessment tools, such as Verificarlo, check the functions that contribute the most to the final error at reduced precision and verify their stability.
Select Candidate Code Section
With the help of the previous steps, prune the sections with low return of investment value, and the ones providing inconsistent results with reduced precision.
Step 3
Step 4
Import to Mixed Precision
Import the selected code sections to mixed precision.
Check the accuracy of the new implementations.

Figure 4.1: Methodology for enabling sustainable computing in applications.

Adapting an application to use mixed-precision arithmetic is a complex process that involves a thorough understanding of the application's structure, including its data structures, mathematical operations and their related libraries, and communication library calls. To streamline this process and outline the steps necessary for achieving a mixed-precision application, we have adopted the methodology proposed by Chen et al. [3] outlined in Figure 4.1.

First, we assess whether the application benefits from precision reduction by using profiling tools to identify performance hotspots and mapping these onto the Roofline Model, explained in Section 2.3. If hotspots show potential for improvement with single-precision computations, they are considered as a candidate for mixed-precision optimization.

Subsequently, we focus on determining precision requirements and numerical hotspots by examining key variables that impact or are affected by numerical behavior. These variables, ideally present in each cycle of the application's run-time, contribute to floating-point error accumulation. Tracking them allows us to measure the effects of these accumulated errors.

Using Verificarlo's VPREC backend, we identify the minimal virtual precision needed for each function to maintain accuracy and pinpoint routines that can be executed with reduced precision without exceeding acceptable accuracy limits. This is done by comparing VPREC results with the reference implementation in original (double) precision or by following the application's convergence and stoping criteria.

We also use Verificarlo's MCA backend to identify numerically unstable routines and to test numerical sensitivity. This helps us assess how randomization affects numerical stability and locate areas where precision adjustments can enhance performance while ensuring reliability. At each step of this examination process, we systematically eliminate code sections that are unlikely to benefit from precision reduction. Finally, we implement the mixed-precision code, continuously monitoring both accuracy and the application's time-to-solution, while also measuring energy-to-solution.

5 Reactor Simulator Benchmark

The Reactor Simulator [18] is a probabilistic system developed using Monte Carlo simulation to model the interactions between particles emitted from a neutron source and a slab. The application performs a series of iterative processes, incorporating conditional logic to manage particle states.

This chapter begins with an overview of the benchmark, followed by an examination of the code using two various analysis tools. Lastly, we discuss the implementation details of mixed-precision techniques and present the results obtained.

5.1 Description of Benchmark

The rector is found at the center of our logical three dimensional space. At each iteration the reactor emits a particle with random initial direction and energy selected from $[E_{min}, E_{max}]$ with a $\frac{1}{\sqrt{E}}$ distribution with the help of `source()` function. The shield has a thickness in the x direction previously specified in the program, and it extends to infinity in the y and z directions.

Then, the distance that the particle can travel through the slab before colliding, based on its current energy, is computed with the help of `dist2c()` function given a particle's energy, and assuming that it is inside the shield. The collision distance is computed by estimating a cross section then randomly selecting a distance that is logarithmically distributed. Therefore this part of the computation heavily relies on mathematical library functions such as `log(), exp()`. `Update()` function is responsible for updating the particle's position based on the particle's direction using `sqrt(), sin(), cos()` functions from the shared mathematical library.

In this simulation, particles are emitted sequentially, with each particle being launched only after the previous one has reached its final state, thereby neglecting any interactions between particles. A particle can exist in one of four possible states, three of which result in the termination of its trajectory: reflection by the slab, absorption by the slab, or transmission through the slab. These outcomes mark the end of the particle's journey. Additionally, a particle may enter a fourth state where it is scattered by the slab. In this state, the particle is redirected in a new random direction and with reduced energy, due to the energy loss incurred during the collision with the slab.

The interaction between the slab and the particles that are emitted at each iteration is described by the local variables that are generated once and subsequently updated by each function and overwritten at the launch of each particle. The main loop tracks the number of particles absorbed, reflected, or transmitted through the slab using integer variables. Additionally, it logs the energy associated with these particles by summing the results of each particle in double precision. Since each particle's fate is determined within the same iteration in which it is emitted, an explicit log of individual particles is unnecessary.

5.2 Inspection and Implementation

5.2.1 Initial Code Inspection With Tools

We start by analyzing the structure of our benchmark, identifying the key variables to evaluate accuracy. Next, we use Callgrind to assess bottlenecks. Lastly, we observe how the simulation performs with varying numbers of elements and precision levels, using the Verificarlo VPREC backend. Even though the simulation is stochastic by nature, in order to observe the effects of FP errors, we are fixing the seed and making the application deterministic.

In this simulation there is only one variable that can be considered as a significant variable. However, relying on just this one variable provides limited insight. Additionally, the simulation lacks a convergence criterion, which might otherwise offer a different approach for assessing accuracy.

As we mentioned in Section 5.1, this simulation puts an extensive usage of shared library functions. Most time consuming functions are coming from the shared mathematics library such as `sin_fma()` and `exp_fma()`, consuming respectively 28.78 % and 17.50 % of execution time according to Callgrind [1] when compiled with `g++ -O2`. The `r8_uniform_01()` function which exceptionally takes up 18.12% of the execution time but it is a mainly integer arithmetic function with only one casting to double at the end. Therefore, this function cannot be instrumented with Verificarlo. Hence, when necessary in the instrumentation phase, it is hard coded to into the application. Additionally, when we check the source code we notice that the Fused Multiply Add (FMA) implementations are not called directly from the code, although they are

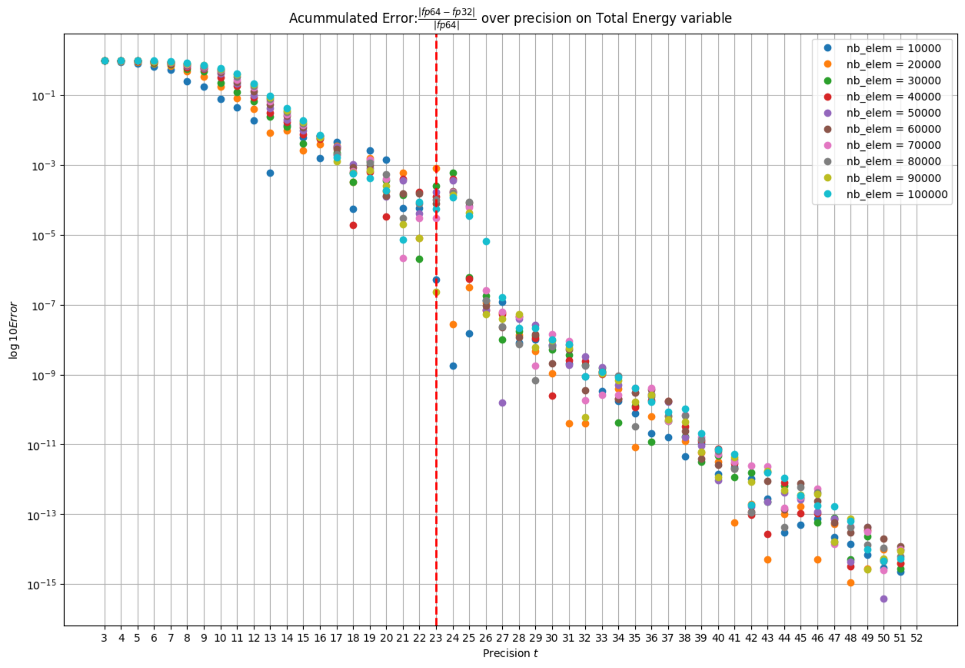


Figure 5.1: Evolution of Forward Error on Total Energy variable with different number of elements *nb_elem* and with different precision *Precisions* settings.

[1]Note, this is not exactly time but number of instructions.

the courtesy of the library being compiled to take the best advantage of the machine they are working on. Therefore, the library is making a call to this optimized version of `sin()` with FMA.

In this simulation, increasing number of elements is proportional to increasing number of cycles since each cycle launches a particle and the tracked variable `Total Energy` is accumulated through these cycles. Thus, increasing number of elements naturally increases the error accumulated. Figure 5.1 shows evaluation of the error through number of elements and increasing precision with the help of Verificarlo VPREC instrumentation backend. Unlike the observation made by Chen et al. [3] with Nekbone mini-app, here we do not observe any smoothness around an arbitrary precision value that could guide us to applying single precision computations directly. Instead, we observe an almost linear relationship between precision and accuracy; as the number of mantissa bits increases, the accuracy of the results improves.

5.2.2 How We Conducted Measurements

The timing measurements reported in this section were conducted on Kebnekaise compute nodes, which is described in the Appendix A.1, unless stated otherwise. On Kebnekaise, we compiled the benchmark with `g++` version 10.2, and on Laptop we compiled with 11.4.0, with `-O2` optimization flag on both of them. To obtain statistically significant results, the application was executed 32 times on both machines with performance governor. Energy measurements, which covers the entire duration of the application run, were collected also 32 times. They are obtained on our Laptop by measuring the counters every 5 ms with the help of a custom application detailed in Chapter 4. We selected the cores domain as the basis for our measurements because the specific benchmark does not involve any execution on the integrated GPU. Additionally, the system domain is relatively new, undocumented, and lacks sufficient validation, while the package domain was not used because our code does not generate memory traffic with DRAM.

For timing, we used the GNU `time` command, which is suitable for applications that run more than a second, to record only user and system time, which reflects the CPU time dedicated to executing the specific process. Additionally, to ensure a sufficiently extended measurement period, we used an input size of 10,000,000 elements on Kebnekaise and on our Laptop, and we used 10,000 elements to measure accuracy on both machines.

To check the accuracy of the results, we compile our code with Verificarlo, using optimization level `-O2` to prevent any potential changes due to compiler optimizations. We use Verificarlo version 1.0.0 [27] with the VPREC backend set to its default parameters, which maintains the default precision and range settings. For experiments where precision is adjusted, we use the option `-precision-binary64=wanted_precision` to instrument double precision at the wanted precision level.

5.2.3 Detailed Code Inspection and Mixed-Precision Implementations

In Section 5.2.1, we detected that the shared library functions had the most time inputs in this application. Therefore, we proceeded with the help of Verificarlo VPREC back-end to instrument these trigonometric functions at different precision settings.

Using the VPREC Backend, we were able to specify both the parent and child functions where we wanted to reduce precision. Initially, we compiled the code with Verificarlo, using the function instrumentation flag `-inst-func`. Then, by setting the backend runtime parameter `-prec-output-file`, we generated a precision profiling output file with Verificarlo. This output file provided detailed information, including the parent functions of the targeted functions, their respective line numbers, the precision of input arguments along with their respective ranges, and the frequency of function calls at each call site. For each function we wanted to instrument, we modified this precision profiling output file by adjusting the input and output precision parameters accordingly. Finally, we used the modified file as a runtime parameter to apply the desired instrumentation to each specific function.

We noticed that the `cross()` function contains two calls to `sin()`. Therefore, we opted to focus our first time analysis to lower the precision of `sin()` whose parent is `cross()`.

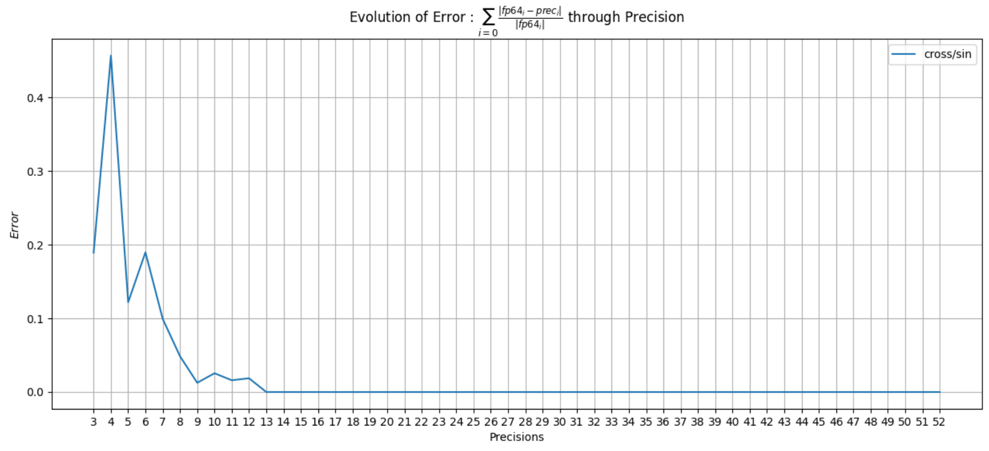


Figure 5.2: Accumulated Forward Error at every 500 iterations for a different number of bits in mantissa for `sin()` in `cross()`.

Figure 5.2 points out that, when only `sin()` is instrumented the plot becomes smooth and reaches a stable state after 13 bits of precision in the mantissa. We also examined the part contributed by the `update()` function separately and noticed that there is no change in the forward error when we reduce the precision of `sin()` by this function. Therefore, `sin()` in both `cross()` and `update()` can be reduced to FP32.

The `exp()` function is only called by `cross()`. We followed the methodology as previously with `sin()`. Figure 5.3 shows that there is no accuracy loss after 18 bits of precision. Lastly, we have experimented with `update()/cos()` and `update()/sqrt()` functions. Experiments with different precisions yielded no change in results, meaning the precision of these functions does not have an impact in the end result. Therefore, both can directly be computed in FP32.

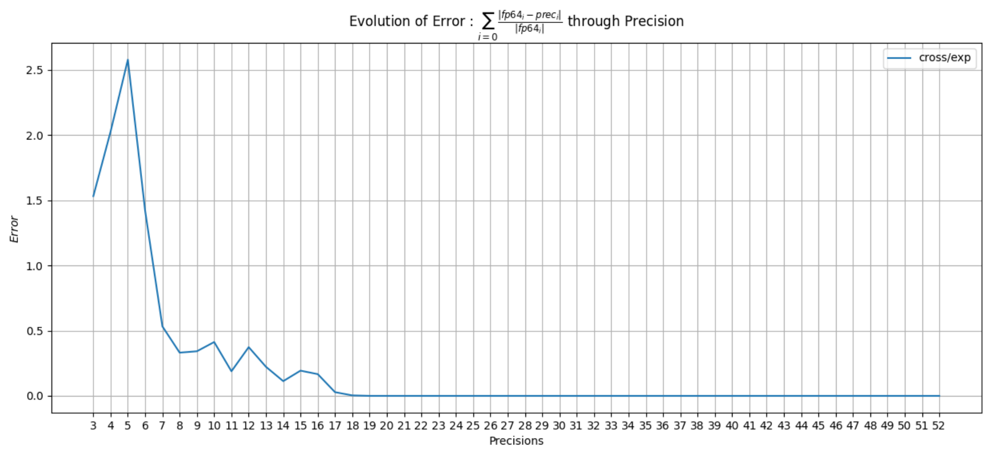


Figure 5.3: Accumulated Forward Error at every 500 iterations vs a different number of bits in mantissa for `exp()` in `cross()`.

We present our precision cropping approach implemented in Reactor Simulator by stages.

Stage 1: `sin(), cos(), exp()`

Based on previous observation, we start by implementing `sin(), cos(), exp()` functions in single precision. These functions as explained above are called in `cross()` and `update()`. As predicted by our instrumentation with the help of Verificarlo, we observe no decrease in accuracy while saving 4 % in time-to-solution on Kebnekaise. Additionally, the savings are 8.4 % in time-to-solution and 8.7 % in energy-to-solution on our Laptop.

Stage 2: `cross(), update()`

We then proceed to lower the precision of computation of the functions that call `sin(), cos(), exp()` ensuring that potential time savings are maximized without compromising accuracy. Considering our observations of the original application with the help of Callgrind, Stage 1 and Stage 2 comprehends almost 60 % of instructions executed in the program. By reducing the precision in the 12 lines of code that contain these functions, on Kebnekaise, we achieved 10 % savings in time-to-solution while also reducing the number of instructions counted by Callgrind, which are associated with `cross()` and `update()`, to 45 %. And, on our Laptop we achieved 13.4 % savings in time-to-solution and also 12.8 % in energy-to-solution.

After examining the shared library functions that most significantly contribute to the benchmark's execution time, we proceed to instrument each function individually at single precision, while leaving the others in double precision. The shared library functions and castings were excluded from this instrumentation since our code inspection in the previous stages determined that using single precision for these elements does not lead to any loss of accuracy. By following this approach we explore the extent to which we can reduce precision. Figure 5.4 shows the instrumented functions and the amount of error each contributed to the end result under this approach.

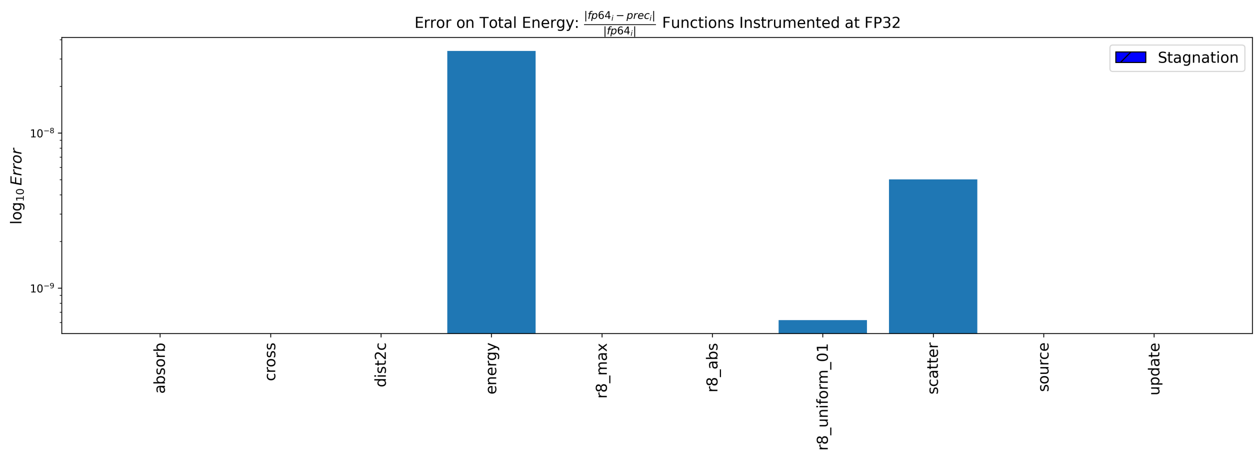


Figure 5.4: Forward Error of Functions instrumented in FP32.

Stage 3: Excluding `scatter()`, `energy()` and `r8_uniform_01()`

We observe that the functions other than `energy()`, `scatter()` and `r8_uniform_01()` do not contribute to the forward error. Therefore, we extended our implementation by applying single precision computation to all functions except these three.

```
    mu_d = mu;
    azm_d = azm;
          scatter ( seed, e, mu_d, azm_d );
    mu = mu_d;
    azm = azm_d;
```

Listing 5.1: Copying Variables for Scatter Function. `mu_d` and `azm_d` are in double and correspondents are in single precision.

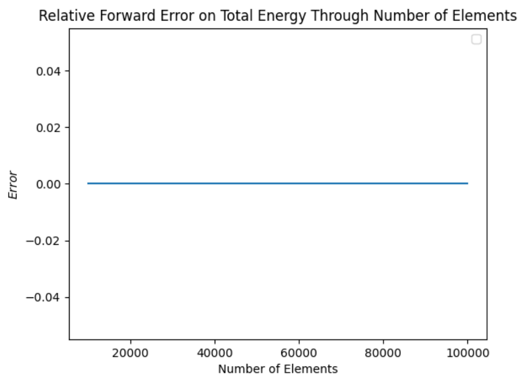


Figure 5.5: Error on Total Energy Variable

Consequently, `r8_uniform_01()` is called in double precision in these `scatter()` and `energy()` functions and is called in single precision in the rest of the application and the variables in the function `main()` are changed to float. Thus, we had to implement copying of some of these elements necessary for the `scatter()` function as shown in Listing 5.1 since they are passed by reference to the function for these variables. Even with the cost of these copies, we were able to achieve 16.6% in savings in time-to-solution on Kebnekaise and 16.1% on Laptop, with an additional saving of 14.9% on energy-to-solution and no loss in accuracy as shown in the Figure 5.5.

Type	**Energy (J)**	**Time (s)**
Double	8075,75	3.35
Mixed	6869,16	2.81
Saving	14.9%	16.1%

Table 5.1: Savings achieved with Stage 3 on Laptop.

Stage 4: Excluding `energy()`

On this phase, we included the `scatter()` function into our low precision computations. Therefore in this phase, `energy()` function is the only function that is left aside, not implemented in single precision. Thus, `r8_uniform_01()` is accessed in single precision by all the functions. Additionally, the functions that are accessing the energy variable in double by copy are changed to single precision, in the functions that accept it by reference still accessing it in double precision from the main function, therefore the castings to single precision are handled by the functions. This approach helped us to achieve 14.3% savings in time-to-solution on Kebnekaise while keeping the accuracy intact up to 50.000 elements as shown in the Figure 5.6 which depicts the difference of accuracy when the whole application is instrumented in single precision comparing to our implementation.

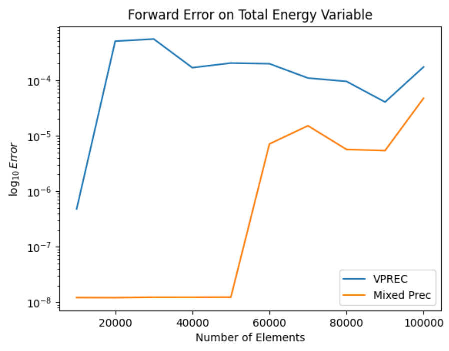


Figure 5.6: Forward Error Comparison VPREC-FP32 versus Stage 4.

Stage 5: All in FP32

As we notice from the Figure 5.6 there is still room for precision reduction opportunity for up to 50.000 number of elements. Therefore in order to exploit this gap, we included the `energy()` function to our low precision computation, making the application completely in single precision, i.e the energy variable explained before is also changed to single precision and reference passings are handled accordingly. As we can deduce from Table 5.2 and Figure 5.7, this stage comes with a compromise on accuracy starting from the small number of elements and with increasing loss faster than the example we see on Stage 4 with Figure 5.5.

Type	Energy (J)	Time (s)
Double	8075.75	2.23
Mixed	6966.3	1.85
Save	13.7%	17.7%

Table 5.2: Savings achieved with Stage 5 on Laptop.

Roofline Inspection For Performance

With the help of Intel Advisor we were able to observe our application's characteristics and showcase the return of investment that we have achieved. As discussed before, the application always allocates the same number of bytes no matter the number of elements, which can fit into L2 cache, and always uses that already allocated variables. Additionally, upon inspection we notice that it does not change between mixed-precision and double precision implementation. This means that the denominator part of the Arithmetic Intensity stays stable between different precision settings.

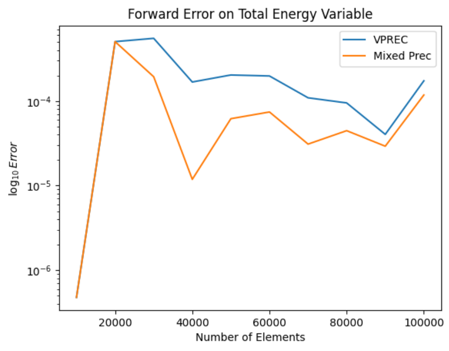
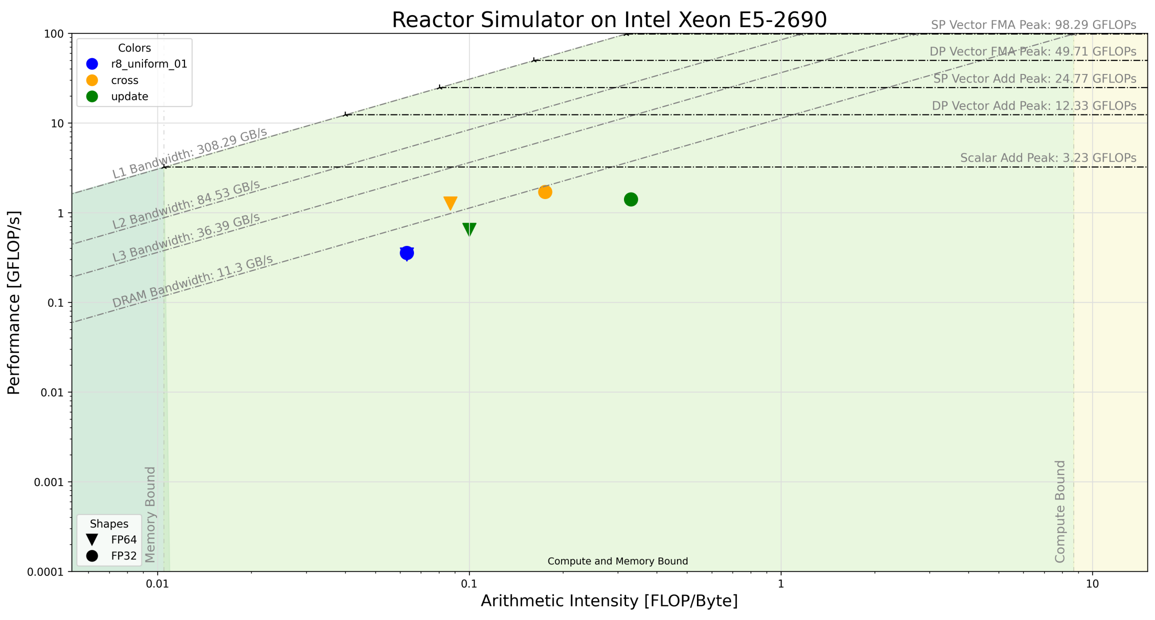


Figure 5.7: Forward Error Comparison VPREC-FP32 versus Stage 5.



z

Figure 5.8: Roofline Comparison of Stage 5 and Double Precision on Kebnekaise.

Moreover, we notice that our application can not be vectorizable since it does not contain a repetitive computation over a variable, therefore only scalar instructions are present in the assembly code as we checked the binary. The consequences of this structure can be seen from the Figure 5.8, being bounded by Scalar Add Peak. We can see that after the modifications we achieved better Arithmetic Intensity as single precision instructions tend to be faster than double precision instructions. Lastly, we observe no change in the most time consuming part of the application, `r8_uniform_01()`. As this function operates only on integer elements, it is not affected by the modifications.

5.3 Discussion

Type	**Stage 1**	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**
Double Median	4.87	4.81	4.92	4.82	4.87
Mixed Median	4.63	4.28	4.28	4.12	4.11
Savings	4.7%	11.0%	16.6%	14.3%	15.7%
Std. Dev.	2.5%	1.1%	1.2%	1.5%	2.6%
Error^2	0	0	0	10^{-8}	10^{-7}

Table 5.3: Achieved Savings With Mixed-Precision Versions on Reactor Simulator on Kebnekaise

We demonstrated that we could obtain up to 16.6% gain in time-to-solution and also in energy-to-solution with no loss of accuracy in our application as shown in Table 5.3 with Stage 3. We showcased a modular approach to pin point the low precision computation areas of the application with the help of Verificarlo's VPREC backend and presented how this information can be leveraged. We started with the most time consuming parts of the application, monitored the possible accuracy loss and time gains in case of single precision computation, and we expended this approach to the functions they are called from, consequently finalizing it with the implementation of related functions. We observed with the different versions of the functions that the results that are obtained with Verificarlo's instrumentation are holding up with the real life implementations.

Type	**Stage 1**	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**
Double Median	3.33	3.34	3.29	3.29	3.29
Mixed Median	3.06	2.89	2.81	2.9	2.8
Savings	8.4%	13.4%	16.1%	12.6%	14%
Std. Dev.	1.1%	1.1%	1.2%	1.3%	2.3%

Table 5.4: Achieved Savings With Mixed-Precision Versions on Reactor Simulator on Laptop

Type	**Stage 1**	**Stage 2**	**Stage 3**	**Stage 4**	**Stage 5**	**Double**
Cores Median	7372.93	7034.02	6869.165	7138.684	6966.302	8075.75
Savings	8.7%	12.8%	14.9%	11.6%	13.7%	-

Table 5.5: Achieved Energy (in Joules) Savings With Mixed-Precision Versions on Reactor Simulator on Laptop

In our measurements on the Laptop, we successfully observed the effects of precision reduction not only on time-to-solution but also on energy-to-solution. Our results indicate that Stage 3 offers the best performance in both categories with 16.1% on increase in performance and 14.9% decrease in energy. Moreover, the order of time savings remained consistent with energy savings, likely reflecting the intrinsic relationship between power, energy, and time.

By adhering to the Swedish principle of lagom, which advocates for "just enough," we were able to reduce both time and energy consumption by applying just enough precision without compromising accuracy.

6 LULESH

LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) is a proxy application for ALE3D developed by LLNL [19][20][24] to replicate the computational behavior of a hydrodynamics code, allowing for realistic performance and scalability evaluations. It is a standalone application without external dependencies. This makes it a valuable candidate for developing strategies to optimize performance and energy efficiency in scientific computing.

This chapter describes the benchmark, provides a code overview, investigates the code using arithmetic and profiling tools, and explores two approaches for implementing mixed-precision techniques.

6.1 Description of the Benchmark

LULESH, implements Lagrangian hydrodynamics formulations to solve the governing equations for a single-material Sedov blast problem in three dimensions, though the code uses the concept of regions to support multiple material types. It has an analytical solution and the problem set up is hard-coded for this specific problem. It discretizes these equations by dividing the spatial domain into a collection of mesh-defined elements, the details can be found in Section B.2, coupling them through stencil operations, and incorporating material properties and the equation of state. Within a single domain, there can be multiple regions, each representing a different material or part of the simulation space, allowing array indirections in key computational kernels. Finite element approximation is used to compute spatial gradients needed for solving the governing equations.

The following steps summarize the initialization and execution process of the simulation, more details can be found in the Section B.1:

1. Necessary number of domains and their related variables for a portion of the mesh, also the material index set is created
2. Previously allocated variables are defined according to the initial problem state.
3. After the initialization we have information about the shape of the mesh. Boundary conditions to model the mesh symmetries are created.
4. *While* $t^n < t_{stop}$:
   (a) Compute Δt^n with the help of CFL in the function `TimeIncrement()`.
   (b) In the function `LagrangeLeapFrog()` advance the solution first in the nodes, then in the elements.

6.2 Inspection and Implementation

6.2.1 Initial Inspection With Tools

In this section, we will examine the LULESH sequential application implemented in double precision and identify opportunities for precision cropping using the methodology from

Chapter 4.

For this benchmark, we focus on three forces (f_x, f_y, f_z), the internal energy e at the origin, and Δt computed using CFL. Tracking multiple variables ensures that the system's physical properties are accurately captured, helps us understand error propagation, element sensitivity to precision changes, and the impact of reduced precision on physical outcomes.

With no convergence criterion, we assess accuracy by comparing results with the number of time steps and the final internal energy at the origin, as outlined in the technical manual.

With 20 elements per edge, resulting in 20^3 elements in total, we observe that as precision increases, the error decreases, as shown in Figure 6.1. Unlike the behavior observed in the conjugate gradient method presented by Chen et al. [3], where error reduction smooths out after a certain precision level, we do not observe such smoothness in our results. Instead, as we observed with Reactor Simulator in Chapter 5, there is a roughly linear relationship with the error introduced at reduced precision computation with the number of significant bits.

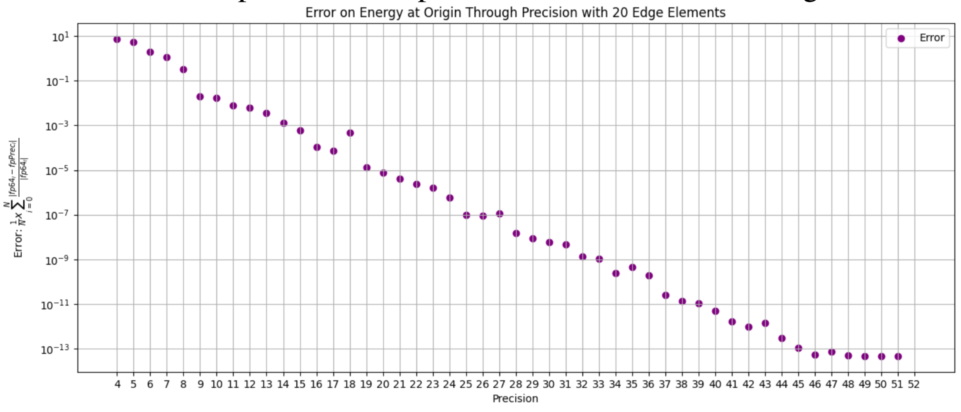


Figure 6.1: Evolution of Mean Relative Error of Energy over Precision, N : number of iterations.

It is important to note that with certain precision values, the timestep selected did not reach the final time value and instead stagnated. To prevent these cases from running indefinitely, we implemented a timeout mechanism to terminate the process if the number of iterations significantly exceeded that of the original application. Consequently, Figure 6.1 shows the mean error per iteration of LULESH in computing the energy of the element at the origin. We observe that at a precision level of 23 bits (corresponding to FP32), the application stagnates, which will be further investigated in the future.

Additionally, as as shown in Figure 6.2, the behavior varies as the number of elements increases. Therefore, we extended our research to identify opportunities for precision reduction while maintaining an acceptable level of accuracy.

The reason for calculating the error contribution per iteration is that it allows us to estimate the magnitude of the error. By multiplying this value by the expected number of iterations provided in the manual, we can approximate the worst-case scenario for the expected floating-point error, adopting a deterministic approach.

LULESH consists of 40 functions that include floating-point operations (excluding C++ functions such as those for vector allocation). Therefore, it is essential to carefully identify and

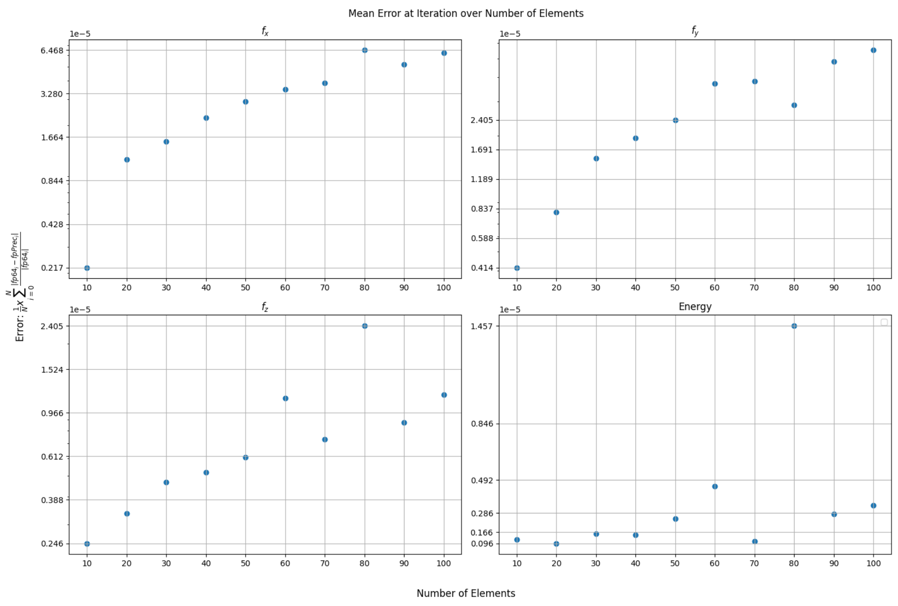


Figure 6.2: Evolution of Error on *Significant Variables* over Number of Elements at FP32

filter out sections that may become unstable when using single precision. This approach helps to reduce development time while ensuring reliable accuracy. To assist in this process, Verificarlo's MCA backend is used to detect unstable code sections through stochastic rounding. Figure 6.3 shows that the error committed on Energy variable around the mean value is minimal in both RR and MCA modes. In the MCA mode, there are no significant fluctuations in the application's final results, as indicated in Table 6.1, suggesting that the accumulation of floating-point errors is likely negligible. Additionally we observe that with 35 MCA runs, the number of significant digits in the final result remains around 5 when the virtual precision of choice is 23 i.e single precision. Interestingly, we did not observe any stagnation in the application when using either MCA or RR modes, even though the instrumentation was performed in single precision. However, some repetitions required more iterations to complete, which will be explored further in future work.

Statistic	**MCA Mode**	**RR Mode**
Mean	96688.7749	96688.7915
Standard Deviation	0.3108	0.1738
Significant Digits	5.4	5.7

Table 6.1: Statistical Analysis of MCA and RR Modes When Instrumented at FP32

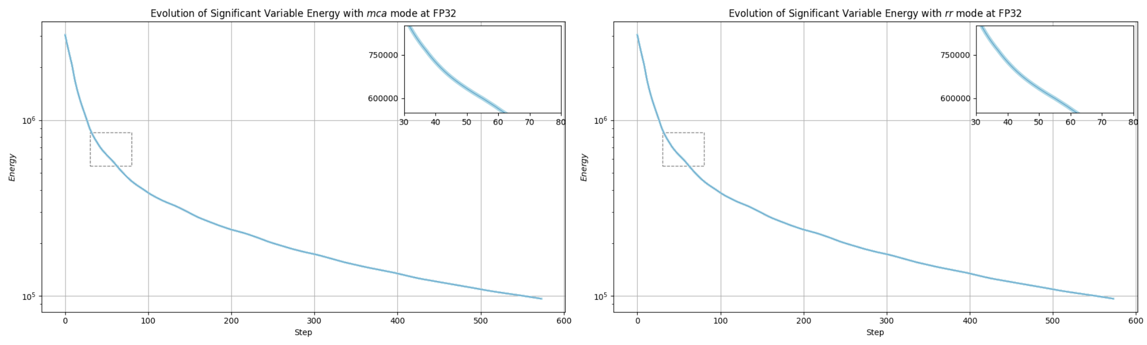


Figure 6.3: Evolution of Energy with MCA (left) and RR (right) modes at FP32

6.2.2 How We Conducted Measurements

Performance measurements for this benchmark were conducted on the Kebnekaise system using `g++` version 10.2 with the `-O2` optimization flag. To account for potential accuracy impacts from compiler optimizations, we also compiled the reference implementation with Verificarlo. Following the application manual's guidelines, we ran the double precision application with various element configurations and checked if the iteration counts matched the manual. For cases where the application stalled, only the original iteration count was used for accuracy measurements, and these instances were noted. Unless stated otherwise, all measurements were done with 20 elements per edge to ensure consistency. Time-to-solution was measured 32 times for each application, and time savings were based on the median results.

6.2.3 Detailed Code Inspection and Mixed-Precision Implementations

Figure 6.4 shows the behavior of forward error of two significant variables, f_x and *energy* when the functions named on the x axis are instrumented in single precision while the rest of the application is in double precision. That way, we are able to determine which functions can be implemented in low precision without compromising accuracy or which regions must stay in double in order to preserve it.

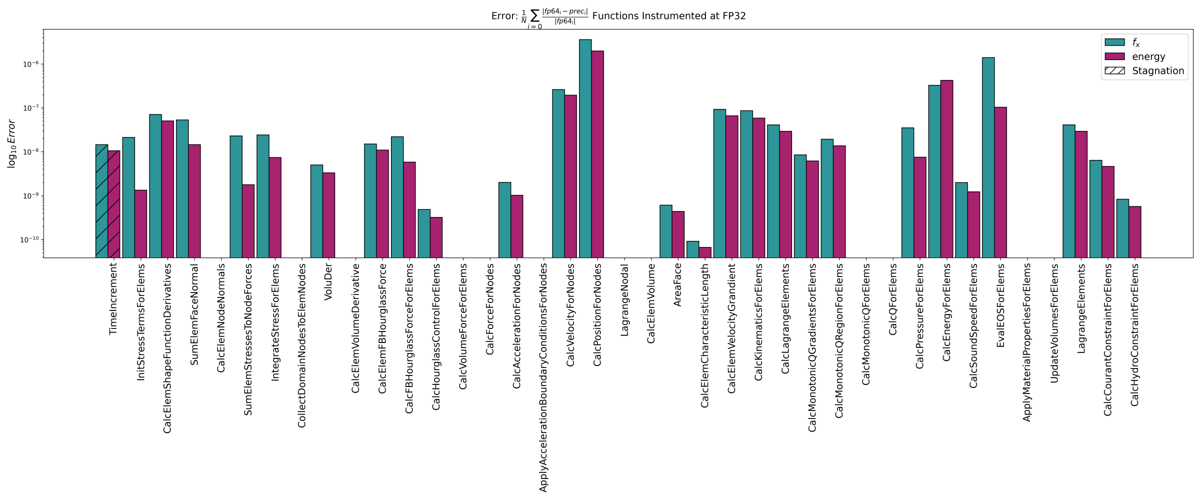


Figure 6.4: Error contribution by low precision computation per iteration for each function on Energy and f_x variables.

We observed that the error patterns are consistent across force variables, though there are notable differences between force and *energy* variable. Specifically, errors are generally more pronounced in force variables compared to *energy* variable. Consequently, when determining areas for precision reduction, we prioritized minimizing error in *energy* variable because it is the objective of the simulation to compute.

Changing an application from double to mixed-precision is complex and requires balancing accuracy and performance. Each application has a unique structure, so our approach must adapt accordingly. The main challenge in mixed-precision strategies is managing the trade-off between the speed benefits of lower precision and the overhead of copying and casting between

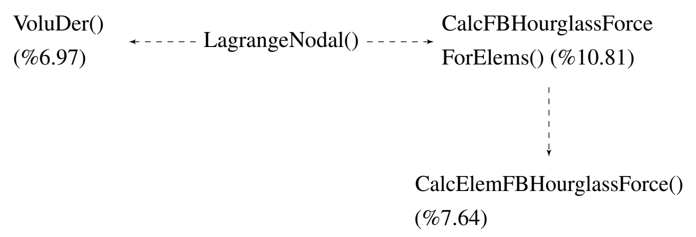


Figure 6.5: Self Times and Call Relations on LULESH

variable types. Ideally, lower precision should be applied where it has minimal impact on accuracy and in the most time-consuming parts to maximize time savings. However, this balance is not always achievable, as functions may not meet both criteria, and the overhead of casting and copying can sometimes negate performance gains.

For this reason, we chose to implement two different approaches with this benchmark. We will call one the inclusion approach and the other one the exclusion approach. While inclusion consists of including more single precision computation at the code regions that consume the most time while keeping the rest of the application in double precision and aims to reduce time, in the exclusion approach we keep rest of the application in single precision and introduce double precision at the code regions that are estimated to contribute the most error when they are singled out and instrumented at single precision with Verificarlo's VPREC backend.

Inclusion Approach

In this method, we maintained double precision for the majority of the application while converting only the most time-consuming quartile, shown in Section 6.2.3 and its associated functions to single precision.

Stage 1: `CalcElemFBHourglassForce()`

```
static inline void
CalcElemFBHourglassForce(Real_t *xd, Real_t *yd, Real_t *zd,  Real_t hourgam[][4],
    Real_t coefficient, Real_t *hgfx, Real_t *hgfy, Real_t *hgfz )
```

Listing 6.1: CalcElemFBHourglassForce() Function signature

We start our conversion stages with this function with the intention of expanding its application to outer functions. It is called only by `CalcFBHourglassForceForElems()` for each element to compute the hourglass forces for each element. As the function signature shows, it accepts arguments as separate array sets rather than as class members. Therefore, for some arguments, we had to handle them by casting and copying each value outside of this function. We notice the effects of these supplementary operations on the performance as the application run time is extended by 4.5%, meaning mixed-precision implementation is running longer than the double precision version.

Stage 2: `CalcFBHourglassForceForElems()`

This function calculates the hourglass control force for each element, involving a large block of code with loop unrolling, array indirections, and reductions.

We focus on this function because it has the highest self-time in the application and contains the previously modified function, reducing some casting and copying operations. Due to its size, we implemented this function in three phases.

In the first phase, the function accepts arguments in double precision but converts or copies them to single precision within each call. All local variables are converted to single precision, except for members of the Domain class, which remain in double precision. Other variables are either declared directly in single precision or copied and cast as needed, keeping only Domain class members in double precision. With this double-precision implementation, we observed a 14% increase in runtime on Kebnekaise compared to the pure double-precision application. In the second phase, we introduced temporary single-precision arrays to store domain elements and continued computations with these. Instead of dynamically allocating a large array (since the number of elements is unknown), we allocated an 8-element array on the stack, maintaining indirect access to the main array. We also added copying steps between these temporary and main arrays before the original computation.

This implementation led to a 16% increase in runtime for the mixed-precision application compared to the double-precision version on Kebnekaise. The overhead from copying between indirectly accessed arrays outweighed the gains from using single precision, so we will avoid this in future implementations.

Finally, we modified the function to use single precision in the function signature and moved copy and cast operations to the caller function, `CalcHourGlassControlForElems()`. Local variables and passed variables were also converted to single precision. This change resulted in an 18.1% reduction in runtime while maintaining a forward relative error of 9.3×10^{-10} at the last element.

Stage 3: `CalcHourglassControlForElems()`

This routine calculates the hourglass control contribution for each element. It calls three functions, one of which is the single-precision function we converted in the previous stage. The second function is `CollectDomainNodestoElemNodes()`, which does not contribute to the error when instrumented alone, as shown in Figure 6.4. The third function is `CalcVolumeDerivative()`, which also shows no error contribution when isolated without its child functions. This is because `CalcVolumeDerivative()` serves primarily as a wrapper, calling only the `VoluDer()` function without having any internal variables itself. The `VoluDer()` function ranks in the second half of functions contributing the most to error and in the first quartile of the most time-consuming functions, making it a good candidate for conversion to single precision. Consequently, we proceed to convert these two functions to single precision. And finally we obtain a 12.3% decrease in time-to-solution on Kebnekaise, while achieving 1.9×10^{-9} relative forward error at the last element.

Type	Stage 1	Stage 2	Stage 3
Functions	CalcElemFB-HourglassForce()	CalcFB-HourglassForceFE()	CalcHourglassControlFE() CollectDomainNtoElemN() VoluDer()
Double Median	5.27	5.27	5.25
Mixed Median	5.51	4.21	4.6
Savings	-4.5%	20.2%	12.3%
Std. Dev.	2.8%	3.5%	1.2%
Error[1]	1.2×10^{-9}	6.9×10^{-9}	9.6×10^{-9}

Table 6.2: Achieved Savings With Mixed-Precision Versions on LULESH on Kebnekaise with Inclusion Approach

In this approach, we increased the use of single precision computations, starting from the most time consuming inner functions and extending outwards. Contrary to the Reactor Simulator's immediate reduction in time-to-solution, our method did not yield instantaneous results. Instead, we observed significant impacts due to the indirect access of arrays, which caused additional memory access costs. To mitigate these issues, we developed strategies to minimize supplementary data copying and type casting. Persistent application of these strategies throughout the project led to a substantial reduction in time-to-solution by 20.1% on the Kebnekaise system. We observed that performance and energy consumption metrics on the Laptop showed similar timing results to those on Kebnekaise, details in B.3. Although the exact proportions differed, the energy consumption followed the same trend as the time consumption, as shown in Table 6.4, consistent with what we saw on the Reactor Simulator.

Type	Stage 1	Stage 2	Stage 3	Double
Cores Median	10932.41	9165.34	9861.84	10187.16
Savings	-7.3%	10%	3.19%	-

Table 6.3: Achieved Energy (in Joules) Savings With Mixed-Precision Versions on LULESH on Laptop

As a result, we were driven to investigate other possible strategies that might provide a simpler and more direct approach to immediately improved performance. The preliminary results and the approaches discussed below are still in the initial stages of exploration and will be examined more thoroughly in future studies to ensure a comprehensive understanding and validation of the proposed changes.

Exclusion Approach

Although Verificarlo's VPREC backend with 23 bits of mantissa indicates that the application stagnates regardless of the number of elements, we tested the application's behavior when fully implemented in single precision. This was straightforward to implement, as all data types are defined using `typedef`, allowing us to simply replace the definition with `typedef float Real_t;`.

Unlike Verificarlo's VPREC backend results, this experiment did not show stagnation, we plan to explore further. Instead, it provided a 18.8% time gain compared to the double precision version, as shown in Table 6.4, in Group 0 row.

We used this implementation as the baseline for accuracy and time gain, with all experiments built upon it. While some parts of the application had reduced accuracy, we improved it by applying double precision to the most error-prone functions, as shown in Figure 6.4, advocating for precision where needed. The "Single Error" column lists the error contributions of these functions individually in Table 6.4 which details the functions excluded at each step grouped by their interrelated nature, with each group containing functions from the previous one. We focused on the top quartile of error contributors in single precision, ranging from 1.15×10^{-8} to 1.77×10^{-6}.

When analyzing the self-time statistics of functions, we see that the last quartile ranges from 2.84% to 10.22% self-time. This helps us identify potential candidate functions and adjust our strategy. Notably, half of the functions have less than 1.15% self-time, which suggests we should rethink our conversion design. Only one function appears in both the last quartile of error-prone functions and the first quartile of time-consuming functions. Since our goal is to apply double precision to the functions that benefit most, this overlap is minimal. Thus, we decided to use double precision only for this function to reduce error, while keeping the rest of the application in single precision, estimating minimal time gains after considering copying and casting costs.

Groups	**Function Name**	**Single Error**	**SelfTime(%)**	**Time Gain**
Base	Complete FP64 Application	0	-	-
Group 0	Complete FP32 Application	1.95×10^{-6}	-	18.8 %
Group 1	Time Increment()	10^{-8}	$< 0.1\%$	
Cumulative Error		1.95×10^{-6}		14.4 %
Group 2	CalcPositionFN()	1.77×10^{-6}	0.48%	
Improvement		1.95×10^{-6}		13.7%
Group 3	CalcEnergyFE()	3×10^{-7}	5.08%	
	CalcPressureFE()	3.5×10^{-8}	2.91%	
Improvement		1.89×10^{-6}		11%
Group 4	EvalEOSFE()	1.2×10^{-6}	2.47%	
	CalcSoundSpreedFE()	10^{-9}	$< 0.1\%$	
Improvement		2.08×10^{-6}		10.3%
Group 5	CalcKinematicFE()	7.8×10^{-8}	3.47%	
	CollectDomainNtoElemN()	0	4.51%	
	CalcElemVolume()	0	1.79%	
	CalcElemShapeFuncDeriv()	6.5×10^{-8}	3.47%	
	CalcVelocityGradient()	8×10^{-8}	1.71%	
Improvement		1.33×10^{-6}		7.9%
Group 6	CalcVelocityFN()	1.8×10^{-7}	0.81%	
Improvement		1.8×10^{-6}		6.8%

Table 6.4: Detailed Function Statistics and Cumulative Implementation Steps of Mixed-Precision Application with Achieved Error Improvements and Time Gain on Kebnekaise

Group 1: `TimeIncrement()`

When this function is instrumented in single precision using the Verificarlo VPREC backend—whether in isolation or alongside other functions—we observe that the application stagnates. This occurs despite the fact that the computed error for the same number of iterations with the double precision application is approximately 9×10^{-9}. Even though this stagnation does not occur with our implementation we implemented in double precision.

Group 2: `CalcPositionForNodes()`

The function is relatively small and primarily involves fused multiply-add operations (FMAs). To compute a single position, it requires one copy of dt outside the function and two additional copies for each node element—one before and one after the computational section. This redundant copying contributes to performance loss.

Although this function is in the third quartile with only 0.48% self-time, it is the most error-prone function when instrumented alone. This discrepancy highlights that the error contribution of a function is not necessarily correlated with its number of instructions. Consequently, we decided to prioritize transforming this function to double precision.

Group 3: `CalcEnergyForElements()`

It is the second most error-prone function when evaluated in isolation, To address these issues and reduce the overhead associated with data handling, we decided to convert both this function and `CalcPressureForElems()` to double precision. This change helps to reduce the performance loss from casting and copying while maintaining accuracy. Additionally, this function is invoked from `EvalEOSForElems()`, which is in Group 4, therefore helps to reduce the mentioned performance loss when converted in this order.

Group 4: `EvalEOSForElems()`

In addition to its significant self-time, the total time spent in this function, combined with the time spent in the functions discussed previously, accounts for 12.83% of the total runtime for this reason it is preferable to maintain single precision in the underlying functions for performance concerns. Moreover, the function calls `CalcSoundSpeedForElems()`, which is so small that, if not converted to double precision, the overhead from conversions, copies, and casts, it becomes unprofitable. Therefore, this function is also converted to double precision.

By including these two functions in double precision, we achieved an improvement in accuracy compared to the full single precision application in Group 0, while also keeping a considerable gain in performance at 10.3% .

Group 5: `CalcKinematicForElems()`

This function makes call to its sub functions listed in the table. from total time perspective, it constitutes of 17.48% time of the application. We observe that the functions listed with 0 error contribution are the ideal candidate to keep in single precision considering their selftimes as one is in the forth quartile and the other one is in the third. With this implementation, at the expense of 8% loss in time gain comparing to our initial implementation of Group 0, we were able to retrieve some accuracy back.

Group 6: `CalcVelocityForNodes()`

Finally, we conclude by converting this function to double precision. Despite being the third most error-prone function when considered individually, it is implemented last because it is more isolated from the closely related functions we addressed earlier.

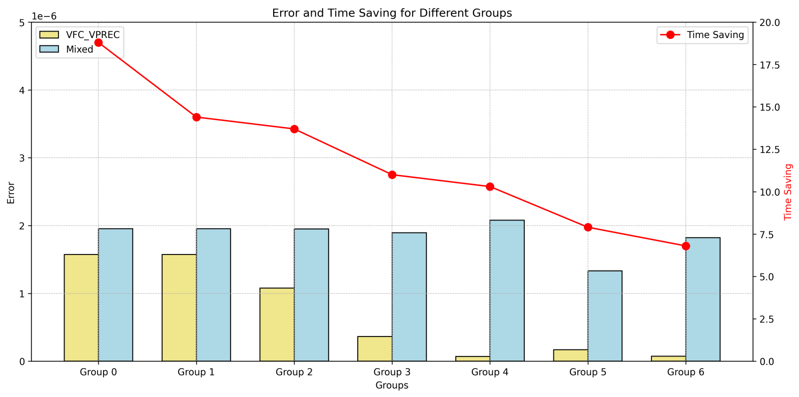


Figure 6.6: Error Comparison between Verificarlo Instrumentation and Implementation Stages given with Time Saving

Figure 6.6 shows a comparison of the mean forward error per iteration for the energy variable, as measured by Verificarlo VPREC and our implementation. In this comparison, for each group, we excluded the functions that are part of the group and those from previous groups. At compile time, Verificarlo was configured to instrument the remainder of the application in single precision, while the functions included in each group were kept in double precision.

6.3 Discussion

Using the Verificarlo VPREC backend, we analyzed the application to identify sections that introduced the most errors in single precision while keeping the rest in double precision. Based on this and insights into the most time-consuming sections, we developed two methods to address potential issues in precision cropping, one of which is still under development.

With the inclusion approach, we achieved up to a 20% performance gain on Kebnekaise by moving conversions outside functions and reducing copying. Then, to find the best strategy for precision reduction, we used the exclusion approach. This involved converting the entire application to single precision and then selectively reverting the most error-prone functions to double precision to determine which functions should remain at higher precision.

In the exclusion approach, we aimed to reduce precision while handling array access complexities and associated castings. We set limits on acceptable accuracy loss and potential time gains, exploring optimization strategies that balance copying/casting overhead with the speed of low-precision computation, ultimately developing two complementary approaches.

7 Conclusion and Future Work

This internship focused on investigating energy efficiency opportunities in scientific codes through the use of mixed-precision methods and tools. As the emphasis on energy efficient hardware grows, having equally reliable and efficient software is crucial for leveraging these advancements in the hardware-software co-design cycle. On the other hand, we see the increasing need of high accuracy as the numerical simulations grow to take advantage of the exascale era that we are in, while also trying to minimize their energy footprint. With these aspects in mind, mixed-precision computation correspond well to the current demands, it provides flexibility in maintaining accuracy while also offering increased performance and reduced energy consumption.

To address this need, we first developed a best practice guide aimed to help incorporating energy-to-solution as a key performance metric. This comprehensive guide assists researchers in understanding energy measurements, using available tools, and presenting their findings, with practical examples from the CEEC projects on European HPC systems. We also contributed to an interdisciplinary collaboration space where researchers from various fields could share insights and methodologies.

Building on this foundation, we developed our scientific approach further by examining the trade-offs between precision, accuracy, and performance in scientific applications. We used a holistic methodology to maintain accuracy while optimizing time-to-solution and energy-to-solution. Using computer arithmetic tools and performance profilers, we identified opportunities for precision reduction and explored the trade-offs associated with converting applications to mixed precision.

Using Callgrind, we identified the performance hotspots in the application. Next, with the Roofline Model, we found code regions that could benefit from precision reduction. We then used Verificarlo to identify numerical hotspots. At each step, we excluded regions that were not suitable for reduced precision computation and then applied mixed-precision to the selected code regions.

We evaluated our methodology on two benchmarks: In the Reactor Simulator, we reduced time-to-solution and energy-to-solution by up to 15%, while maintaining the same accuracy as the original double precision version, supporting our approach of using just enough precision.

In the LULESH benchmark, we developed two mixed-precision strategies. The first, the inclusion approach, kept double precision across the application but reduced precision in time-intensive regions. This method allowed us to explore the trade-offs between the speed of single-precision computations and the overhead of casting and copying. It achieved up to a 20% improvement in time-to-solution and 10% in energy-to-solution. In the second, called as the exclusion approach, we experimented with various mixed-precision implementation strategies to find the optimal balance between double and single precision. By leveraging the application's initial design, we kept the entire application in single precision while gradually applying double precision to the code regions that produced the most errors when computed in single precision. This approach is still under investigation. Our initial efforts are focused on understanding the discrepancies in instrumentation with the Verificarlo VPREC backend to identify any potential limitations that may arise from this exclusion approach.

In the final weeks of this internship, the focus will be on investigating the differences in single precision instrumentation between certain stages of the Reactor Simulator and LULESH, particularly concerning the stagnating function. We would like to identify the potential benefits of finding an optimal approach and implement it to parallel versions of LULESH. Additionally, we plan to continue exploring our exclusion approach, as discussed in Section 6.2.3, to determine suitable application types for this method. Another area worth exploring to further reduce the energy footprint of scientific codes is the use of adaptive precision techniques over the course of computation.

While the HPC environment has extensively developed and well-established performance models, there is a lack of standardization in representing and modeling energy-to-solution metrics, as this aspect of the field is still in its infancy. Therefore, we aim to explore the insights that the original Roofline Model [4] and Cache-Aware Roofline Model [14] can provide for energy modeling. Additionally, we are interested in investigating how the Execution-Cache-Memory Model [12] can be applied to energy measurements and how Amdahl's Law [29] can be used to model energy consumption.

Bibliography

[1] Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, and et al. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *The International Journal of High Performance Computing Applications*, 35(4):344–369, 2021. doi: 10.1177/10943420211003313.

[2] Yohan Chatelain, Eric Petit, Pablo de Oliveira Castro, Ghislain Lartigue, and David Defour. Automatic exploration of reduced floating-point representations in iterative methods. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing*, pages 481–494, Cham, 2019. Springer International Publishing. ISBN 978-3-030-29400-7.

[3] Yanxiang Chen, Pablo de Oliveira Castro, Paolo Bientinesi, and Roman Iakymchuk. Enabling mixed-precision with the help of tools: A nekbone case study, 2024. URL https://arxiv.org/abs/2405.11065.

[4] J. Choi, D. Bedard, R. Fowler, and R. Vuduc. A roofline model of energy. *IPDPS*, 2012. URL https://smartech.gatech.edu/xmlui/handle/1853/45737.

[5] Pablo de Oliveira Castro. *High Performance Computing code optimizations: Tuning performance and accuracy*. Habilitation à diriger des recherches, Université Paris-Saclay, October 2022. URL https://theses.hal.science/tel-03831483.

[6] Christophe Denis, Pablo De Oliveira Castro, and Eric Petit. Verificarlo: Checking floating point accuracy through monte carlo arithmetic. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 55–62, 2016. doi: 10.1109/ARITH.2016.31.

[7] J Dongarra, J Hittinger, J Bell, L Chacon, R Falgout, M Heroux, P Hovland, E Ng, C Webster, and S Wild. Applied mathematics research for exascale computing. 2 2014. doi: 10.2172/1149042. URL https://www.osti.gov/biblio/1149042.

[8] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mar 1991. ISSN 0360-0300. doi: 10.1145/103162.103163. URL https://doi.org/10.1145/103162.103163.

[9] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 896–904, 2015. doi: 10.1109/IPDPSW.2015.70.

[10] Nicholas J. Higham and Theo Mary. A new approach to probabilistic rounding error analysis. *SIAM Journal on Scientific Computing*, 41(5):A2815–A2835, 2019. doi: 10.1137/18M1226312. URL https://doi.org/10.1137/18M1226312.

[11] Nicholas J. Higham and Theo Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica*, 31:347–414, 2022. doi: 10.1017/S0962492922000022.

[12] HPC Group at Friedrich-Alexander University Erlangen-Nürnberg (FAU). Execution-cache-memory (ecm) model, 2024. URL https://hpc.fau.de/research/ecm/. Available at: https://hpc.fau.de/research/ecm/ [Accessed: 2024-09-06].

[13] Roman Iakymchuk, Gülçin Gedik, Kajol Kulkarni, Yanxiang Chen, Daniel Kempf, Samuel Kemmler, Dimitris Papageorgiou, Damaskinos Konioris, Sally Kiebdaj, Julita Corbalan, and Harald Köstler. Best Practice Guide – Harvesting energy consumption on European HPC systems: Sharing Experience from the CEEC project, August 2024. URL https://doi.org/10.5281/zenodo.13306639.

[14] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2014. doi: 10.1109/L-CA.2013.6.

[15] Intel Corporation. Intel® advisor - roofline analysis guide, 2023. URL https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html. Accessed: 2024-09-02.

[16] Intel Corporation. Intel® advisor - vectorization, threading, memory, and offload optimization, 2024. URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html. Accessed: 2024-09-02.

[17] Niclas Jansson, Martin Karp, Adalberto Perez, Timofey Mukha, Yi Ju, Jiahui Liu, Szilárd Páll, Erwin Laure, Tino Weinkauf, Jörg Schumacher, Philipp Schlatter, and Stefano Markidis. Exploring the ultimate regime of turbulent rayleigh–bénard convection through unprecedented spectral-element simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092. doi: 10.1145/3581784.3627039. URL https://doi.org/10.1145/3581784.3627039.

[18] David Kahaner, Cleve Moler, Steven Nash, and John Burkardt. Reactor simulator, 1989. URL https://people.math.sc.edu/Burkardt/cpp_src/reactor_simulation/reactor_simulation.html.

[19] Ian Karlin, Abhinav. Bhatele, Bradford L. Chamberlain, Jonathan. Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charle H. Still, Felix Wang, and Daniel Wong. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, Livermore CA, December 2012.

[20] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, Livermore, CA, August 2013.

[21] Michael Malms, Laurent Cargemel, Estela Suarez, Nico Mittenzwey, Marc Duranton, Sakir Sezer, Craig Prunty, Pascale Rossé-Laurent, Maria Pérez-Harnandez, Manolis Marazakis, Guy Lonsdale, Paul Carpenter, Gabriel Antoniu, Sai Narasimharmurthy, André Brinkman, Dirk Pleiter, Utz-Uwe Haus, Jens Krueger, Hans-Christian Hoppe, Erwin Laure, Andreas Wierse, Valeria Bartsch, Kristel Michielsen, Cyril Allouche, Tobias Becker, and Robert Haas. ETP4HPC's SRA 5 - Strategic Research Agenda for High-Performance Computing in Europe - 2022, November 2022. URL https://doi.org/10.5281/zenodo.7347009.

[22] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefevre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhauser, Basel, Switzerland, 2nd edition, May 2018.

[23] D. Stott Parker, Brad Pierce, and Paul R. Eggert. Monte carlo arithmetic: how to gamble with floating point and win. *Computing in Science and Engg.*, 2(4):58–68, jul 2000. ISSN 1521-9615. doi: 10.1109/5992.852391. URL https://doi.org/10.1109/5992.852391.

[24] M. B. Gokhale R. D. Hornung, J. A. Keasler. Hydrodynamics challenge problem, 2011. URL https://www.osti.gov/servlets/purl/1117905.

[25] Daniel A. Reed and Jack Dongarra. Exascale computing and big data. *Commun. ACM*, 58 (7):56–68, jun 2015. ISSN 0001-0782. doi: 10.1145/2699414. URL https://doi.org/10.1145/2699414.

[26] TOP500. Green500 list, 2024. URL https://www.top500.org/lists/green500/. Accessed: 2024-08-30.

[27] Verificarlo Project. Verificarlo: A floating point arithmetic checker. https://github.com/verificarlo/verificarlo, 2024. Available at: https://github.com/verificarlo/verificarlo [Accessed: 2024-09-02].

[28] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, apr 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

[29] Dong Hyuk Woo and Hsien-Hsin S. Lee. Extending amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, 2008. doi: 10.1109/MC.2008.494.

A Machines

This section provides system description of the machines we worked on.

A.1 Kebnekaise

Type	Value
OS	Ubuntu 5.4
CPU	Intel(R) Xeon(R) Gold 6132
Base Frequency	2.60 GHz
Sockets	2
Num. Cores	14
Num. Threads	28
L1 Cache	32 KiB
L2 Cache	1 MiB
L3 Cache	19.25 MiB
TDP	140 W

Figure A.1: Kebnekaise Compute Node Details.

Kebnekaise is part of High Performance Computing Center North (HPC2N) which is a Swedish national center for Scientific and Parallel Computing. It is a heterogeneous system, consisting of several different node types such as Skylake CPUs, Skylake V100 GPU nodes, 3TB memory large memory nodes, AMD Zen3 nodes, and A100 GPU nodes, equipped with high bandwidth, low latency EDR/FDR InfiniBand interconnect.

Kebnekaise offers energy measurements with the help of RAPL on both AMD nodes and Intel nodes. On Intel nodes it has only Package and DRAM domains available, on AMD nodes it has Package and Core domains. SLURM is not configured with energy measurement support, additionally PAPI is also not configured to cover energy measurements, therefore we needed to have sudo access to read these values, which the administrators refused to give. Therefore we were not able to provide energy measurement results from this machine.

A.2 Laptop

The laptop runs Ubuntu 6.8 and is equipped with an Intel Core i7 processor with a base frequency of 3 GHz. It features one socket, 4 cores, and 2 threads. The CPU has a 32 KiB L1 cache, a 1.25 MiB L2 cache, and a 12 MiB L3 cache. The thermal design power (TDP) is 28 W, and it uses the intel_pstate driver for CPU frequency scaling.

B LULESH

B.1 Physics Details

To compute the nodal forces, stress terms for each element is initialized. Consequently, volumetric stress contribution for each element is integrated by a loop over the elements and summed the contributions in the local data elements. It is followed by the compute of hourglass contributions over a loop for each element. Lastly, hourglass filter is applied to control force for each element and independently added directly to domain level forces, completing the force computation. Following the forces, the accelerations are calculated with the help of second law of motion.

Due to the symmetry conditions, the normal component of the accelerations at the boundaries is set to zero, ensuring that the normal component of the velocity remains stable over time. As a result, symmetry boundary conditions are applied to nodes on the mesh boundaries.

Then the velocities are updated according to the accelerations. During this phase a cut-off value is applied to prevent floating point errors near zero values. This introduces a performance cost while preventing untrustworthy mesh motion. It is followed by the update of the node positions.

Artificial viscosity terms are computed using displacement and velocity gradients, therefore requiring a low amount of conditional dependencies. Pressure and internal energy are updated to their current values by looping though the elements and putting them into temporary arrays. The equation of state shown in Equation is also computed at this section. This computation is then followed by the computation of a sound constant which helps us to determine the maximum timestep to be taken for the next step. Then, the volumes are updated to their previously calculated values while also applying a cut-off to each element.

B.2 Code Details

The code uses the concept of regions to support multiple material types, even though it is primarily designed to solve a single-material Sedov blast problem, though the code uses the concept of regions to support multiple material types. Within a single domain, there can be multiple regions, each representing a different material or part of the simulation space. This region-based implementation allows for array indirections in key computational kernels, optimizing memory usage by dynamically allocating memory based on the size of each region rather than making a large allocation for the entire domain. Another type of array indexing found in this simulation stems from the structure of the mesh, or the lack thereof. These arrays store indices that indicate how elements are connected to each other and to the nodes helping to explain the neighbors, traversing the mesh, applying boundary conditions etc.

The code uses various loops, including short loops over the 8 nodes of an element. A domain is a mesh partition made up of a rectangular collection of elements and their nodes. There can be multiple domains, each containing multiple elements. The application uses one material, but an index set is created for each domain to approximate real-life applications and access material properties.

The regions are implemented using an outer loop over the regions and an inner loop over the elements within each region. Each region applies the same material model as in the original LULESH code. However, these regions differ in size and have different (artificially introduced) computational costs.

The initial timestep is determined by the mesh resolution. To keep energy distribution consistent and ensure uniform shock wave location across mesh resolutions, the initial energy deposition is scaled to the mesh resolution. In the updated code, the shock wave is set to be about 1.0 cm from the origin at the simulation's end, serving as a reference point for comparing shock locations across different mesh resolutions.

The mesh matches the physical problem domain by dividing it into disjoint material elements that correspond to the material's spatial domain, tracking changes over space and time. Each element is a hexahedron in three-dimensional space, defined by eight surrounding nodes.

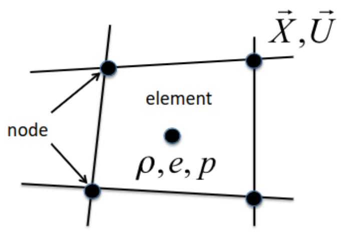


Figure B.1: Staggered Mesh

Thermodynamic variables like density ρ, internal energy e, and pressure p are represented at element centers, while kinematic variables such as displacement $\vec{X}$ and velocity $\vec{U}$ are at nodes, forming a potentially distorted element, as shown in Figure B.1. These thermodynamic variables are piece-wise constant, averaging over the entire element. Spatial gradients for solving the governing equations are computed using finite element approximation. The problem is evaluated by integrating the governing equations over time using an explicit time-stepping method with single-point quadrature for the mesh elements. The maximum allowable time step is determined by the Courant-Friedrichs-Lewy (CFL) condition, which ensures the stability of the simulation. Additionally, artificial viscosity is introduced to address numerical challenges, such as maintaining stability, and to accurately simulate energy-conserving and momentum-conserving equations.

An hourglass mode filter is used to address issues that arise from using single-point quadrature over hexahedral elements. Single-point quadrature under-integrates the elements, which improves runtime efficiency but reduces accuracy compared to other methods. Since fully integrating the elements is computationally expensive and can be less robust for large simulations, internal hourglass forces are applied to correct these inaccuracies. Consequently, at each iteration, the maximum allowable time step is calculated, followed by advancing the node variables and updating the elements based on the new node values.

B.3 LULESH Laptop Time-to-Solution Results

Type	**Stage 1**	**Stage 2**	**Stage 3**
Double Median	2.91	2.92	2.8
Mixed Median	3.11	2.67	2.7
Savings	-6.5%	8.5%	4.5%
Std. Dev.	3.8%	2.9%	3.1%

Table B.1: Achieved Savings With Mixed-Precision Versions on LULESH on Laptop with Inclusion Approach

Glossaries

hourglass mode Hourglass (HG) modes are nonphysical, zero-energy modes of deformation that produce zero strain and no stress. [1] . 37

[1]https://ftp.lstc.com/anonymous/outgoing/jday/hourglass.pdf