UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

GABRIEL FREYTAG

Improving Performance of Iterative Applications through Interleaved Execution of Approximated CUDA Kernels

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Advisor: Prof. Dr. Philippe O. A. Navaux
Coadvisor: Prof. Dr. Paolo Rech

Porto Alegre
April 2023

CIP — CATALOGING-IN-PUBLICATION

Freytag, Gabriel

Improving Performance of Iterative Applications through Interleaved Execution of Approximated CUDA Kernels / Gabriel Freytag. – Porto Alegre: PPGC da UFRGS, 2023.

166 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2023. Advisor: Philippe O. A. Navaux; Coadvisor: Paolo Rech.

1. Approximate computing. 2. Mixed-precision. 3. Half-precision. 4. Accuracy loss profile. 5. Energy efficiency. I. Navaux, Philippe O. A.. II. Rech, Paolo. III. Título.

ACKNOWLEDGMENTS

First and foremost, I would like to express my utmost gratitude to my beloved wife, Andréia Luisa Friske, for her unwavering support throughout this insane and tumultuous thesis. Secondly, I would like to thank my parents, Rosane and Silvério Freytag, for instilling in me early on the importance of education and striving for better opportunities. I would also like to thank my advisor, Philippe Navaux, for agreeing to guide me and always steering me in the right direction during this madness. I would also like to thank my co-advisor, Paolo Rech, for his assistance in researching an area outside my comfort zone, approximate computing. In addition, I would like to thank my in-laws, Ivanir and Arteno, for their support and partnership. Finally, I would like to thank my colleagues in the research group "Grupo de Processamento Paralelo e Distribuído" (GPPD). Thank you all for your help and support.

ABSTRACT

Approximate computing techniques, particularly those involving reduced and mixed precision, are widely studied in literature to accelerate applications and reduce energy consumption. Although many researchers analyze the performance, accuracy loss, and energy consumption of a wide range of application domains, few evaluate approximate computing techniques in iterative applications. These applications rely on the result of the computations of previous iterations to perform subsequent iterations, making them sensitive to precision errors that can propagate and magnify throughout the execution. Additionally, monitoring the accuracy loss of the execution in large datasets is challenging. Calculating accuracy loss at runtime is computationally expensive and becomes infeasible in applications with a considerable volume of data. This thesis presents a methodology for generating interleaved execution configurations of multiple kernel versions for iterative applications on GPUs. The methodology involves sampling the accuracy loss profile, extracting performance and accuracy loss statistics, and offline generating interleaved execution configurations of kernel versions for different thresholds of accuracy loss. The experiments conducted on three iterative applications of physical simulation in three-dimensional data domains demonstrated the capability of the methodology to extract performance and accuracy loss statistics and generate interleaved execution configurations of kernel versions with speedups up to 2 and reduction of energy consumption up to 60%. For future work, we suggest studying different optimization strategies for generating interleaved execution configurations of kernel versions, such as using neural networks and machine learning.

Keywords: Approximate computing. Mixed-precision. Half-precision. Accuracy loss profile. Energy efficiency.

Melhorando o Desempenho de Aplicações Iterativas por meio da Execução Intercalada de Kernels CUDA Aproximados

RESUMO

As técnicas de computação aproximada, particularmente aquelas envolvendo precisão reduzida e mista, são amplamente estudadas na literatura para acelerar aplicações e reduzir o consumo de energia. Embora muitos pesquisadores analisem o desempenho, perda de precisão e consumo de energia de uma ampla gama de domínios de aplicação, poucos avaliam técnicas de computação aproximada em aplicações iterativas. Essas aplicações dependem do resultado dos cálculos das iterações anteriores para realizar iterações subsequentes, tornando-as sensíveis a erros de precisão que podem se propagar e amplificar durante a execução. Além disso, monitorar a perda de precisão da execução em grandes conjuntos de dados é desafiador. Calcular a perda de precisão em tempo de execução é computacionalmente caro e se torna inviável em aplicações com um volume considerável de dados. Esta tese apresenta uma metodologia para gerar configurações de execução entrelaçadas de múltiplas versões de kernel para aplicações iterativas em GPUs. A metodologia envolve amostrar o perfil de perda de precisão, extrair estatísticas de desempenho e perda de precisão, e gerar offline configurações de execução entrelaçadas de versões de kernel para diferentes limiares de perda de precisão. Os experimentos realizados em três aplicações iterativas de simulação física em domínios de dados tridimensionais demonstraram a capacidade da metodologia de extrair estatísticas de desempenho e perda de precisão e gerar configurações de execução entrelaçadas de versões de kernel com speedups de até 2 e redução do consumo de energia de até 60%. Para trabalhos futuros, sugerimos estudar diferentes estratégias de otimização para gerar configurações de execução entrelaçadas de versões de kernel, como o uso de redes neurais e aprendizado de máquina.

Palavras-chave: Computação aproximada. Precisão mista. Meia precisão. Perfil de perda de acurácia. Eficiência energética..

LIST OF ABBREVIATIONS AND ACRONYMS

AC	Approximate Computing
AI	Artificial Inteligence
AVX	Advanced Vector Extensions
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient
CPU	Central Processing Unit
CSV	Comma-Separated Values
CUDA	Compute Unified Device Architecture
DNN	Deep Neural Network
DSP	Digital Signal Processors
EP	Embarrassingly Parallel
FA	Full Adder
FMA	Fused Multiply-Add
FP	Floating-Point
FPU	Floating-Point Unit
GB	Gigabyte
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GPU	Graphics Processing Unit
HPC	High Performance Computing
INT	Integer
ISA	Instruction Set Architecture
J	Joules
LBM	Lattice Boltzmann Method

LU	Lower-Upper Decomposition
LULESH	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
MAE	Mean Absolute Error
MIMD	Multiple Instruction Multiple Data
MSE	Mean Squared Error
NPU	Neural Processing Unit
NUMA	Non-Uniform Memory Access
NVCC	Nvidia CUDA Compiler
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessors
SMP	Symmetric Multiprocessing
SP	Scalar Penta-diagonal
SSE	Streaming SIMD Extensions
TOQ	Target Of Quality
W	Watts

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

It's no secret that the demand for computing power is rapidly increasing. As humanity moves towards digitization and interconnected intelligent systems, the need for storage, processing, and data analysis continues to grow significantly. Despite significant improvements in computational power and storage capacity generation after generation, the demand for storage and computing capacity still vastly exceeds the available resources, including budget resources (MITTAL, 2016). Moreover, existing and new application areas will require computation with energy efficiency orders of magnitude higher than the current state of the art (SHALF, 2020). Therefore, achieving higher computational power benchmarks economically will require more efficient hardware, algorithms, and methods (THOMPSON et al., 2020).

In recent years, approximate computing has become a popular and promising approach for improving the efficiency and performance of computer systems (XU; MYTKOWICZ; KIM, 2015). The basic idea is to trade off computational accuracy for better system performance and reduced power consumption (MITTAL, 2016). By designing hardware and software to tolerate errors and allowing minor errors or variations in the output of computations, approximate computing can reduce computational power requirements. This approach can achieve faster computing times and lower energy consumption than traditional methods, resulting in cost savings.

Reduced precision and mixed precision are among the most popular approximate computing techniques. These techniques involve using fewer bits to represent floating-point numerical values. For example, instead of using 64 bits to represent a number, we can use only 32 bits. By doing so, we can reduce the amount of memory needed to store a value and the amount of data that needs to be transferred and processed. This can lead to reductions in both computation time and energy consumption (JIN et al., 2017).

To prevent compatibility issues and calculation inaccuracies, IEEE standardized the representation of floating-point numbers in binary in computer hardware and software. The IEEE 754 (IEEE..., 1985) standard defines a consistent and universally recognized format for representing floating-point numbers. This allows for accurate and reliable calculations across different systems and programming languages. The revisions of the IEEE 754-2008 (IEEE..., 2008) specify several floating-point formats, including the half-precision, single-precision, double-precision, and quadruple-precision formats. These formats use 16, 32, 64, and 128 bits to represent floating-point numbers in binary.

The main difference between the floating-point formats is the number of bits used to represent the number, which influences the precision and range of numbers that can be represented. For example, the single-precision format uses 32 bits to represent a floating-point number. It has a precision of about seven decimal digits, while the double-precision format uses 64 bits and has a precision of about 15 decimal digits. The more bits allocated to the mantissa, the more precise the number can be represented. However, the decimal value of 0.1 cannot be represented precisely in binary using any finite number of bits (GOLDBERG, 1991). Therefore, any binary representation of 0.1 will be an approximation. For instance, in single-precision format, 0.1 is approximated as 0.100000001490116119384765625, while in double-precision format, it is approximated as 0.1000000000000000055511151231257827021181583404541015625.

When working with lower-precision floating-point formats, such as in reduced and mixed-precision approximate computing techniques, errors can occur due to the representation of decimal numbers in binary and casting between different floating-point formats. Casting a number to a lower-precision format can result in rounding or truncation, leading to the loss of some digits. Conversely, casting to a higher-precision format may add additional zeros, but the precision of the original number remains unchanged. As a result, the accuracy of application execution is directly impacted by the number of castings between different floating-point formats.

Another factor affecting the accuracy of floating-point calculations is the difference in error introduced by different arithmetic operations. For example, subtracting two numbers that are very close in value can result in a loss of precision due to the limited number of significant digits in the floating-point format. Multiplying two numbers with very different magnitudes can also result in inaccurate results due to overflow or underflow. In general, addition and subtraction are less prone to errors than multiplication and division because addition and subtraction do not amplify errors to the same extent as multiplication and division (GOLDBERG, 1991).

Therefore, controlling and limiting errors when working with floating-point numbers and approximations is crucial because they can lead to inaccurate results and significantly impact the overall accuracy of application execution. Consequently, it is essential to carefully consider the precision and accuracy requirements of the specific application and use appropriate techniques to minimize the introduction and accumulation of errors. This may involve using higher precision formats, proper arithmetic operations, and error correction and compensation techniques. Doing so makes improving the accuracy and re-

liability of floating-point computations possible, which is essential for many applications in science, engineering, and other domains.

1.1 Motivation

Despite being widely explored in the literature, techniques and tools for fine-tuning the precision of floating-point operations are primarily focused on non-iterative applications. The main application domains are computer graphics, machine learning, signal processing, finance and numerical computing (BAEK; CHILIMBI, 2010; RUBIO-GONZ et al., 2013; RUBIO-GONZALEZ et al., 2016; KHUDIA et al., 2015; CHERUBIN et al., 2020; ROY et al., 2014), robotics, compression (KHUDIA et al., 2015), clustering and classification, time series, regression problems (ZHANG et al., 2014). The selection of these applications is typically based on their real-world workload representativeness and their notable resilience to floating-point errors (MITTAL, 2016).

Although some works explore iterative scientific applications that are equally representative, such as the CG (Conjugate Gradient), EP (Embarrassingly Parallel), and FP (Fourier Transform) kernels and the SP (Scalar Penta-diagonal) and LU (Lower-Upper Gauss-Seidel) pseudo-applications of the suite of NAS parallel benchmarks (RUBIO-GONZ et al., 2013; RUBIO-GONZALEZ et al., 2016; SAMPSON et al., 2011; GRAILLAT et al., 2019), jetEngine and turbine (CHIANG et al., 2017; MENON; LAM, 2019), Lattice Boltzmann Method (LBM) (HO et al., 2017), iterative applications are significantly more sensitive to floating-point errors, making it considerably more challenging to tune the precision of floating-point operations.

Iterative applications are algorithms that repeat a specific set of instructions multiple times until a particular condition is met (BU et al., 2010). This technique is used for problems that cannot be solved analytically or when the complexity of the solution to a problem makes it unfeasible to compute in a single step (CARSON; STRAKOŠ, 2020). In this technique, the output of one iteration becomes the input for the next. While this makes the applications more efficient, it also means that any approximations made in one iteration are carried over to the next. These errors can then be amplified, leading to inaccuracies in the final result. Therefore, it is essential to carefully monitor and validate the output of the iterations to ensure the accuracy of the final result (ZHANG et al., 2014).

Ensuring output quality in iterative applications can be challenging due to their repetitive nature and typical use of multidimensional data domains. To control output

quality using approximation techniques, a standard method is monitoring accuracy loss during runtime (MITTAL, 2016). However, monitoring accuracy loss during runtime in iterative applications may be impractical due to the volume of data involved. In scientific simulation applications, for example, to calculate accuracy loss in each iteration on a 2D data domain with 512 cells on the x-axis and y-axis, 262144 values must be compared against the correct values. Using a 3D data domain with 512 cells on each side increases the number of cells to 134217728. Since iterative scientific simulation applications can have significantly larger data domains, the computational cost of checking for accuracy loss at runtime can easily exceed the application's running cost.

1.2 Objectives

Our research focuses on developing an efficient method of accelerating the execution of applications using approximate computing techniques. We are interested in iterative applications, where a given set of operations is applied recurrently over a large dataset. The execution of iterative applications demands great computational power due to the number of operations and the volume of data on which the operations are performed (BU et al., 2010; CARSON; STRAKOŠ, 2020). However, these characteristics make GPUs ideal for accelerating the execution of iterative applications since their architecture allows the simultaneous execution of operations on an extensive dataset. Therefore, our work will focus on studying further application acceleration through approximate computing techniques on GPUs.

In general, approximations of applications on GPU devices are achieved by fine-tuning the floating-point precision of operations or by using less precise architecture-specific methods (SAMADI et al., 2014a; SAMADI et al., 2014b; LAGUNA et al., 2019). The aggressiveness of the approximations is relative to the user-determined Target Output Quality (TOQ). For each new TOQ, a new analysis is performed to determine which less accurate methods or floating-point precisions will be used in each operation to ensure that the TOQ is respected. Although this approach enables significant performance improvements, the tuning search space grows significantly as larger and more complex applications are used. One way to avoid the cost of recurrent tuning is to use multiple code versions with different accuracies, varying the selection or execution order of the kernel versions at runtime according to how close the execution accuracy is to the TOQ.

This research explores the hypothesis that the interleaved execution of multiple

approximated kernel versions, based on their accuracy loss profile, can improve the performance of iterative applications. The accuracy loss profile refers to the variation of loss along the execution of the application. Typically, multiple versions of kernels with different accuracies are scaled based on measurements of accuracy loss at runtime or kernel calibrations performed before or during execution (LAGUNA et al., 2019; KOTIPALLI et al., 2019; HO; SILVA; WONG, 2021). However, checking accuracy at runtime in larger applications, such as scientific simulations that use 2D or 3D data domains, can be impractical. Our proposed solution is to analyze the accuracy loss profiles of running multiple CUDA kernel versions with different accuracies to generate an execution configuration that alternates the execution of different kernel versions. This configuration prioritizes the execution of the fastest version respecting the user-defined TOQ.

This work's main contributions are:

1. A new methodology for approximate computing in iterative applications in GPU architectures is proposed. Based on the accuracy profile of multiple approximated kernel versions, an interleaving execution configuration of the kernels can be generated entirely offline. This can be done for an arbitrary number of TOQs if the set of kernels and data inputs is the same, without additional measurements.
2. The evaluation of the performance and energy efficiency of the proposed methodology on three well-known iterative applications of physical simulation in three-dimensional domains.
3. The main insight from the thesis: interleaved execution of multiple approximated kernel versions based on their accuracy loss profile can improve the performance and energy efficiency of iterative applications.

1.3 Document Organization

The remaining thesis is organized as follows. Chapter 2 provides a solid understanding of state-of-the-art techniques in approximate computing. We differentiate between the different levels of existing approximate computing techniques and dissect the approaches used in the literature for using approximate computing techniques at the application level on GPU architectures. Chapter 3 introduces the concept of interleaved execution of multiple versions of approximate kernels in GPU architectures. Chapter 4 presents the methodology developed in this thesis for extracting the accuracy loss profile

of multiple approximate kernel versions and generating execution configurations based on accuracy loss and performance statistics. This is done in a fully offline manner. Chapter 5 presents the experimental results of execution configurations generated from the methodology presented in Chapter 4 for different iterative physical simulation applications using different problem sizes and accuracy loss thresholds. Chapter 6 analyzes and discusses the performance and energy efficiency of the results presented in Chapter 5. Finally, in Chapter 7, we offer the final considerations of our work and suggestions for future work.

2 BACKGROUND AND STATE OF THE ART

Approximate Computing (AC) refers to a range of techniques that trade off the precision of results for improvements in application performance and reduced energy consumption (MITTAL, 2016). By omitting specific calculations, reducing hardware resource usage, using faster but less accurate hardware, or optimizing computational methods, these techniques considerably reduce the computational power required to run applications. This, in turn, enhances application performance and reduces energy consumption. However, omitting or approximating certain operations during execution can lead to inaccurate results and, in some cases, completely corrupt the final output of the application.

Approximate computing techniques are tools and methods that explore the intrinsic resilience of applications to numerical errors. However, the influence of a given technique on the final result can vary significantly from one application to another due to the inherent characteristics of each application. While a particular technique may only minimally change the result of one application, it could result in a significant divergence between the expected value and the value obtained in another application. Intrinsic resiliency refers to an application's ability to produce acceptable output despite some of its underlying calculations being incorrect or approximate (CHIPPA et al., 2013).

However, an application's resilience to errors does not guarantee an acceptable result when using any applicable AC technique. Though such resilience can naturally reduce the impact of approximations introduced by these techniques, the final result with specific techniques may still be unacceptable (CHIPPA et al., 2013). This is due to the variation in the approximation level introduced by each technique in an application's operations, which can even vary from one application to another (XU; MYTKOWICZ; KIM, 2015). Despite the negative impact of AC techniques on an application's final result, there are established ways to explore the potential for improving performance and energy consumption while minimizing impacts on result accuracy.

Computing has entered an era of complex systems that requires sophisticated algorithms to deliver good enough answers quickly, at scale, and with energy efficiency, and approximation is often the only way to meet these goals (Xu, 2016). This Chapter will detail the different levels of Approximate Computing techniques, with a particular focus on the more general application-level techniques and AC in GPU architectures.

2.1 Circuit-Level Approximation

The circuit-level approximation is a technique for designing circuits that perform approximate computations instead of exact computations. This is achieved by introducing approximations into the circuit design, which reduces the computations' accuracy and the energy consumed by the circuits (MITTAL, 2016). Most proposals for approximate circuits modify the original function of circuits to balance accuracy and energy (XU; MYTKOWICZ; KIM, 2015). Since addition and multiplication are critical arithmetic operations, most research has focused on approximating adder and multiplier circuits.

Gupta et al. (2011) propose reducing logic complexity as an alternative to voltage scaling to maximize the relaxation of numerical accuracy. The authors suggest using various imprecise or approximate Full Adder (FA) cells with reduced complexity at the transistor level to design approximate multi-bit adders. Their technique results in significantly shorter critical paths, enabling voltage scaling. The proposed approximate arithmetic units are then used to design architectures for video and image compression algorithms. The authors evaluate these designs and demonstrate their efficacy through simulations, which show significant power and area savings and an insignificant loss in output quality compared to other implementations.

Kim, Zhang and Li (2013) propose an adder with a carry skip scheme that achieves higher carry prediction accuracy, enabling faster addition operations or reduced energy dissipation by lowering the supply level. This technique leverages information from less significant input bits in a parallel through-carrying prediction, significantly improving error rate and critical path delay. The authors claim that their adder design is flexible, allowing low-overhead error correction logic to be included for error-free operations at the cost of one more clock cycle. The proposed design is faster and more energy-efficient than traditional adders and has minor approximation errors in the training processes of a neuromorphic character recognition chip using unsupervised learning.

Liu, Han and Lombardi (2014) proposed a multiplier design for high-performance Digital Signal Processing (DSP) applications with lower power consumption and a shorter critical path than traditional multipliers. This design uses an approximate adder with limited carry propagation to the nearest neighbors for fast partial product accumulation. The design also achieves different levels of accuracy through a configurable error recovery using different numbers of Most Significant Bits (MSBs) for error reduction. Although there are errors in the multiplier, most of them are not significant in magnitude. By imple-

menting the multipliers, the authors showed that an appropriate error recovery results in the proposed approximate multiplier achieving processing accuracy similar to traditional exact multipliers but with significantly improved power and performance.

2.2 Architecture-Level Approximation

In addition to approximating circuit designs, researchers have explored approximation in essential components of computer systems such as processors, memory, and storage. Computer architects aim to balance performance and energy efficiency under various constraints imposed by a given technology, such as chip area and processors' power, balance density (cost per bit), and performance on memory and storage (XU; MYTKOWICZ; KIM, 2015; MITTAL, 2016). However, simultaneously improving these components' performance, energy efficiency, and density is challenging because improving one often sacrifices the others (XU; MYTKOWICZ; KIM, 2015).

Venkataramani et al. (2013) propose a quality programmable processor that codifies the notion of quality in the Instruction Set Architecture (ISA). The ISA contains instructions associated with quality fields that specify the accuracy level required during the execution of the instructions. This approach allows the control of instruction execution accuracy and dramatically enhances the scope of approximate computing, making it applicable to more significant portions of programs. While the micro-architecture of a quality programmable processor contains hardware mechanisms that translate instruction-level quality specifications into energy savings, it can also expose the actual error incurred during the execution of each instruction back to the software. Results show that using instruction-level quality specifications can lead to significant energy savings with virtually no loss in application output quality and only a modest impact on output quality.

Esmaeilzadeh et al. (2012) propose a learning-based approach for accelerating approximate programs through the Parrot transformation. This program transformation selects and trains a neural network to mimic a region of imperative code. During the learning phase, the compiler replaces the original code with an invocation to a Neural Processing Unit (NPU), a low-power accelerator that speeds up machine learning algorithms. The NPU is tightly coupled to the processor pipeline to accelerate small code regions. The authors define a neural network programming model that allows programmers to identify code that can produce imprecise but acceptable results. The approximable code regions can then be offloaded to NPUs, executing it faster and more energy-efficiently

than the original code. This approach can significantly improve performance and energy consumption with an acceptable quality loss.

Palframan, Kim and Lipasti (2014) presented a precision-aware soft error protection scheme for the GPU execution logic and register file. The system combines selective gate hardening, an inexpensive checker circuit, and precision-aware encoding to improve soft-error resilience with low overhead. The authors extended this approach to integer (INT) computations by modifying the architecture and treating integers similarly to floating-point numbers. They showed that precision-aware protection has low overhead and minor error magnitudes in the event of a soft error compared to approaches that do not target error magnitudes. The technique significantly reduces errors while maintaining a low area overhead compared to a traditional approach with the same area overhead.

Sampson et al. (2014) propose a mechanism for storing data approximately in applications, demonstrating that it can improve the performance, lifetime, or density of solid-state memories. The authors suggest two mechanisms. The first mechanism allows errors in multilevel cells by reducing the number of programming pulses used to write the cells. The second mechanism mitigates wear-out failures by mapping approximate data to blocks that have exhausted their hardware error correction resources, thus extending the memory's endurance. Simulations show that writes with reduced precision in multilevel phase-change memory cells are faster, and using failed blocks improves the array lifetime with an acceptable quality loss.

2.3 Application-Level Approximation

Several software-level techniques and approaches have been explored to improve various applications' performance and energy efficiency (XU; MYTKOWICZ; KIM, 2015). These techniques include approximate languages, approximate compilers, and optimization strategies for reducing computational work. By decreasing the precision of computations, application-level approximations can significantly improve the runtime and lower power consumption without the need for specialized hardware, albeit at the cost of some accuracy loss in the output of applications (MITTAL, 2016).

2.3.1 Popular Techniques

The most widely used approximate computing techniques for improving application performance and reducing energy consumption aim to efficiently exploit resources available in modern computing systems (MITTAL, 2016). These techniques involve reducing system resource usage or minimizing high computational cost tasks for executing applications. Other techniques diversify the use of available computational resources, using those with lower computational costs or distributing the workload among a more significant number of resources.

One of the most popular approximate computing techniques focused on reducing system resource usage is loop perforation. This general application-level approximation technique trades the accuracy of the output for performance by transforming loops to execute only a subset of their iterations (HOFFMANN et al., 2009). This technique aims to reduce the computational work of applications by filtering out loops whose perforation produces unacceptable behaviors and identifying loops where the perforation makes more efficient yet accurate computations (SIDIROGLOU-DOUSKOS et al., 2011). The perforation of loops is usually conducted through space exploration algorithms, statistical analysis, perforation at regular intervals, or even randomly.

Other techniques, such as memoization, load value approximation, task skipping, and memory access reduction, also aim to reduce the use of system resources (MITTAL, 2016; XU; MYTKOWICZ; KIM, 2015). Like loop perforation, skipping task execution and memory access reduces the number of operations performed by applications. However, instead of proceeding with subsequent operations, these techniques replace the results of skipped computations with previously computed or stored data. For example, memoization stores the result of functions so they can be reused later by other similar functions or functions with identical input data (RAHIMI; BENINI; GUPTA, 2013; KERAMIDAS; KOKKALA; STAMOULIS, 2015). On the other hand, the load value approximation technique estimates the load value in cases of cache loading failure, which avoids searching in different levels of cache or main memory and reduces significant latency (MIGUEL; BADR; JERGER, 2014; SUTHERLAND; MIGUEL; JERGER, 2015).

These techniques do not require additional or specialized resources, making them applicable to various computing systems. Their implementation is relatively simple, usually involving the omission of specific procedures or the storage and approximation of values. However, modern computing systems consist of a heterogeneous mix of components

and resources with different purposes and utilities that can also be exploited to optimize application performance and energy consumption (MITTAL, 2016; XU; MYTKOWICZ; KIM, 2015).

One example is the different floating-point representation formats present in modern systems. The most popular are the half, single, and double-precision binary formats (IEEE..., 2008). While larger formats offer more precise representations of floating-point values (with more decimal digits), larger representation sizes require more space in system memory. The information about each value increases with larger sizes, making reading and transferring the data more time-consuming. For instance, a value in single-precision floating-point format uses 32 bits of memory. In contrast, in double-precision format, it uses twice as much space (64 bits) and twice as much memory bandwidth during transfers compared to the single-precision representation.

During the development process of most applications, it is common practice to use a floating-point representation format with higher precision than what is minimally necessary to represent the data (RUBIO-GONZ et al., 2013). This is because correctly sizing the minimum format needed to represent each variable in an application accurately can be complex, and using a larger, more accurate format for the application as a whole becomes the more viable choice (MITTAL, 2016; XU; MYTKOWICZ; KIM, 2015). However, while larger floating-point formats can provide a margin of safety in executing floating-point operations, they can significantly degrade performance and energy consumption, especially in compute- and memory-intensive applications (RUBIO-GONZ et al., 2013).

Reduced and mixed precision is one of the most popular techniques used to reduce the amount of data in computations and increase the performance and efficiency of computations in applications. Based on the observation that the use of higher precision floating-point formats than necessary is a common practice in application development, the idea of reduced and mixed precision techniques is to use floating-point representations with smaller sizes for variables and arithmetic operations, trading off the accuracy of the output for improved performance and energy efficiency (RUBIO-GONZ et al., 2013).

Due to the complexity of existing numerical programs, manually tuning their floating-point precision may be prohibitively expensive or even impossible (RUBIO-GONZ et al., 2013). To overcome this, precision tuning tools were developed to automate the process of precision tuning in applications.

2.3.2 Popular Automated Tools for Precision Tuning

Sampson et al. (2011) propose using type qualifiers to declare data that may be subject to approximate computation, isolating them from the parts that must be precise. Using these types, the system can automatically map approximate variables to low-power storage, use low-power operations, and apply more energy-efficient algorithms provided by the programmer. This approach statically guarantees the isolation of the precise parts from the approximate ones, allowing programmers to explicitly control the flow of information from approximate to accurate data. This eliminates the need for dynamic checks and further improves energy savings. The authors developed an extension to Java that adds approximate data types named EnerJ. They show that the extension is expressive and compelling, as a small number of annotations can lead to significant energy savings with an acceptable accuracy loss.

Rubio-gonz et al. (2013) present Precimonious, a dynamic program analysis tool designed to assist in tuning the precision of floating-point programs. The tool searches for floating-point program variables and attempts to lower their precision based on accuracy constraints and performance goals. It recommends a type of instantiation that uses lower precision while producing an accurate enough result. The authors evaluated Precimonious on several widely used functions from the GNU Scientific Library, two NAS Parallel Benchmarks, and three other numerical programs. The experimental results for these functions and applications demonstrate that the tool can reduce precision, leading to significant performance improvements.

Schkufza, Sharma and Aiken (2014) present a method for overcoming the complicated semantics of floating-point instruction sets. Compilers often treat floating-point optimization as a stochastic search problem, which forces them to preserve programs as written. The authors show that it is possible to produce high-performance optimizations specialized to the range of live inputs of a code sequence and the desired precision of its live outputs by repeatedly applying random transformations to floating-point binaries produced by a production compiler or by hand. Furthermore, they demonstrate that the combined result of tens of millions of arbitrary transformations is sufficient to create novel and often non-obvious optimizations that could otherwise be missed. Experiments show that the method can generate reduced-precision implementations with significant performance improvements.

Chiang et al. (2017) developed an automated tool called FPTuner, which rigor-

ously allocates precision based on formal analysis via Symbolic Taylor Expansions and error analysis based on interval functions. This tool generates and solves a quadratically constrained program to obtain a precision-annotated version of a given expression, automatically introducing all the required precision casting operations. FPTuner also offers flexible control over precision allocation using constraints to restrict the number of high-precision operators and group operators to allocate the same precision to facilitate vectorization. To evaluate the tool, the authors tuned several benchmarks. They measured the proportion of lower-precision operators assigned for increased error thresholds and energy consumption reduction from executing mixed-precision tuned code. The results showed significant energy savings using mixed-precision tuning.

2.4 State of the Art

Kotipalli et al. (2019) present AMPT-GA, an automated mixed-precision floating-point tuning framework for GPU applications. The framework selects application-level data precisions to maximize performance while adhering to user-defined accuracy constraints. It combines static analysis for casting-aware performance modeling and dynamic analysis for modeling and enforcing precision constraints. The authors further improve the optimizations performed by the framework with application-aware mutations using a genetic algorithm-based search function. Experimental results indicate that the framework improves the performance efficiency of a LULESH application implementation by 14-63% more than the prior state-of-the-art approach called Precimonious.

Laguna et al. (2019) have presented GPUMixer, a tool for tuning mixed-precision floating-point calculations on scientific applications running on GPU architectures. The tool utilizes performance-driven approximations and introduces a new static analysis method that identifies sets of operations that minimize the conversion between floating-point types. Additionally, the authors propose shadow computation analysis to estimate the relative error introduced by using mixed precision on GPUs. Experimental results indicate that the tool can improve the performance of GPU-based applications by up to 46.4%.

Samadi et al. (2014b) present a framework, SAGE, which uses an automated approach to generate a set of CUDA kernels with varying levels of approximation. The framework iteratively selects the kernels that provide the best performance while adhering to a user-defined output quality target. SAGE consists of two main steps: an offline compilation step and a runtime kernel management step. During the offline compilation step,

the input code is analyzed to identify opportunities to trade accuracy for performance. This analysis automatically generates approximate CUDA kernel versions using three GPU-specific optimizations. These optimizations systematically detect and skip expensive GPU operations. Each optimization has its tuning parameters, which the framework uses to manage the performance-accuracy trade-off.

The runtime management step dynamically selects the approximate kernel. It consists of three parts: tuning, preprocessing, and optimization calibration. Tuning uses a greedy algorithm to find the fastest kernel with better quality than the target output quality defined by the user. Preprocessing ensures that the data needed by the approximate kernels are ready before execution. During calibration, the framework monitors the accuracy and performance dynamically. If the output quality does not meet the target, it chooses a less aggressive approximate kernel to improve the output quality. The results demonstrate that the framework improved performance by up to 2.5x on average with less than 10% quality loss in ten machine learning and image processing applications on an NVIDIA GTX 560 GPU.

Samadi et al. (2014a) introduce Paraprox, a pattern-based approximation framework for data-parallel applications. The framework transparently approximates data-parallel patterns on OpenCL or CUDA kernels by creating a parameterized approximate kernel tuned at runtime to maximize performance while adhering to a target output quality specified by the user. The approximations are made by identifying common computation idioms found in data-parallel programs (such as Map, Scatter/Gather, Reduction, Scan, Stencil, and Partition) and replacing them with approximations.

Once the approximated kernels are generated, the framework tunes a set of variables to control the accuracy and performance of the approximations. If the target output quality specified by the user is not met, the framework performs a new parameter tuning or uses a less aggressive approximated kernel for subsequent executions. Experimental results indicate that the framework improves the performance of 13 soft data-parallel applications by up to 2.7x on an NVIDIA GTX 560 GPU and 2.5x on an Intel Core i7 quad-core processor compared to accurate execution, with an accuracy loss of no more than 10%.

Ho, Silva and Wong (2021) present a framework for dynamically mixing floating-point precision in GPU applications. The framework consists of two steps. The first step generates three kernel versions of a given CUDA application with different levels of approximation using FP32 and FP16. Based on a user-defined target output quality,

Table 2.1 – Applications used by previous works and their respective domains.

Application	Domain
Histogram, Image Binarization, Dynamic Range Compreesion, Mean Filter, Gaussian Smoothing, Gamma Correction, Gaussian Filter, Image Denoising, Billateral Filter, MRI-q	Image Processing
Kmeans, Naive Bayes, SVM, Fuzzy Kmeans, Means Shift, Kernel Density Estimation, Back Propagation	Machine Learning
Matrix Multiply, Newtonraph, Arclength	Numerical Analysis
LavaMD, CoMD, CP	Molecular Dynamics
LULESH, CFD	Fluid Dynamics
Quasirandom Generator, BoxMuller	Statistics
Blackscholes / Cumulative Frequency Histograms / Nbody / Inversek2j	Financial / Signal Processing / Physics / 3D Gaming

the framework selects only two kernels for execution, one with higher and one with lower accuracy, to ensure that neither exceeds the target output quality. The second step consists of merging the two versions into one and finding the degree of approximation needed to maintain the target output quality.

The approach proposed by the authors differs from most state-of-the-art methods in how the approximation is performed. Instead of searching for a subset of variables where the precision of the floating-point representation can be lowered, the authors try to change the precision of a subset of data elements of the application at runtime. To do this, they assign groups of threads to perform the computations using the higher-precision version of the kernel or the lower-precision one. Experimental results show that this approach can improve performance by up to 2 times with an accuracy loss of up to 10%.

Table 2.1 shows that most applications used in these works are non-iterative. Although some iterative applications and applications that use multidimensional data domains are used, most applications iterate only once over the data domain and do not represent applications with significant computational demand. There may be several reasons for this, including the complexity involved in tuning the precision and monitoring the accuracy loss, especially during runtime, in iterative applications.

All works follow a two-stage approach, where the first offline approach generates different approximated kernel versions or performs different approximations for every new TOQ. The second stage involves online calibration and accuracy loss monitoring.

Using these approaches in iterative applications can result in significant overhead during execution. However, our approach is fully offline and uses multiple static approximation versions of CUDA kernels for different TOQs defined by users. By extracting the kernel versions' accuracy loss profile, we can generate interleaved execution configurations for arbitrary TOQs as long as the same input data is used. This way, we can remove the overhead of calibration and accuracy loss monitoring, which can introduce a significant overhead for the execution, especially in iterative applications.

3 INTERLEAVED EXECUTION OF APPROXIMATED CUDA KERNELS

In this Chapter, we introduce the concept of executing multiple approximate versions of CUDA kernels in an interleaved manner to enhance the performance of interactive applications. We begin by discussing how to measure and manage the accuracy loss of approximations, especially in iterative applications with multidimensional data domains. Next, we explore modern heterogeneous GPU architectures with hardware support for approximate computing. We also discuss applying application-level approximate computing techniques to create multiple approximate kernel versions with different levels of accuracy. By executing these versions in a way that respects the Target Output Quality (TOQ) and maximizes the execution of faster kernel versions, we can improve the performance of iterative applications.

3.1 Measuring Accuracy Loss

There are several metrics to measure the error of approximate computations. The most commonly used metrics for numerical problems are Mean Absolute Error (MAE), Mean Squared Error (MSE), and Relative Error (MITTAL, 2016). MAE measures the absolute difference between the approximate and exact results, MSE measures the squared difference, and Relative Error measures the difference between the approximate and exact results relative to the exact computation as a percentage. These metrics are used to quantify the loss of accuracy in approximate computing, and the choice of metric depends on the specific application.

The Frobenius norm is a method for measuring accuracy loss in approximate computing. It is a matrix norm that measures the difference between a matrix computed using exact arithmetic and a matrix computed using approximate arithmetic. The Frobenius norm is defined as the square root of the sum of the absolute squares of the matrix's elements. For a matrix A, the Frobenius norm is given by the following formula:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

Here, a_{ij} denotes the element in the ith row, jth column of A, and m and n are the number of rows and columns in A, respectively. The Frobenius norm is a measure of a matrix's "size". It is a valuable tool for quantifying the magnitude of a matrix and

Figure 3.1 – Code example for computing the accuracy loss using the Frobenius norm.

```
1 double sumSqC, sumSqA;
2
3 for (int i = 0; i < N; i++)
4     sumSqC += pow(Aapprox[i] - A[i], 2);
5     sumSqA += pow(A[i], 2);
6
7 double res = sqrt(sumSqC) / sqrt(sumSqA);
```

Source: The Authors

has a wide range of applications in various fields of mathematics and science. As our primary focus is on iterative scientific applications that typically involve multidimensional matrices, we will use the Frobenius norm to measure accuracy loss in our work.

Figure 3.1 provides an example of using the Frobenius norm to calculate accuracy loss in a vector. The process involves first calculating the sum of the differences between the approximate computation and the exact computation for each element in the vector (i.e., the absolute error) and the sum of the exact vector. With these sums, the square root of the sum of the absolute error (i.e., the sum of the difference between the elements in the approximate vector and the elements in the exact vector) and the sum of the exact vector is calculated. The resulting value is then divided by the square root of the sum of the exact vector to obtain the relative error (or relative accuracy loss). This yields a value between 0 and 1, where 0 indicates no accuracy loss, and 1 indicates the complete dissimilarity between the approximate and exact computations.

3.2 Management of Accuracy Loss

Managing loss of accuracy is a crucial step in developing approximate applications. While some techniques may have acceptable accuracy loss in specific applications, others may exceed the limit and invalidate the final result. However, this does not necessarily mean the technique cannot be used in these applications. By incorporating a loss management method, the loss can be prevented from exceeding an acceptable threshold.

One popular approach is to set predetermined loss of accuracy limits during the development of the approximate application. The method or technique adjusts itself based on the predetermined loss limit, ensuring the final result is within the acceptable loss limit. The aggressiveness of the approximation is relative to the limit: wider limits allow for a more aggressive approach, while fairer limits allow only milder approaches.

It is also possible to use a predetermined loss of accuracy threshold as a barrier be-

yond which no approximation is performed. In this case, the approximations are executed only while the application's loss of accuracy is below this limit. Operations are executed without approximation when the limit is reached to prevent further loss of accuracy.

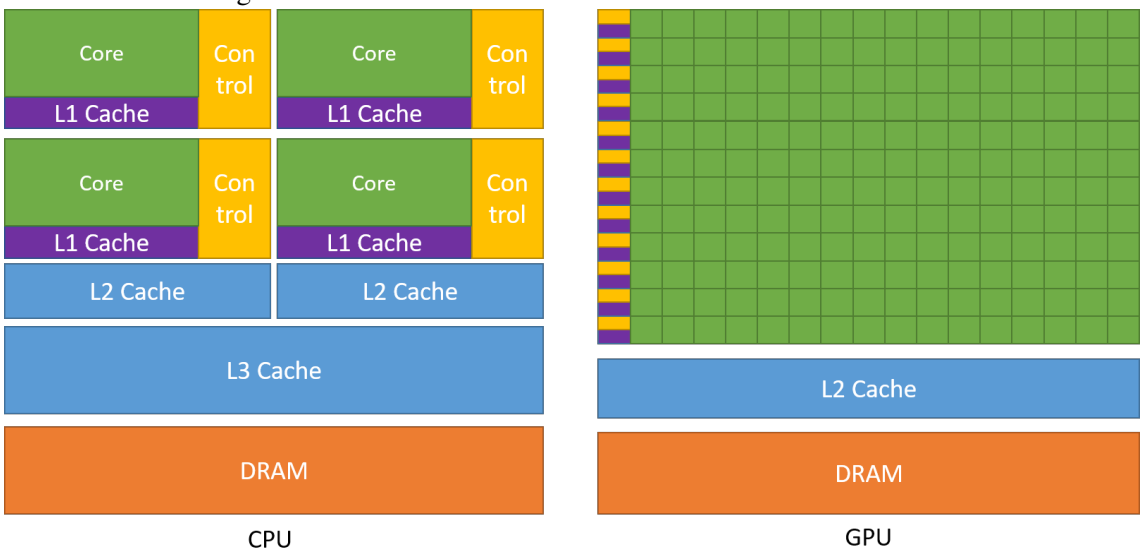
Correction and compensation methods are other popular approaches. Mathematical methods are usually employed to correct or compensate for the loss of accuracy introduced by approximation techniques.

3.3 Heterogeneous Architectures

Modern computing systems comprise various architectures developed to perform specific tasks optimally. One example of such an architecture is the Graphics Processing Unit (GPU), initially designed to process and render images. Today, GPUs are used in computationally demanding tasks, such as executing physical and mathematical simulation models (e.g., climate prediction and fluid dynamics) and training and inference of Artificial Intelligence (AI) algorithms.

Despite being capable of being used in various applications, GPUs were designed to perform specific tasks efficiently. Therefore, they cannot perform all computable tasks independently, requiring a central agency responsible for the control and management of execution - the CPUs.

Figure 3.2 – Difference between CPU and GPU architectures.



Source: NVIDIA

The main difference between a CPU and a GPU is the number of cores and the size of each core's instruction set, as shown in Figure 3.2. CPUs typically have between 2

and 128 cores per chip, while GPUs can have tens of thousands of cores on a single chip. This difference in core count is due to the difference in the instruction set size of their cores. CPUs are designed to handle various tasks quickly but are limited in the number of concurrent tasks they can perform. Conversely, GPUs are designed to take a small and specific number of tasks with a lower speed but massive parallelism in execution.

Additionally, CPUs are MIMD (Multiple Instruction, Multiple Data) architectures, while GPUs are SIMD architectures. On a CPU, each core can execute different instructions on different data sets simultaneously. Each CPU core can execute its instruction on multiple data sets in parallel (SIMD) thanks to vector instruction sets. However, on a GPU, all cores run the same instruction over various data sets. Therefore, GPUs are best suited for repetitive and highly parallel tasks where instructions must be executed over a large dataset.

Advancements in semiconductor lithography technologies have enabled the manufacture of faster and more powerful CPUs and GPUs than ever before. With each new iteration of the technology, the number of circuits that can be added to the same silicon area has increased considerably. While CPU developers used the space to increase the amount of cache memory, add new instructions and more complex instructions, and increase the number of cores, GPU developers used the additional space mainly to increase the number of cores. These cores are specialized in rendering and floating-point calculation, as well as new cores specialized in specific tasks. For example, NVIDIA GPUs have seen a significant increase in cores on a single chip in recent years, including reduced precision cores (FP16) and cores specialized in computing Fused Multiply-Add operations (FMA), which are widely used in HPC and AI.

Figure 3.3 illustrates the structure of a Streaming Multiprocessor (SM) from NVIDIA Ampere architecture. Each SM consists of 4 warps, with eight double floating-point cores, 16 single floating-point cores, 16 integer cores, and one tensor core. There are 32 FP64 cores, 64 FP32 cores, 64 INT32 cores, and 4 Tensor cores. A complete chip can contain up to 128 SMs, providing 4096 FP64 cores, 8192 FP32 cores, and 512 Tensor cores per chip. Despite the significantly lower number of tensor cores, each can perform 256 mixed-precision FP16/FP32 FMA operations per clock. This amounts to 1024 FMA operations per clock per SM, totaling 131,072 operations in an entire chip with 128 SMs.

The Ampere architecture supports half-precision (FP16) floating-point operations, where each FP32 core can perform FP32 and FP16 operations. Furthermore, each FP32 core can perform two FP16 operations simultaneously, which results in twice as many

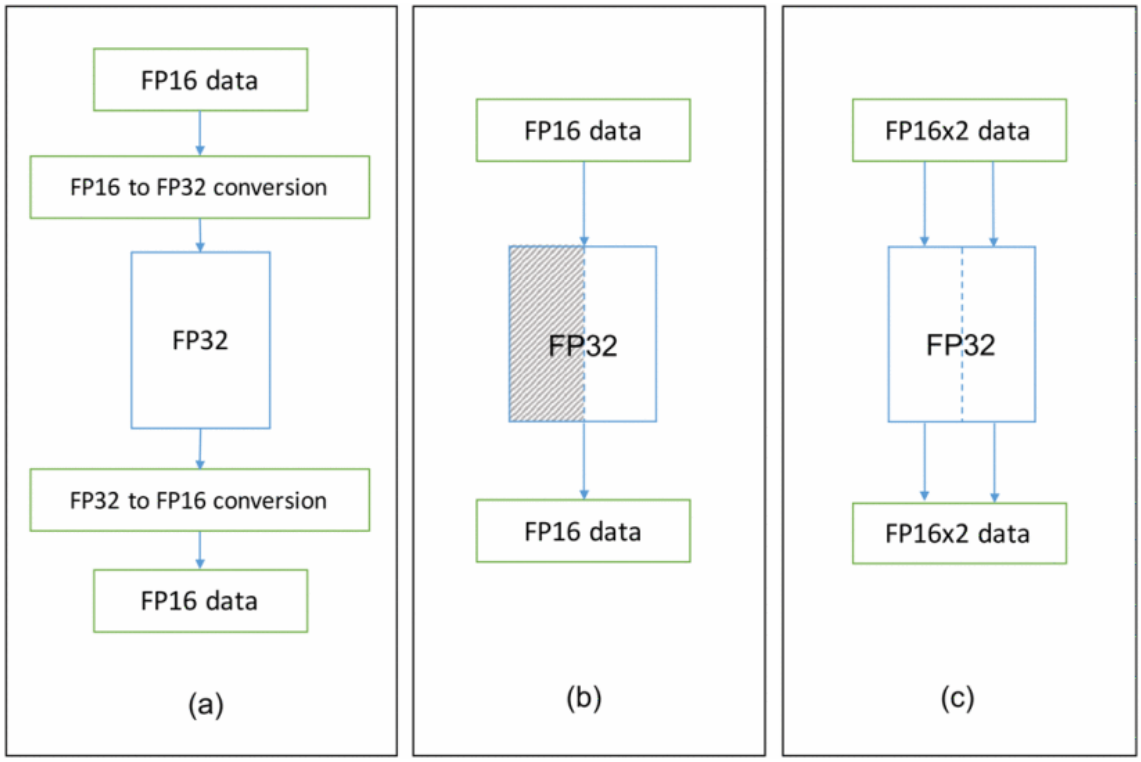
Figure 3.3 – NVIDIA's A100 GPU Streaming Multiprocessor (SM).

Source: NVIDIA

FP16 operations compared to FP32, as shown in Figure 3.4. Therefore, each warp can perform 32 FP16 operations, up to 128 operations per SM, for 16384 FP16 operations on an entire chip with 128 SMs. The Pascal architecture introduced support for running two simultaneous operations in FP16 on an FP32 core, found in GPUs such as NVIDIA P100, V100, and A100.

Although the tensor cores are optimized to compute only FMA operations, the ability to perform operations at reduced precision (FP16) and high throughput makes

Figure 3.4 – Evolution of NVIDIA's GPU half-precision FPUs.



Source: (HO; WONG, 2017)

it possible to speed up the execution of other arithmetic operations in applications that rely heavily on other arithmetic operations. Therefore, AC techniques such as reduced precision and mixed precision can benefit from the high parallelism present in GPUs, especially the native support of FP16 reduced precision. These techniques reduce the size of the floating-point representation format in applications, in addition to reducing memory pressure and increasing parallelism in the execution of arithmetic operations. The reduction from FP32 to FP16 on modern NVIDIA GPUs represents a two-fold increase in the total number of operations that can be performed in the same period. Therefore, optimizing the use of FP16 operations in reduced-precision and mixed-precision techniques on GPU devices can significantly leverage the performance gains of these AC techniques.

3.4 Execution of Approximated Kernel Versions in Iterative Applications

GPUs are massively parallel devices that enable the execution of thousands of operations simultaneously on large data sets. Applications that benefit most from this parallelism are scientific, numerical analysis, neural networks, and machine learning applications.

In particular, scientific and numerical analysis applications require significant computational power, as they usually perform complex calculations on large data sets. Examples of applications such as fluid dynamics simulation, seismic analysis, heat transfer simulation, and weather prediction demand high computational power and benefit from the high parallelism available in GPUs.

These applications are iterative, meaning they perform operations repeatedly on a data set until an acceptable result is obtained or until the desired number of repetitions is reached. Due to the repetitive execution of operations on large datasets, AC techniques that reduce the size of the floating-point representation format in these applications can significantly optimize performance on GPUs that have native support for smaller formats, such as FP16.

However, incorporating reduced and mixed precision techniques can be labor-intensive, especially in iterative applications. Although the reduced precision technique usually requires only tiny modifications to the types of application variables and, in the case of NVIDIA GPUs, adjustments in vector and matrix indexing, in many cases, the loss of accuracy in the final result of the application exceeds the maximum acceptable limit. In these cases, using the mixed-precision technique allows the loss of precision to be controlled by tuning the variables that will or will not have their floating-point representation size reduced (or increased). This tuning becomes extremely costly in larger applications, such as numerous iterative applications.

Moreover, refining and tuning the variables and operations that should or should not have reduced representation size is necessary at each newly defined accuracy loss limit. While higher loss limits allow more operations to be performed at lower accuracies, tighter limits restrict this amount due to their more significant impact on accuracy. Additionally, new input data may require re-tuning tuning to ensure the loss of accuracy limit is respected.

Due to the repetitive nature of iterative applications, combining reduced or mixed precision techniques with executing multiple inexact programs can be an alternative. Instead of developing an approximate implementation based on the loss of accuracy limit by adjusting the size of the representation format of floating-point variables individually, two or more approximate implementations with different loss of accuracy rates are developed. One of the faster versions is executed until the loss of accuracy limit is reached. Then the rest of the iterations are performed in the exact version to avoid exceeding the loss of accuracy limit. This approach allows the same implementations to be used for new

thresholds of loss of accuracy or different data sets, eliminating the need to refactor the application code for new thresholds or data.

As a case study, we will use the HotSpot3D simulation application from the well-known Rodinia benchmark suite (CHE et al., 2009). This application estimates the temperature of processors based on an architectural blueprint and simulated power measurements. Thermal simulations are performed where differential equations are iteratively computed per block. The resulting matrix represents the average temperature value for each cell in the area corresponding to the processor chip.

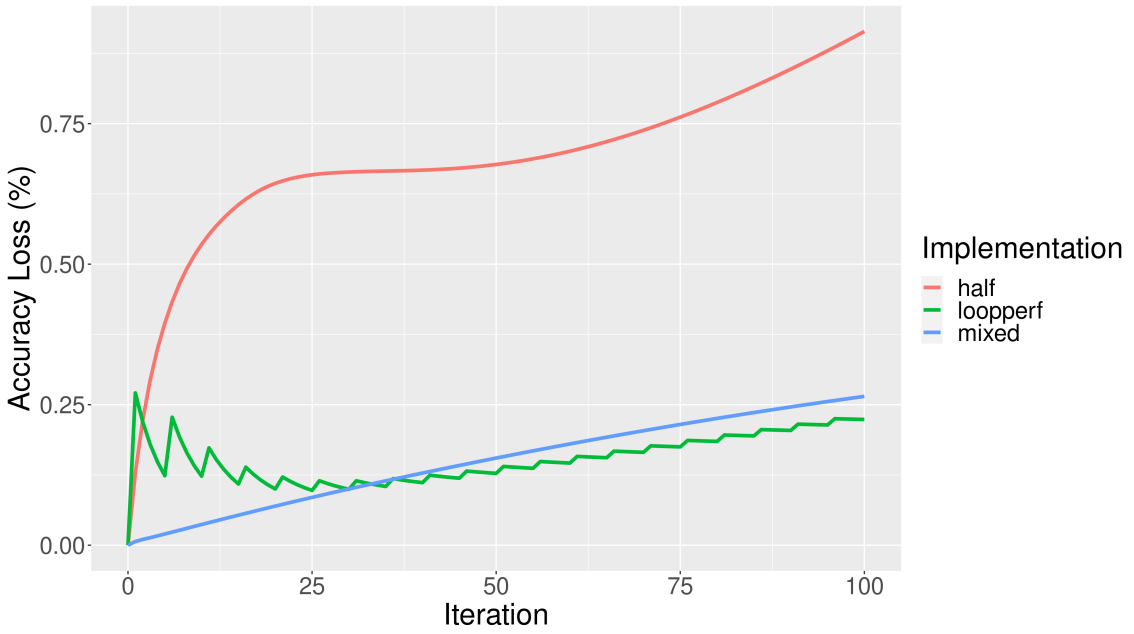
To evaluate the performance of reduced precision, mixed precision, and multiple inexact programs, we will use a system equipped with 4 NVIDIA P100 GPUs natively supporting two-way FP16 precision operations on a 32-bit FPU. Each GPU has 1792 FP64 cores and 3584 FP32 cores, totaling 7168 FP16 cores and 16 GB of global memory. The system also has 2 POWER8NVL 1.0 CPUs, each with ten physical cores and 80 threads (a total of 160 threads) running at a base frequency of 2394 MHz and a maximum frequency of 4023 MHz, and 128 GB of main memory. Although we have multiple GPUs, we will only use one in our experiments.

For the experiments, we will use a 3D input dataset of 512x512x8, which means 512 cells on the X axis, 512 cells on the Y axis, and eight cells on the Z axis, respectively, making a total of 2,097,152 cells. We will use 100 application iterations; the presented results are the average of 10 executions of each experiment.

Figure 3.5 illustrates the evolution of accuracy loss about exact execution in FP32 over 100 iterations of applying three approximate implementations, using the techniques of loop perforation (loopperf), reduced precision FP16 (half), and mixed precision FP32 / FP16 (mixed). The loss of accuracy is expressed as a percentage, with 0% indicating that the result in each cell of the approximate implementation result matrix is identical to the result of the corresponding cell in the exact implementation and 100% indicating that the result in each cell of the approximate implementation is entirely different from the result of the corresponding cells in the exact implementation.

As the execution progresses, each implementation's accuracy loss profile differs. In the loopperf implementation, there is a more significant loss right at the beginning, which retracts and increases from the second half of the iterations to the end. In the mixed precision implementation, the loss of accuracy is continuous during the execution of the application and reaches a final loss of accuracy, like the loopperf implementation, close to 0.25%. On the other hand, the half implementation presents a high accuracy loss

Figure 3.5 – HotSpot3D accuracy loss using FP16 (half), FP16+FP32 (mixed) and loop perforation (loopperf).



Source: The Authors

right at the beginning of the execution, which continues to increase, but at a lower rate, until the end of the execution, reaching a loss of 0.91%.

Despite the significantly more significant loss of accuracy, half implementation has the lowest runtime. While the exact implementation runtime in FP32 is 6.91 milliseconds (ms), the implementation runtime is only 4.05 ms. Loopperf got the second-best time, 5.54 ms, while mixed had a running time close to the exact implementation, 6.87 ms.

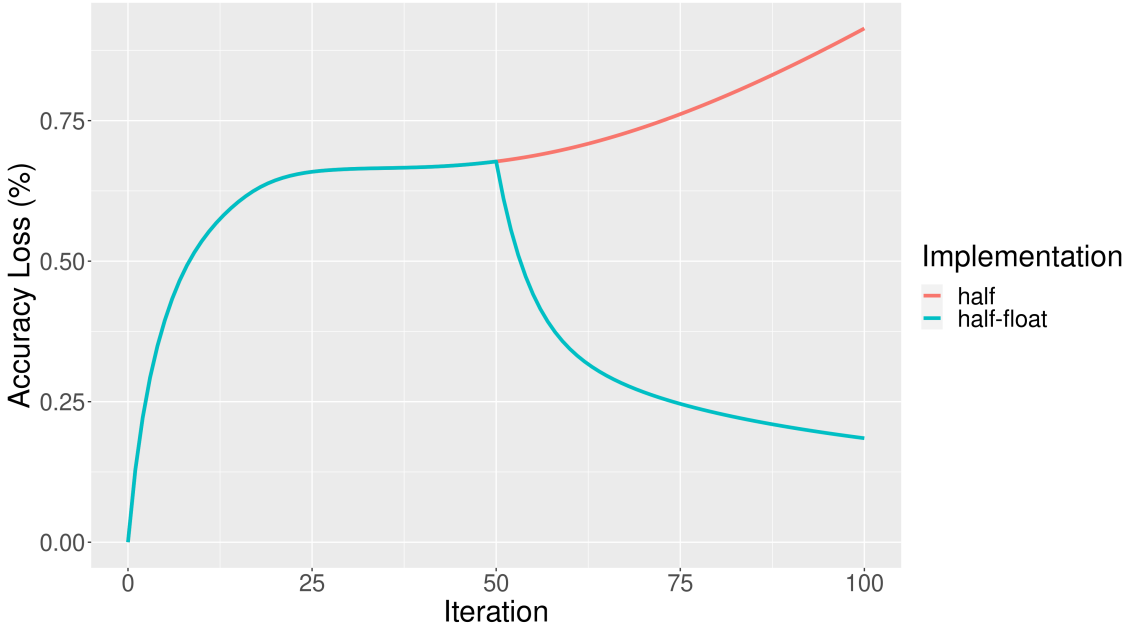
Comparing the runtime of the three approximate implementations, it is possible to observe that the mixed implementation did not significantly improve in runtime compared to the exact implementation in FP32 despite performing some of its arithmetic operations in FP16. This implementation stores all information in structures with FP32 precision. Therefore, when arithmetic operations are performed in FP16, they depend on the conversion of data referring to the computation from FP32 to FP16, which introduces an overhead in the execution. As only a few operations are performed in FP16, the use of FP16x2 incurs more overhead, which ends up harming the execution furthermore, resulting in a performance very close to the exact implementation.

On the other hand, despite the implementation taking good advantage of the parallelism available in FP16x2 operations of the architecture, the storage, and computation of all operations in FP16 compromise the result more. Despite having an accuracy loss

of less than 1%, which in most cases is acceptable, if the accuracy loss limit were 0.7%, for example, this implementation would have exceeded the limit, and therefore the final result would not be considered acceptable. However, a solution to continue using this implementation would be to execute the initial iterations until reaching the loss limit of 0.7%, and from then on, execute the rest in float (exact implementation in FP32), thus avoiding exceeding the limit of loss.

Figure 3.6 presents the result of an execution where the accuracy loss limit is 0.7% with the replacement of implementations when the limit is reached. This image shows the loss of accuracy profile of the half application (in red) where all iterations are executed in FP16 and the profile of the loss of accuracy of the execution of the half implementation until reaching the loss limit and, from then on, the execution using the float implementation of the rest of the iterations. As the float implementation replaces the half implementation, the loss of accuracy starts to decrease, retracting from 0.7% to just 0.18%. Despite the reduction in accuracy loss, the runtime increased from 4.06 ms to 5.51 ms.

Figure 3.6 – HotSpot3D accuracy loss using FP16 (half) and FP16 up to 0.7%, then switching to FP32 for the rest of the iterations (half-float).

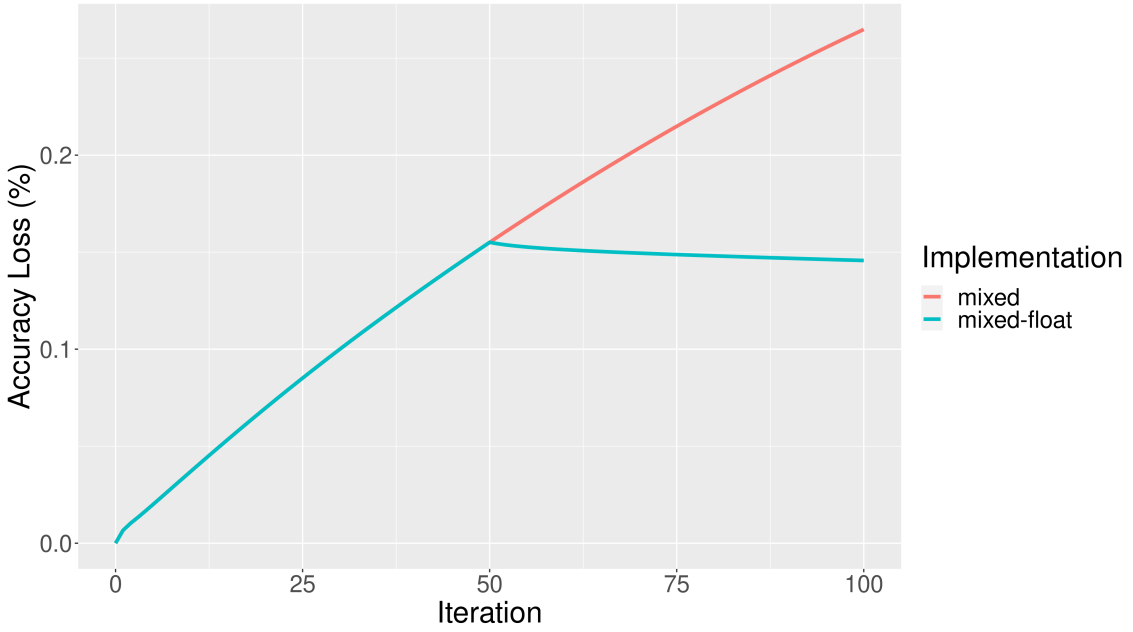


Source: The Authors

Suppose we perform the same experiment with the mixed and loopperf implementations. In that case, we observe that the loss of accuracy profile behaves quite differently from that of the half implementation, as shown in Figures 3.7 and 3.8. Using an accuracy loss limit of 0.16% for the mixed implementation and 0.14% for the loopperf implemen-

tation and replacing these implementations with the float implementation when this limit is reached, we see that the accuracy loss stabilizes around these limits until the end of the execution, with a slight decrease in the loopperf implementation. These results suggest that replacing FP16 implementations with higher-precision implementations can reduce precision loss introduced by the FP16 implementation due to accumulation.

Figure 3.7 – HotSpot3D accuracy loss using FP16+FP32 (mixed) and FP16+FP32 up to 0.16%, then switching to FP32 for the rest of the iterations (mixed-float).



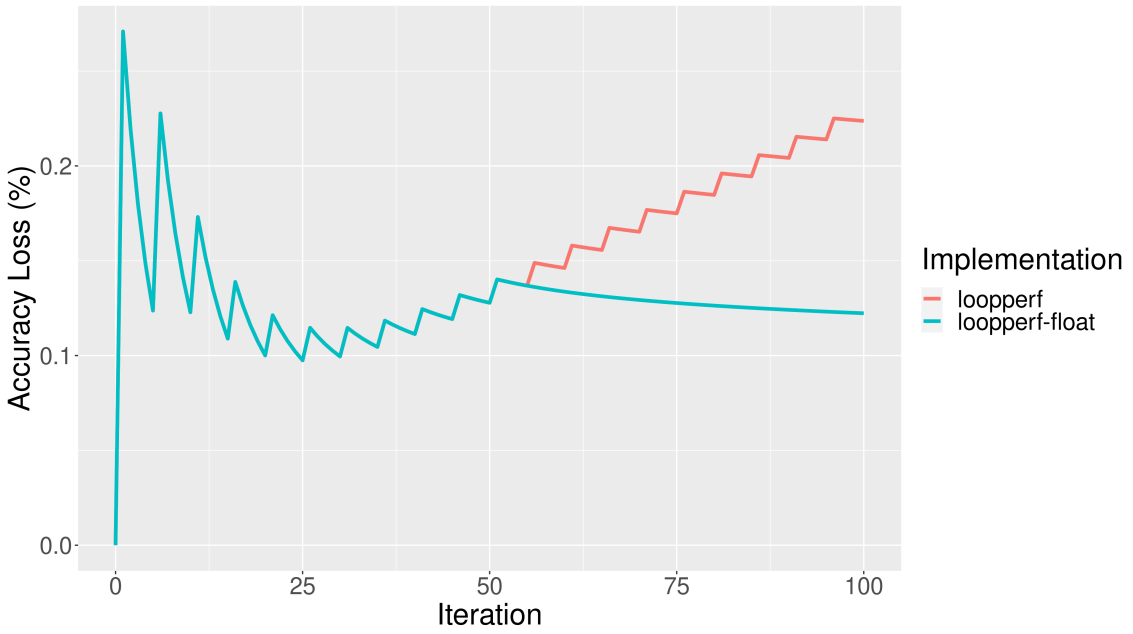
Source: The Authors

Replacing the mixed and loopperf implementations with the float implementation increases their running time. In the mixed implementation, the time increases from 6.87 ms to 6.89 ms. In the loopperf implementation, the replacement causes the time to rise from 5.54 ms to 6.16 ms.

Although the final result is significantly smaller than the 0.7% loss of accuracy threshold that triggered the replacement of the half implementation with the float implementation, it does not necessarily mean that the same run is valid in a case where the loss of accuracy limit is 0.2%, for example. Despite the final result being less than 0.2%, the loss of accuracy during execution was significantly higher. Therefore, it cannot be guaranteed that this result is correct. However, exploiting this feature can result in significant performance gains.

To evaluate the potential performance gain in interleaving the half and float implementations so that the accuracy loss limit is not exceeded during the entire execution, we will use three different limits. Figure 3.9 presents the profile of the loss of accuracy of

Figure 3.8 – HotSpot3D accuracy loss using FP16+FP32 (mixed) and FP16+FP32 up to 0.16%, then switching to FP32 for the rest of the iterations (mixed-float).



Source: The Authors

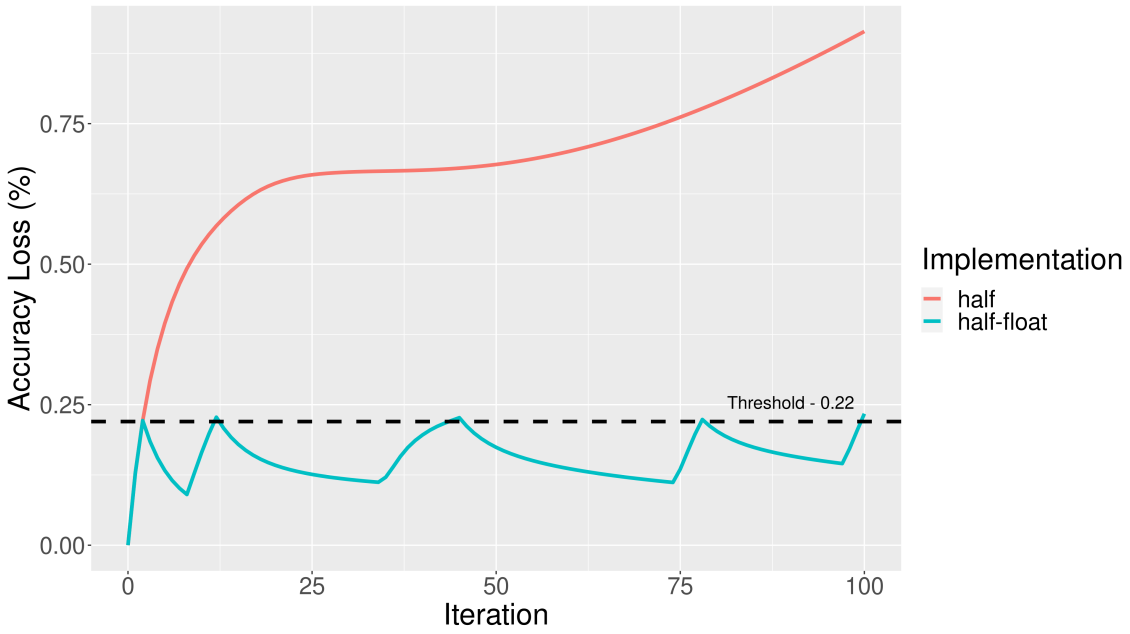
the interleaving of the half and float implementations in a loss limit of around 0.22%. We start the execution with the half implementation and reach the accuracy loss limit already in the third iteration, at which point we replace the half implementation with the float. After a few iterations, we rerun the half implementation, successively interspersing them until the execution is finished. This way, it was possible not only to respect the limit of loss of accuracy but also to achieve a relatively lower runtime than the exact execution, 6.44 ms.

By setting a higher accuracy loss limit, precisely 0.44%, we can observe that the interleaving between half and float implementations decreases. However, the half implementation requires considerably more iterations, as shown in Figure 3.10. With more iterations running in FP16x2, the runtime is significantly reduced compared to the previous loss limit of 0.22%, dropping from 6.44ms to 5.53ms.

Consequently, by increasing the precision loss limit to 0.68%, we observe that the float implementation performs only a tiny fraction of iterations, as shown in Figure 3.11. This results in a runtime that is very close to using only half implementation, with only 4.3ms compared to 4.03ms.

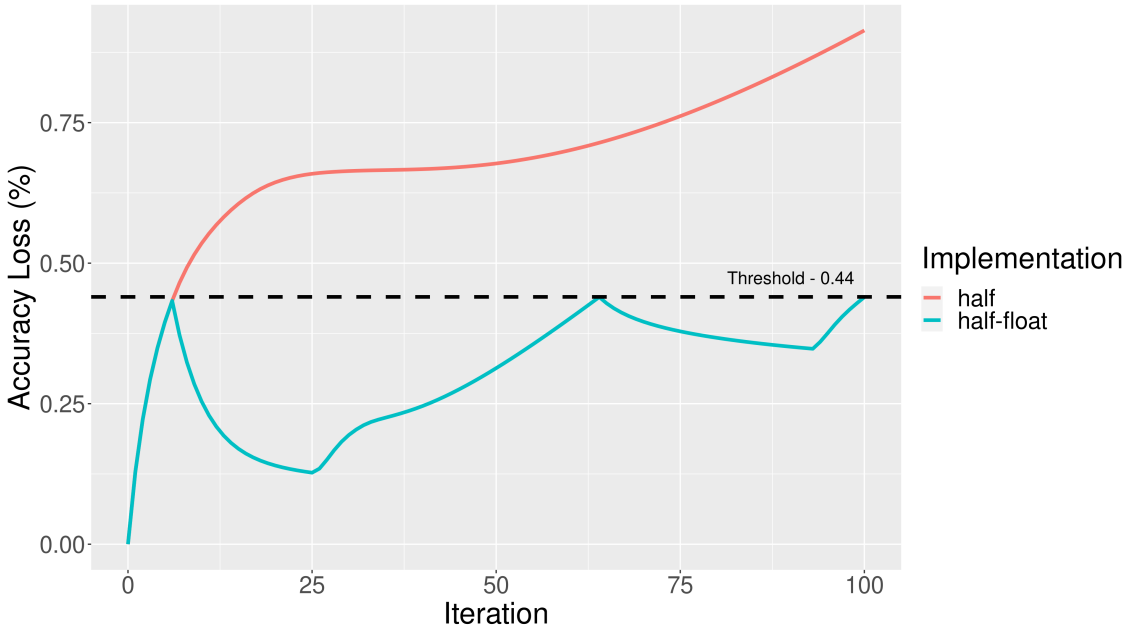
The interleaved execution of the exact and approximate implementations allows code reuse without requiring changes for different accuracy loss configurations. Using a predetermined accuracy loss limit as the basis for interleaving between the two imple-

Figure 3.9 – HotSpot3D accuracy loss using FP16 and switching to FP32 each time the accuracy loss threshold of 0.22% is reached (half-float) compared to the accuracy loss of full FP16 execution (half).



Source: The Authors

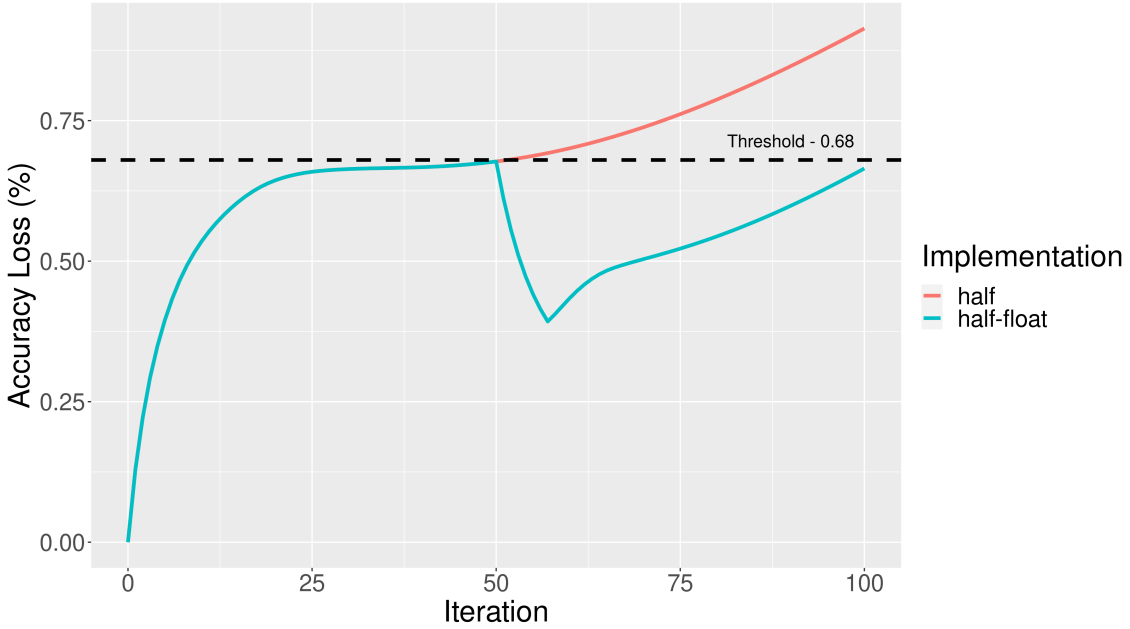
Figure 3.10 – HotSpot3D accuracy loss using FP16 and switching to FP32 each time the accuracy loss threshold of 0.44% is reached (half-float) compared to the accuracy loss of full FP16 execution (half).



Source: The Authors

mentations maximizes the use of the faster but less accurate implementation, resulting in a considerable acceleration of the application's execution within the accuracy loss limit. This approach allows optimal utilization of the FP16x2 architecture while respecting de-

Figure 3.11 – HotSpot3D accuracy loss using FP16 and switching to FP32 each time the accuracy loss threshold of 0.68% is reached (half-float) compared to the accuracy loss of full FP16 execution (half).



Source: The Authors

sired precision loss limits.

During execution, a considerable variation in the accuracy loss evolution profile can be observed as intercalations occur. With a limit of 0.22%, Figure 3.9 shows that the rate of accuracy loss differs in the first two parts where the half implementation is used compared to the third part. In the first two parts, the loss of accuracy goes from 0% to 0.22% and from 0.09% to 0.22%, respectively, in just 2 and 4 iterations. However, the third execution of the half implementation experiences a loss of accuracy from 0.11% to 0.22% over a range of 11 iterations. Finally, in the fourth and fifth executions of the half implementation, the loss rate increases again, reaching 0.22% in smaller intervals of iterations, only 4 and 3 iterations, respectively.

4 A METHODOLOGY FOR INTERLEAVED EXECUTION OF APPROXIMATED CUDA KERNELS BASED ON ACCURACY LOSS PROFILES

This Chapter discusses the challenges related to the interleaving of multiple kernel versions and presents the proposed methodology in this thesis. The methodology is discussed in detail, including the steps of defining measurement points, generating performance statistics and accuracy loss of subsections, and generating the execution configuration based on the statistics for a specific accuracy loss limit defined by the user.

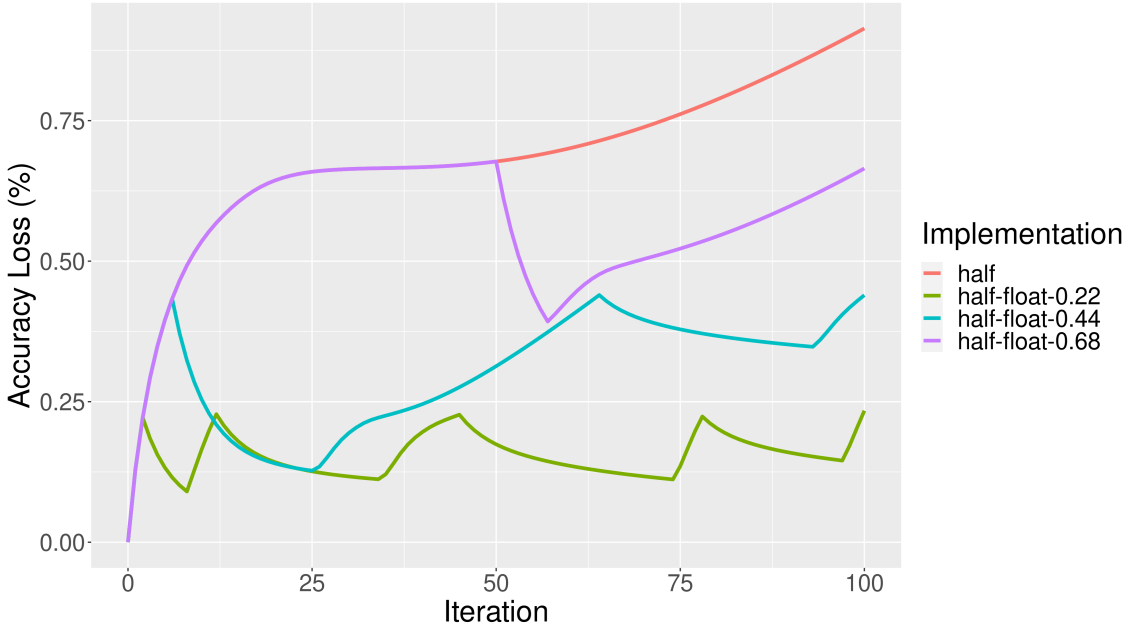
4.1 Challenges in Interleaved Execution of Kernel Versions

In Section 3.4, we analyzed the impact of three of the most popular approximate computing techniques on the accuracy loss of the HotSpot3D heat transfer simulation application from the Rodinia benchmark suite. Additionally, we showed how it is possible to accelerate application execution within predetermined accuracy loss thresholds by using multiple kernel versions with different accuracy loss and performance profiles. For this purpose, we used a kernel implemented in FP16 (called "half") developed to efficiently utilize modern NVIDIA GPU architectures, where the same FP32 FPU can perform two FP16 operations simultaneously. We also used an exact kernel implemented in FP32 ("float").

However, as mentioned in Section 3.4, the accuracy loss profile varies considerably during execution. Figure 4.1 shows the accuracy loss profiles of the half kernel implementation and the interleaved execution of the half and float kernel implementation with accuracy loss thresholds of 0.22%, 0.44%, and 0.68%. If we compare the behavior of the accuracy loss when the half kernel implementation is executed, we see that the loss rate differs in every execution. The same occurs when the exact kernel implementation is executed, which can quickly reduce in just a few iterations or slowly reduce over a few dozen iterations. This behavior shows that the initial state of the data (i.e., the value) at the time the kernel implementations start their execution significantly impacts the accuracy loss rate throughout the interleaved execution of the implementations. Thus, a loss rate measured at the beginning of execution may not accurately represent the accuracy loss rate throughout the remainder of the application's execution.

To improve the method's accuracy in generating execution configurations, it is

Figure 4.1 – HotSpot3D accuracy loss profile of the interleaved execution of FP16 and FP32 kernel versions with accuracy thresholds of 0.22%, 0.44%, and 0.68%.



Source: The Authors

crucial to consider the difference in accuracy loss behavior at various points during execution. One way to achieve a more accurate measurement of accuracy loss at different moments of execution is to use the accuracy loss profile of the execution of the approximate kernel version with the highest accuracy loss as a basis. Divide the cartesian plane represented by the total number of iterations and the maximum accuracy loss of the execution into N subsections. In Figure 4.1, for example, there are a total of 100 iterations and a maximum loss of 0.91%. This will obtain the average accuracy loss rate for each kernel version in each subsection, allowing for more precise accuracy loss estimates throughout application execution.

To measure the accuracy loss profile, it is not necessary to measure all subsections resulting from dividing the area related to the total number of iterations and the maximum accuracy loss value. Typically, the accuracy loss profile expands upward to the right of the Cartesian plane, represented by the total number of iterations and the maximum accuracy loss (as shown in Figure 4.1). In practice, some subsections are not intercepted by the base accuracy loss profile (i.e., the profile of the kernel version with the highest accuracy loss) or by the subsequent measurements taken. Therefore, measuring only the subsections intercepted by one accuracy loss profile is sufficient.

Although measuring the accuracy loss rate in multiple subsections requires significant additional work, these measurements are necessary only once for the same input data

set and kernel versions. Performance and accuracy loss statistics can then be extracted for a data set and kernel versions. With these statistics, the method can generate execution configurations for any desired accuracy loss limit without requiring new measurements. This eliminates the need to alter the kernel version code for different accuracy loss limits.

In this work, we will use only two versions of the kernel of applications: the exact version and an approximate version with reduced floating-point precision or loop perforation technique. This choice was made for simplification since our work aims to evaluate measurements in multiple subsection configurations and accuracy loss limits. To achieve this goal, we will use only the exact version and an approximate version that offers a significant gain in runtime since the idea is to interleave multiple versions with different runtimes and seek to maximize the utilization of the kernel version with the best performance to reduce the application's runtime as much as possible. Following this line, our method of generating execution configurations will always start with using the approximate kernel version to maximize the execution of the kernel with the lowest runtime within the specified accuracy loss limit.

4.2 Our Methodology

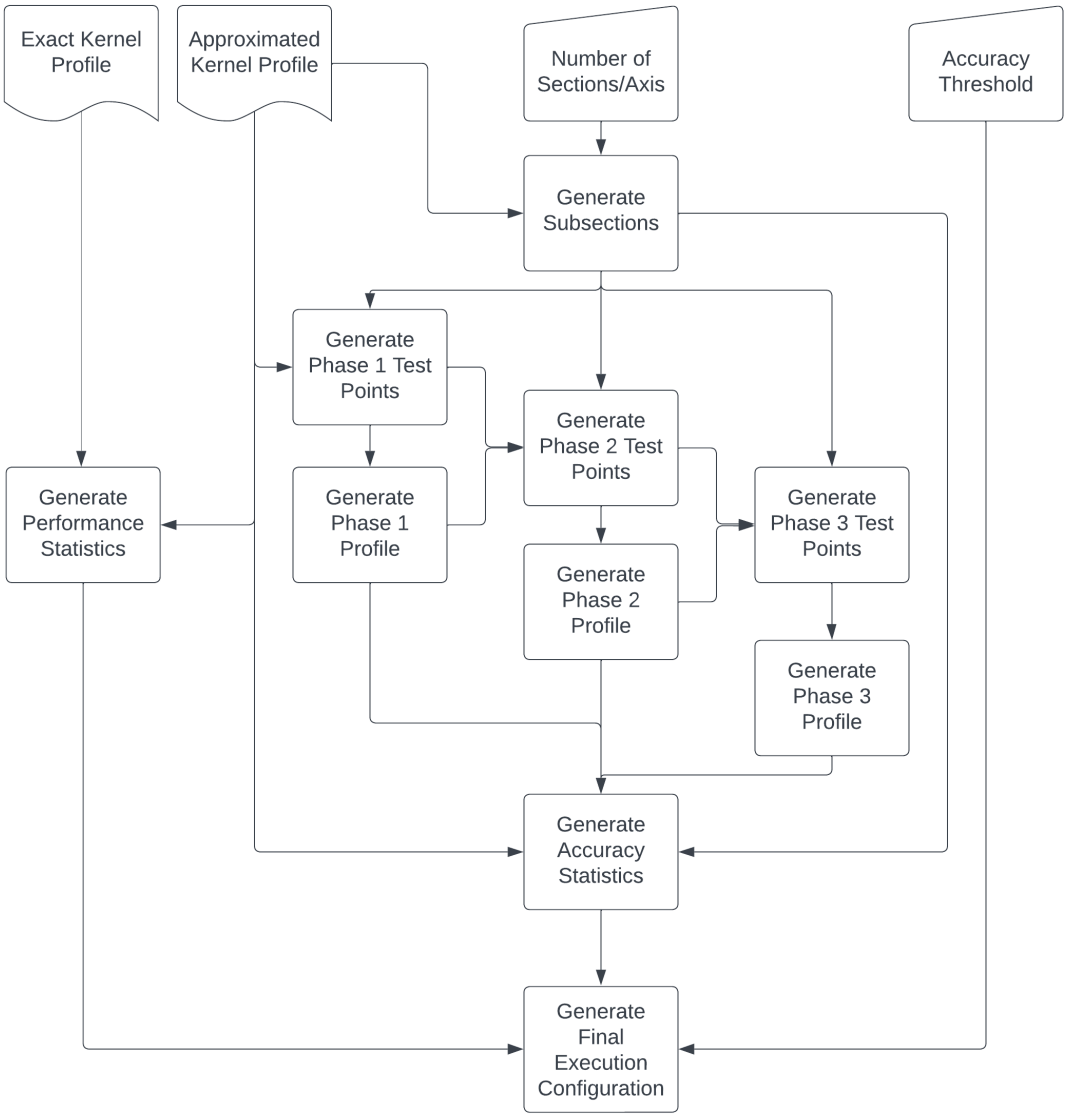
Figure 4.2 shows a flow diagram of the entire process, which includes measuring performance and loss of accuracy and generating the execution configuration of our method. There are six main steps:

1. Generating Accuracy Loss Profiles
2. Generating Subsections Configuration
3. Measuring Performance and Accuracy Loss per Subsections
4. Generating Accuracy Loss Statistics
5. Generating Performance Statistics
6. Generating Execution Configurations

4.2.1 Step 1 - Generating Accuracy Loss Profiles

The first step is to measure the loss of accuracy and runtime of each iteration of the application for both kernel versions. To do this, you must first instrument the application

Figure 4.2 – Flowchart of our methodology.



Source: The Authors

with a loss of accuracy measurement function to compute the Frobenious norm defined in Section 3.1 of Chapter 3 and duplicate the primary data structures. These are the structures that contain the input and output data modified during the execution of the application. The data duplication is necessary to perform the control computation and the approximate computation of the application side by side. This enables you to compute the loss of accuracy of the computation as the execution progresses.

The pseudo-code in Figure 4.3 illustrates the instrumentation process. In lines 1 and 2, the code duplicates all necessary data structures modified during execution, isolating the data for control execution (line 5) and approximate execution (line 6). Within the application's main loop, after each iteration's control and approximate execution, the

Figure 4.3 – Code instrumentation to perform accuracy loss profiling.

```
1  reference_data
2  approximate_data
3
4  main iterative loop:
5      compute_exact_kernel(reference_data)
6      compute_approximate_kernel(approximate_data)
7
8      wait()
9      copy_data_from_device_to_host()
10
11     compute_accuracy_loss(reference_data, approximate_data)
12
13     store_profile()
```

Source: The Authors

Figure 4.4 – Profiling output file format.

```
1  kernel, iteration, start_timestamp, end_timestamp, runtime,
       accuracy_loss
2  half, 1, 1675188784836839782, 1675188784837324423, 0.480255991220474,
       0.001458076294512
```

Source: The Authors

computation results are copied from the GPU's global memory to the host's main memory (lines 8 and 9). Once the results are copied to the host's main memory, the loss of accuracy of the execution up to that point is calculated (line 11), and the information is stored in a file (line 13). The process then continues with the execution of the next iteration and repeats the cycle.

The data is stored in a CSV (Comma-Separated Values) file at each iteration of the application. Before the execution starts, any loss of accuracy is recorded to track losses arising from conversions between different floating-point data types. Data conversion is only required when different floating-point representation formats are used, since different approximate computing strategies can be used with the same floating-point format. In addition to the loss of accuracy, the kernel name, the iteration, and the start and end timestamps of each iteration's execution are also stored. The file format is shown in Figure 4.4.

Instrumentation can be performed on each kernel version individually or in a single application. If the application has kernel version control, we can perform instrumentation on only one application instance. We can then generate the accuracy profile for each kernel version in two runs, using only one of the versions in each run. If the application does not have kernel version control, we need to perform instrumentation for each kernel version in a different application instance. This will allow us to extract the accuracy loss

profile for each kernel version.

4.2.2 Step 2 - Generating Subsections Configuration

In step 2, we divided the area corresponding to the display into a graph that shows the loss of accuracy throughout the execution of the application. As mentioned earlier, to obtain a more accurate estimate of the loss of accuracy as the execution progresses, we will measure the loss of accuracy profile of multiple interleaves of the kernel versions at different stages of the execution. This is necessary because the behavior of the loss of accuracy changes throughout the execution. To accomplish this, we use the loss of accuracy profile of the kernel version with the highest loss to create subsections in the area corresponding to the loss of accuracy profile when displayed on a Cartesian plane.

We extract the total number of iterations and the maximum loss of accuracy reached during the execution of the application from the approximate kernel's loss of accuracy profile (represented by the "Approximate Kernel Profile" object in the flow diagram of Figure 4.2). Then, we divide these values by the number of subsections per axis the user enters. Using this information, we generate a CSV file containing the subsection identifier, start and end points for both axes, and the center of each subsection. By default, we use the X axis for the number of iterations and the Y axis for the loss of accuracy.

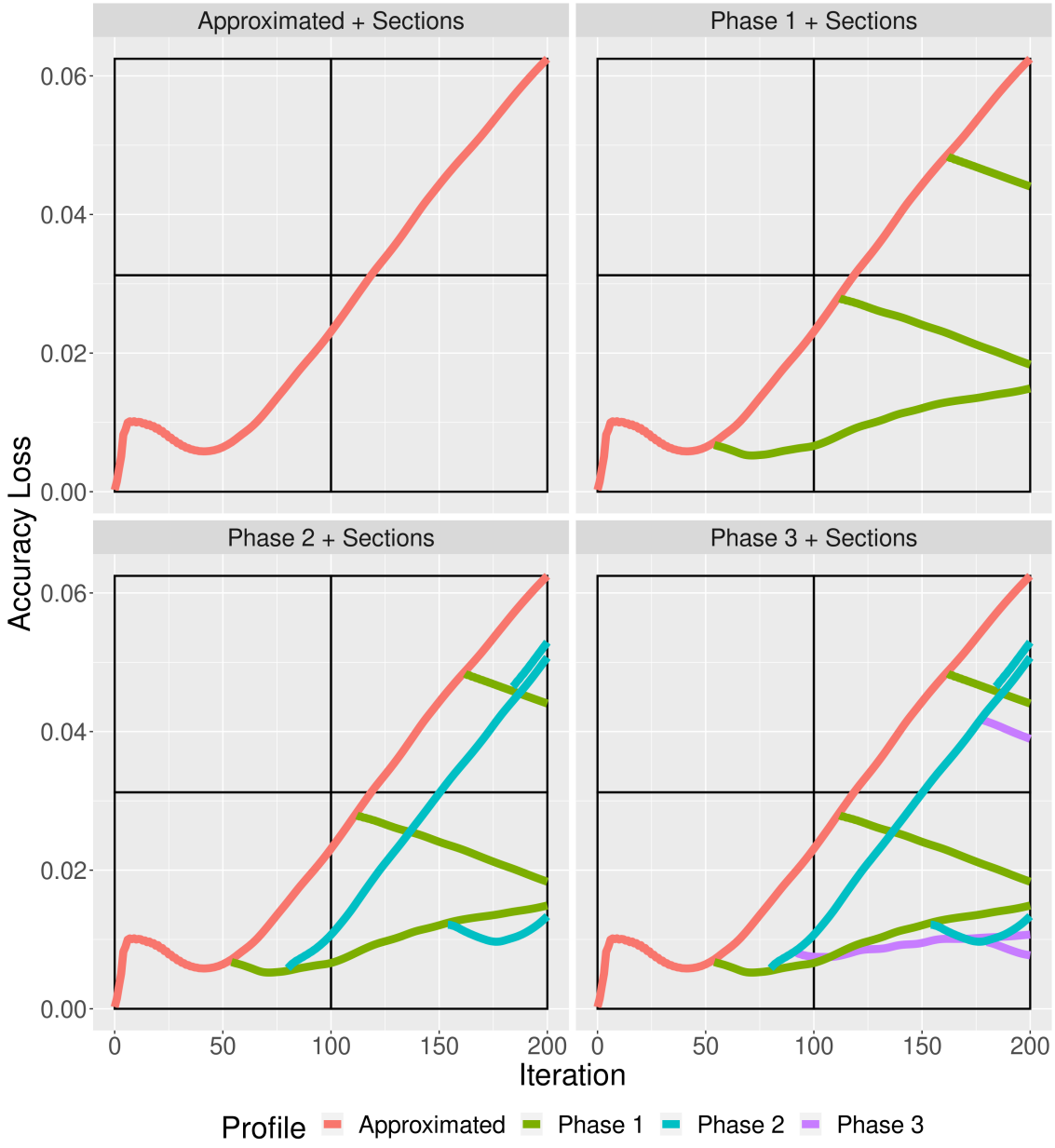
Figure 4.5 illustrates the accuracy loss profile in the "Approximate + Sections" facet and the corresponding subsections generated from this profile. The X-axis displays the iterations, while the Y-axis displays the accuracy loss (measured using the Frobenius norm). In this instance, each axis was split into two subsections, resulting in four subsections.

4.2.3 Step 3 - Measuring Performance and Accuracy Loss per Subsections

In step 3, we collected samples of the accuracy loss at different stages of the application's execution and with different interleaving of kernel versions. These measurements were performed in three phases, based on the subsections generated in step 2. However, before starting the measurements, we must decide where and when to take them.

The first phase involves measuring the accuracy loss that results from interleaving the approximate kernel with the exact kernel. To do this, we use the loss of accuracy

Figure 4.5 – Illustration of the main steps of our methodology.



Source: The Authors

profile of the approximate version and the subsection information to identify which subsections are intercepted by the profile. This allows us to determine which iterations of the approximate kernel version's loss of accuracy profile belong to each subsection (as seen in the "Approximated + Sections" facet of Figure 4.5) and then define from which iteration to switch to the exact kernel version for execution.

Our goal is to find a more accurate estimate of the loss of accuracy in each subsection. We chose the average iteration among the iterations intersecting each subsection for the first phase measurements to achieve this. This strategy avoids measuring iterations near section boundaries. After deciding on the iterations of the approximate version from

which the measurements of the loss of accuracy of the float version will be carried out, we store these points (iterations) in a text file. This is represented by the "Generate Phase 1 Test Points" object in Figure 4.2.

Figure 4.6 illustrates the instrumentation required to conduct interleaving loss-of-accuracy measurements of the approximate kernel version against the exact version. In lines 1-3, we read the iterations stored in a file and insert them into a vector. In lines 5-6, we duplicate the data structures used by the kernels to allow separate execution of the control (from now on called the “reference” execution) and approximate executions, enabling loss-of-accuracy measurement at each iteration. The computation of the exact and approximate execution using the approximate kernel is performed in lines 9-10, continuing until the current iteration is present in the vector, which signals the need to interleave the two kernel versions in the approximate execution.

After identifying the current iteration in the vector, we copy the updated execution data to new data structures. Then, a new execution starts on line 16 from the following iteration. Instead of the approximate kernel version used in the earlier computation, we use the exact kernel version (lines 17 and 18) with the data from the approximate execution of the loop in line 8. This way, we can measure the loss of accuracy starting from the iteration that generated the interleaving of the two kernel versions until the end of the execution of the loop of line 8.

We repeat this process for each iteration present in the vector. At the end of each iteration of the loop in line 16, we calculate and store the loss of accuracy (line 20). This is represented by the "Generate Phase 1 Profile" object in Figure 4.2. In the end, we will have the accuracy loss profile of the interleaving of the approximate kernel version with the exact one in the different subsections, which is intercepted by the loss profile of the approximate version used as the baseline. You can observe the result in the "Phase 1 + Sections" facet of Figure 4.5.

In phase one, we duplicate the data in lines 13 and 14 by copying the data from the current execution to new data structures. This second duplication is necessary because we are performing a fork of execution. When an iteration is present in the vector, the execution of the loop in line 8 is paused, and the loop in line 16 runs from that iteration up to the total number of iterations of the execution. Once the loop in line 16 is finished, the loop in line 8 resumes execution. Therefore, it is essential to preserve the data in the structures of lines 9 and 10 for the correct continuity of the execution of the loop in line 8.

Figure 4.6 – Phase 1 accuracy loss profiling code instrumentation.

```
1  std::ifstream tpfile(filename)
2  std::istream_iterator<int> start(tpfile), end
3  std::vector<int> test_points(start, end)
4
5  reference_data
6  approximate_data
7
8  for (int i = 0; i < numiter; i++)
9      compute_exact_kernel(reference_data)
10     compute_approximate_kernel(approximate_data)
11
12     if (std::count(test_points.begin(), test_points.end(), i))
13         reference_data_phase1 = reference_data
14         approximate_data_phase1 = approximate_data
15
16         for (int j = i+1; j < numiter; j++)
17             compute_exact_kernel(reference_data_phase1)
18             compute_exact_kernel(approximate_data_phase1)
19
20             compute_accuracy_loss(reference_data_phase1,
                   approximate_data_phase1)
```

Source: The Authors

Phase two involves measuring the loss of accuracy when interleaving the exact kernel version by the approximate kernel version. We use the loss of accuracy information collected in phase one to determine which subsections and iterations to collect this information from. Similar to phase one, we identify the subsections intersected by the different loss of accuracy profiles measured in phase one. If multiple profiles intersect the same subsection, we use the proximity of the profile to the center of the intercepted subsection as a criterion. This enables us to choose a profile that intersects the subsection more evenly instead of only at the boundaries of the subsection.

After identifying the subsections intercepted by each phase one profile and choosing the profile closest to the center of the subsection in case of multiple intercepts, we need to determine from which iteration to perform measurements on each intercepted subsection. Similar to phase one, we choose the average iteration of the set of iterations of the profile that intersects the subsection.

Finally, we create a file that includes the iterations chosen for phase one and the iterations chosen to carry out the measurements of phase two. This allows for reproducing the measurements of phase one and new measurements to be carried out. The "Generate Phase 2 Test Points" object in Figure 4.2 represents this stage.

Figure 4.7 depicts the instrumentation for phase two. We begin by reading the file that contains the iterations in which the new measurements of phase two will be carried

out. This file consists of two columns. The first column lists the iterations of phase one corresponding to the chosen profiles, and the second column lists the iterations from which the measurements of phase two will be conducted. This information is stored in a vector where column one represents the index of the vector, and column two represents the values of the respective indexes.

Figure 4.7 – Phase 2 accuracy loss profiling code instrumentation.

```
1  std::vector<int> test_points[numiter]
2  std::ifstream tpfile(filename)
3  std::string line
4
5  while(std::getline(tpfile, line))
6
7      std::stringstream    linestream(line)
8      int                  val1, val2
9
10     linestream >> val1 >> val2
11
12     test_points[val1].push_back(val2)
13
14 reference_data
15 approximate_data
16
17 for (int i = 0; i < numiter; i++)
18     compute_exact_kernel(reference_data)
19     compute_approximate_kernel(approximate_data)
20
21     if (!test_points[i].empty())
22         reference_data_phase1 = reference_data
23         approximate_data_phase1 = approximate_data
24
25         for (int j = i+1; j < numiter; j++)
26             compute_exact_kernel(reference_data_phase1)
27             compute_exact_kernel(approximate_data_phase1)
28
29             if (std::count(test_points[i].begin(), test_points[i].end()
                   , j))
30                 reference_data_phase2 = reference_data_phase1
31                 approximate_data_phase2 = approximate_data_phase1
32
33                 for (int k = j+1; k < numiter; k++)
34                     compute_exact_kernel(reference_data_phase2)
35                     compute_approximate_kernel(approximate_data_phase2)
36
37                     compute_accuracy_loss(reference_data_phase2,
                           approximate_data_phase2)
```

Source: The Authors

The iterations in the first column only include the iterations of the profiles from which new measurements will be performed in phase two. Therefore, not all iterations tested in phase one will be included. Additionally, if a loss profile extracted from a given

iteration in phase one intersects multiple subsections and more than one measurement in phase two is conducted from that iteration, the iteration of phase one is repeated in the first column with the respective iteration of phase two from which measurements will be taken.

From lines 14 to 27, we repeat the same steps as in phase one. The objective of phase two is to measure the loss of accuracy when interleaving the exact kernel version with the approximate version from certain iterations of the loss of accuracy profiles of phase one. To achieve this, we need to reproduce the execution of these profiles until these iterations to have updated data referring to the execution of the phase one profiles.

After duplicating data to isolate the reference execution and the approximate execution in lines 14 and 15, the main loop begins in line 17. In lines 18 and 19, we perform the reference execution always using the exact kernel version and the approximate execution using the approximate kernel version until the index vector equals the current iteration of the execution, signalling the need to perform the interleave of the approximate kernel version with the exact version (only in the approximate execution). From then on, we duplicate the data again (lines 22 and 23), copying the current state of the data structures used thus far to new structures. A new loop starts in line 25, executing from the iteration that generated the interleaving of the kernels to the total number of iterations of the execution.

If the current iteration of the loop in line 25 exists in the vector with an index equal to the iteration of the loop in line 17 that caused the interleaving of the approximate kernel version with the exact version, a new interleave of the versions will be necessary (line 29). Afterwards, we perform a new measurement of the accuracy loss by interleaving the exact kernel version with the approximate kernel version. This starts with duplicating the data structures in lines 30 and 31. Once the data structures have been copied to new instances, a new loop is started in line 33 from the iteration that caused the second kernel interleave in line 19, using the approximated kernel version in the approximate execution (line 35). At the end of each iteration, the loss of accuracy is calculated, and the results are stored in a file on line 37. The "Generate Phase 2 Profile" object in Figure 4.2 represents this stage.

The results of the measurements are shown in the "Phase 2 + Sections" facet of Figure 5.4. The graph presents the loss of accuracy profiles of the approximate version (represented by the red line), phase one (represented by the green lines), and phase two (represented by the cyan lines). As shown in the lower right subsection (between iterations 100 and 200 and loss of accuracy 0 and 0.31), two phase one loss of accuracy profiles

intersect the subsection, but only the one closest to the center was used to measure the loss of accuracy in phase two. Furthermore, the graph illustrates how the behavior changes for the same kernel version between different subsections.

Phase three involves measuring the loss of accuracy resulting from interleaving the approximate kernel version with the exact kernel version. In phase one, the loss of accuracy resulting from interleaving the approximate kernel version with the exact version is also measured. However, this is based only on the loss of accuracy profile of the approximate kernel version execution, which may cover only some of the subsections. By introducing the third phase, we can increase the coverage of measurements for the loss of accuracy resulting from interleaving the approximate and exact versions. Although the main objective of phase three is to measure the loss of accuracy in subsections not covered in phase one, we will measure all subsections intercepted by the profiles of phase two to improve the results.

In phase two, we collect information on the loss of accuracy for different kernel versions. Based on this information, we determine the subsections and iterations from which to collect loss of accuracy information in phase three. If multiple loss of accuracy profiles intersect the same subsection, we use the proximity of the profile to the center of the subsection as a selection criterion, as we did in phase two.

After identifying the subsections intercepted by each phase two profile and choosing the profile closest to the center of the subsection in case of multiple intercepts, we determine from which iteration to perform measurements on each intercepted subsection. We select the mean iteration of the set of profile iterations that intersect the subsection. Finally, we generate a file based on the measurement iterations of phase one and two and the iterations chosen to carry out the measurements of phase three to enable the reproduction of the measurements of phase one and Two from which the new measurements will be carried out. This stage is represented by the "Generate Phase 3 Test Points" object in Figure 4.2.

Figure 4.8 illustrates the instrumentation for phase three. The file containing the iterations in which the new measurements of phase three will be carried out is read between the lines 1 and 14. This file has three columns: the first column contains the iterations of the respective phase one profiles, the second column contains the iterations of the phase two profiles, and the third column contains the iterations of the phase two profiles from which the measurements of phase three will be performed.

This information is stored associatively in a *map* structure. The first column cor-

Figure 4.8 – Phase 3 accuracy loss profiling code instrumentation.

```
1  std::map<int, std::map<int, std::vector<int>>> test_points
2  std::ifstream tpfile(filename)
3  std::string   line
4
5  while(std::getline(tpfile, line))
6      std::stringstream   linestream(line)
7      int                 val1, val2, val3
8
9      linestream >> val1 >> val2 >> val3
10
11     if (test_points[val1][val2].empty())
12         test_points[val1][val2] = std::vector<int> {val3}
13     else
14         test_points[val1][val2].push_back(val3)
15
16 reference_data
17 approximate_data
18
19 for (int i = 0; i < numiter; i++)
20     compute_exact_kernel(reference_data)
21     compute_approximate_kernel(approximate_data)
22
23     if (test_points.count(i))
24         reference_data_phase1 = reference_data
25         approximate_data_phase1 = approximate_data
26
27         for (int j = i+1; j < numiter; j++)
28             compute_exact_kernel(reference_data_phase1)
29             compute_exact_kernel(approximate_data_phase1)
30
31             if (test_points[i].count(j))
32                 reference_data_phase2 = reference_data_phase1
33                 approximate_data_phase2 = approximate_data_phase1
34
35                 for (int k = j+1; k < numiter; k++)
36                     compute_exact_kernel(reference_data_phase2)
37                     compute_approximate_kernel(approximate_data_phase2)
38
39                     if (std::count(test_points[i][j].begin(),
                           test_points[i][j].end(), k))
40                         reference_data_phase3 = reference_data_phase2;
41                         approximate_data_phase3 =
                               approximate_data_phase2
42
43                         for (int l = k+1; l < numiter; l++)
44                             compute_exact_kernel(reference_data_phase3)
45                             compute_exact_kernel(
                                   approximate_data_phase3)
46
47                             compute_accuracy_loss(reference_data_phase3
                                   , approximate_data_phase3)
```

Source: The Authors

responds to the iteration of phase one and is the index of a second map structure. This second map structure has an index corresponding to the iteration of phase two stored in the second column of the file. The index of the second map structure contains a vector with the iterations of the third column of the file. These iterations will be used to carry out the measurements of phase three.

From lines 16 to 37, we repeat the same steps as in phase two. The objective of phase three is to measure the loss of accuracy resulting from the interleaving of approximate and exact kernel versions at certain iterations of the loss of accuracy profiles from phase two. Thus, as in phase two, we need to repeat the execution of the profiles from phases one and two up to these iterations to obtain updated data regarding the profile executions.

After duplicating the data to isolate the reference execution and the approximate execution on lines 16 and 17, the executions begin on lines 20 and 21 within the main loop of line 19. The approximate execution, using the approximate kernel version, progresses until the current iteration is found at the index of the first map structure (line 23), indicating the first interleave of kernel versions from the approximate kernel version to the exact version (in the approximate execution only). From there, we duplicate the data again in lines 24 and 25, copying the current state of the data structures to new structures, and begin executing a new loop in line 27. In lines 28 and 29, we carry out the reference execution and the approximate execution with the exact kernel version (lines 28 and 29) from the iteration that generated the kernel's interleave up to the total number of iterations of the execution.

The loop on line 27 continues until the current iteration is found in the index of the second map structure on line 31. At this point, the second interleave of the kernel versions begins. This involves duplicating the data structures on lines 32 and 33 and executing the reference and approximate computations (now with the approximate kernel version) on lines 36 and 37 within the loop on line 35. This loop starts from the iteration on line 27 that produced the interleaving of the kernels, and continues for the total number of iterations of the execution.

As the loop on line 35 progresses, and the current iteration is present in the index vector equal to the iterations of the loops on lines 27 and 35, the measurements of phase three begin. A new duplication of the data in lines 41 and 42 is then carried out, followed by the start of the loop execution on line 43 and the reference and approximate executions on lines 44 and 45. In the third step, the approximate computation on line 45 is performed

in the exact kernel version. After each iteration, the loss of accuracy and other profile information from step three are calculated and stored on line 47. This cycle is repeated until all measurements are completed. The "Generate Phase 3 Profile" object in Figure 4.2 represents this stage.

The results of the measurements are displayed in the “Phase 3 + Sections” facet of Figure 4.5. The graph depicts the loss of accuracy profile for the approximate version (red line), as well as those for phase one (green lines), phase two (cyan lines), and phase three (purple lines). An additional measurement is taken in each section where the phase two loss of accuracy profiles intersect, allowing a measurement to be taken for the lower right section where there was no measurement in phase one. Furthermore, the significant difference in behavior between executions of the phase three exact kernel version in different subsections is observable.

4.2.4 Step 4 - Generating Accuracy Loss Statistics

The fourth step involves generating statistics on accuracy loss for each kernel version in the different measurement subsections. These statistics are derived from measurements taken in the three phases of the step three and from the approximate kernel version loss of accuracy profile that we used as baseline to perform the measurements in step three. The results are stored in a CSV file containing each subsection’s information, such as its identifier, start, end, and center on runtime of the iterations for the baseline execution (approximate kernel version loss of accuracy profile), for the approximate kernel version based on measurements taken in phase two of step three, and for the exact kernel version based on measurements taken in phases one and three of step three.

To calculate the loss of accuracy statistics, the first step is to identify which profiles intersect each subsection. Then, for each subsection, we can identify the loss of accuracy profiles whose measurements in step three started in that subsection and calculate the average loss of accuracy for each kernel version. The average loss of accuracy (or rate of loss of accuracy) is calculated for the set of iterations intersecting a subsection. If more than one measurement started in the same subsection, an average of the average loss of accuracy of iterations of each measurement is performed. This calculation is also performed for the baseline execution of the approximate kernel version. For each subsection intercepted by the baseline loss of accuracy profile, the average loss of accuracy is calculated.

In addition to calculating the rate of accuracy loss for each kernel version in the measurements performed in step three and the baseline execution, we also calculated the average runtime in each subsection. This average is calculated using only the iterations of the execution that originated in the subsection. If there is more than one measurement starting from the same subsection, we calculate the average of the runtime of each measurement. For the baseline execution, however, the average is calculated for each subsection intercepted by the loss of accuracy profile. This step is represented by the "Generate Accuracy Statistics" object in Figure 4.2.

4.2.5 Step 5 - Generating Performance Statistics

The fifth step involves generating performance statistics for the two kernel versions. These statistics are derived from each kernel version's accuracy loss profiles, represented by the "Exact Kernel Profile" and "Approximate Kernel Profile" objects in Figure 5.1. For each kernel version, we calculate the cumulative runtime of iterations in the accuracy loss profile and four runtime targets based on the difference in runtime between the two kernel versions. The results are stored in a CSV file containing the iteration, approximate kernel version and exact kernel version cumulative runtime, and the four targets.

These targets will later be used to generate an interleaved execution configuration of kernel versions in step six. They serve as reference points to minimize the impact on the runtime of interleaving the approximated kernel version with the exact kernel version, which has a higher runtime than the approximate version. The targets are calculated by adding a quarter of the difference in the runtime of the two kernel versions to the accumulated runtime of the iterations of the approximated kernel version.

The first target is the sum of the cumulative runtime of the previous iterations of the approximate version with a quarter of the difference between the cumulative runtime of the exact version. The second target is the sum of two quarters, the third of three quarters, and the fourth target is the sum of the cumulative runtime of the approximate version with the total difference between the two versions (which, in turn, is equal to the cumulative runtime of the exact version). This approach provides a four-level estimate of the potential impact of interleaving the execution of the exact kernel version with the approximate kernel version. The "Generate Performance Statistics" object in Figure 4.2 represents this step.

4.2.6 Step 6 - Generating Execution Configurations

In the sixth and final step, represented by the "Generate Execution Configuration" object in Figure 4.2, we generate the execution configuration based on the performance and accuracy loss statistics generated in steps four and five, respectively. Using this information, we conduct a simulation starting with the approximate kernel version and the statistics generated for the first subsection, which is always located at the bottom left by definition, as shown in Figure 4.5.

The simulation involves iterating over the total number of iterations of the execution and, at each iteration, add the subsection loss rate to the simulation's accuracy loss and the subsection's execution average to the simulation's runtime. As the simulation progresses, if the iteration or loss of accuracy (or both) exceeds the boundaries of the current subsection, we identify the next subsection to which the iteration and loss of accuracy values belong. Then we use the statistics of this next subsection.

If the accuracy loss reaches or exceeds the limit established for the simulation, a new interleaving of kernel versions is performed. After starting the simulation, executing the approximate kernel version is interleaved with the exact version when the loss limit is reached. The exact version will run until it reaches the first goal of step four or the simulation's accuracy loss has retracted more than 50% of the established limit.

If the first target of step four is reached and the loss of accuracy has not decreased, the exact version will continue to be executed until the next target is reached. If the loss of accuracy has not decreased after reaching the second target, the execution will continue until the third target is reached. If the loss of accuracy has not decreased after reaching the third target, the execution will continue until the fourth target or until the end of the execution if the loss of accuracy does not decrease after reaching the fourth target of step four.

If the simulation of the exact version's execution results in a retraction greater than or equal to 50% of the established limit, a new interleave of kernel versions is performed, regardless of whether any targets of step four have been reached. We established this 50% retraction limit to prevent the exact version from running an extended period when it only minimally impacts the simulated runtime and does not reach the first objective of step four even after having retracted significantly. This way, we can prioritize the execution of the approximate version, which offers a reduced runtime compared to the exact version's runtime.

This process continues until the total number of iterations of the execution is reached. In the end, we will have a set of iterations in which the simulation performed interleaves between kernel versions based on the performance statistics and loss of accuracy of the measurements performed in step three. This set of iterations and the respective kernel versions that started to be used in these iterations is stored in a text file. The file contains in the first line the total number of iterations of the application's execution, in the second line, the iteration number 0, indicating the kernel version with which the execution will start. On the remaining lines will be the iteration number and the kernel version into which the previous version should be interleaved. This step is represented by the "Generate Final Execution Configuration" object in Figure 4.2.

Figure 4.9 illustrates the instrumentation of the application for reading and executing interleaves according to the file generated in this step. Lines 1 to 8 read the file with the execution configuration information. The first line, containing the total number of iterations of the application's execution, is stored in a variable of type integer (line 6). The remaining lines are stored in a vector, where the index represents the iteration in which an interleaving of kernel versions will be necessary, and the value represents the kernel version that should be used from that iteration on (line 17).

The execution of the application's main loop begins on line 19. From then on, we check if the current loop iteration is in the vector index and if data conversion is necessary in case a new kernel version interleave occurs. Between lines 20 and 34, we check if the current loop iteration is in the vector index. If the iteration is present and the version to be used from that point on differs from the one used until then, we update the control variable of the current kernel version and convert data from the floating-point precision used in the previous kernel version to the floating-point precision used in the new kernel version. This conversion is only necessary if the precision used in the two kernels is different. From then on, the application continues executing (lines 36 to 39) using the current kernel version until a new interleaving is necessary.

Figure 4.9 – Execution configuration code instrumentation.

```
std::ifstream ecfile(filename)
std::string line

std::getline(ecfile, line)
std::stringstream linestream(line)
linestream >> iterations

std::vector <std::string> exec_config[iterations]

while(std::getline(ecfile, line))
    std::stringstream   linestream(line)
    int                 val1
    std::string         val2

    linestream >> val1 >> val2

    exec_config[val1].push_back(val2)

currentKernel = "approximate"

for (int i = 0; i < numiter; i++)
    if (!exec_config[i].empty() &&
        std::find(exec_config[i].begin(), exec_config[i].end(),
        "approximate") != exec_config[i].end())

        previousKernel = currentKernel
        currentKernel = "approximate"
        convert_data_from_approximate_to_exact_fp()

    else if (!exec_config[i].empty() &&
        std::find(exec_config[i].begin(), exec_config[i].end(),
        "exact") != exec_config[i].end())

        previousKernel = currentKernel
        currentKernel = "exact"
        convert_data_from_exact_to_approximate_fp()

    if (currentKernel == "approximate")
        compute_approximate_kernel()
    else
        compute_exact_kernel()
```

Source: The Authors

5 EXPERIMENTS

This Chapter presents the results of the experiments conducted using our methodology. We provide a detailed description of the experimental configurations and the applications used in the experiments. We then show the interleaved execution configurations of multiple kernel versions generated by our methodology for different problem sizes and accuracy loss limits.

5.1 Experiment Setup

We run all experiments on a compute node of the Grid5000 (BOLZE et al., 2006) grid computing infrastructure. The node, named Grouille-2, comprises two AMD EPYC 7452 CPUs, each with 32 cores and 64 threads, for a total of 128 threads. These CPUs run at a base frequency of 2.35GHz and maximum boost frequency of 3.35GHz and have a cache memory of 128MB each.

Additionally, the node comprises two NVIDIA A100 GPUs, each with 40GB of HBM2 global memory and a memory bandwidth of 1,555GB/s. Each GPU has 3456 FP64, 6912 FP32, and 13824 FP16 cores, i.e., a ratio of FP64, FP32, and FP16 cores of 1:2:4. Moreover, each GPU has a total of 432 tensor cores, which are cores specialized in performing FP16/FP32 mixed-precision Fused Multiply-Add (FMA) operations. Lastly, the CUDA computing capability of these GPUs is 8.0.

The node has as its operating system a Debian GNU/Linux version 11 (Bullseye) with kernel version 5.10.0-20 and a main memory of 128GB. In addition, it uses the GCC C/C++ compiler version 10.2.1 and NVIDIA CUDA NVCC compiler version 11.2.152.

To evaluate the performance and effectiveness of the method proposed in this work, we performed experiments using two different problem sizes and two configurations of the number of iterations for the execution of the main loop for each application, as well as three different thresholds of loss of accuracy. While we defined the problem sizes based on the availability of data sets, the number of iterations was determined based on the default number adding a second run with twice as many iterations. Finally, we defined the three thresholds of loss of accuracy based on the maximum loss of accuracy of the approximate (less precise) kernel version. We divide this value into four parts and use the three initial ones (e.g., for a loss of accuracy of 8%, we would use as accuracy loss thresholds 2%, 4%, and 6%, since 0% corresponds to an exact execution and 8%

corresponds to the execution of the approximate kernel itself).

Finally, to divide the area corresponding to the number of iterations and the loss of accuracy of the approximate kernel version (or the version with the most significant loss of accuracy in case of using two approximate kernel versions), we use five configurations of subsections. To obtain the subsections, we divide the number of iterations and the loss of accuracy of the kernel version with the highest loss into 2, 3, 4, 5, and 6 parts, resulting in 4 (2x2), 9 (3x3), 16 (4x4), 25 (5x5) and 36 (6x6) subsections. In this way, we can evaluate the efficiency of generating the execution configurations of our method with different amounts of measurements since as the number of subsections increases, the greater the number of measurements on which the method will generate the execution configurations.

The runtime of each application is measured using the *gettimeofday* function from the *sys/time.h* library of the C++ programming language. To achieve this, we record the timestamp at the start of the main loop of the iterative application and immediately after the completion of the main loop execution. We then calculate the execution time in milliseconds for each execution from this information.

For measuring energy consumption during application execution, we utilize the NVProf tool. NVIDIA developed this profiling tool for analyzing the performance of CUDA applications. It provides detailed information about GPU activities, such as memory transfers, kernel launches, and energy consumption, allowing developers to measure the performance of their CUDA code. The tool intercepts CUDA API calls and instruments the CUDA runtime and driver libraries to collect detailed information about GPU activities.

Our experiments collect information such as memory usage and bandwidth, compute capability utilization, and energy consumption. We instruct the tool to collect data as quickly as possible, every 1 ms, and also collect the timestamp of the collection. This way, we obtain the subsection of measurements corresponding to the start and end interval of the main loop execution of the running application through the respective timestamps. We can then accurately calculate the resource utilization and energy consumption of the application execution.

5.2 Applications

We will conduct experiments using three different physical simulation applications to assess the effectiveness of the methodology proposed in this work. These applications are all iterative but execute a fixed number of iterations instead of running until a set of conditions is met. Additionally, they rely on computations performed on the GPU using three-dimensional data domains. By testing the methodology in multiple applications, we can better understand its generalizability and applicability to various scenarios. We will also carefully analyze our experimental results to identify potential areas for improvement and optimization.

5.2.1 LBM3D

The Lattice Boltzmann Method (LBM) is a powerful Computational Fluid Dynamics (CFD) simulation technique that models fluid flows in three-dimensional domains. The method uses a grid-based approach to simulate fluid flow, dividing the fluid into a lattice of discrete points. Mathematical models describe how fluid particles interact and accurately simulate their behavior under various conditions. Each particle has properties that describe the fluid's state at a particular point in time, including density, velocity, and temperature. Figure 5.1 shows the pseudo-code of the application's main loop and kernels instrumented to run execution configurations of our methodology.

Figure 5.1 – LBM3D application pseudo-code.

```
1  initialize_data()
2
3  for (int i = 0; i < numiter; i++)
4      if (current_kernel == "half")
5          redistribution_kernel_half()
6          propagation_kernel_half()
7          bounceback_kernel_half()
8          relaxation_kernel_half()
9      else
10         redistribution_kernel_float()
11         propagation_kernel_float()
12         bounceback_kernel_float()
13         relaxation_kernel_float()
```

Source: The Authors

The LBM3D application consists of four kernels, as seen in the pseudo-code. While the bounceback and propagation kernels handle the propagation of forces and col-

lisions with obstacles, respectively, without performing arithmetic operations, the redistribution and relaxation kernels perform a significant amount of arithmetic operations. For our experiments, we used two kernel versions: an approximate version using 16-bit floating-point arithmetic operations (half) (lines 5 to 8) and an exact version in 32 bits (lines 10 to 13). The interleaving between the two kernel versions is performed simply by identifying the moment of interleaving and the kernel version to be executed from then on through the execution configurations generated by our method and read before the main loop execution of the application on line 3.

5.2.2 Euler3D

Euler3D is a Computational Fluid Dynamics (CFD) solver that uses an unstructured grid to compute Euler equations for compressible flow for the three-dimensional domains. The method is used by researchers to simulate fluid flow in applications such as aircraft design, wind turbines, and astrophysics. It uses numerical methods and high-order accurate schemes for the spatial and temporal discretization to solve the equations on a grid of points in three-dimensional space. Figure 5.2 shows the pseudo-code of the application's main loop and kernels instrumented to run execution configurations of our methodology.

Figure 5.2 – Euler3D application pseudo-code.
Source: The Authors

```
1  initialize_data()
2
3  for (int i = 0; i < numiter; i++)
4      if (current_kernel == "loopperf")
5          if (5 == count)
6              count = 0
7          else
8              compute_step_factor_kernel_float()
9              compute_flux_kernel_float()
10             time_step_kernel_float()
11             count++
12     else
13         compute_step_factor_kernel_float()
14         compute_flux_kernel_float()
15         time_step_kernel_float()
```

Source: The Authors

he Euler3D application consists of three kernels, as shown in the pseudocode. The first kernel performs data associations and preparations, while the second kernel com-

putes the method's equations and incurs a significant computational load of arithmetic operations. The third kernel manages the simulation's time advancement. For our work's experiments, we used two kernel versions: an approximate version that uses the loop perforation technique to skip one iteration of the main loop every five iterations (lines 5 to 11) and an exact version in 32 bits (lines 13 to 15). Our method generates execution configurations that are read before the main loop execution of the application on line 3. The interleaves between the two kernel versions are performed simply by identifying the moment of interleaving and the kernel version to be executed.

5.2.3 HotSpot3D

HotSpot3D is a benchmark that simulates heat propagation in a three-dimensional space. It estimates processor temperature based on an architectural floor plan and simulated power measurements. The HotSpot3D application calculates a chip's or circuit board's temperature profile, considering power dissipation, thermal conductivity, and other parameters. It generates output data that can be used to analyze thermal performance and optimize system design. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip. Figure 5.3 shows the pseudo-code of the application's main loop and kernels instrumented to run execution configurations of our methodology.

Figure 5.3 – HotSpot3D application pseudo-code.
Source: The Authors

```
1 initialize_data()
2
3 for (int i = 0; i < numiter; i++)
4     if (current_kernel == "half")
5         compute_hotspot_kernel_half()
6     else
7         compute_hotspot_kernel_float()
```

Source: The Authors

The HotSpot3D application consists of only one kernel, as the pseudo-code shows. This kernel is responsible for all of the simulation's computation, requiring a significant computational load of arithmetic operations. For our experiments, we used two versions of the kernel: an approximate version using 16-bit floating-point arithmetic operations (half) (lines 5) and an exact version using 32-bit floating-point arithmetic (lines 7). Our method generates execution configurations, which are read before the start of the main

application loop on line 3. The interleaves between the two kernel versions are performed simply by identifying the moment of interleaving and the kernel version to be executed.

5.3 Results and Analysis

5.3.1 LBM3D

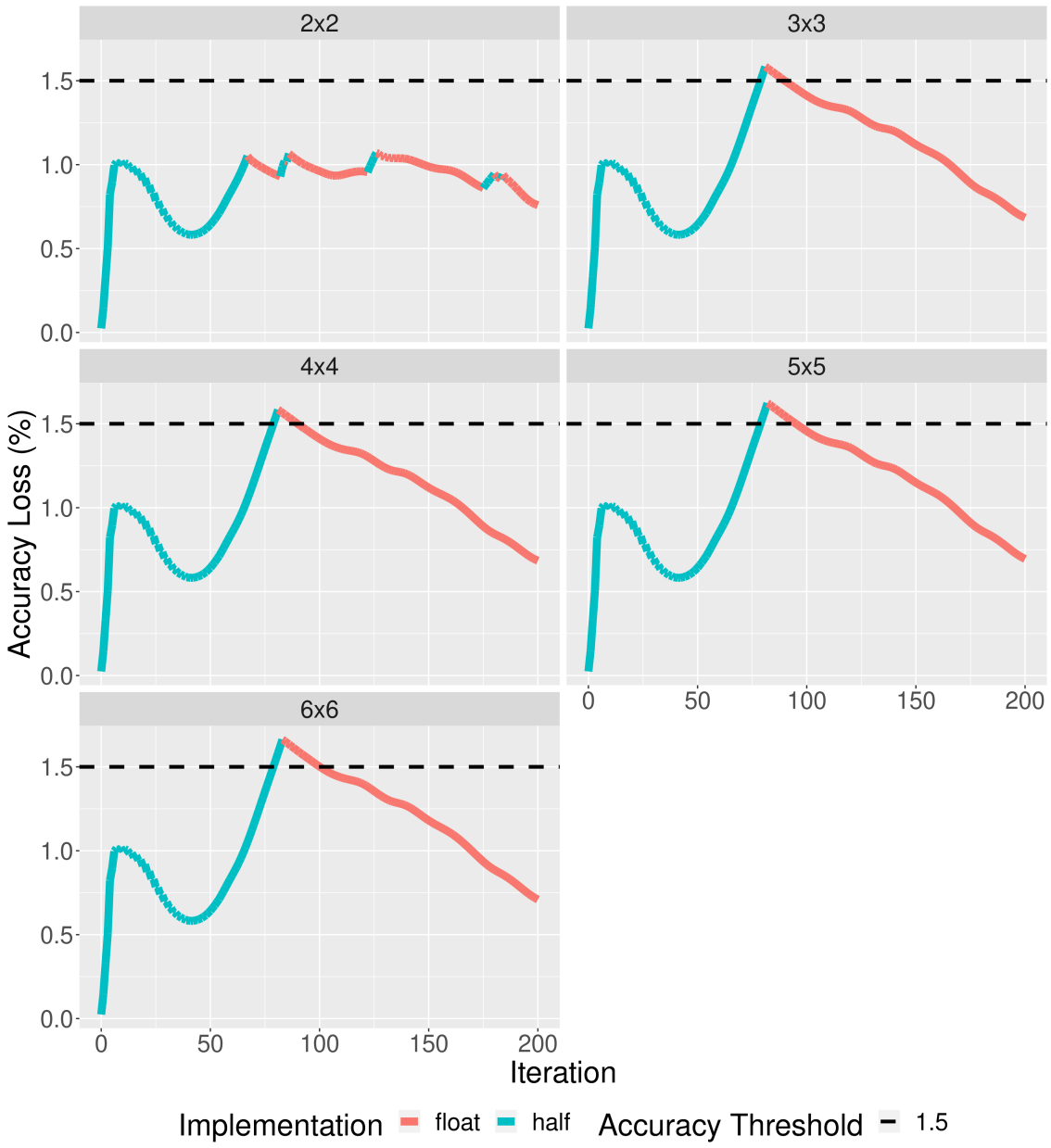
Figure 5.4 shows the loss of accuracy profile of the LBM3D application for the execution configuration generated by our method using 1.5% as a loss of accuracy threshold. The size of the problem used was a 3D data domain of dimensions 64x64x64 and a quantity of 200 iterations. The figure shows, on the X axis, the evolution of the execution over time (iterations) and the respective loss of accuracy of the progress of the execution on the Y axis on each of the five subsection configurations (2x2, 3x3, 4x4, 5x5 and 6x6).

By using an accuracy loss threshold of 1.5% for the execution with 200 iterations, it is possible to observe that the execution configuration produced by the method is identical using both subsections, 3x3, 4x4, 5x5, and 6x6. Despite slightly exceeding the defined threshold of accuracy loss, in both cases, the execution configuration produced by the method consists of executing approximately 80 iterations using the half version of the kernel and, from then until the end of the execution, using the float version (exact) of the kernel. In the 2x2 subsections configuration, the method produced an execution configuration that interleaves the half and float kernels multiple times. Furthermore, this configuration kept the accuracy loss at approximately 1.1%, below the established threshold.

While the execution configuration generated with the 2x2 subsections configuration had room to run a few additional iterations before reaching the 1.5% threshold, in the remaining subsections, there are no multiple interleaves of the two kernel versions to increase the execution of the fastest half kernel version. The execution configuration generated by the method could have interleaved the half and float kernel versions more times to take advantage of the retraction in the loss of accuracy after the initial change from the half version to the float version. Similar to subsection 2x2, where after the first interleave in iteration 67, there are multiple subsequent interleaves between both kernel versions.

Figure 5.5 shows the loss of accuracy profile for a 3.1% loss of precision threshold for the same problem size and number of iterations of the LBM3D application. Contrary to the previous limit of 1.5%, in both configurations of subsections, the method generated

Figure 5.4 – LBM3D accuracy loss profile using a problem size of 64, 200 iterations, and an accuracy loss threshold of 1.5%.

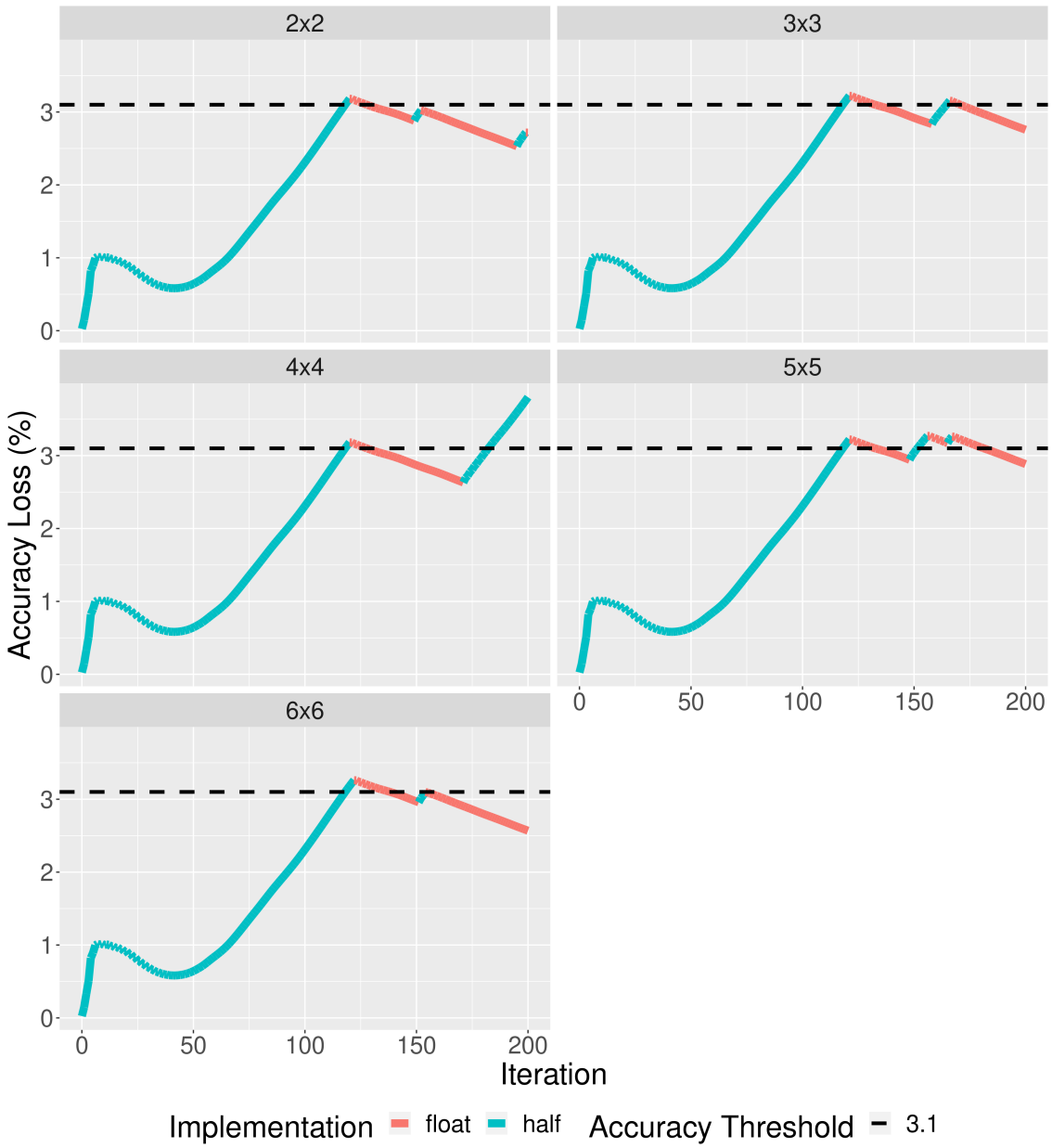


Source: The Authors

an execution configuration with two or more interleaves of the two versions of kennels. Although the configuration of 4x4 subsections (16 measurement subsections) exceeds the determined accuracy loss threshold, in the other configurations of subsections, there are only minor deviations from the threshold.

Because it is a higher threshold, it allows the execution of a more significant number of iterations using the half kernel version at the start, reaching the determined threshold only after surpassing 100 iterations. From then on, the method creates an execution with interleaves between the half and float versions of the kernel to accelerate the execu-

Figure 5.5 – LBM3D accuracy loss profile using a problem size of 64, 200 iterations, and an accuracy loss threshold of 3.1%.



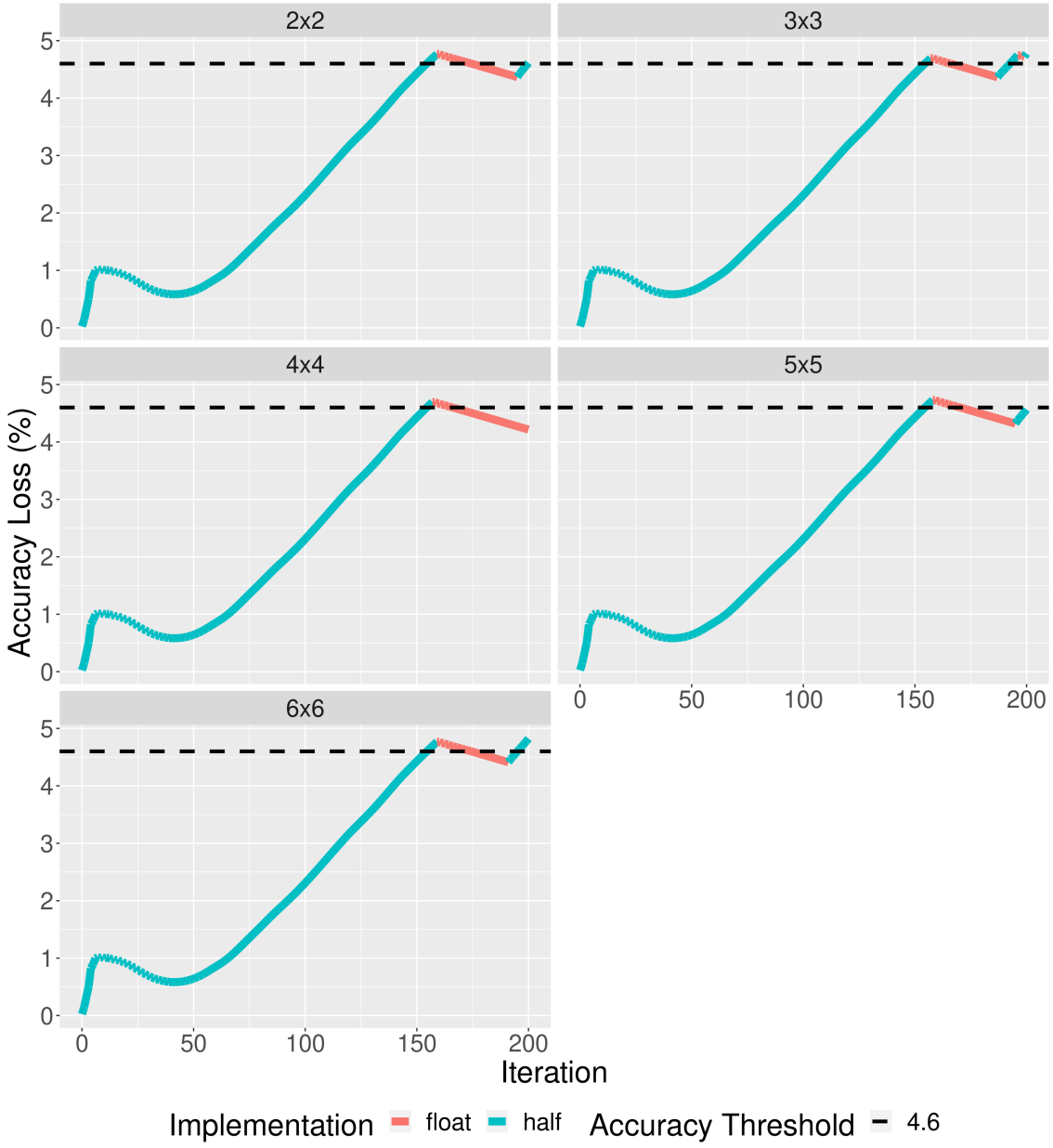
Source: The Authors

tion through a higher use of the half version. However, due to the slow retraction of the loss of accuracy during periods of execution of the float version of the kernel, subsequent executions of the half version occur during only a few iterations, quickly returning to the float execution.

With a threshold of 4.6% for the same problem size and number of iterations, the same is achieved only after the initial run using the half kernel version for more than three-quarters of iterations, as shown in Figure 5.6. Although the method generates an execution configuration that respects the given threshold and interleaves the two kernel

versions, there is little room for executing a significant number of iterations in half before reaching the threshold again.

Figure 5.6 – LBM3D accuracy loss profile using a problem size of 64, 200 iterations, and an accuracy loss threshold of 4.6%.

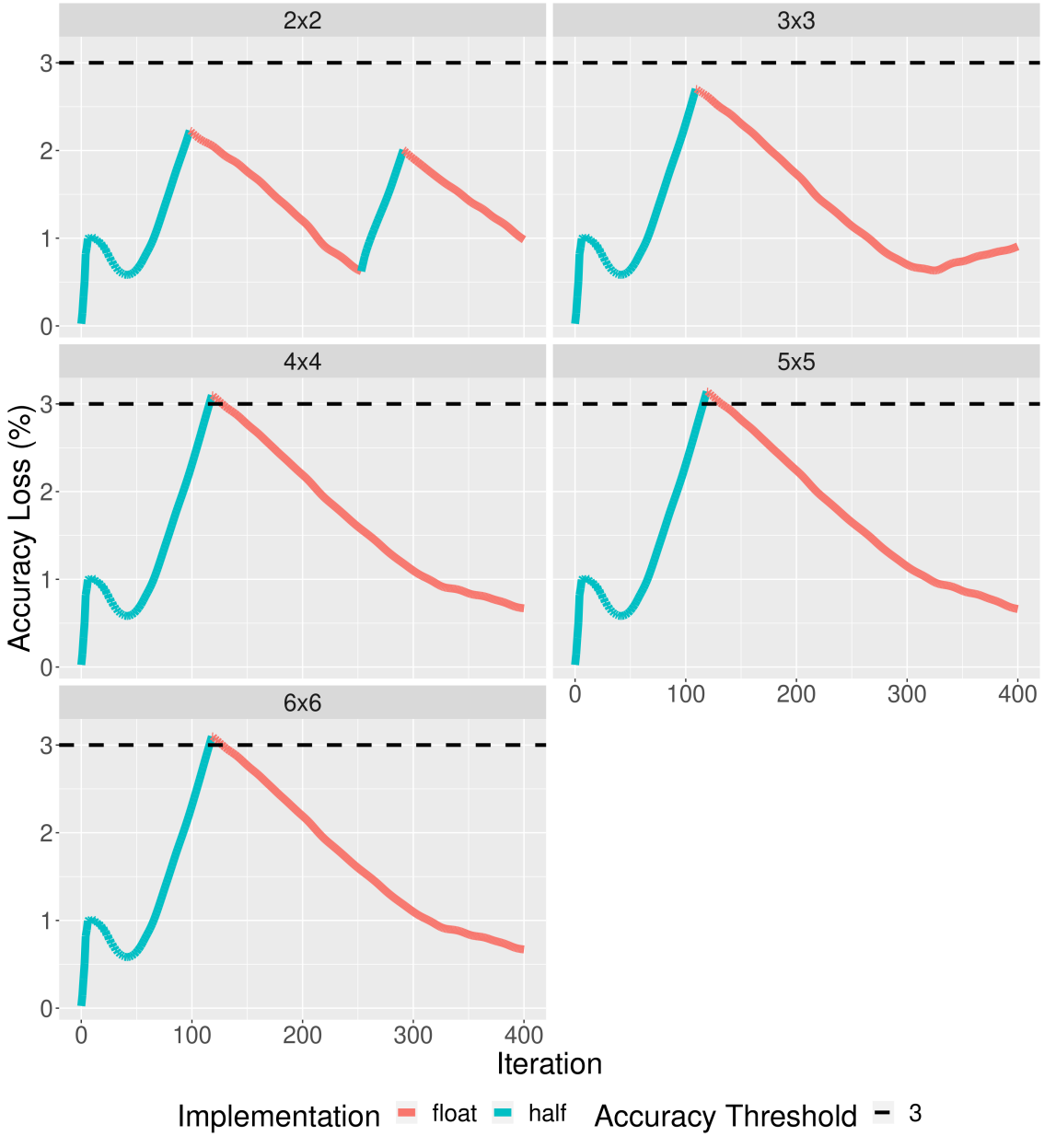


Source: The Authors

Figure 5.7 presents the loss of accuracy profile for the problem size 64x64x64, now with 400 iterations for the application execution and a threshold of loss of accuracy of 3%. In a similar way to the execution with only 200 iterations in Figure 5.4, in the 2x2 subsections configuration, the execution configuration generated by the method interleaves both kernels more than once. In the other subsections configurations, as soon as the execution reaches the threshold, the rest of the iterations are performed only on the

float kernel version. Furthermore, in the 2x2 subsection configuration, the generated configuration stays below the loss threshold with a significant distance where the faster kernel version half could run a more significant number of iterations to improve the execution.

Figure 5.7 – LBM3D accuracy loss profile using a problem size of 64, 400 iterations, and an accuracy loss threshold of 3%.



Source: The Authors

Figures A.1 and A.2 show the loss of accuracy profile for the thresholds of loss of accuracy 6% and 9% with 400 iterations for the execution of the application. As can be seen, in both cases, there is only one interleave of the two kernel versions when the accuracy loss reaches the determined thresholds in both five measurement configurations. From that moment on, the rest of the execution is performed using only the float kernel

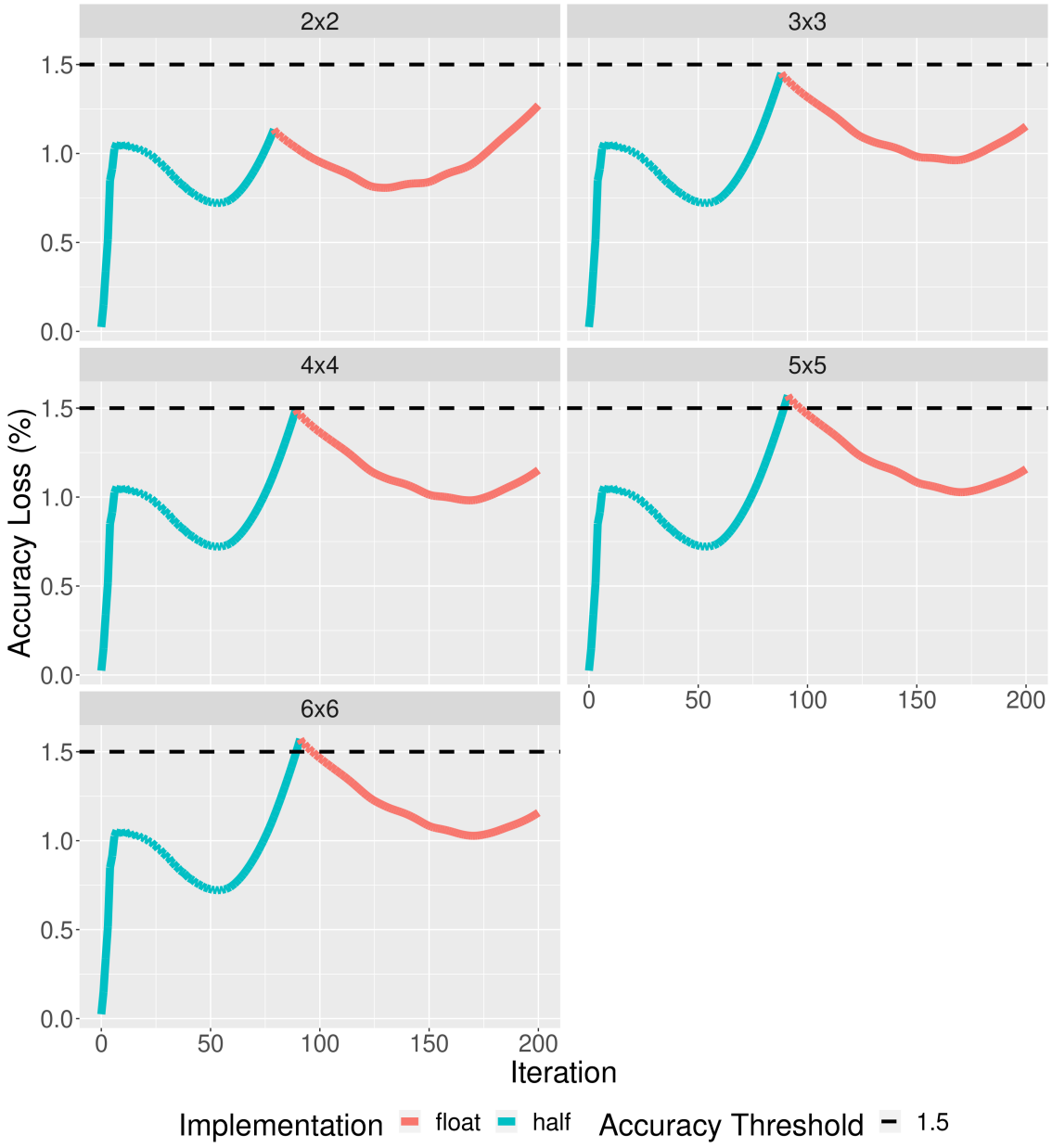
version. While in the 6% threshold, there is space to perform at least one more interleave, in the threshold of 9%, the half version executes practically three-quarters of the execution, and the low retraction in the rest of the execution with the float version limits the performance of more interleaves.

Now, using a larger problem size, 128x128x128, and 200 iterations to run the application, the behavior of the loss of accuracy profile changes considerably. By using a 1.5% loss of accuracy threshold, the method generated an execution configuration with a single interleave of the two kernel versions for the five configurations of measurement subsections, as shown in Figure 5.8. In both cases, as the loss of accuracy approaches or reaches the threshold, the half kernel version is replaced by the float version for the remainder of the execution. In this case, although there is a relative space for more interleaving, the loss of accuracy of the float implementation itself starts to increase after some iterations, making it challenging to include more interleaving between the half and float kernel versions.

With 400 iterations to run the application and an accuracy loss threshold of 3.3%, the results, shown in Figure 5.9, are similar to problem size 64 with 400 iterations in Figure 5.7. However, in the case of size 128 and 400 iterations, there are two interleaves of the kernel versions in the 5x5 and 6x6 configurations and three interleaves in the 2x2 measurement subsections configuration. Although interleaving does not take advantage of the available space for executing a more significant number of iterations when the float kernel version is replaced by the half version in 5x5 and 6x6 configurations, in the 2x2 configuration, there is a good use of space after the first interleave of the half by the float.

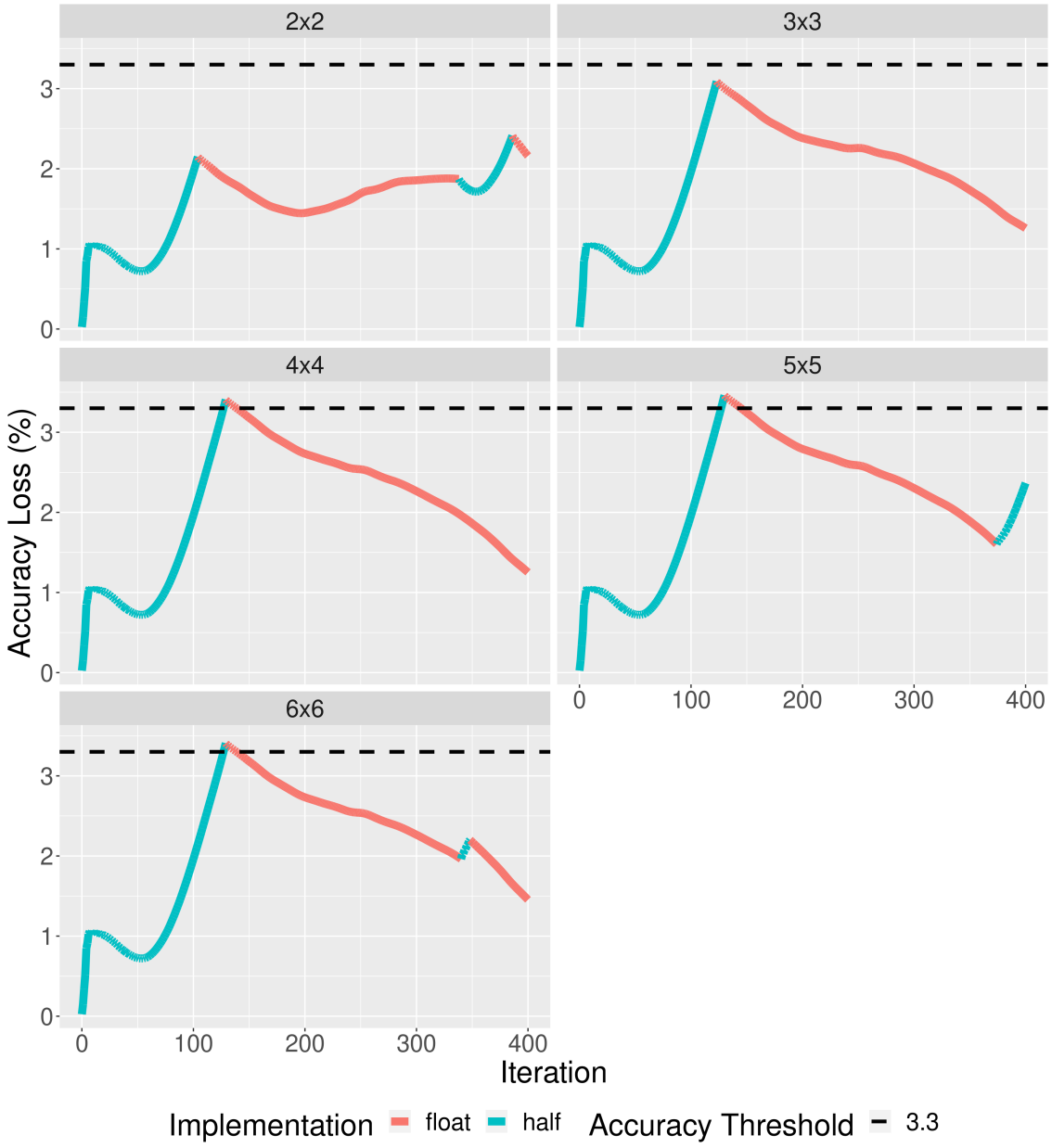
When using the threshold of loss of accuracy 3.1% and 4.6% with 200 iterations for the execution of the application (Figures A.3 and A.4) and 6.6% and 9.9% with 400 iterations (Figures A.5 and A.6) in problem size 128, in almost all five subsections configurations there is only one interleaving of kernel versions. Except for the configurations of subsections 3x3 of Figures A.3 and A.4, the execution configuration generated by the method was not able to interleave the two kernel versions more often to aim to optimize the application execution. The lack of further interleaves is due to the low retraction with a threshold of 4.6% and 9.9% in runs with 200 and 400 iterations, respectively. However, this is not the case at the thresholds of 3.1% and 6.6% with 200 and 400 iterations, respectively, where there is notable room for performing at least one more interleave to increase the number of iterations executed in the half version.

Figure 5.8 – LBM3D accuracy loss profile using a problem size of 128, 200 iterations, and an accuracy loss threshold of 1.5%.



Source: The Authors

Figure 5.9 – LBM3D accuracy loss profile using a problem size of 128, 400 iterations, and an accuracy loss threshold of 3.3%.



Source: The Authors

5.3.2 Euler3D

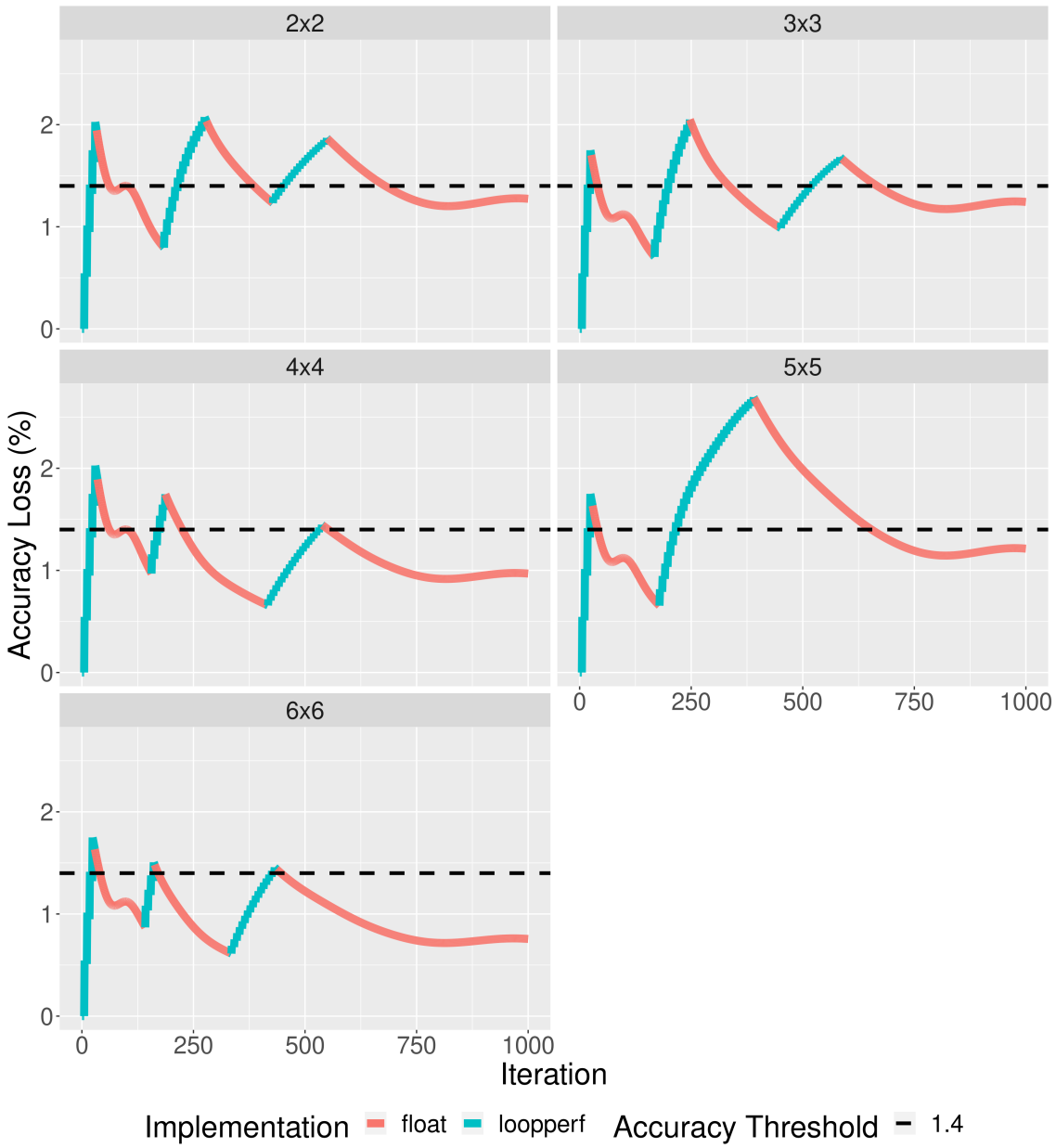
Figure 5.10 presents the loss of accuracy profile of the execution configurations generated by our method for the Euler3D application in a problem size of 97152 elements, 1000 iterations, and a loss of accuracy threshold of 1.4%. One of the first observations is that in the five measurement subsection configurations, the execution configuration generated by the method exceeds the defined accuracy loss threshold. While in the 5x5 subsection configuration, the threshold is exceeded by approximately 1.5%, in the rest of the subsection configurations, it is approximately 0.5% or less.

By analyzing the interleaves of the *loopperf* and *float* kernel versions, it is possible to observe that the method generated execution configurations with multiple interleaves in the five configurations of measurement subsections. Except for the 5x5 configuration, with only three interleaves of the kernels, in the other configurations, there are five interleaves of the two kernels with a considerable balance of the execution peaks in the *looperf* kernel, despite this balance being slightly above the established threshold. The highlight is the execution configuration generated with the data collected in the configuration of 6x6 subsections, where there is the slightest deviation from the determined threshold with peaks and valleys of kernel interleaves at similar levels.

Figure 5.11 shows the loss of accuracy profiles of the run configurations for the 2.8% loss of accuracy limit in problem size 97152 and 1000 iterations. Again, as with the 1.4% loss of accuracy limit in Figure 5.10, the configuration of 5x5 measurement subsections was the greatest due to the determined limit, approximately 1.1% higher. However, in the other configurations of subsections, the dues of the loss of accuracy limit were light. Moreover, the method generated runtime configurations with multiple merges of the *loopperf* and *float* kernel versions, emphasizing the 3x3 configuration with good use of space, significantly respecting the established limit.

With an accuracy loss threshold of 4.2%, however, in the measurement configurations subsections of 2x2 and 3x3, the threshold is exceeded by approximately 1.5%, as shown in Figure 5.12. In contrast, in the remaining ones, the execution configuration considerably respects the threshold. In this case, there is limited space for performing multiple interleaves. However, the method generated configurations for the 4x4, 5x5, and 6x6 subsection configurations with high utilization of the faster *loopperf* kernel version, especially for the 5x5, where the generated execution configuration does not exceed the threshold and runs as many iterations as possible with the *loopperf* kernel version.

Figure 5.10 – Euler3D accuracy loss profile using a problem size of 97152, 1000 iterations, and an accuracy loss threshold of 1.4%.

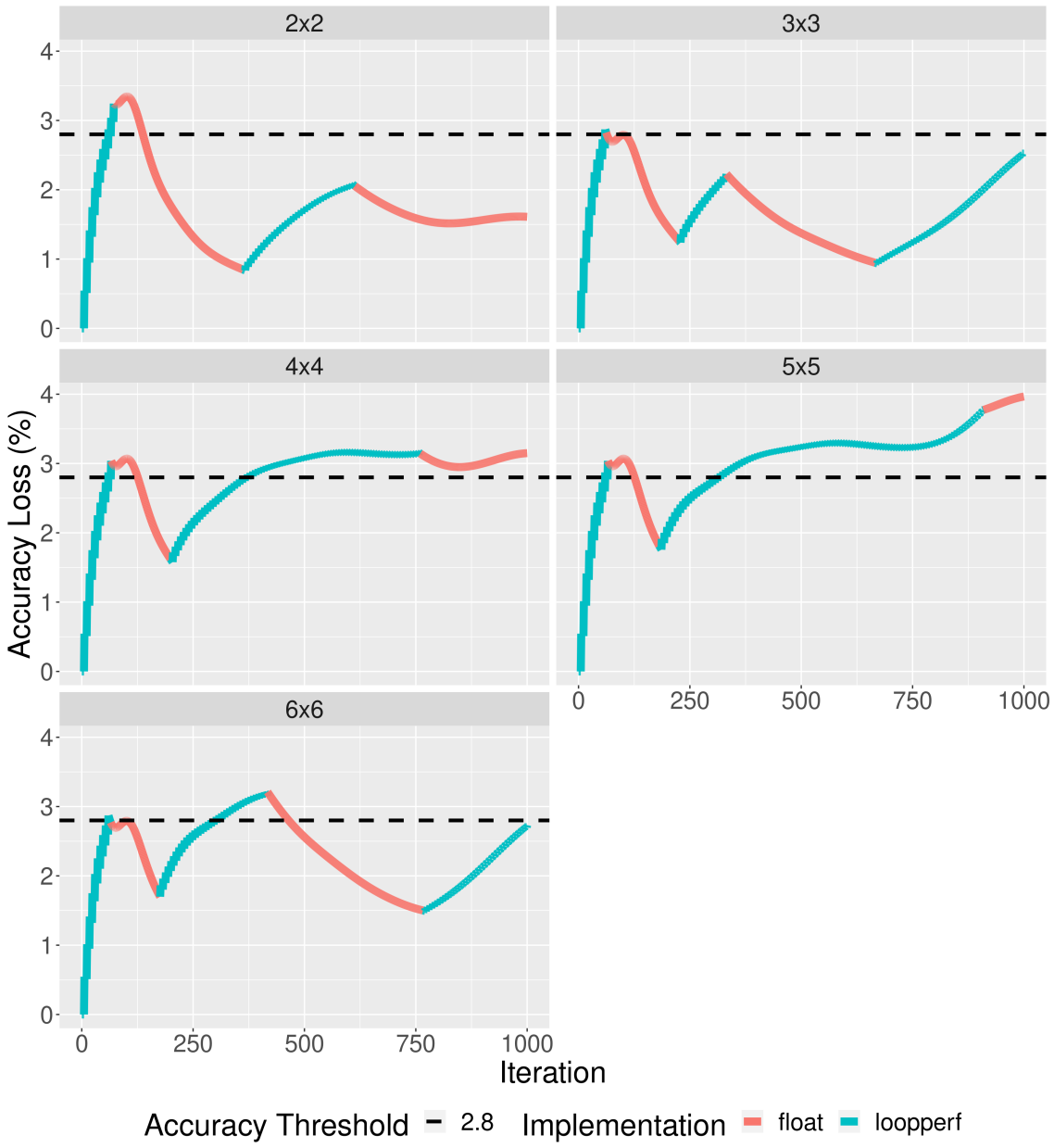


Source: The Authors

Figure 5.13 presents the profile of loss of accuracy in a problem size of 97152 elements, 2000 iterations for the application run, and a threshold of loss of accuracy of 1.4%. As in Figure 5.10 with 1000 iterations, the accuracy loss threshold is considerably exceeded in the five measurement sections configuration, especially in the 3x3 configuration, where the deviation above the threshold exceeds 0.5%.

However, in this case, the behavior of executing the two kernel versions at the end of the execution changes slightly. Whereas with 1000 iterations running in most subsection configurations, the interleaves have balanced tops and valleys at similar levels,

Figure 5.11 – Euler3D accuracy loss profile using a problem size of 97152, 1000 iterations, and an accuracy loss threshold of 2.8%.

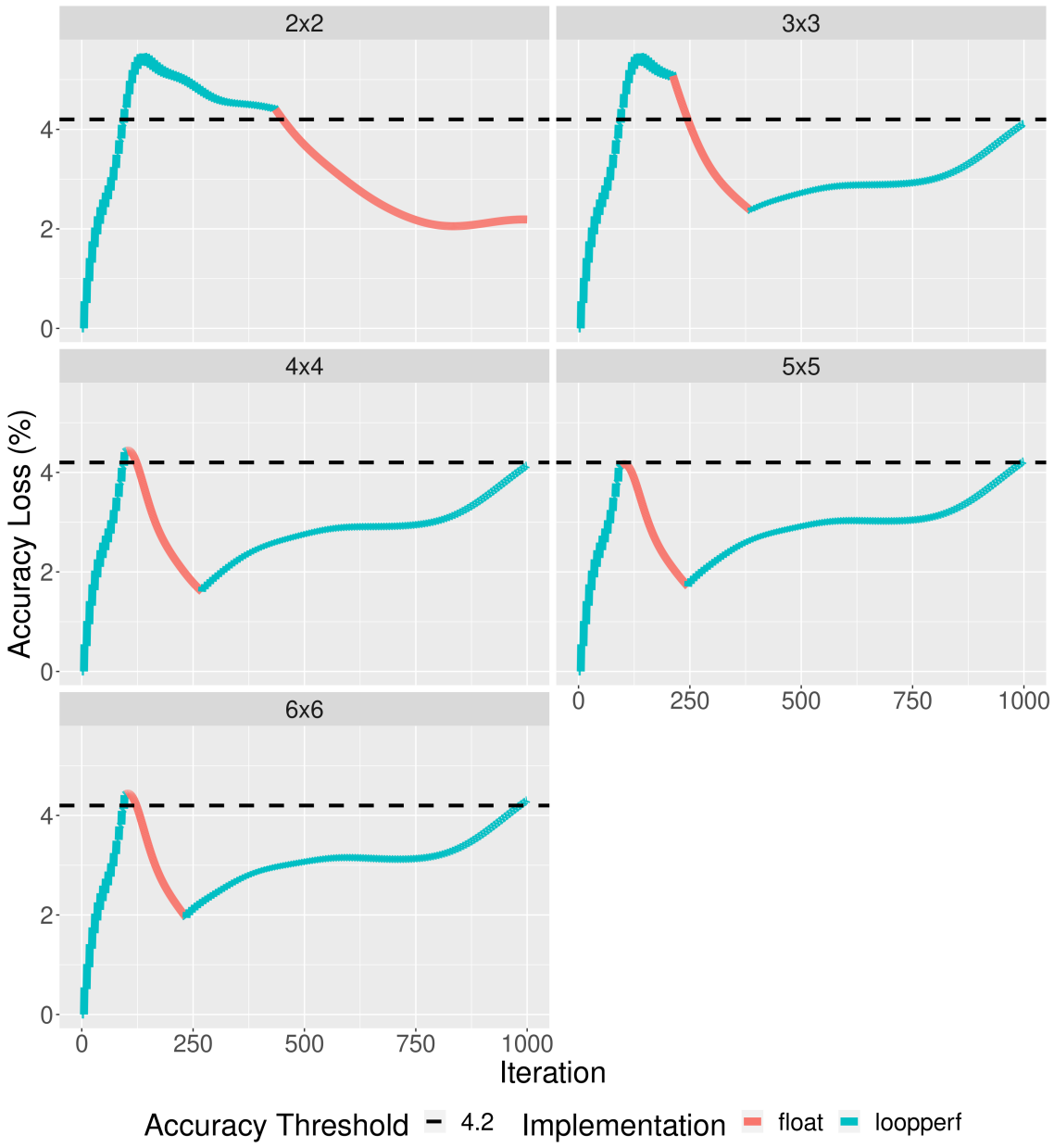


Source: The Authors

this is not the case with 2000 iterations. If we look at the execution after half of the iterations, in configurations of subsections such as 2x2, 4x4, and 6x6, the trend of loss of accuracy is to recede. In the case of the 4x4 configuration, from iteration 1100 onwards, the rest of the execution uses the *loopperf* kernel version. The loss of accuracy oscillates within the same range and does not approximate the thresholds.

The same occurs with an accuracy loss threshold of 2.8%, as shown in Figure A.7, where the execution configuration generated by the method and the behavior of the *loopperf* kernel version are similar to the ones for threshold 1.4% in Figure 5.13.

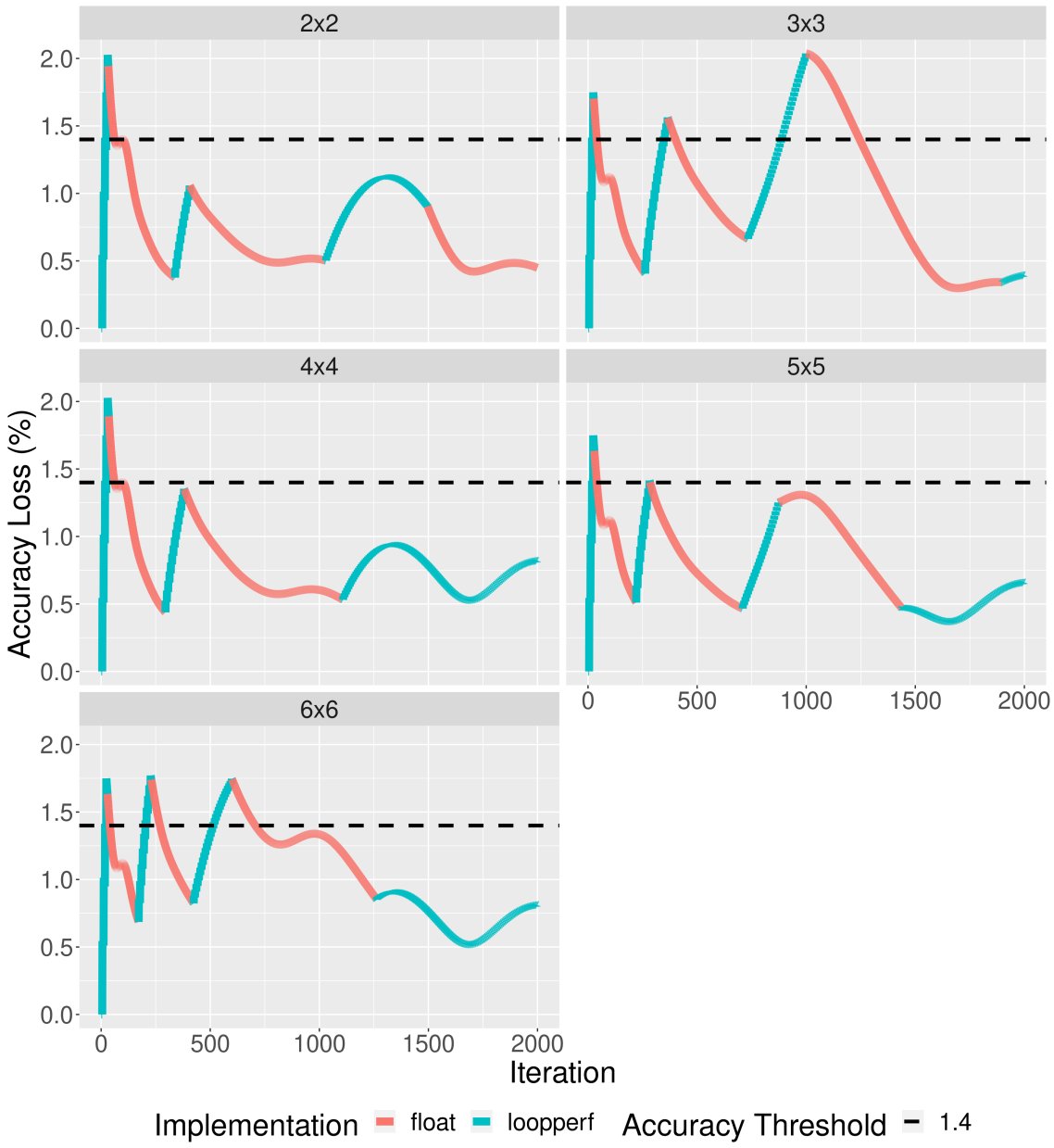
Figure 5.12 – Euler3D accuracy loss profile using a problem size of 97152, 1000 iterations, and an accuracy loss threshold of 4.2%.



Source: The Authors

With an accuracy loss threshold of 4.2%, the *loopperf* kernel's execution behavior changes even more toward the end of execution. In the configurations of 2x2, 3x3, and 6x6 subsections, for example, despite the iterations from 1200 running in the *loopperf* version, the loss of accuracy recedes, contrary to what happens at the beginning of the execution, as can be seen in Figure A.8. Thus, although the method performs some interleaving at the beginning of the execution configuration, it is not necessary for the rest due to the characteristic accumulation of roundings of the *loopperf* kernel version toward the end of the execution.

Figure 5.13 – Euler3D accuracy loss profile using a problem size of 97152, 2000 iterations, and an accuracy loss threshold of 1.4%.

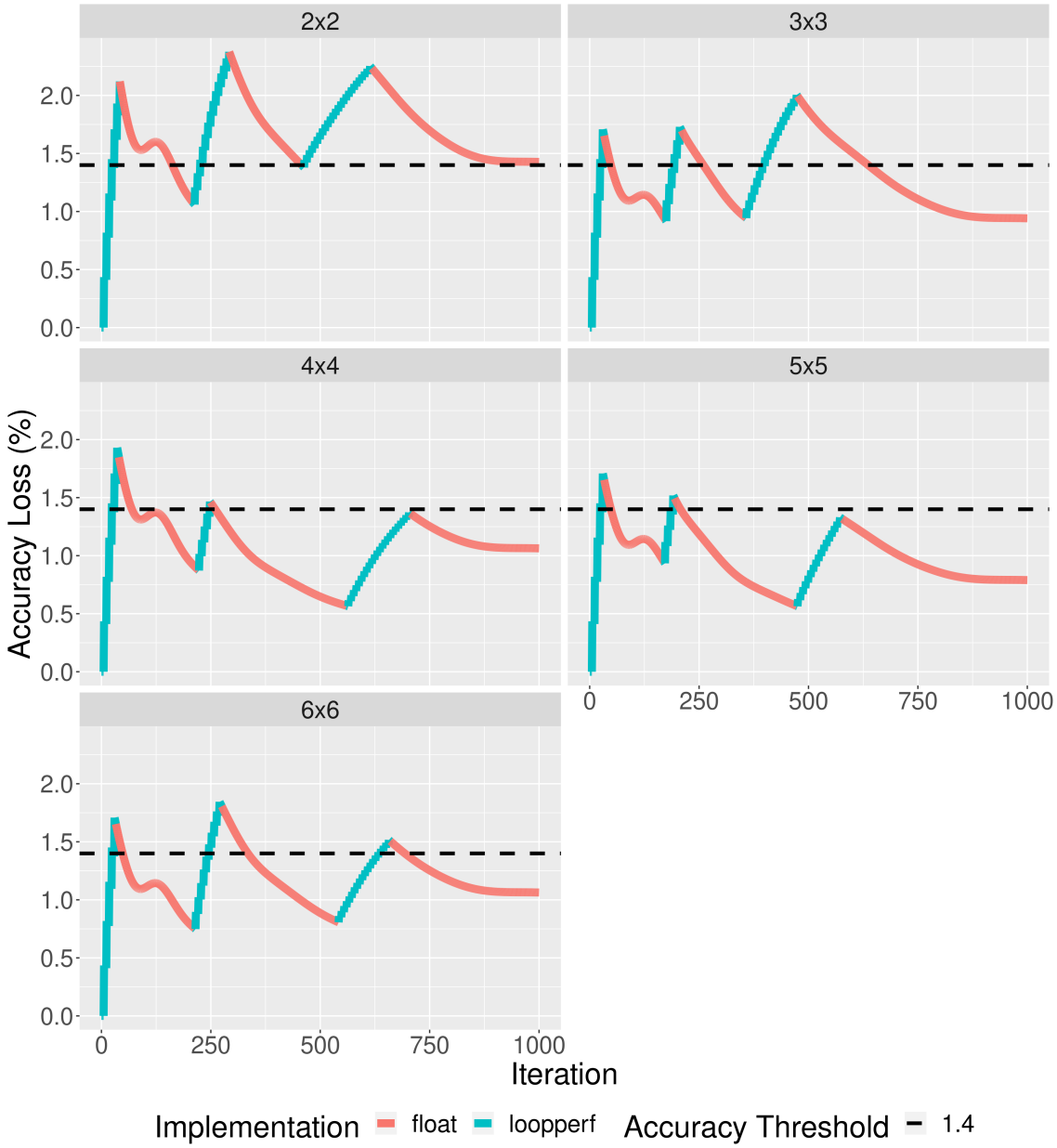


Source: The Authors

Figure 5.14 presents the loss of accuracy profile for a problem size of 193536 elements, 1000 iterations, and a loss of accuracy threshold of 1.4%. The execution configurations generated by the method are similar to those generated for the 97152 element size in Figure 5.10. However, in this case, the 5x5 measurement subsection configuration was not off the loss of accuracy limit by more than two times. On the contrary, the configuration generated for the configuration of 5x5 subsections is the one with the finest balance between peaks and valleys in the interleaving of kernels and even the lowest deviation from the determined loss of accuracy threshold. In contrast, the configuration 2x2

and 3x3 have the most significant deviation, reaching close to 1%.

Figure 5.14 – Euler3D accuracy loss profile using a problem size of 193536, 1000 iterations, and an accuracy loss threshold of 1.4%.

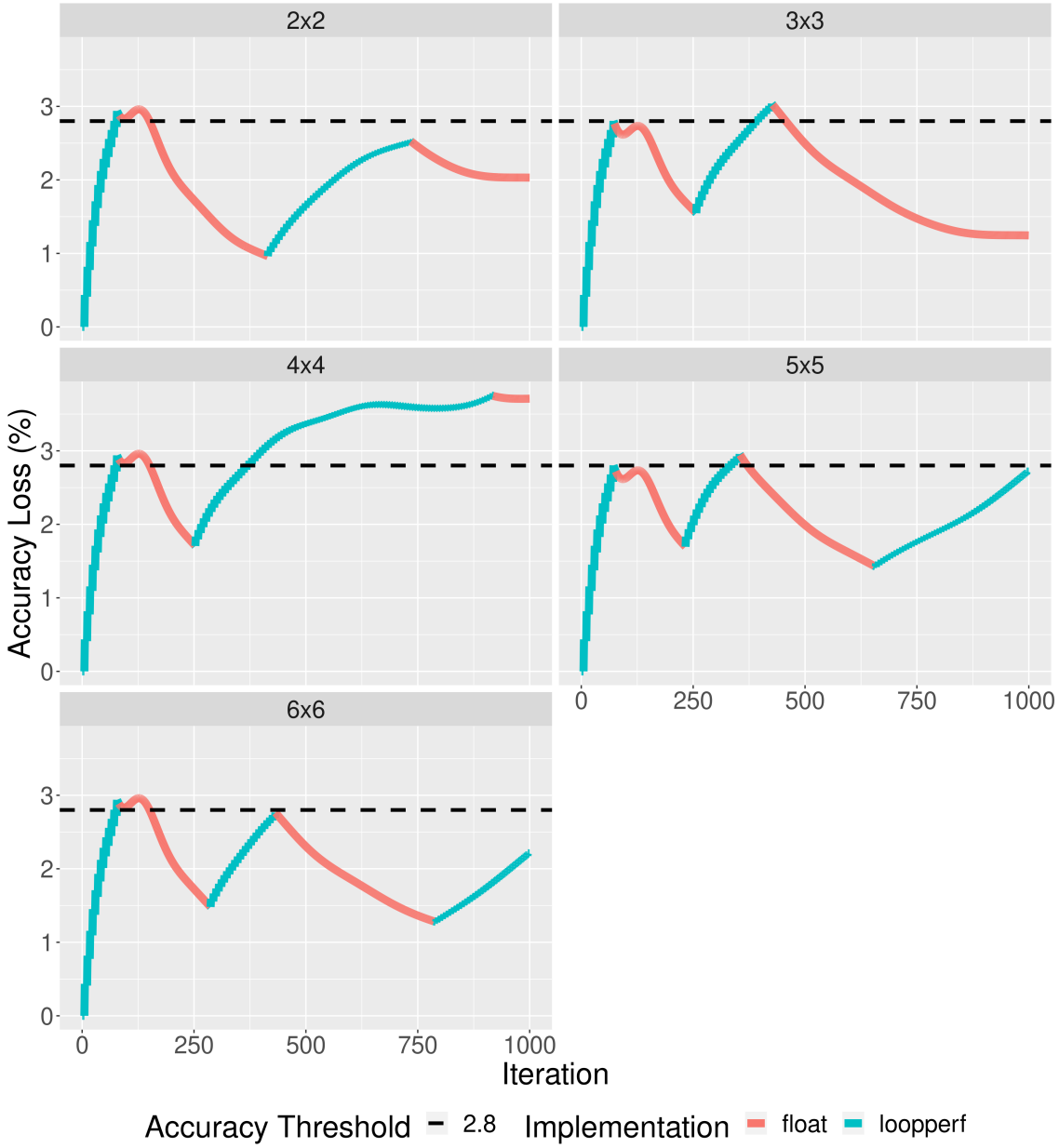


Source: The Authors

Again, with execution configurations identical to those with a problem size of 97152 elements in Figure 5.11, Figure 5.15 shows the loss of accuracy profile of the execution configurations generated by the method for a problem size of 193536 elements, 1000 iterations and an accuracy loss threshold of 2.8%. However, in this case, the only configuration that significantly exceeds the loss of accuracy threshold is the execution configuration for the 4x4 subsections configuration. In the other subsection configurations, the execution configurations generated by the method make reasonable use of the

space provided by the retractions of executions in the *float* kernel version, with multiple kernel interleavings and, consequently, a significant number of iterations executed in the faster kernel version *loopperf*.

Figure 5.15 – Euler3D accuracy loss profile using a problem size of 193536, 1000 iterations, and an accuracy loss threshold of 2.8%.

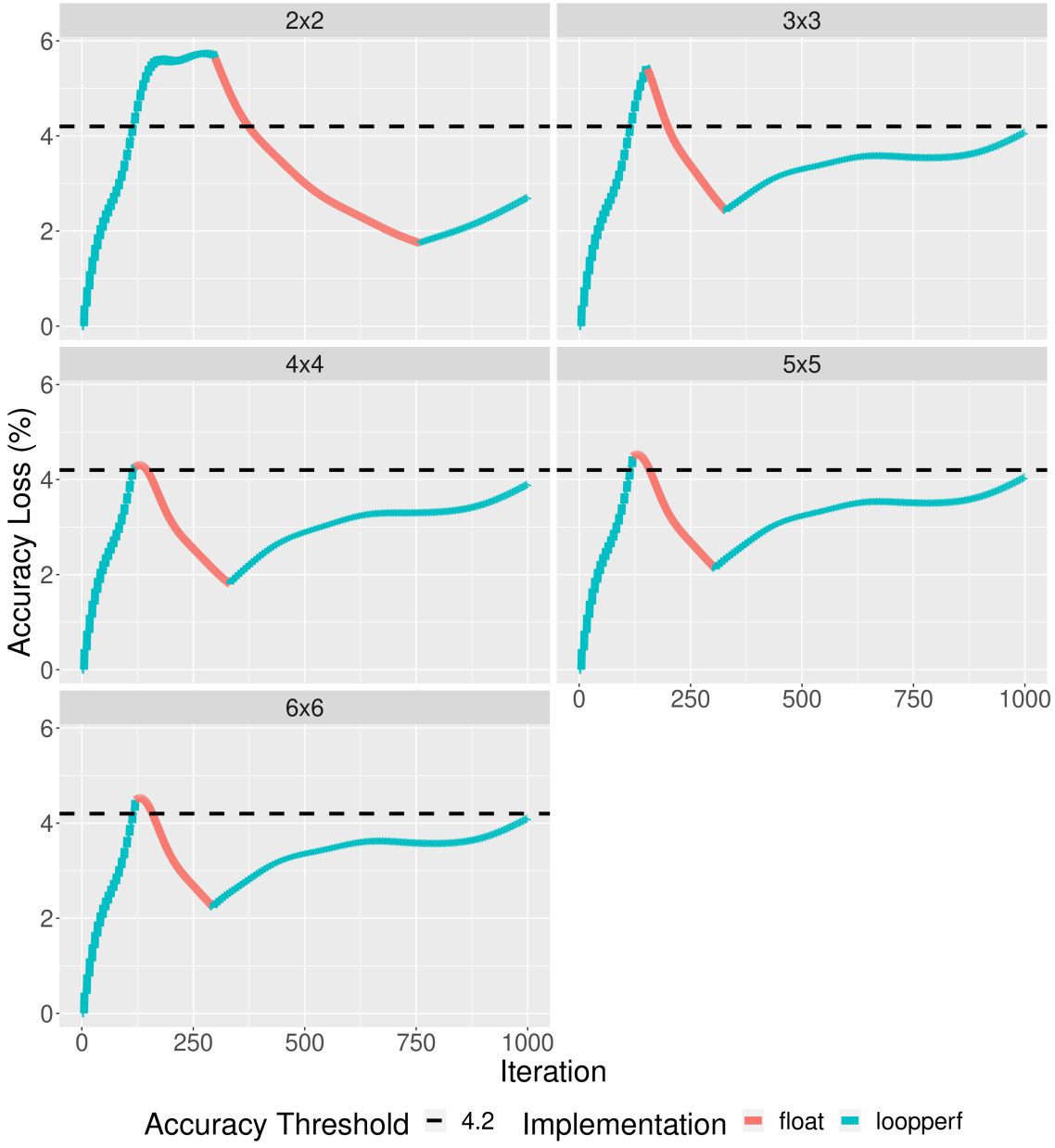


Source: The Authors

Figure 5.16 presents the loss of accuracy profile for a problem size of 193536 elements, 1000 iterations, and a loss of accuracy threshold of 4.2%. As in Figure 5.12 with 97152 elements, the configurations of 2x2 and 3x3 measurement subsections present a significant deviation from the determined loss of accuracy threshold, getting close to 2%. In the rest of the subsections, as with 97152 elements, only two kernel interleaves resulted

in running as many iterations as possible in the *loopperf* kernel version. Furthermore, these configurations did not have considerable deviations from the determined loss of accuracy threshold.

Figure 5.16 – Euler3D accuracy loss profile using a problem size of 193536, 1000 iterations, and an accuracy loss threshold of 4.2%.



Source: The Authors

With a size of 193536 elements and 2000 iterations for the execution of the application, the execution configurations of the five subsection configurations are similar to those with only 1000 iterations, as can be seen in Figures A.9, A.10, and A.11 for the thresholds of loss of accuracy of 1.4%, 2.8%, and 4.2%, respectively. While with 1.4%, there is a more significant number of interleaving between the *loopperf* and *float*

kernel versions and a more substantial deviation from the loss of accuracy threshold in the 2x2 and 3x3 subsection configurations, with a threshold of 2.8%, there are minimal deviations, with multiple interleaves and most of the iterations performed in the *loopperf* kernel version. With a threshold of 4.2%, there is a significant deviation in the 2x2 and 3x3 subsection configurations, with the rest of the configurations respecting the threshold but with the opportunity of executing a higher number of iterations using the faster kernel version *loopperf*.

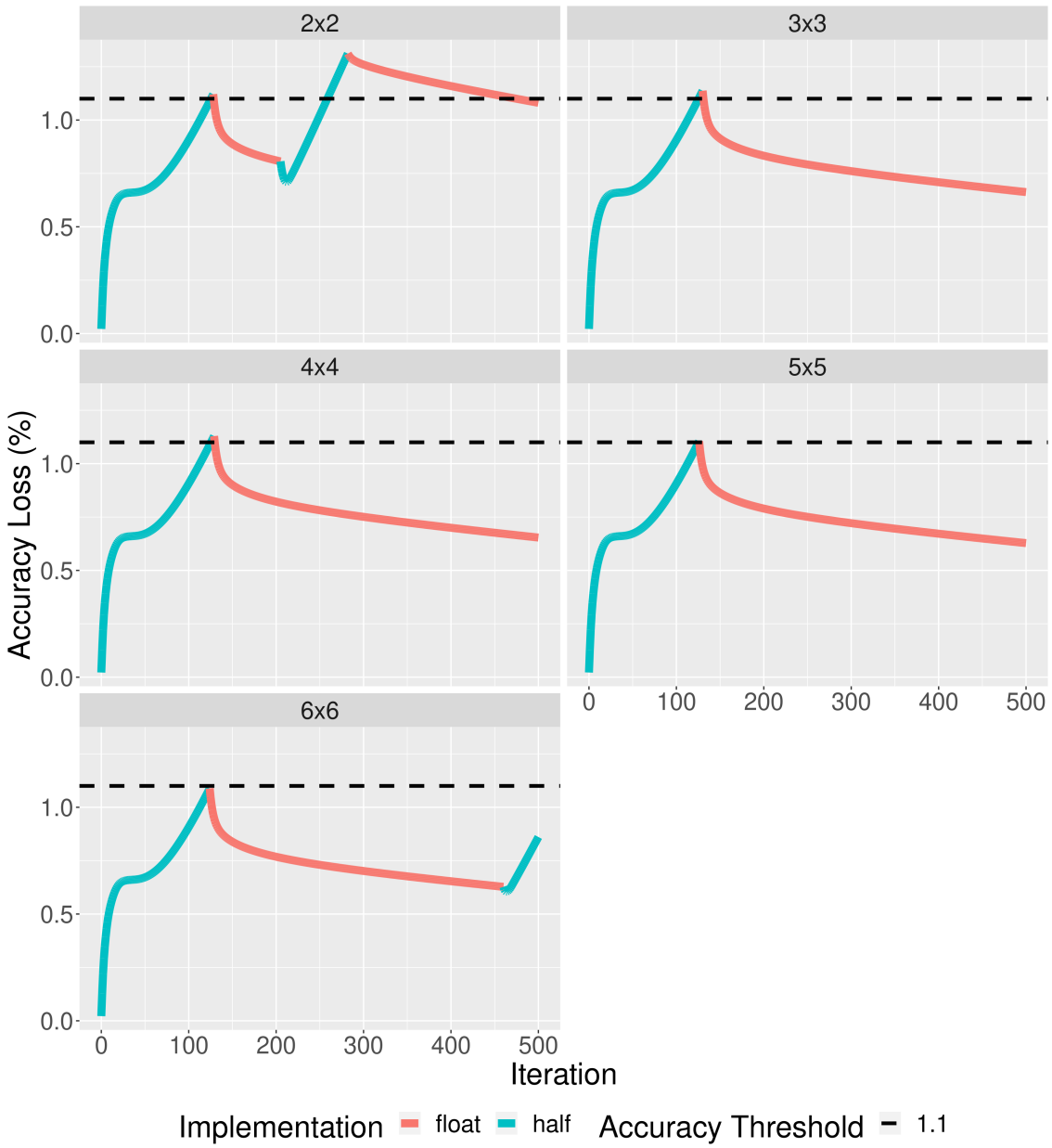
5.3.3 HotSpot3D

Figure 5.17 shows the loss of accuracy profile of the HotSpot3D application using a three-dimensional problem size of 512x512x8, 500 iterations to run the application, and a threshold of loss of accuracy of 1.1%. Unlike the execution configurations of previous applications, where there were usually multiple interleavings of the two kernel versions in most configurations of measurement subsections, in this application, there are only three interleaves in the 2x2 and 6x6 subsections. However, even though there are two instances where the execution configuration generated by the method for these two subsections uses the fastest half kernel version, the amount of additional iterations the kernel performs is reduced. Furthermore, in the 2x2 subsection configuration, there is a deviation from the determined accuracy loss threshold of approximately 0.2%, while for the 6x6 subsection, the generated execution configuration does not exceed the threshold.

With a loss of accuracy threshold of 2.2% and 3.3%, shown in Figures 5.18 and 5.19, respectively, the method could not generate an execution configuration with multiple interleaving between half and float kernel versions. Much of this inability is due to the limited retraction in the loss of accuracy provided by the float version after the first moment in which the half execution reached the specified threshold (approximately at iteration 250), and there was, then, the interleaving between the two kernels. While with a threshold of 3.3% (Figure 5.19), there is no room to perform more half iterations after a period of execution of the float version, with a threshold of 2.2% (Figure 5.18), although reduced, there is a space with potential for running a few more iterations with the half kernel version.

Now with 1000 iterations to run the application on the same problem size of 512x512x8 and an accuracy loss threshold of 1.8%, there is more room for the method to interleave the two kernel versions multiple times. However, from the execution configu-

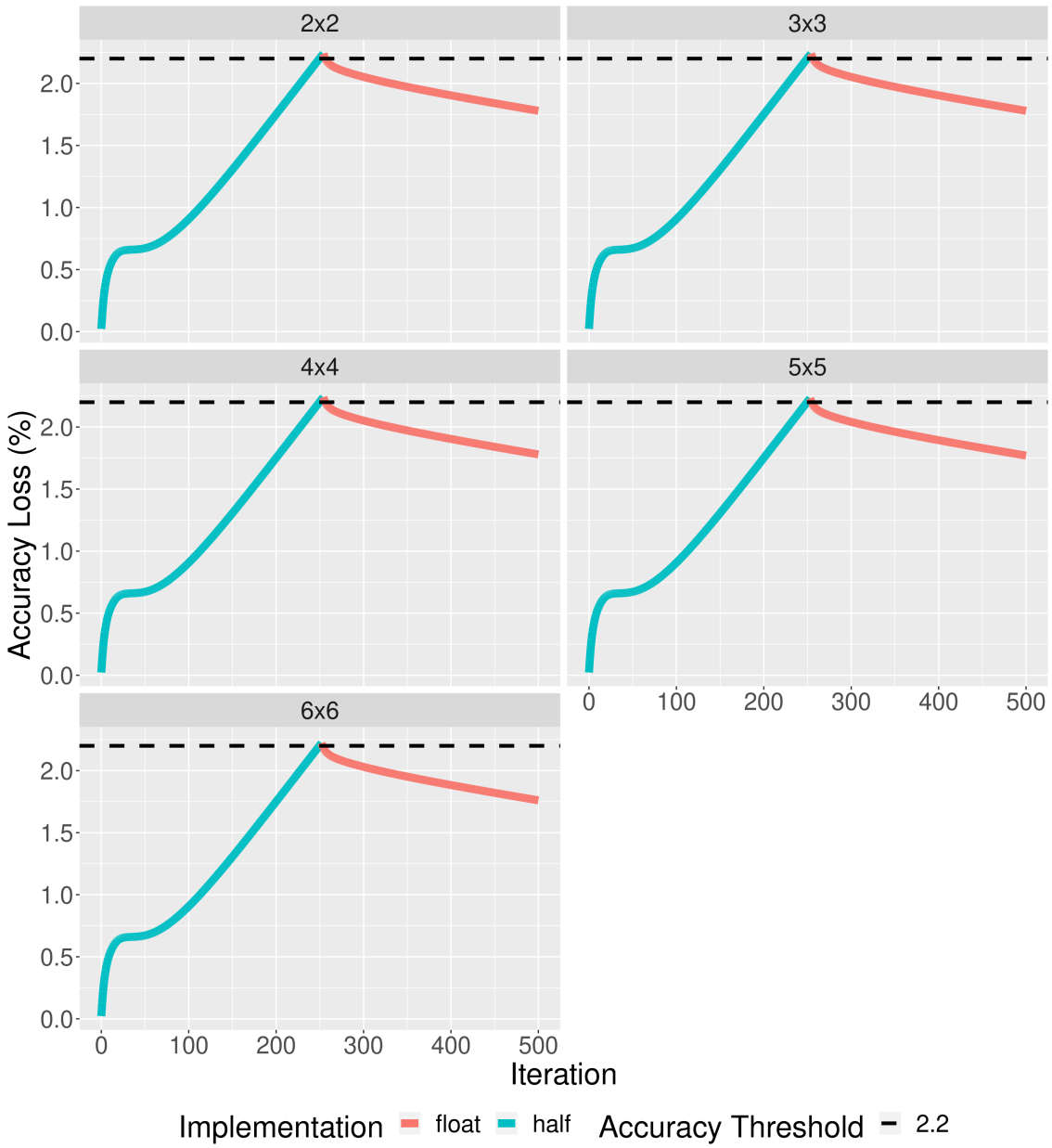
Figure 5.17 – HotSpot3D accuracy loss profile using a problem size of 512, 500 iterations, and an accuracy loss threshold of 1.1%.



Source: The Authors

rations generated by the method for the five configurations of measurement subsections, the method interleaves more than once the two kernels only in the 2x2 subsection, as seen in Figure 5.20. After interleaving the half kernel version with the float version at iteration 208, running on the float version up to iteration 798, and then interleaving the two versions again and running another 33 iterations on the half version, the method was able to create an execution configuration that maximizes the execution of iterations in the half version. Despite slightly exceeding the determined threshold, the method's estimate came very close to the actual value.

Figure 5.18 – HotSpot3D accuracy loss profile using a problem size of 512, 500 iterations, and an accuracy loss threshold of 2.2%.

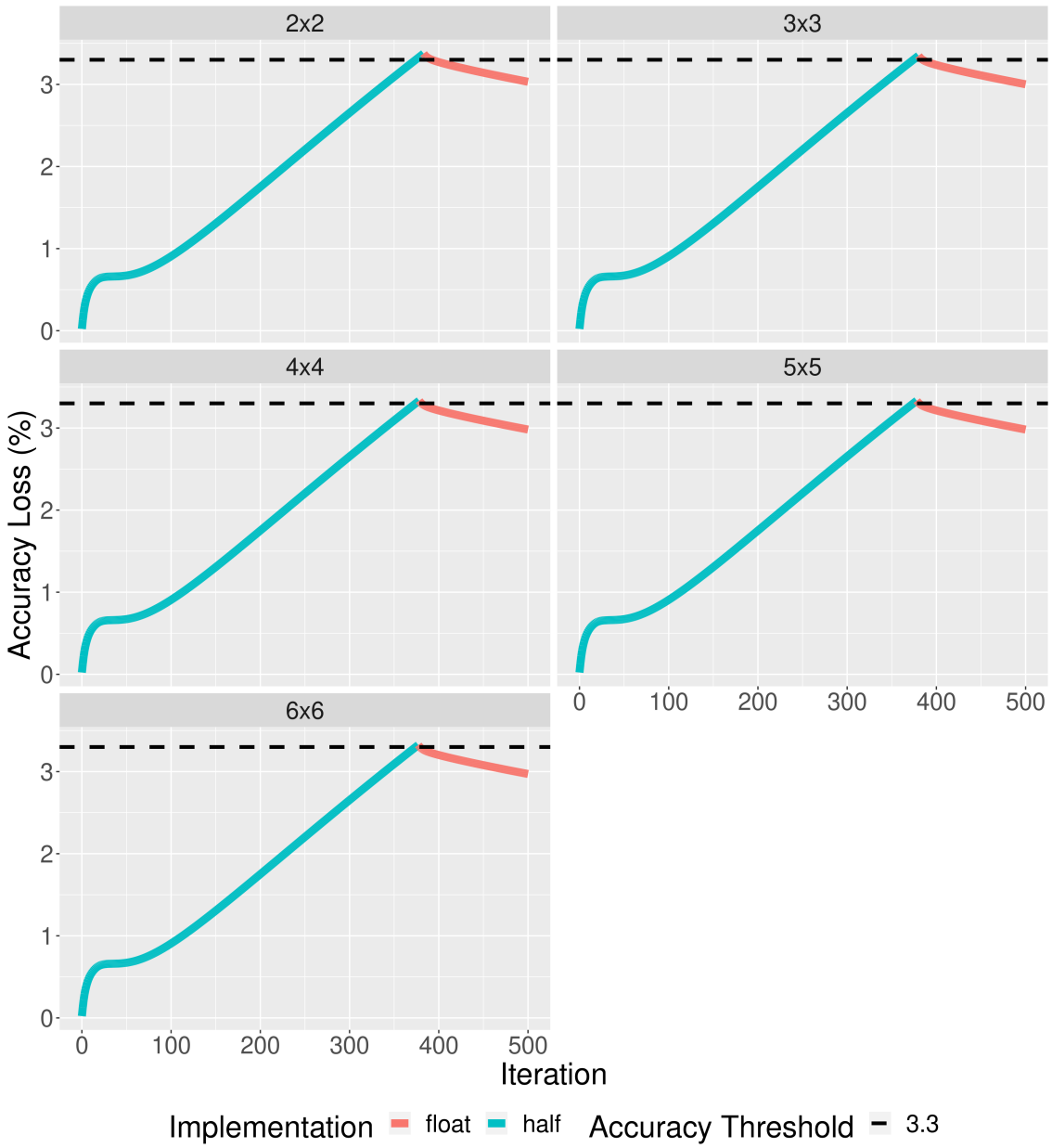


Source: The Authors

Although the method generates an execution configuration that maximizes the number of iterations executed in the fastest half version in the configuration of 2x2 measurement subsections, in the other subsections, there were no more intercalations after reaching the determined loss threshold. Despite identical space to the 2x2 subsection configuration, the method could not seek to add more interleaving between the two kernel versions in these configurations.

The same occurred in the execution configurations generated by the method with a 3.7% and 5.6% threshold, shown in Figures A.12 and A.13. Although the space pro-

Figure 5.19 – HotSpot3D accuracy loss profile using a problem size of 512, 500 iterations, and an accuracy loss threshold of 3.3%.

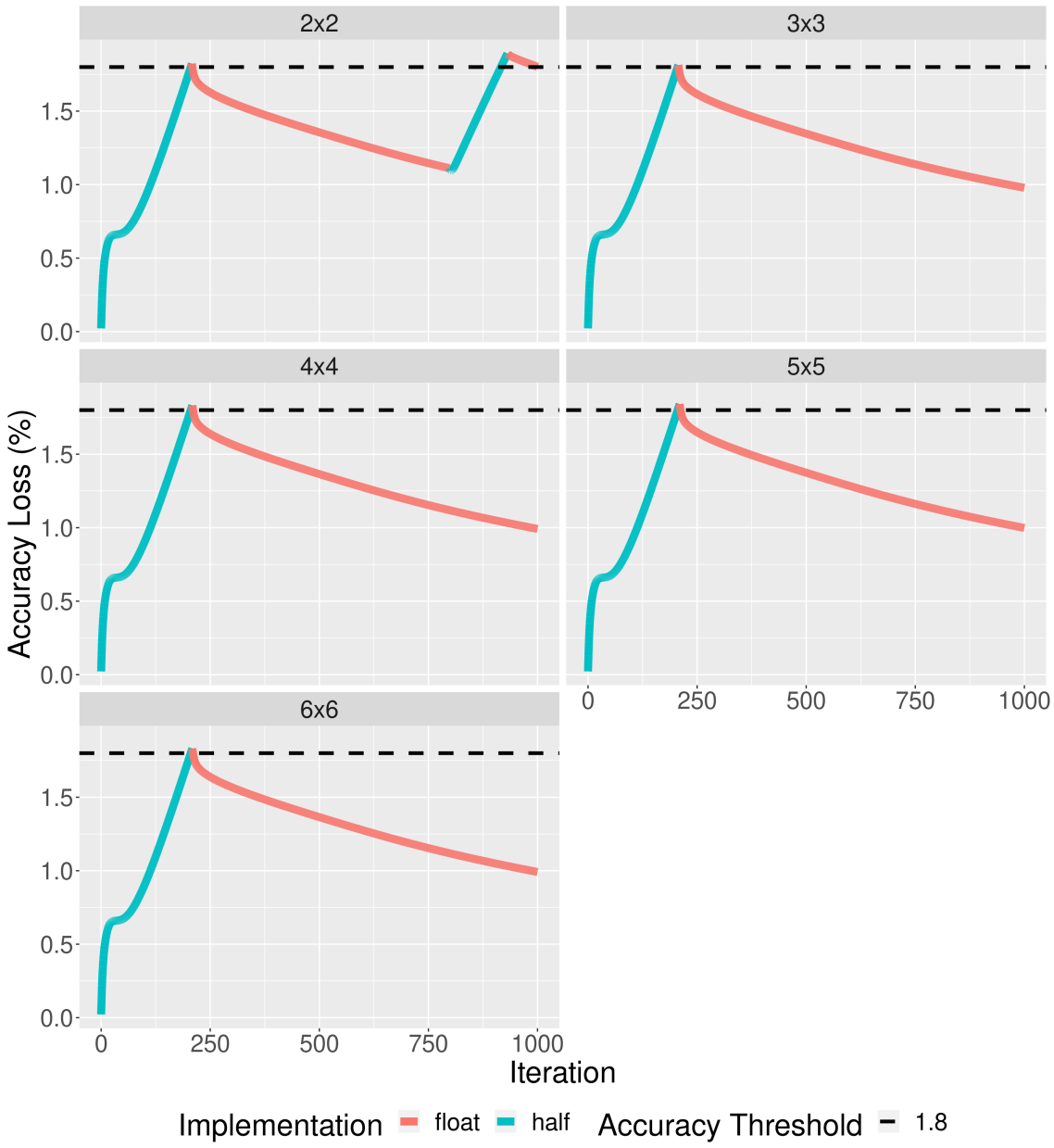


Source: The Authors

vided by running the float version is smaller, with a 3.7% threshold compared to the 1.8% threshold, there is potential to run a few more iterations with the half kernel version. With an accuracy loss threshold of 5.6%, the retraction space provided by executing the float version is even smaller, limiting the inclusion of an additional execution of the half version.

With a problem size of 1024x1024x8, the behavior of the loss of accuracy profile of the float kernel version makes further attempts to execute iterations in the half kernel version unfeasible. As soon as the execution reaches the thresholds of loss of accuracy of

Figure 5.20 – HotSpot3D accuracy loss profile using a problem size of 512, 1000 iterations, and an accuracy loss threshold of 1.8%.



Source: The Authors

5.3%, 10.6%, and 16% with 500 iterations (Figures A.14, A.15, and A.16, respectively) and 10%, 20%, and 30% with 1000 iterations (Figures A.17, A.18 and A.19, respectively), the generated execution configuration interleaves the half and float kernel versions. From then on, the rest of the execution shows minimal or no retraction of the loss of accuracy. Despite this, the execution configurations generated by the method present a reasonable estimate for when the loss of accuracy of the actual execution approaches or reaches the determined threshold.

6 PERFORMANCE ANALYSIS AND DISCUSSION

In this Chapter, we provide an analysis of the performance and energy consumption of the runtime configurations generated by our method for different thresholds of loss of accuracy in three iterative computing applications: LBM3D, Euler3D, and HotSpot3D.

6.1 LBM3D

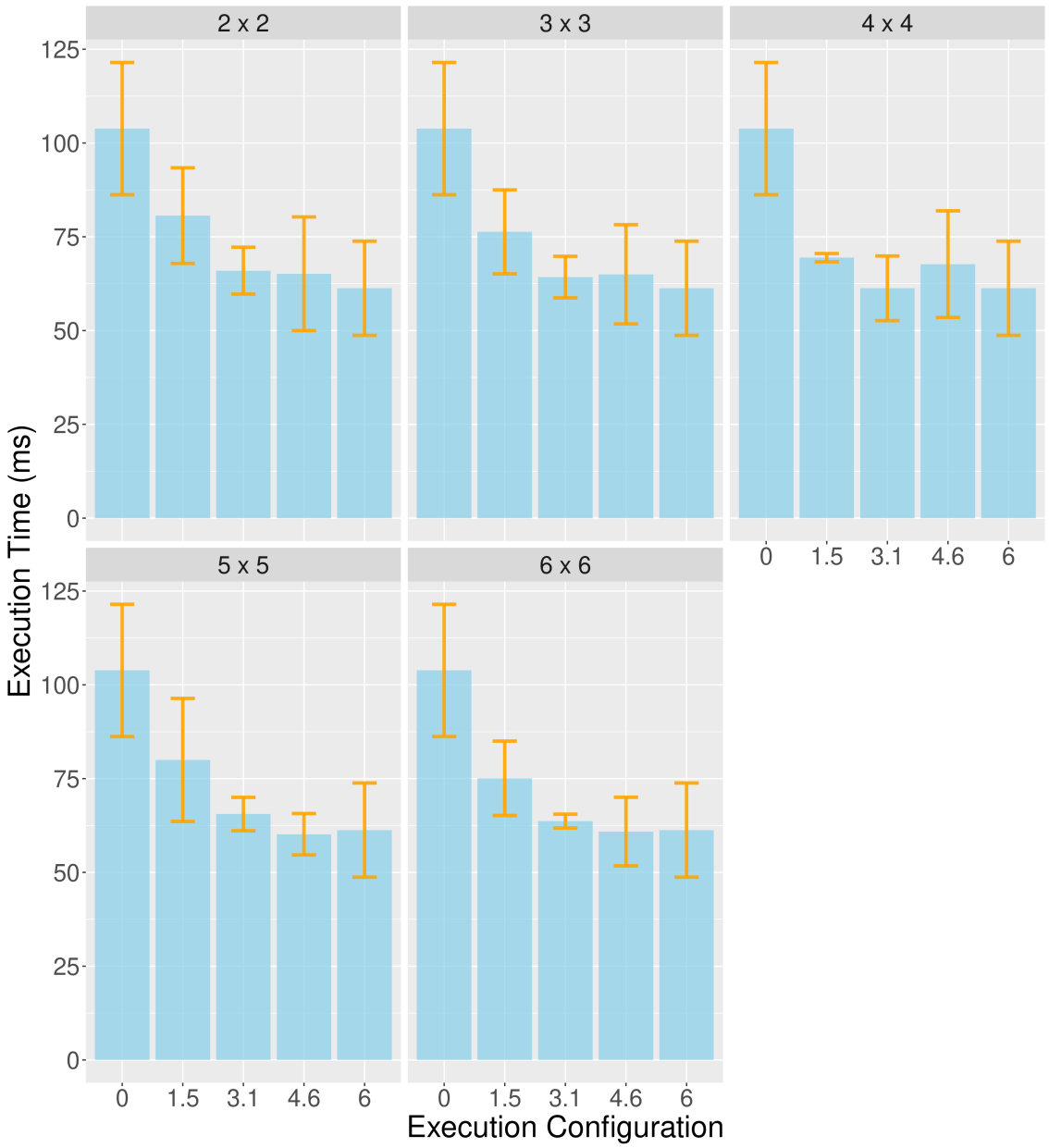
Figure 6.1 presents the runtime of the two kernel versions (half and float) and the execution configurations generated by the method for the three thresholds of loss of accuracy in each of the five configurations of measurement subsections (2x2, 3x3, 4x4, 5x5, and 6x6). On the graph's Y axis, we have the runtime in milliseconds (ms), and on the X axis, the execution configuration used (percentage of loss of accuracy). The orange bar in each graph bar corresponds to each experiment's standard deviation of 10 repetitions.

To present the individual performance results of the two kernels, we used index 0 for the execution of the float kernel since there is no loss of accuracy and index 6 for the results of the half kernel version. This value corresponds to the loss of accuracy of the kernel. Although we present the values for these two indices in the five configurations of subsections, the values are the same and were replicated only to facilitate analysis. The intermediate values correspond to the thresholds of loss of accuracy for which our method generated the respective execution configurations presented in Chapter 5.

Let's start our analysis by comparing the runtime of the two kernel versions in a three-dimensional problem of size 64x64x64 and 200 iterations, shown in Figure 7.1. As mentioned earlier, indices 0 and 6 correspond to float, and half kernel versions, respectively, and are replicated in the five subsection configurations as they correspond to individual kernel execution values. As can be seen, the difference in runtime between the float kernel version and the half version is significant. Comparing the average runtime of each version presented in Table A.1, while the float version has an average time of 103.84 (first line), the half version has an average time of only 61.28 milliseconds (second line of the table). That's a speedup of approximately 1.69.

Now, let's compare the runtimes of the execution configurations generated by the method for the three thresholds of loss of accuracy (1.5%, 3.1%, and 4.6%). We will see that they are closer to the runtime of the kernel half. While the 1.5% threshold has the

Figure 6.1 – LBM3D runtime using a problem size of 64 and 200 iterations for different accuracy loss thresholds.



Source: The Authors

highest runtime of the three thresholds in all five subsection configurations, the runtime of the 3.1% and 4.6% thresholds varies depending on the subsections, with the 3.1% threshold time in most cases being just above the 4.6% threshold runtime. Despite the close runtimes, especially of the 3.1% and 4.6% thresholds, the standard deviation in most subsections is relatively high.

Analyzing in more detail the performance of the execution configuration for the 1.5% loss of accuracy threshold, we will see that the lowest average runtime happens in the 4x4 measurement subsections configuration, 69.43 milliseconds. When analyzing the

loss of accuracy profiles of the execution configurations of this threshold in Figure 5.4, we will see that subsections 3x3, 4x4, 5x5, and 6x6 present practically the same profile. However, the configuration of subsection 4x4 is the one that obtained the lowest runtime. Since the problem size and the number of iterations are relatively low for the execution capability of the GPU, there is a significant standard deviation between the ten repetitions of the performed experiments. Thus, it is inconceivable to state that this was the execution configuration with the finest performance.

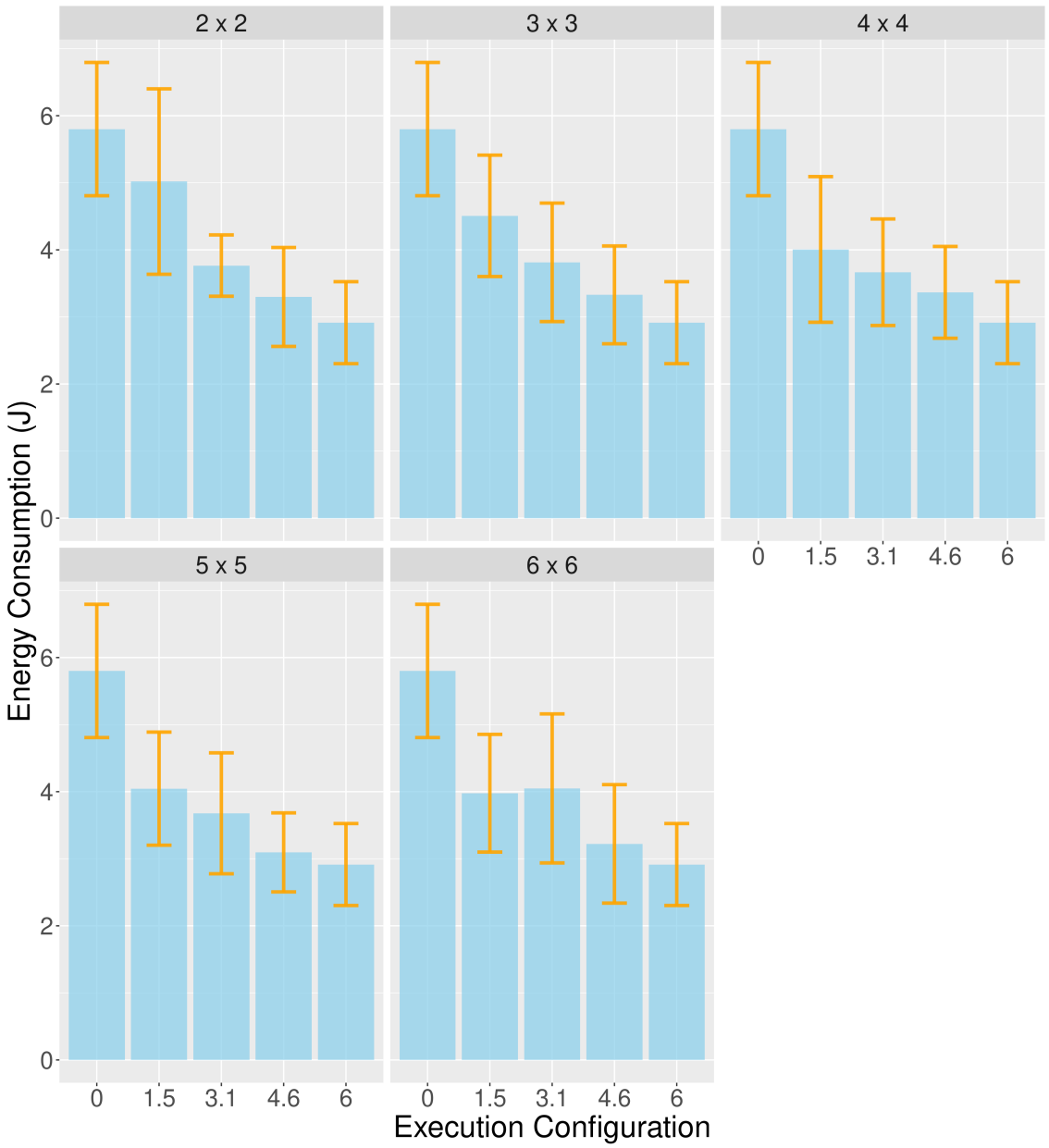
With a threshold of 3.1%, it is possible to observe that the execution configuration in the 4x4 subsection is the one that obtained the fastest runtime, 61.27 milliseconds. Despite coinciding with a higher number of iterations executed in the half kernel version, it is not clear that this is the reason for the performance gain in this execution configuration. The same applies to the 4.6% threshold, where the fastest runtime occurs in the 5x5 subsection, 60.18 milliseconds. However, when comparing the runtimes of the float kernel (index 0) and the half kernel (index 6), it is possible to observe that the runtime for the three thresholds reduces as the number of possible iterations to execute with the fastest half kernel version is more significant with larger thresholds.

Figure 6.2 shows the energy consumption in Joules for a problem of size 64x64x64 and 200 iterations. As we can see, the difference in consumption between the float kernel version (index 0) and the half kernel version (index 6) is substantial. While the float version consumes 5.8 J, the half version only consumes 2.91 J, a reduction of approximately 50%. Furthermore, Table A.1 shows that the average energy consumption significantly decreases between the 1.5%, 3.1%, and 4.6% thresholds, as higher thresholds allow the execution of a more significant number of iterations in the more efficient half kernel version. Despite the considerable standard deviation between the experiments, it is possible to observe that, except for the configuration of 6x6 subsections, the highest deviation value lowers as the accuracy loss threshold increases.

Figure 6.3 shows the runtime at a size of 64x64x64 with 400 iterations. The first observation is that the standard deviation is practically zero in both execution configurations and configurations of measurement subsections due to the higher number of iterations compared to the previous results presented in Figure 6.1 with 200 iterations. In this case, while the runtime of the float kernel version was 165.69 ms, that of the half version was only 98.81 ms, a speedup of 1.67.

Moreover, it is possible to observe, in Figure 7.3, a gradual reduction in the runtime as the loss threshold increases. When observing the runtimes of each threshold in

Figure 6.2 – LBM3D energy consumption using a problem size of 64 and 200 iterations for different accuracy loss thresholds.

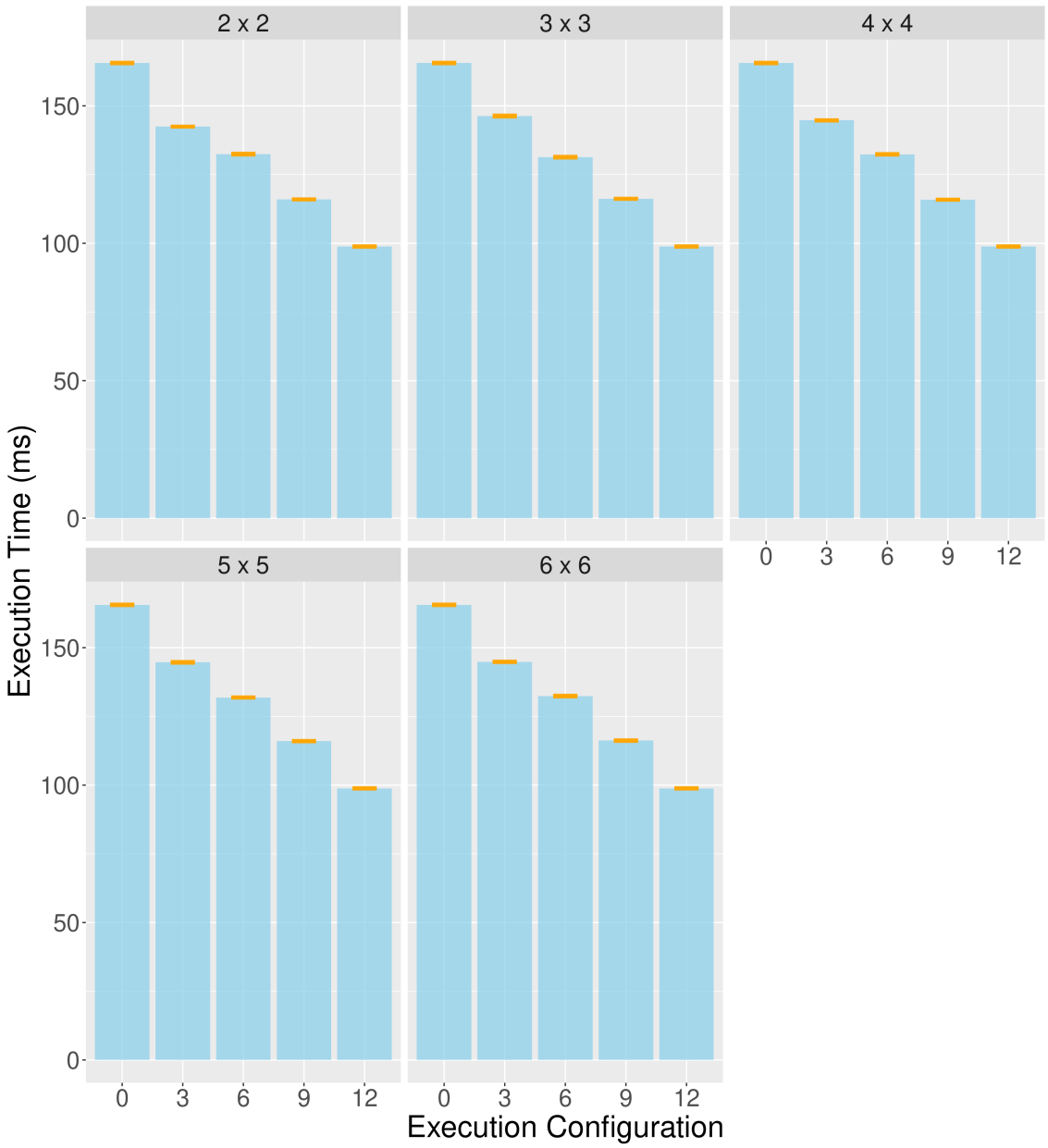


Source: The Authors

the five configurations of measurement subsections in Table A.2, it is possible to see that each threshold's times in different subsections are very close. With a threshold of 3%, the execution configuration of the 2x2 subsection had the lowest runtime, 142.44 ms. Compared to the runtime of the following fastest execution configuration, 144.67ms of the 5x5 subsection, it is a difference of only 1.54%.

However, the execution configuration generated for the two subsections is substantially different. While in subsection 2x2, there is the execution of 98 iterations in the float version and interleaving of the float kernel version by the half version later in

Figure 6.3 – LBM3D runtime using a problem size of 64 and 400 iterations for different accuracy loss thresholds.



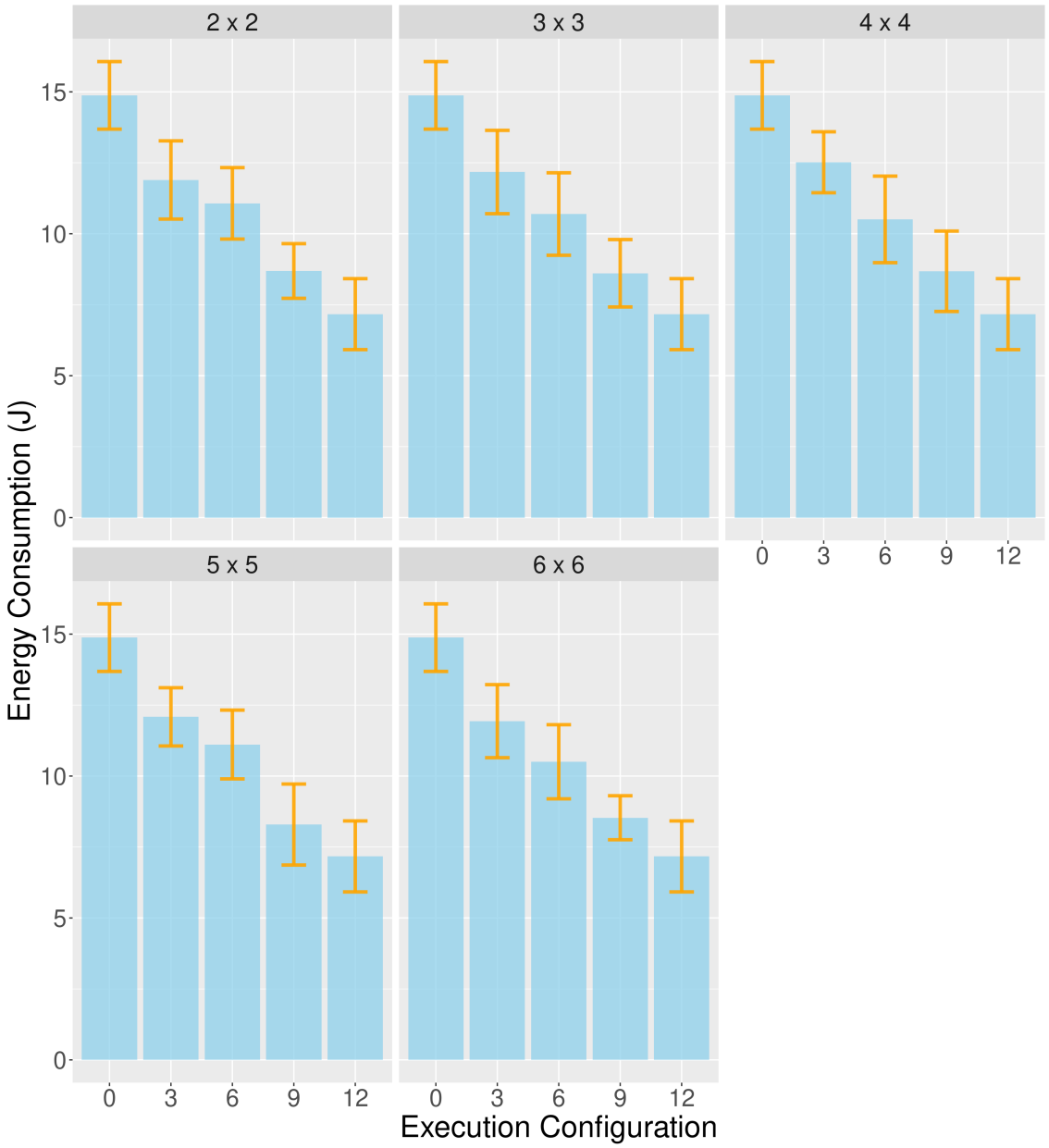
Source: The Authors

iteration 253 with an additional execution of 38 iterations in the half version for a total of 136 iterations, in subsection 5x5 there is the execution of just 119 iterations in the half version at the start of the execution. Even with an additional interleave of the two kernel versions, which involves converting the whole data domain from 32-bit to 16-bit floating-point, executing just 17 additional iterations in the generated execution configuration for the 2x2 subsection resulted in a slight performance gain.

Figure 6.4 presents the energy consumption for a problem size of 64x64x64 and 400 iterations. As with the runtime, it is possible to observe a gradual reduction in energy

consumption starting from the execution only in the float kernel version (index 0), passing through the execution configurations for the thresholds of loss of accuracy of 3%, 6% and 9% up to the energy consumption of running only the half kernel version (index 12). While the float version consumes 14.88 J, the half version only consumes 7.17 J, less than half of the float version, as seen in Table A.2.

Figure 6.4 – LBM3D energy consumption using a problem size of 64 and 400 iterations for different accuracy loss thresholds.



Source: The Authors

Now, we will compare the runtime and energy consumption of the execution configuration with the lowest runtime among the five subsection configurations for the lowest loss of accuracy threshold (3%) with the exact execution, with no loss of accuracy (float).

At the cost of only a 3% loss of accuracy, the execution configuration generated for the 2x2 subsection reduced the runtime by almost 14%, from 165.59 ms to 142.44 ms. In addition, the generated execution configuration reduced the energy consumption by approximately 20%, from 14.88 J to 11.9 J.

Figure 6.5 shows the runtime for a problem size of 128x128x128 and 200 iterations. While the float kernel version had an average runtime of 619.63 ms, the half version had a runtime of only 404.88 ms, a speedup of 1.53. Due to this significant performance gain presented by the half version, there was a gradual reduction in the runtime in the three thresholds of loss of accuracy.

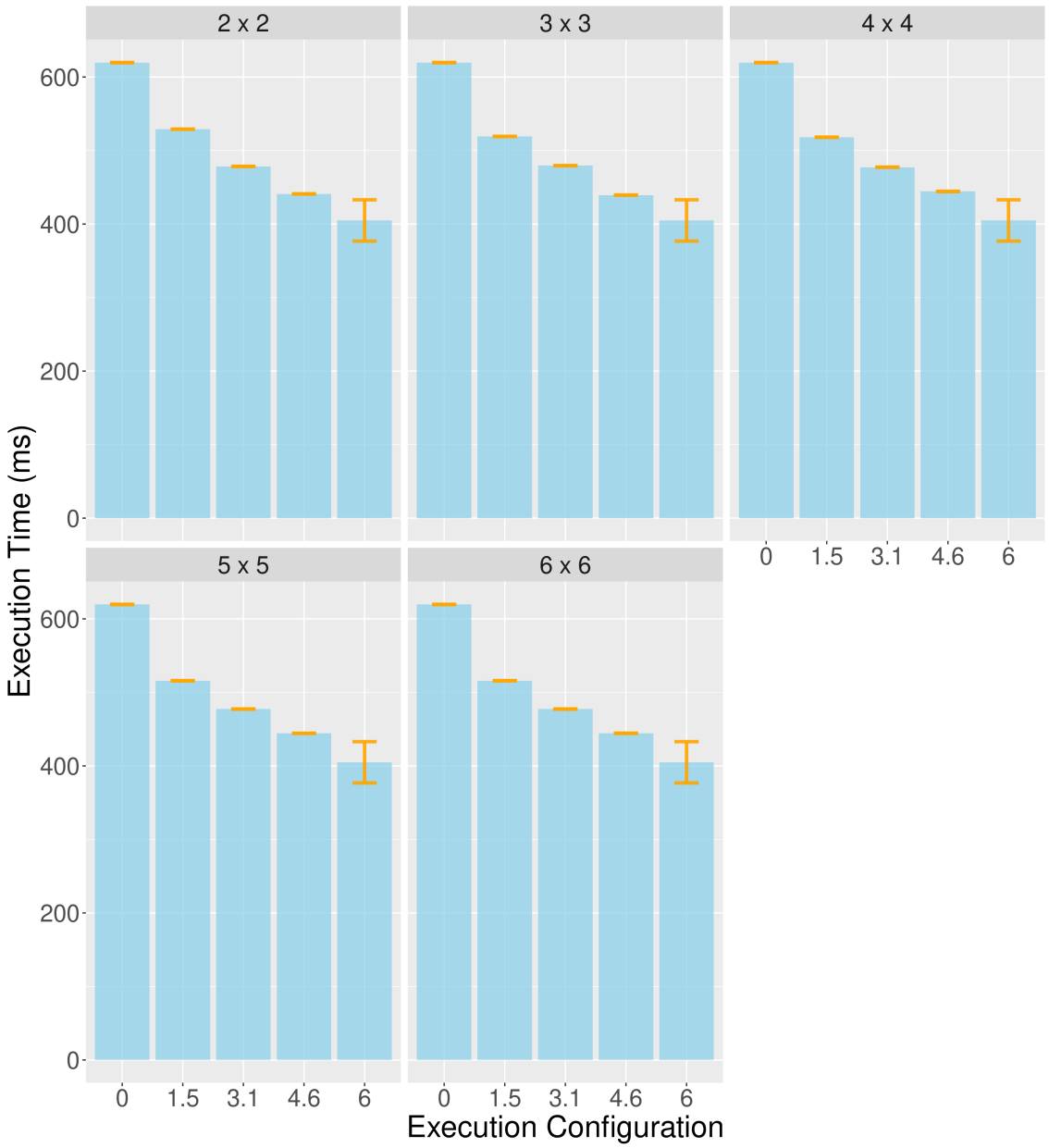
Moreover, it is possible to observe that, for the accuracy loss threshold of 1.5%, the execution configuration generated by the method for the 2x2 configuration of subsections has the highest runtime, as seen in Table A.3. When analyzing the loss of accuracy profiles for each of the five subsection configurations in Figure 5.8, it is possible to observe that, while in the other subsections, the execution configurations generated by the method are identical, in the 2x2 subsection there is low use of the space available for the execution of a more significant number of iterations in the faster half version. Thus, while in the other configurations, the runtimes are similar, in the configuration of subsections 2x2, the runtime is about 1.9% higher than in the other execution configurations.

Figure 6.6 presents the energy consumption for the problem size 128x128x128 and 200 iterations. In this case, the energy consumption of the float kernel version was 72.01 J, while the energy consumption of the half kernel version was only 41.58 J, a reduction of over 42%. In addition, for the 1.5% loss of accuracy threshold, through interleaved execution between the two kernels, it was possible to reduce the runtime from 619.63 ms to 515.82 ms in the 5x5 subsection and the energy consumption from 72.01 J to 57.35 J, a reduction of more than 16% in runtime and more than 20% in energy consumption.

Figure 6.7 shows the runtimes for a problem size of 128x128x128 and 400 iterations. In this case, the runtime of the float kernel version was 1235.83 ms, and the half kernel version was only 787.84 ms, a speedup of 1.56. Furthermore, as seen in Figure 6.8, energy consumption was 154.05 J in the float version and just 91.38 J in the half version, a more than 40% reduction, as seen in Table A.4.

With an accuracy loss threshold of 3.3%, we see that the two execution configurations generated by the method with the lowest runtime are the 2x2 and 5x5 measurement subsection configurations. If we analyze the loss of accuracy profiles for the problem size 128x128x128 and 400 iterations in Figure 5.9, we will see that the execution configu-

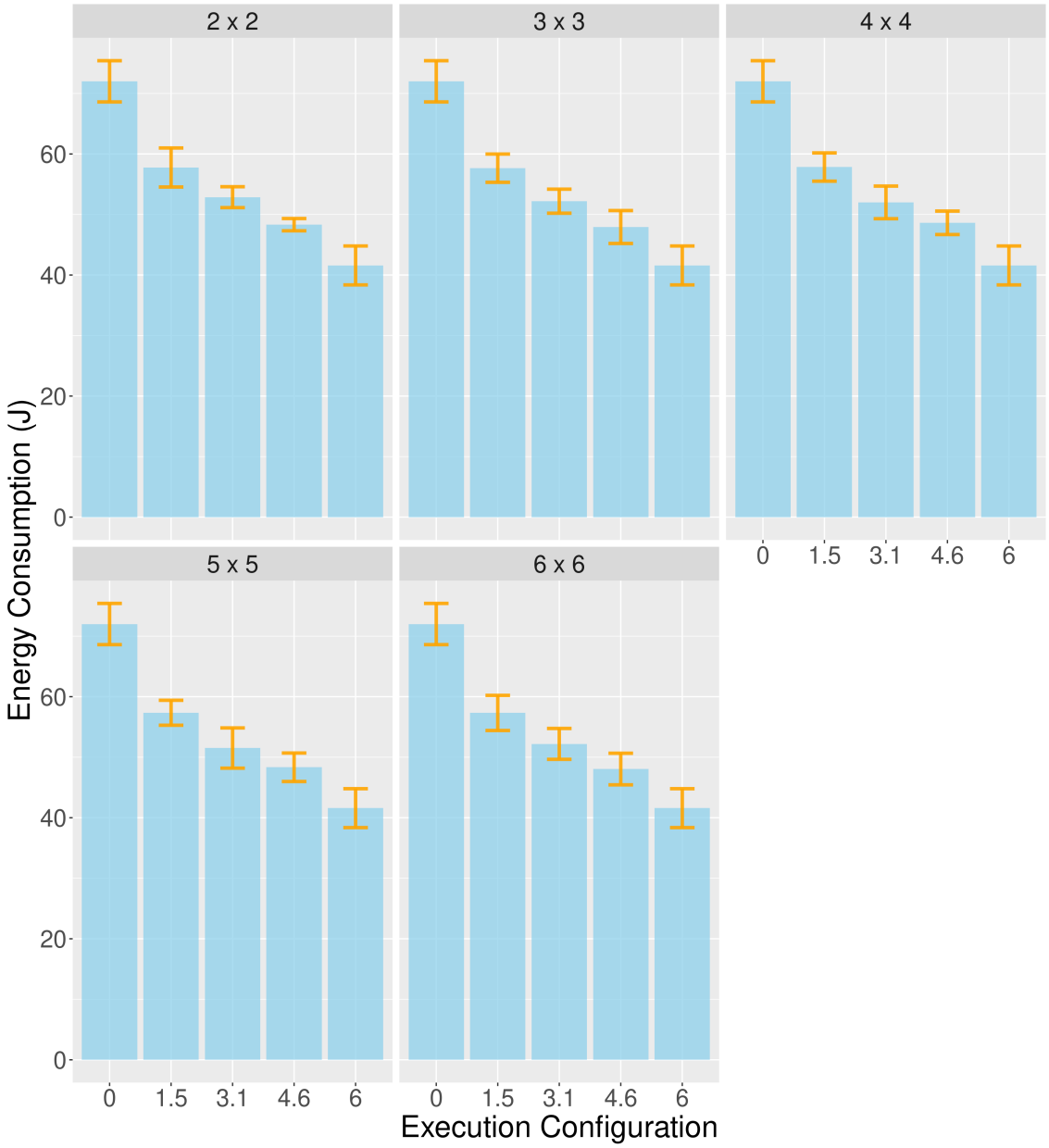
Figure 6.5 – LBM3D runtime using a problem size of 128 and 200 iterations for different accuracy loss thresholds.



Source: The Authors

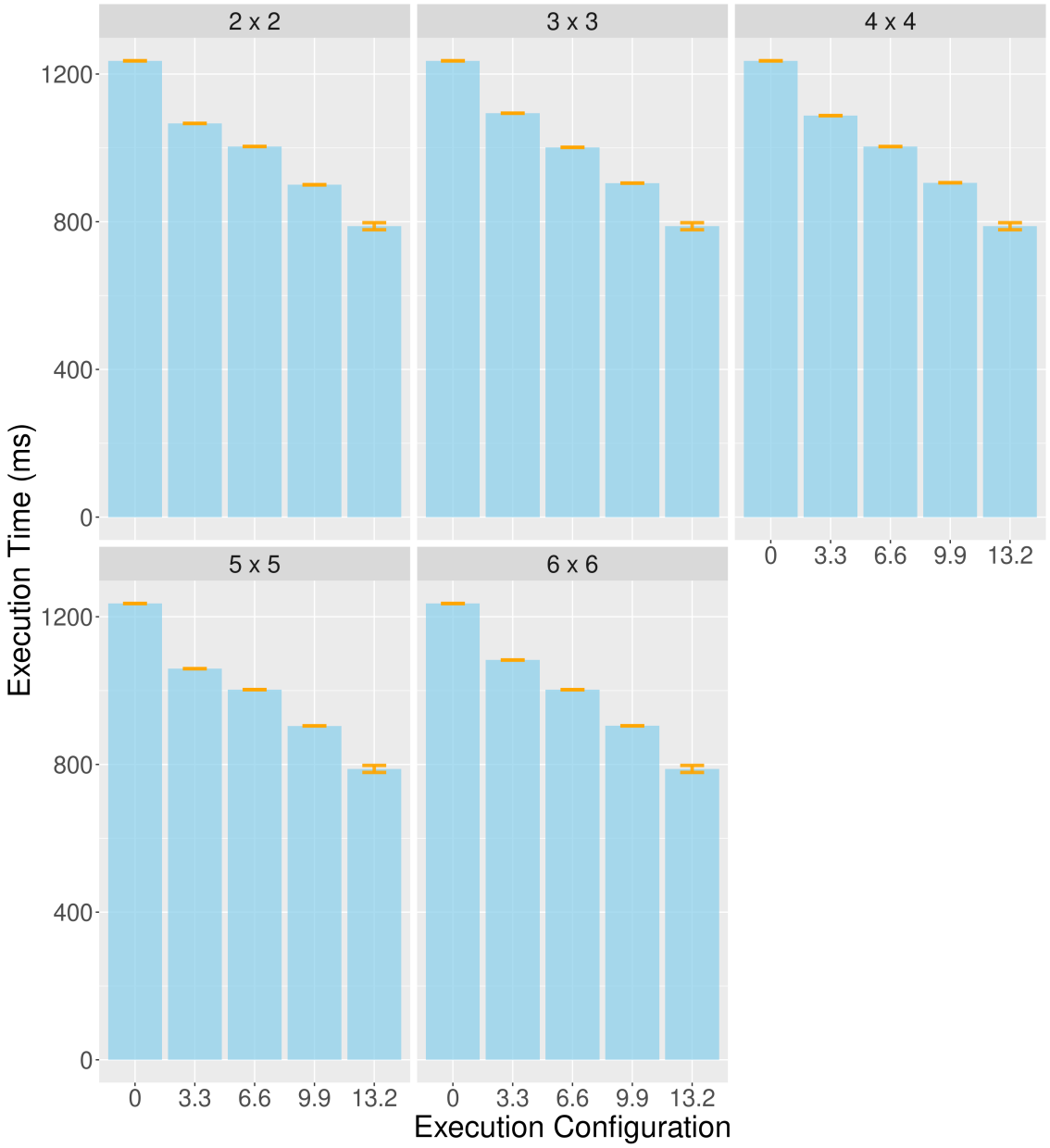
rations generated by the method for these two subsections are precisely those that have the highest number of iterations executed in the fastest version half. The runtime of the execution configuration with the lowest runtime (5x5 subsection) was 1059.31 ms with an energy consumption of just 127.94 J, a more than 14% reduction in the runtime, and a reduction of almost 17% in energy consumption.

Figure 6.6 – LBM3D energy consumption using a problem size of 128 and 200 iterations for different accuracy loss thresholds.



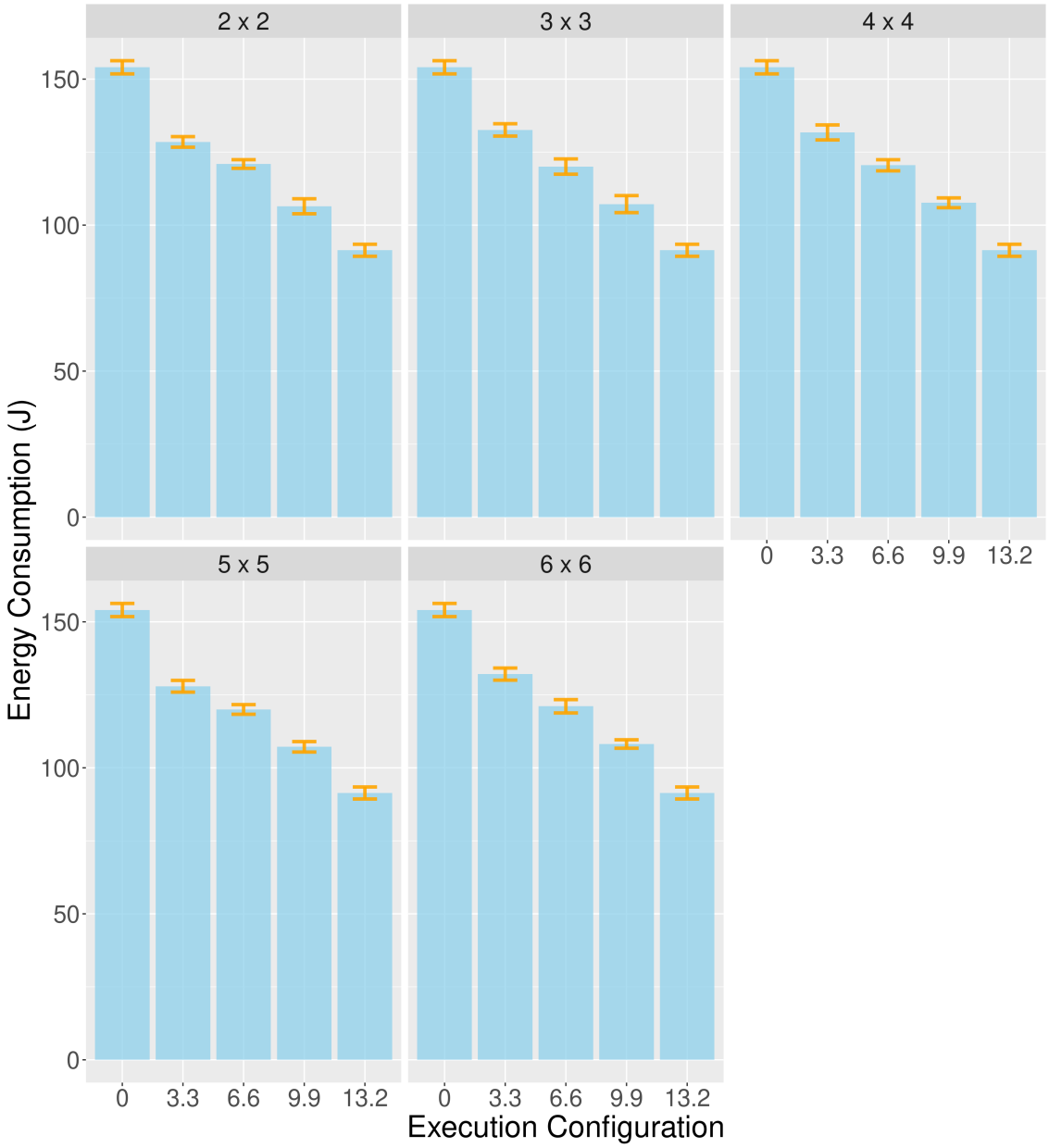
Source: The Authors

Figure 6.7 – LBM3D runtime using a problem size of 128 and 400 iterations for different accuracy loss thresholds.



Source: The Authors

Figure 6.8 – LBM3D energy consumption using a problem size of 128 and 400 iterations for different accuracy loss thresholds.



Source: The Authors

6.2 Euler3D

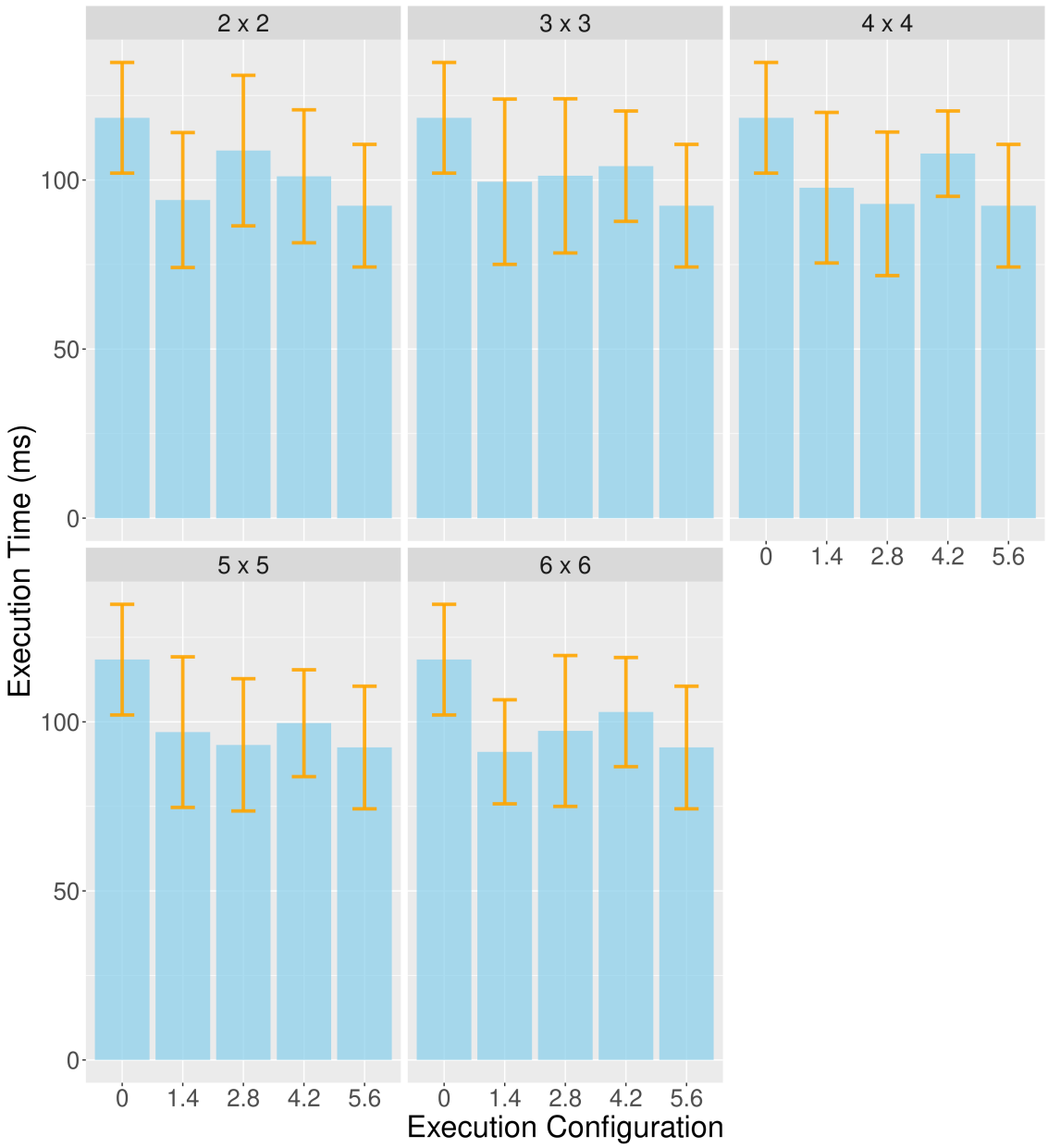
Figure 6.9 presents the runtime of the Euler3D application for a problem with 97152 elements and 1000 iterations for the execution of the application. Again, on the Y axis, we have the runtime in milliseconds (ms). On the X axis, the three thresholds of loss of accuracy (in percentage) used for the generation of execution configurations in our method in five configurations of measurement subsections, where index 0 corresponds to running only the float kernel and index 5.6 corresponds to running only the loopperf kernel version. Despite having a high standard deviation, it is possible to observe that the average runtime of the loopperf kernel version is lower than the runtime of the float kernel version, taking an average of 92.41 ms compared to 118.4 ms of the float version, a speedup of 1.28.

Now, when we analyze the runtimes of the execution configurations generated by the method for the three thresholds of loss of accuracy, we will see that in both cases, the runtime is lower than the runtime of the float version. Furthermore, it is possible to observe that the times vary considerably for the execution configurations generated in each of the five configurations of measurement subsections, despite the profile of the execution configurations being similar (except for the configuration generated for the 5x5 subsection and 1.4% threshold), as can be seen in Figures 5.10, 5.11, and 5.12.

Figure 6.10 presents the energy consumption for a problem size with 97152 elements and 1000 iterations. When comparing the total energy consumption of the float and loopperf versions, it is possible to observe that the loopperf version consumes 22% less energy than the float version, 4.21 J compared to 5.41 J, as can be seen in Table A.5. Furthermore, it is possible to observe that the execution of a set of iterations in the loopperf kernel version contributes to a reduction in energy consumption in the configurations generated by the method for the three thresholds of loss of accuracy, compared to the float version.

In both runtime and energy consumption we see a significant standard deviation. This variation in values from one repetition of the experiments to the following is due to the similarity in the execution configurations generated by the method for the three thresholds of loss of accuracy and the relatively smaller size and number of iterations. Analyzing the computational capacity utilization in the "GPU Util" column and the memory bandwidth utilization in the "Memory Util" column of Table A.5, we observe that this problem size does not exert pressure on the memory and results in a GPU computational

Figure 6.9 – Euler3D runtime using a problem size of 97152 and 1000 iterations for different accuracy loss thresholds.



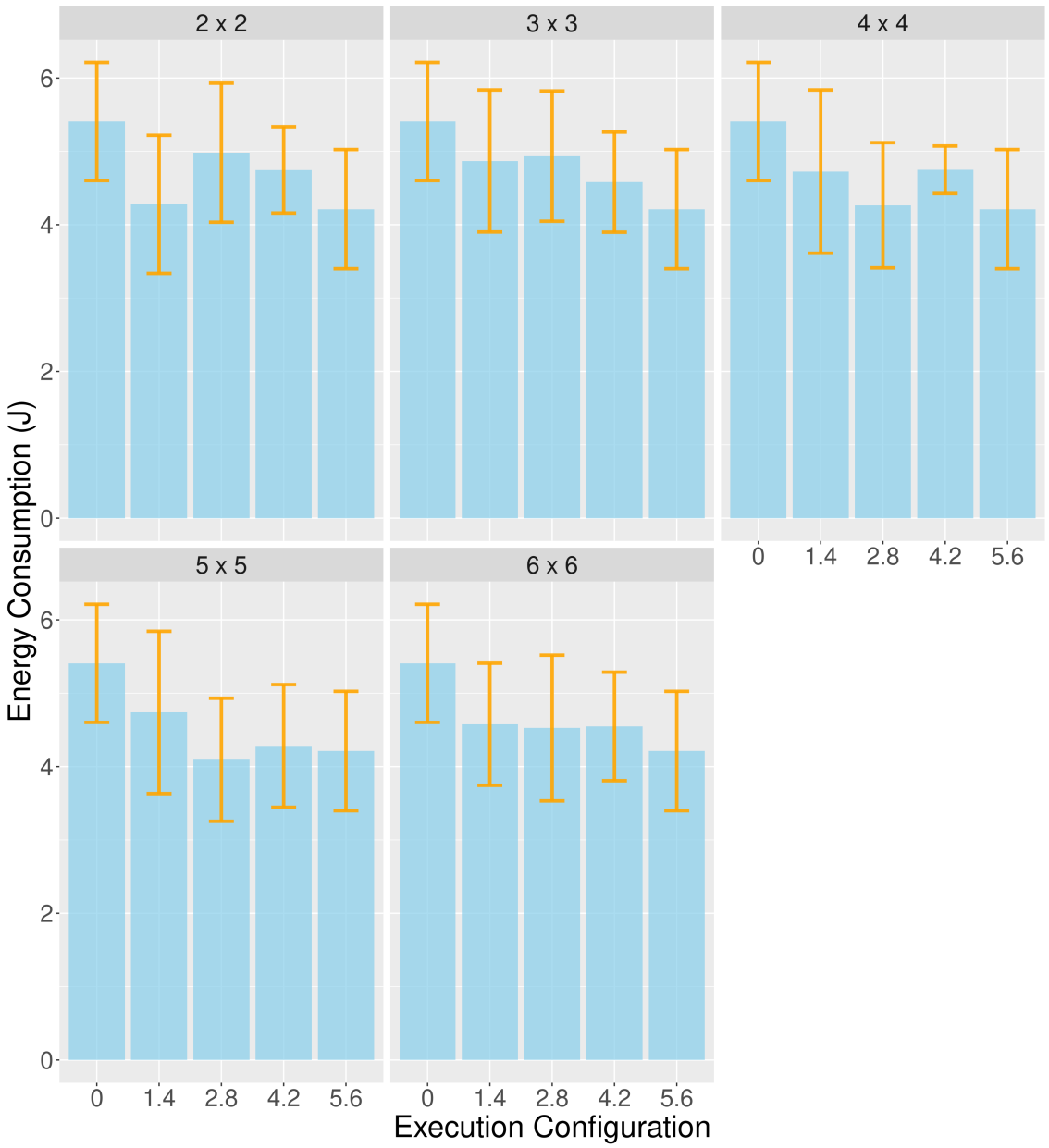
Source: The Authors

capacity utilization of less than 7%.

Figure 6.11 presents the runtime in the same problem size of 97152 but with 2000 iterations for the application execution. Now, with twice as many iterations in the execution of the application, we see that the difference between the runtime of the kernel version loopperf and float had a relative reduction. While execution in the float version finished after an average of 156.42 ms, the loopperf version finished after 140.82 ms, a speedup of 1.11.

Analyzing the runtime of the execution configurations generated by the method for

Figure 6.10 – Euler3D energy consumption using a problem size of 97152 and 1000 iterations for different accuracy loss thresholds.

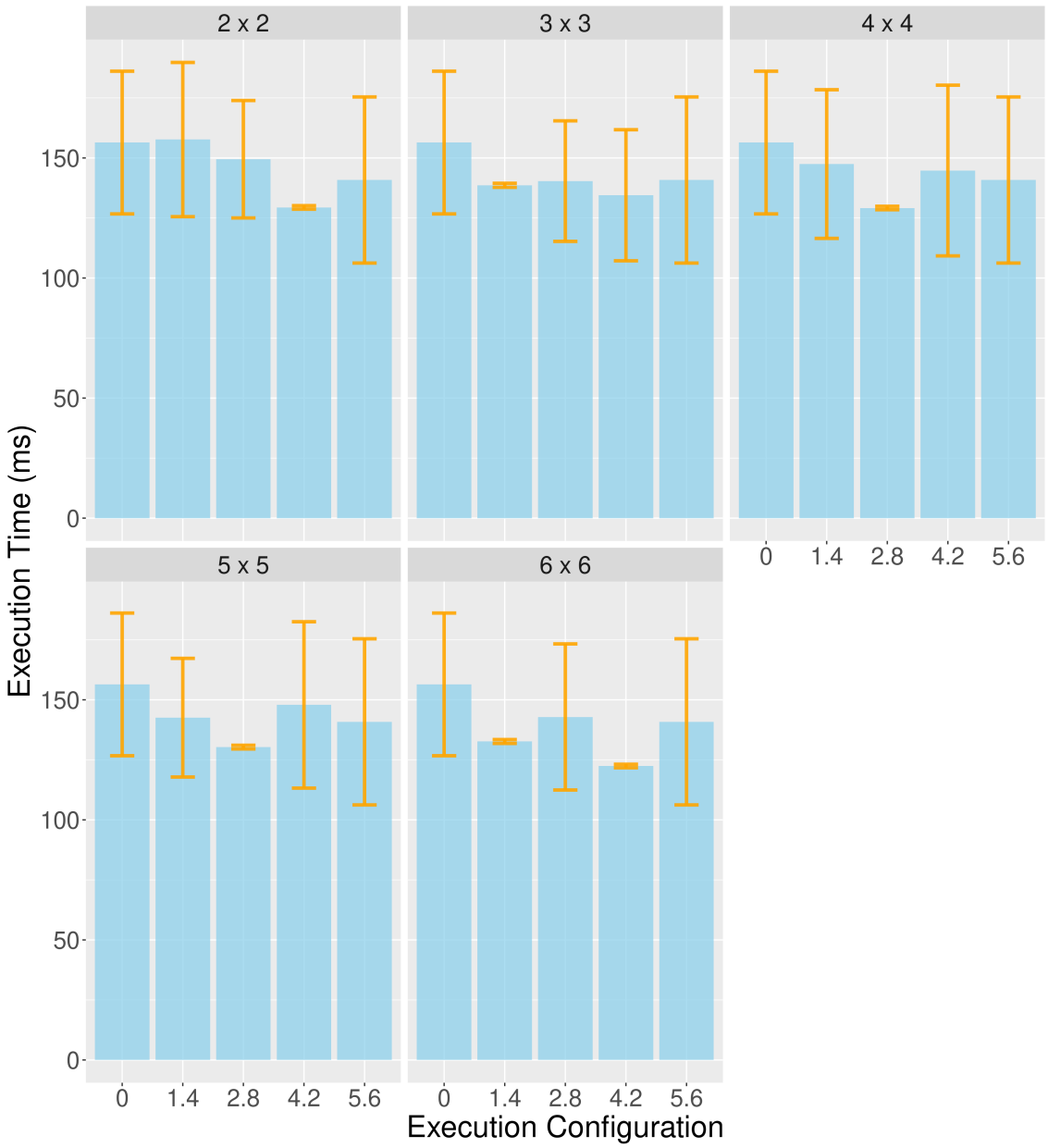


Source: The Authors

the 1.4% threshold in the five configurations of measurement subsections in Table A.6, we see that the configuration generated for the 6x6 subsection achieved the lowest runtime, 132.65 ms. The second configuration with the lowest runtime for the 1.4% configuration is the 3x3 subsection, with 138.61 ms.

When analyzing the loss of accuracy profiles for the 1.4% threshold in Figure 5.13, we see that while the execution configuration for the 6x6 subsection has the highest proportion of iterations executed in the loopperf version, in the 3x3 subsection, it is just the opposite, with one of the highest proportions of executing iterations in the float imple-

Figure 6.11 – Euler3D runtime using a problem size of 97152 and 2000 iterations for different accuracy loss thresholds.



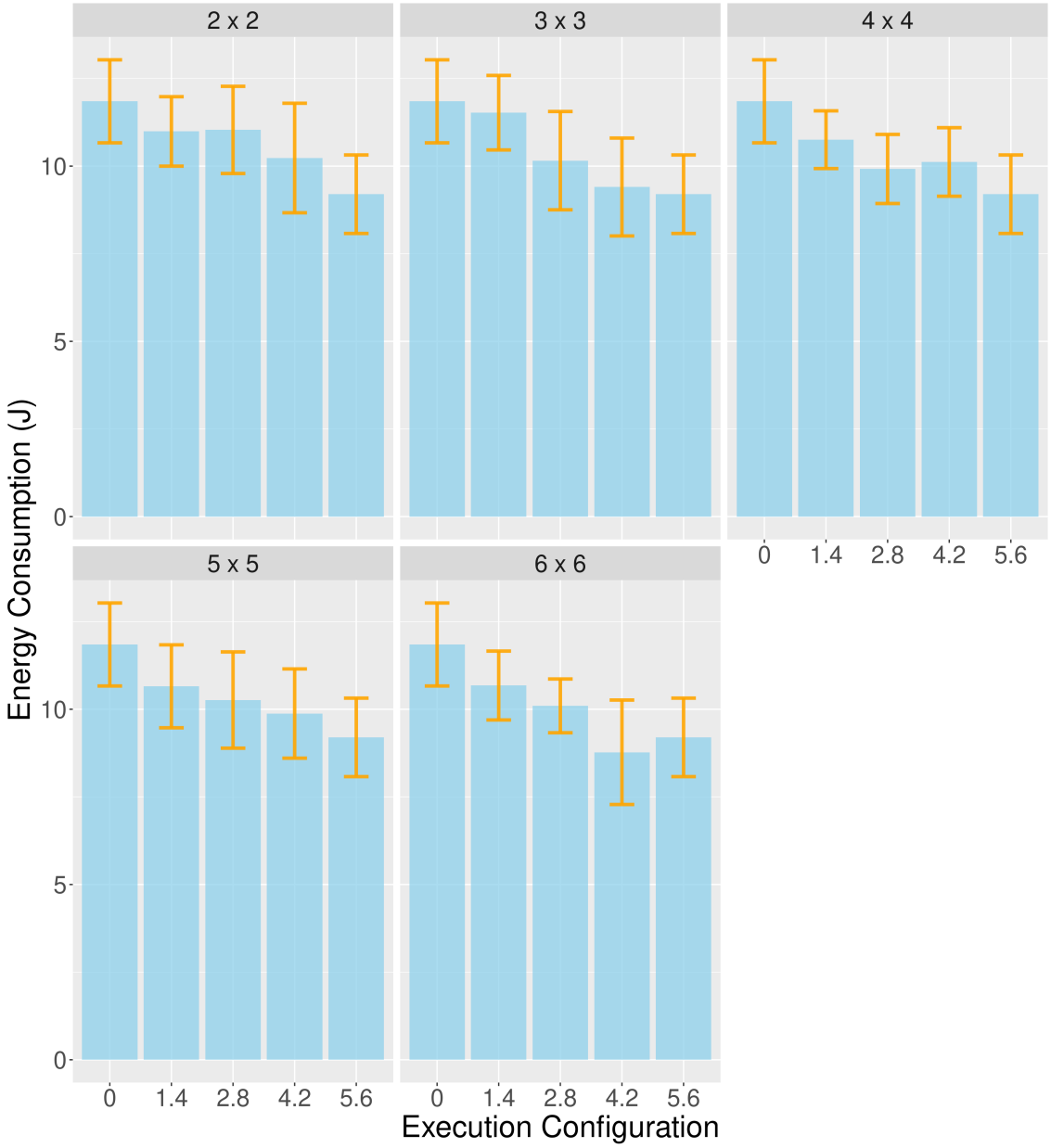
Source: The Authors

mentation. In both cases, the runtimes are lower, with practically zero standard deviation compared to running only the loopperf version. As such, we will have to analyze a more considerable problem size to understand better the impact of the execution configuration on improving the performance of executions.

Before that, let us look at the energy consumption in the problem size 97152 and 2000 iterations shown in Figure 6.12. Contrary to the results with only 1000 iterations presented in Figure 6.10, it is possible to observe a gradual reduction as a higher threshold of loss of accuracy allows the execution of more iterations in the loopperf kernel version,

which in turn presents one of the lowest energy consumptions. While the float version consumes 11.85 J, the loopperf version consumes 9.2 J, a reduction of over 22%.

Figure 6.12 – Euler3D energy consumption using a problem size of 97152 and 2000 iterations for different accuracy loss thresholds.



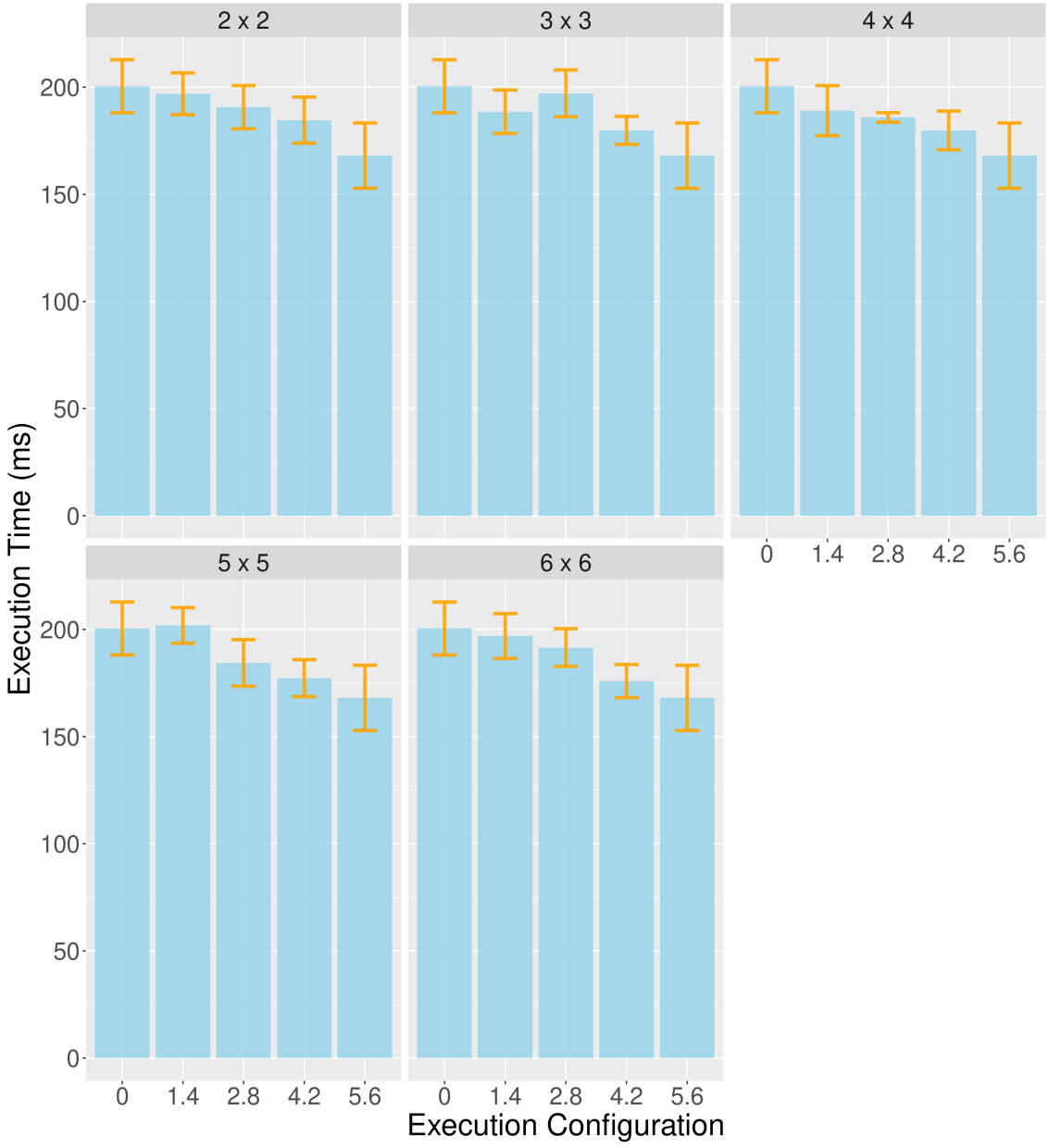
Source: The Authors

However, just as the execution configuration achieves the lowest runtime (only 122.43 ms) for the 4.2% loss threshold in the 6x6 subsection, it also achieves the lowest energy consumption, 8.77 J. When analyzing the profile of loss of accuracy of the execution configuration for the loss threshold of 4.2% in subsection 6x6 of Figure A.8, we see that it is the configuration with the fewest number of iterations executed in the float version. Due to the significant variations, it is inconceivable to state the fastest and most

efficient execution configurations generated.

Figure 6.13 presents the runtimes for a problem size of 193536 elements and 1000 iterations for the execution of the application. As can be seen, the standard deviation reduced considerably with more significant problem size. Moreover, it is possible to observe a gradual reduction of the runtime from the float version to the execution configurations with an increasing loss threshold until the loopperf kernel version. While executions of the float kernel version took an average of 200.46 ms, the loopperf version's average runtime was only 168.07 ms, a speedup of 1.19.

Figure 6.13 – Euler3D runtime using a problem size of 193536 and 1000 iterations for different accuracy loss thresholds.



Source: The Authors

Now, let us compare the runtime of the execution configurations generated by the method for the three loss of accuracy thresholds in the five configurations of measurement subsections. We will see a clear impact on the difference in the number of iterations executed in the loopperf and float versions. While the configurations generated for the 1.4% threshold are significantly similar with close runtimes, with a loss threshold of 2.8%, it is possible to observe that the execution configuration with the highest number of iterations executed in the loopperf version (subsection 5x5) obtained the lowest runtime. The one with the highest number of iterations executed in the float version (subsection 3x3) obtained the highest runtime, 184.37 ms against 197.1 ms, respectively.

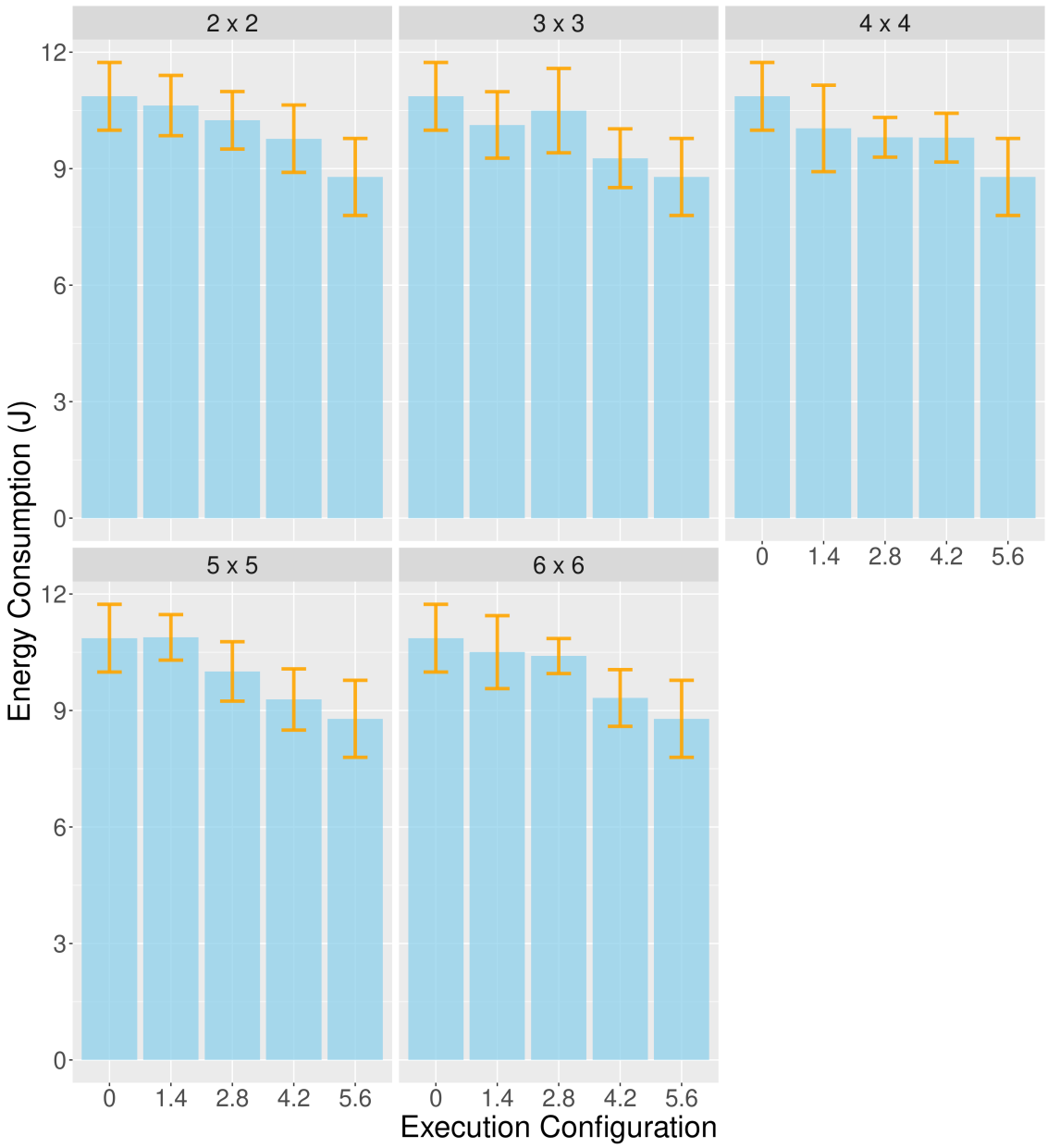
Furthermore, it is possible to observe the same in the 4.2% loss of accuracy threshold. While the execution configuration with the highest number of iterations executed in the loopperf version (subsection 6x6) obtained the lowest runtime, and the one with the highest number of iterations executed in the float version (subsection 2x2) obtained the highest runtime, 175.92 ms against 184.64 ms, respectively, as can be seen in Table A.7.

Figure 6.14 presents the energy consumption for a problem size of 193536 elements and 1000 iterations. Again, the impact of running a more significant number of iterations in the loopperf version on energy consumption is evident with the use of higher accuracy loss thresholds, allowing the execution of more iterations in the more efficient loopperf version. While the float version consumes 10.86 J, the loopperf version only consumes 8.79 J, a reduction of 19%, as seen in Table A.7. In addition, the consumption is also lower in the execution configurations for the 2.8% loss of accuracy threshold in the 5x5 and 3x3 subsections mentioned above, 10.01 J against 10.49, and for the 4.2% threshold in the 6x6 and 2x2 subsections, 9.32 J against 9.77 J.

The results are similar with a problem size of 193536 elements and 2000 iterations, as shown in Figure 6.15. While the float kernel version has an average runtime of 327.2 ms, the loopperf kernel version has a runtime of just 271.24 ms, a speedup of 1.20, as seen in Table A.8. Furthermore, the energy consumption of the float version was 22.66 J. In contrast, the loopperf version was only 18.35 J, a reduction of 19%, as shown in Figure 6.16.

Furthermore, for the 1.4% loss of accuracy threshold, the lowest runtime was achieved by the execution configuration generated for the configuration of 2x2 measurement subsections and the highest runtime in the 4x4 subsection. If we analyze the loss of accuracy profile of the execution configurations generated in these subsections in Figure A.9, it is possible to observe that, apart from the fact that the execution configuration

Figure 6.14 – Euler3D energy consumption using a problem size of 193536 and 1000 iterations for different accuracy loss thresholds.

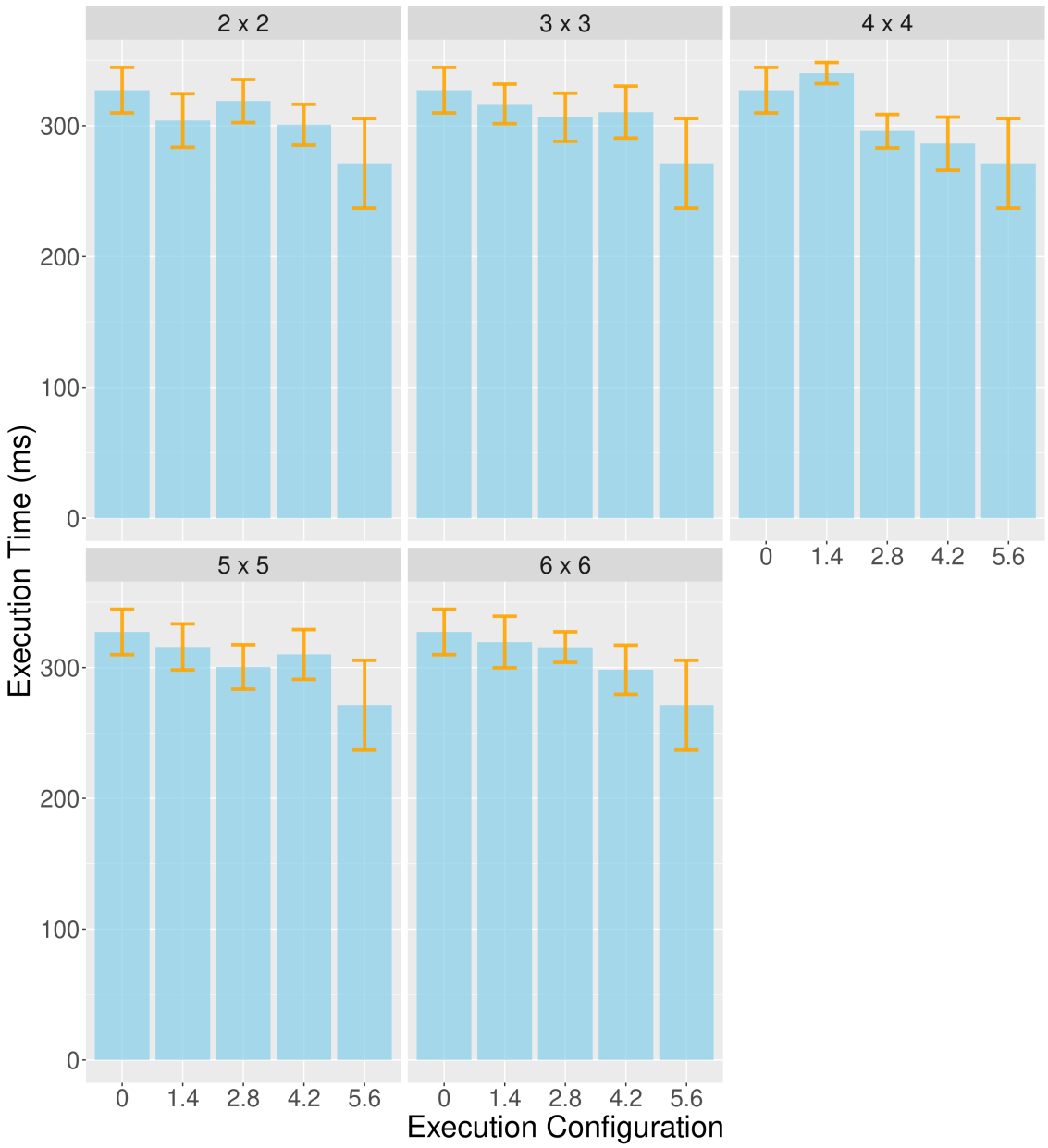


Source: The Authors

exceeded the loss of accuracy threshold significantly, the number of iterations executed in the fastest and most efficient kernel version loopperf is the largest in the 2x2 subsection, while the number of iterations performed in the float version is the largest among the five subsections in the 4x4 subsection.

While in the 2x2 subsection, the average runtime was 303.99 ms with consumption of 20.23 J, in the 4x4 subsection, it was 340.26 ms and 21.36 J. Likewise, at the threshold of loss of accuracy of 2.8%, the execution configurations with the highest number of executions in the loopperf version were those with the lowest runtime, 4x4 and 5x5, as

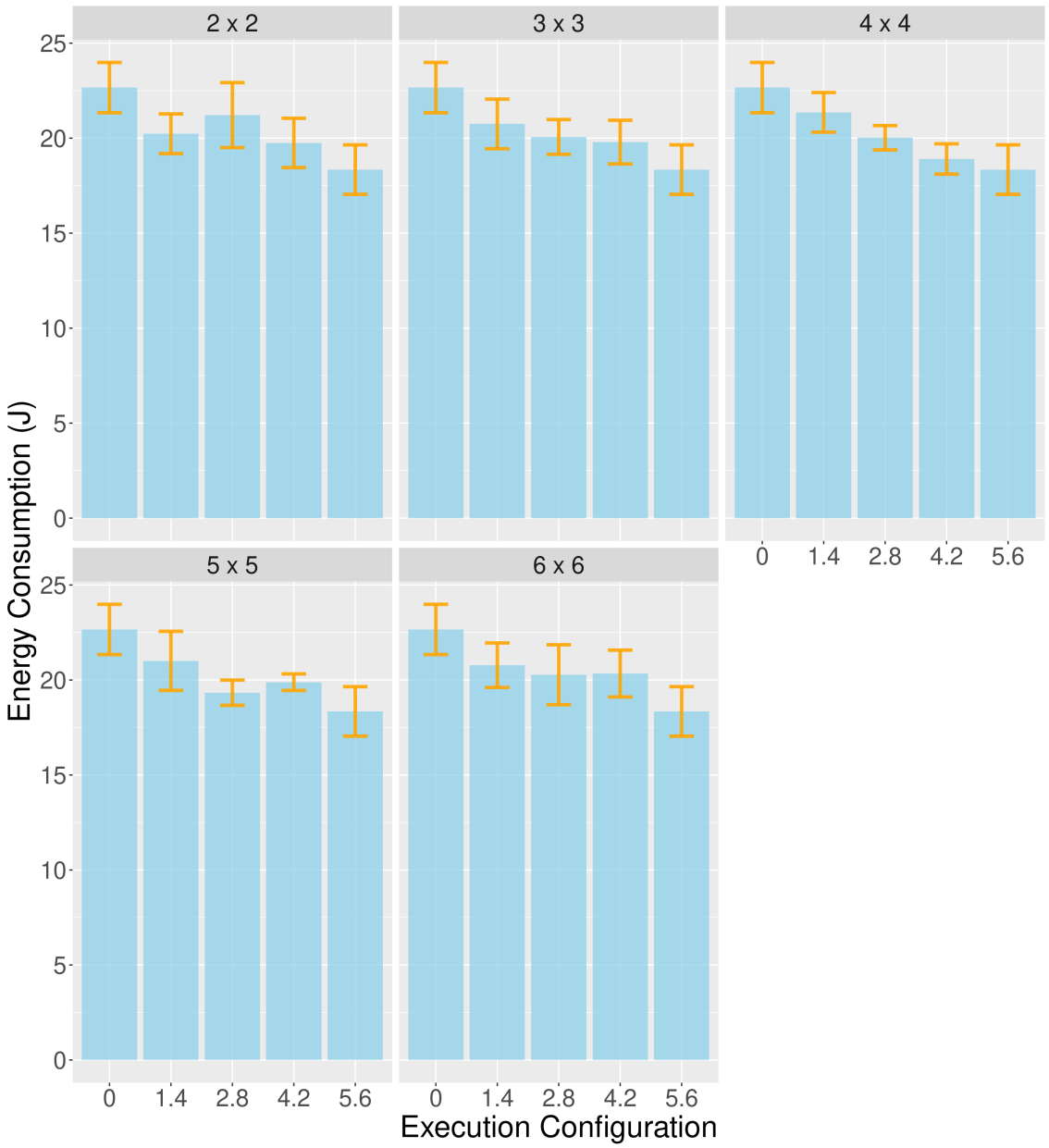
Figure 6.15 – Euler3D runtime using a problem size of 193536 and 2000 iterations for different accuracy loss thresholds.



Source: The Authors

seen in Figure A.10. In the threshold of loss of accuracy of 4.2%, the subsection with the highest number of iterations executed in the loopperf version, 4x4 as seen in Figure A.11, achieved the lowest runtime, as seen in Table A.8.

Figure 6.16 – Euler3D energy consumption using a problem size of 193536 and 2000 iterations for different accuracy loss thresholds.



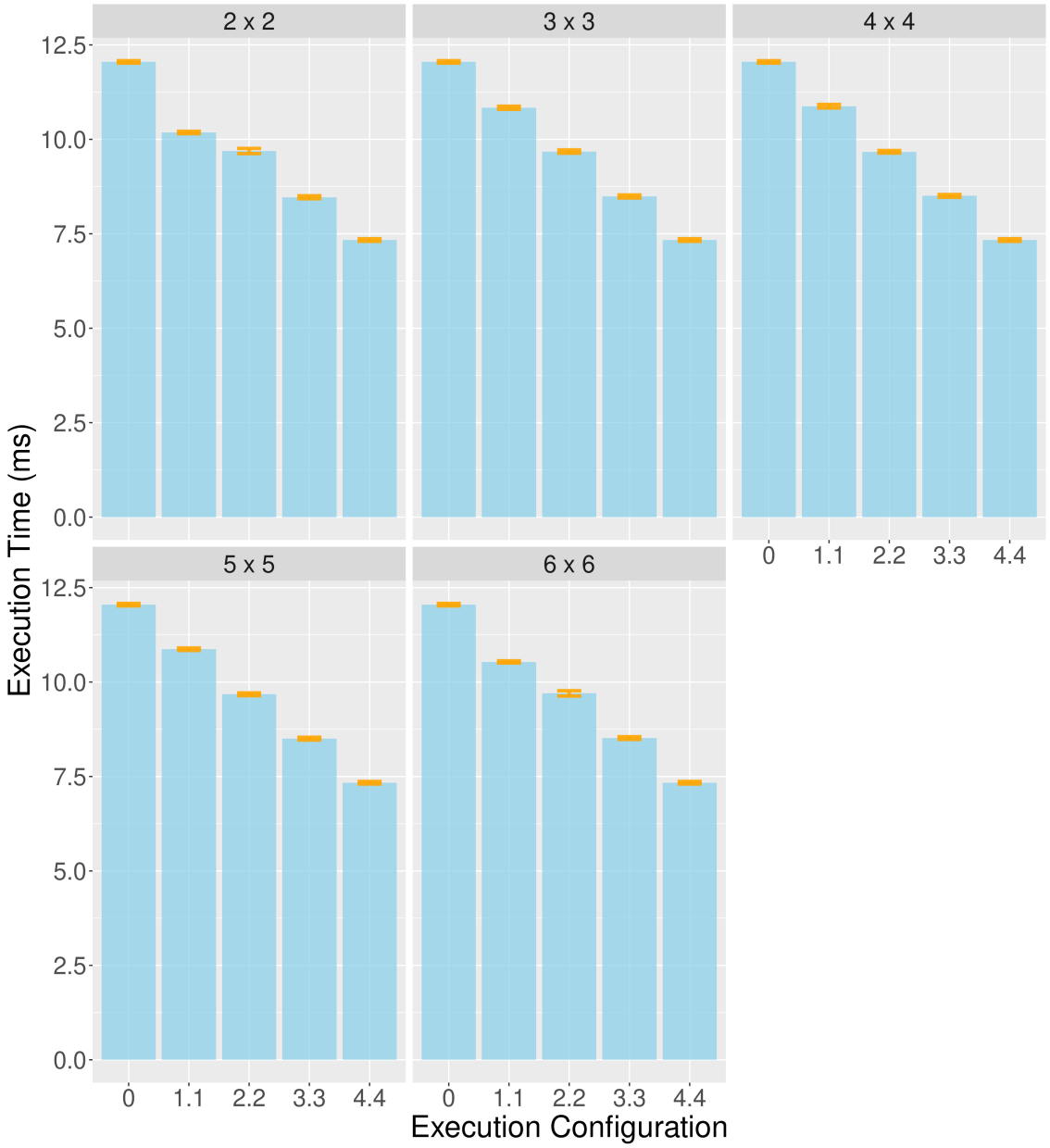
Source: The Authors

6.3 HotSpot3D

Figure 6.17 shows the runtimes of the HotSpot3D application for a three-dimensional problem of size 512x512x8 and 500 iterations for the execution of the application. On the Y axis, we have the runtime in milliseconds (ms). On the X axis, the three thresholds of loss of accuracy (in percentage) used for the generation of execution configurations in our method in the five configurations of measurement subsections, where the index 0 corresponds to running only the float kernel and index 4.4 corresponds to running only

the half kernel version. Comparing the runtime of the float and half kernel versions, it is possible to observe a significant difference. While the float version had an average runtime of 12.05 ms, the half version had an average runtime of just 7.33 ms, a speedup of 1.64.

Figure 6.17 – HotSpot3D runtime using a problem size of 512 and 500 iterations for different accuracy loss thresholds.



Source: The Authors

When analyzing the runtime of the three accuracy loss thresholds (1.1%, 2.2%, and 3.3%), the impact of the possibility of executing a more significant number of iterations in the half version provided by higher loss thresholds is evident. Let us compare the runtimes of the execution configurations generated for the 1.1% threshold in each of

the five measurement subsection configurations in Table A.9. We see that the execution configuration of the 2x2 subsection achieved the lowest runtime, 10.18 ms. Now, if we analyze the loss of accuracy profile for the 1.1% loss threshold in the 2x2 subsection of Figure 5.17, we will see that the execution configuration generated for the 2x2 section was the one with the highest number of iterations executed in the half version.

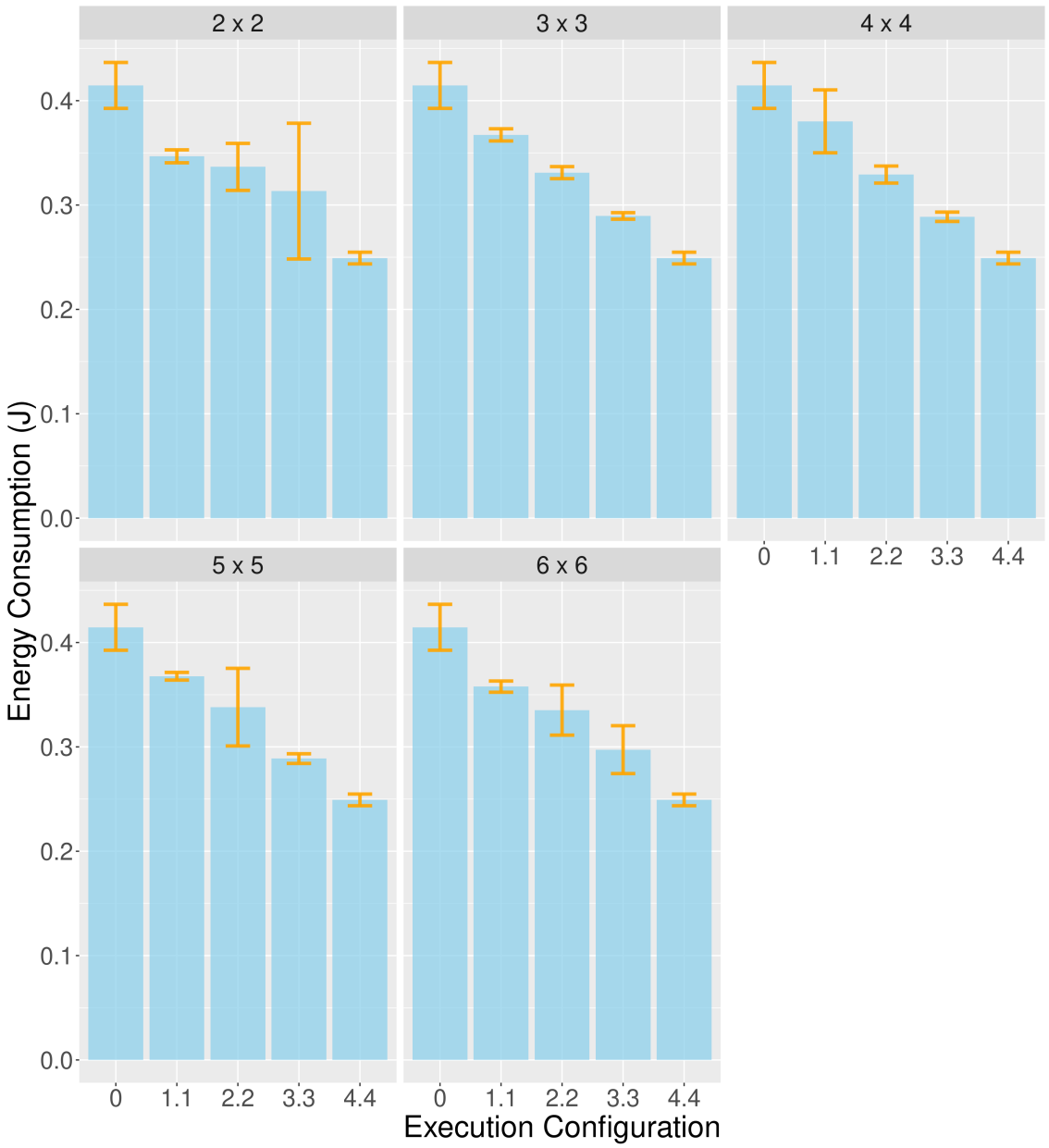
Likewise, let us look at the number of iterations performed on the half kernel versions in the generated configuration for the 1.1% limit in subsection 6x6. We see that it is the second highest. Comparing the runtimes, we see that this execution configuration had the second lowest runtime, 10.53 ms. In both cases, even including the computational cost to convert 32bit to 16bit floating-point data and vice versa, performing a more significant number of iterations in the faster half version compensates for this cost and is still capable of speeding up execution.

Figure 6.18 presents the energy consumption in a problem size of 512x512x8 and 500 iterations. When comparing the energy consumption of the execution configurations generated for the three thresholds of loss of accuracy and of the float and half kernel versions, it is possible to observe a gradual reduction in consumption as the number of iterations executed in the half version increases. As seen in Table A.9, while the float kernel version consumes 0.41 J, the half version consumes 0.25 J, a 39% reduction.

Furthermore, let us compare the energy consumption of the execution configurations to the 1.1% loss threshold of the 2x2 and 6x6 subsections, which had the lowest runtimes. It is possible to observe a significantly lower consumption than in the other subsections. While in the 2x2 subsection, the consumption was 0.35 J, in the 6x6 subsection, it was 0.36 J. In the 3x3, 4x4, and 5x5 configurations, the consumption was 0.37 J, 0.38 J, and 0.37 J, respectively, as seen in Table A.9. However, because these are significantly low values for runtime and energy consumption, we need to analyze the results with a more significant number of iterations and a more extensive problem size to better understand the impact on the performance of running a more significant number of iterations in the half version.

Figure 6.19 presents the running time for a problem of size 512x512x8 and 1000 iterations. As we can see, the gradual reduction in runtime repeats as the half kernel version performs a more significant number of iterations, just as with just 500 iterations at the same problem size in Figure 6.17. While the float kernel version takes an average of 24.52 ms, the half version takes just 15.33 ms, as shown in Table A.10, a speedup of approximately 1.60.

Figure 6.18 – HotSpot3D energy consumption using a problem size of 512 and 500 iterations for different accuracy loss thresholds.

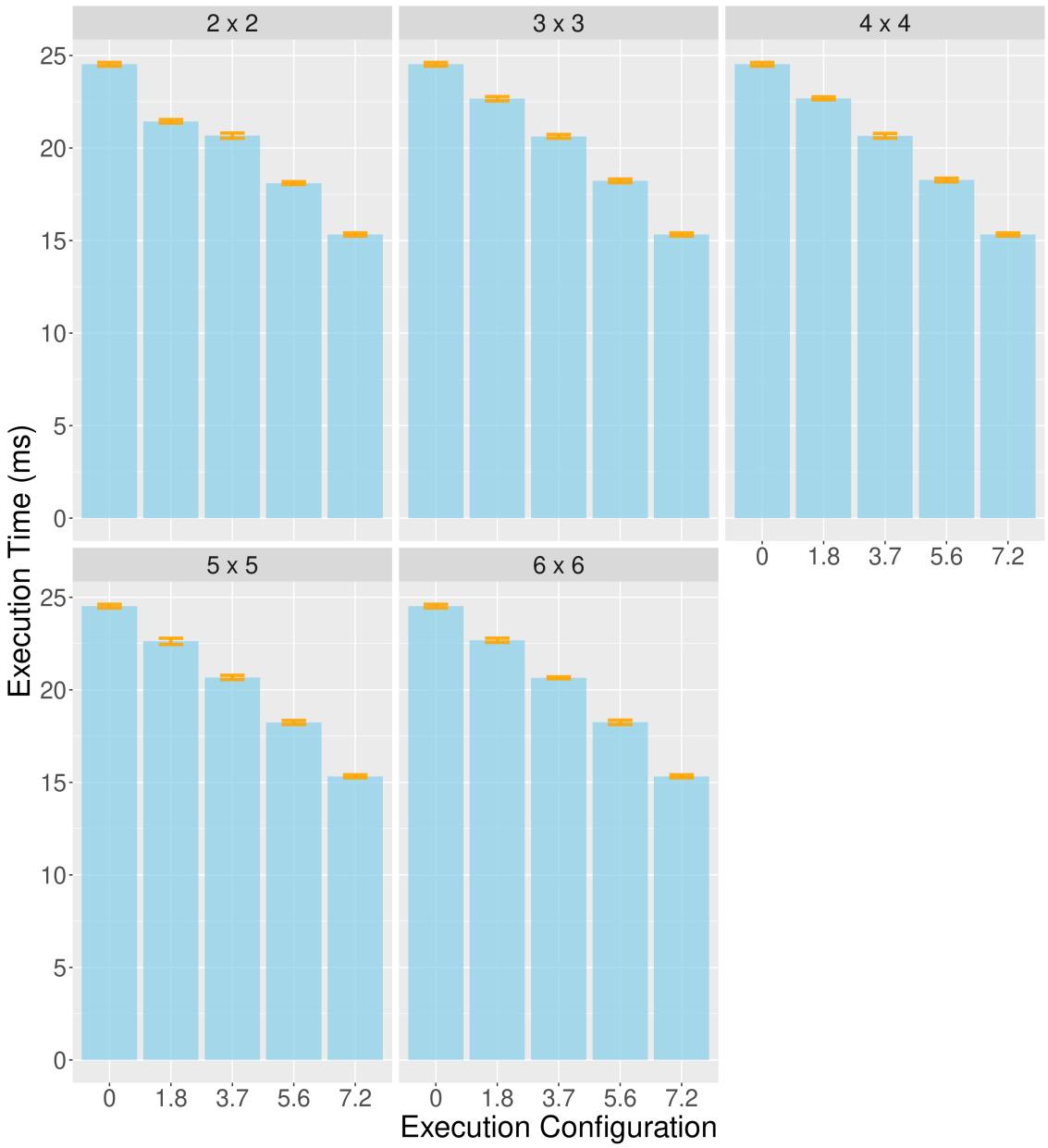


Source: The Authors

By analyzing the loss of accuracy profile generated by the method for the 1.8% threshold in Figure 5.20, we will see that the only subsection where there was a second interleave of the kernels to increase the number of iterations executed in the faster half version was the 2x2 subsection. Comparing the runtime of the 2x2 subsection with the other configurations of measurement subsections, we see that it obtained the lowest runtime, 21.44 ms, against more than 22.66 ms of the others.

Figure 6.20 presents the energy consumption for the problem of size 512x512x8 and 1000 iterations. While the kernel version consumes 0.92 J, the half version only

Figure 6.19 – HotSpot3D runtime using a problem size of 512 and 1000 iterations for different accuracy loss thresholds.

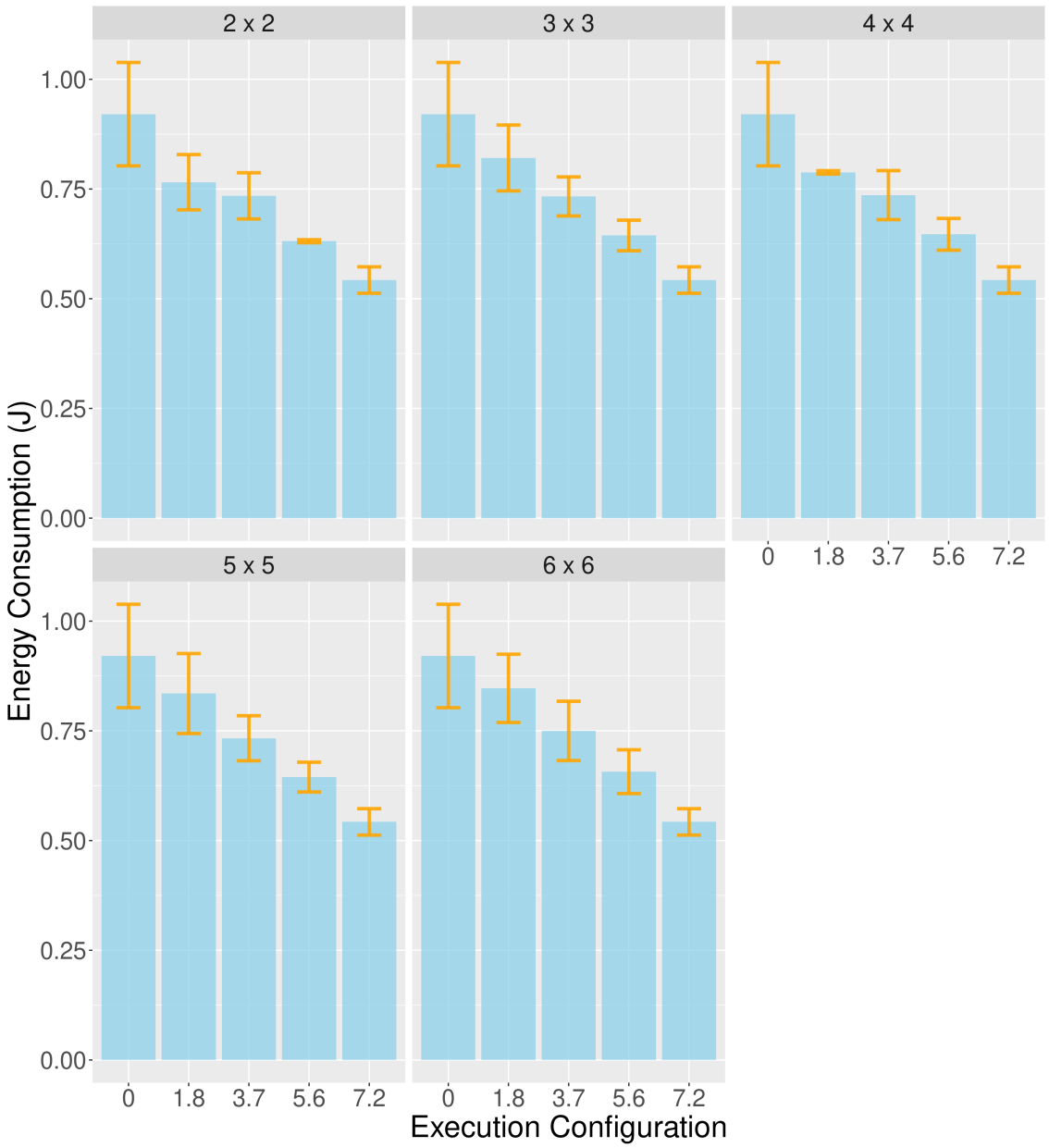


Source: The Authors

consumes 0.54 J, a reduction of over 41%. Furthermore, let us analyze the energy consumption of the execution configuration generated for the 2x2 subsection. We will see that the consumption was 0.77 J compared to more than 0.79 of the other subsections. Although small, it is possible to notice the difference in runtime and energy consumption with multiple interleaving kernel versions. Even with the cost of converting data from 32bit floating-point to 16bit and vice versa included in the time and energy consumption, it can reduce both time and energy.

Figures 6.21 and 6.22 present the runtime and energy consumption for a prob-

Figure 6.20 – HotSpot3D energy consumption using a problem size of 512 and 1000 iterations for different accuracy loss thresholds.

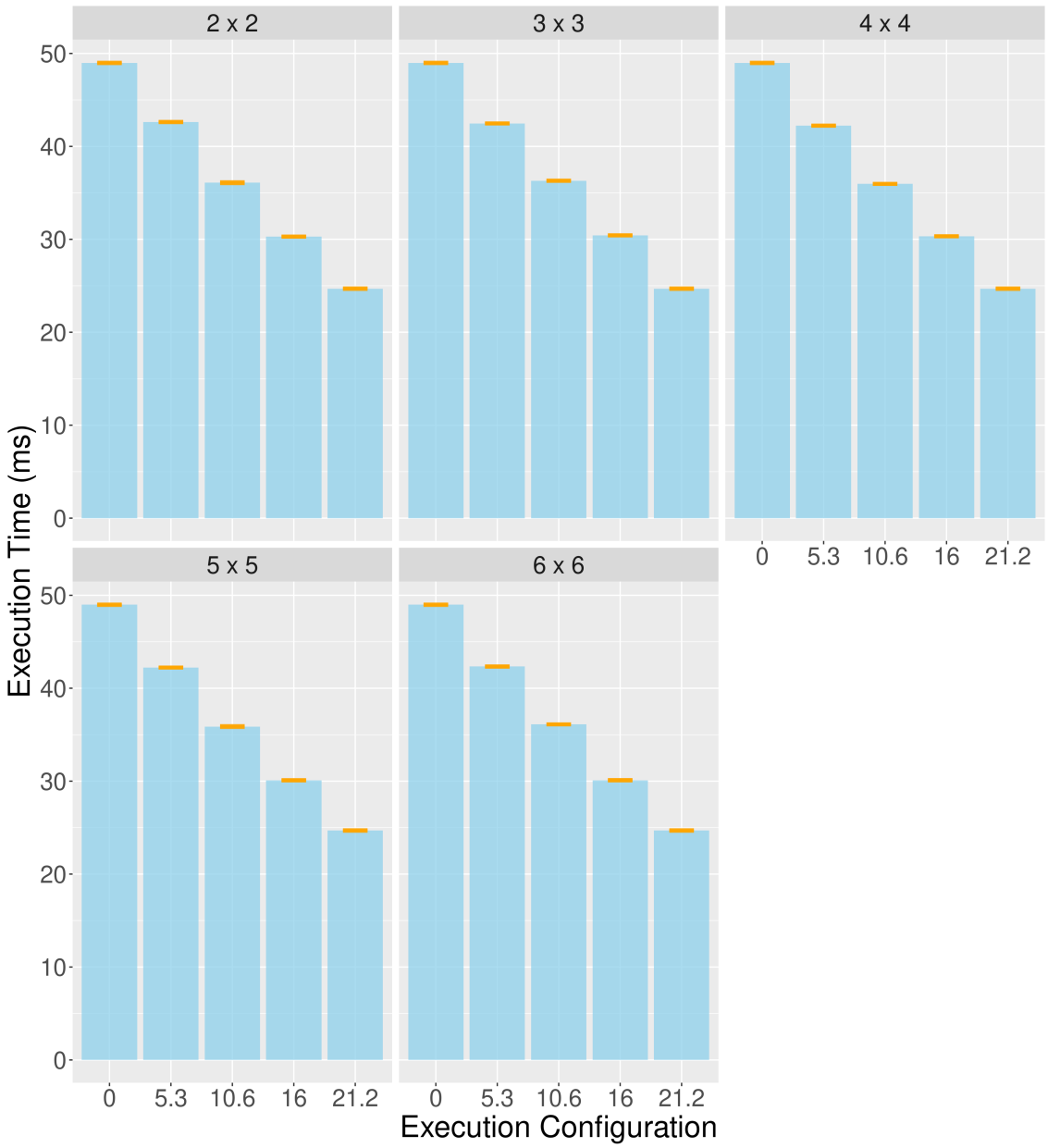


Source: The Authors

lem of size 1024x1024x8 and 500 iterations. Again, it is possible to observe a gradual reduction in runtime and energy consumption as the half kernel version executes more iterations. Comparing the runtime and energy consumption of the float and half kernel versions in Table A.11, it is possible to observe a speedup of 1.98 and an energy consumption reduction of 59% from the float version to the half version. While the float version takes 48.98 ms and consumes 2.32 J, the half version only takes 24.69 ms and consumes only 0.94 J.

The same happens in Figures 6.23 and 6.24, where the runtime and energy con-

Figure 6.21 – HotSpot3D runtime using a problem size of 1024 and 500 iterations for different accuracy loss thresholds.

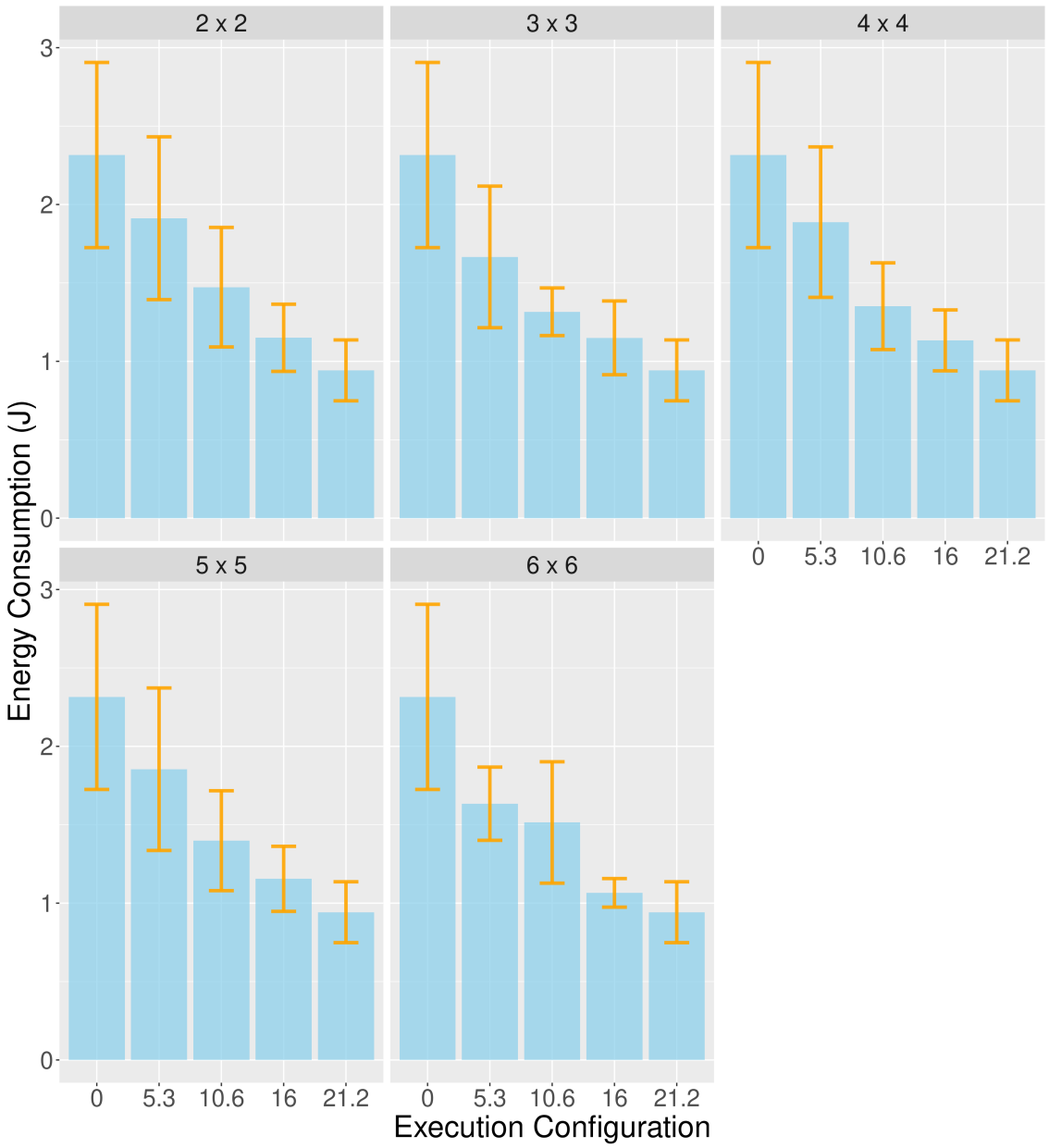


Source: The Authors

sumption are presented for a problem of size 1024x1024x8 and 1000 iterations. While the float kernel version takes an average of 100.65 ms to complete execution and consumes 5.65 J, the half version takes only 50.53 ms and consumes just 2.15 J. It corresponds to a speedup of almost 2.00 and a more than 61% energy consumption reduction.

In both cases, with 500 and 1000 iterations, there was no retraction of the loss of accuracy after interleaving the float kernel version by the half version when reaching the established loss threshold. Although, in this case, it is inconceivable to perform more interleaves, and the method generates consistent configurations based on this, it is possi-

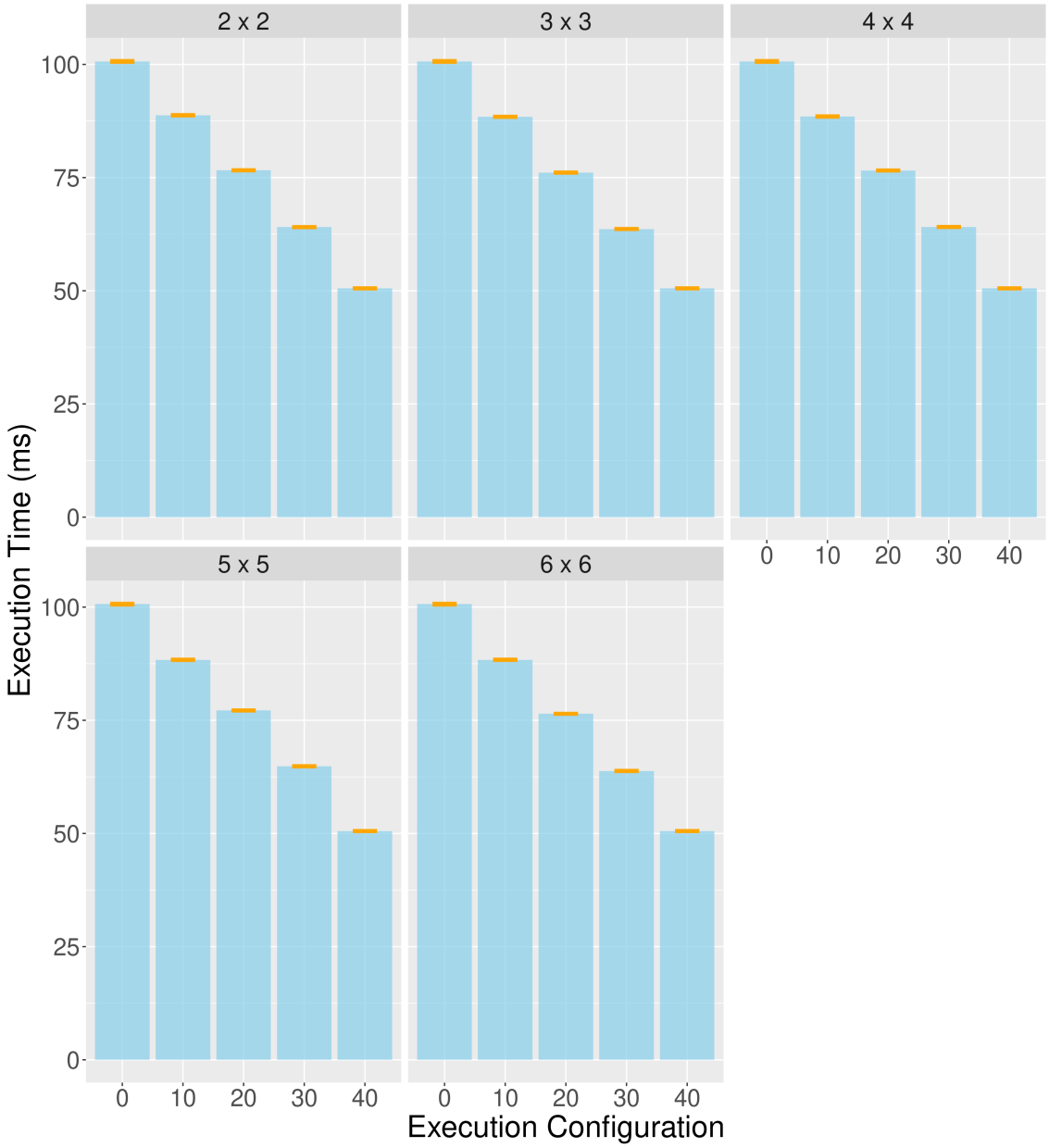
Figure 6.22 – HotSpot3D energy consumption using a problem size of 1024 and 500 iterations for different accuracy loss thresholds.



Source: The Authors

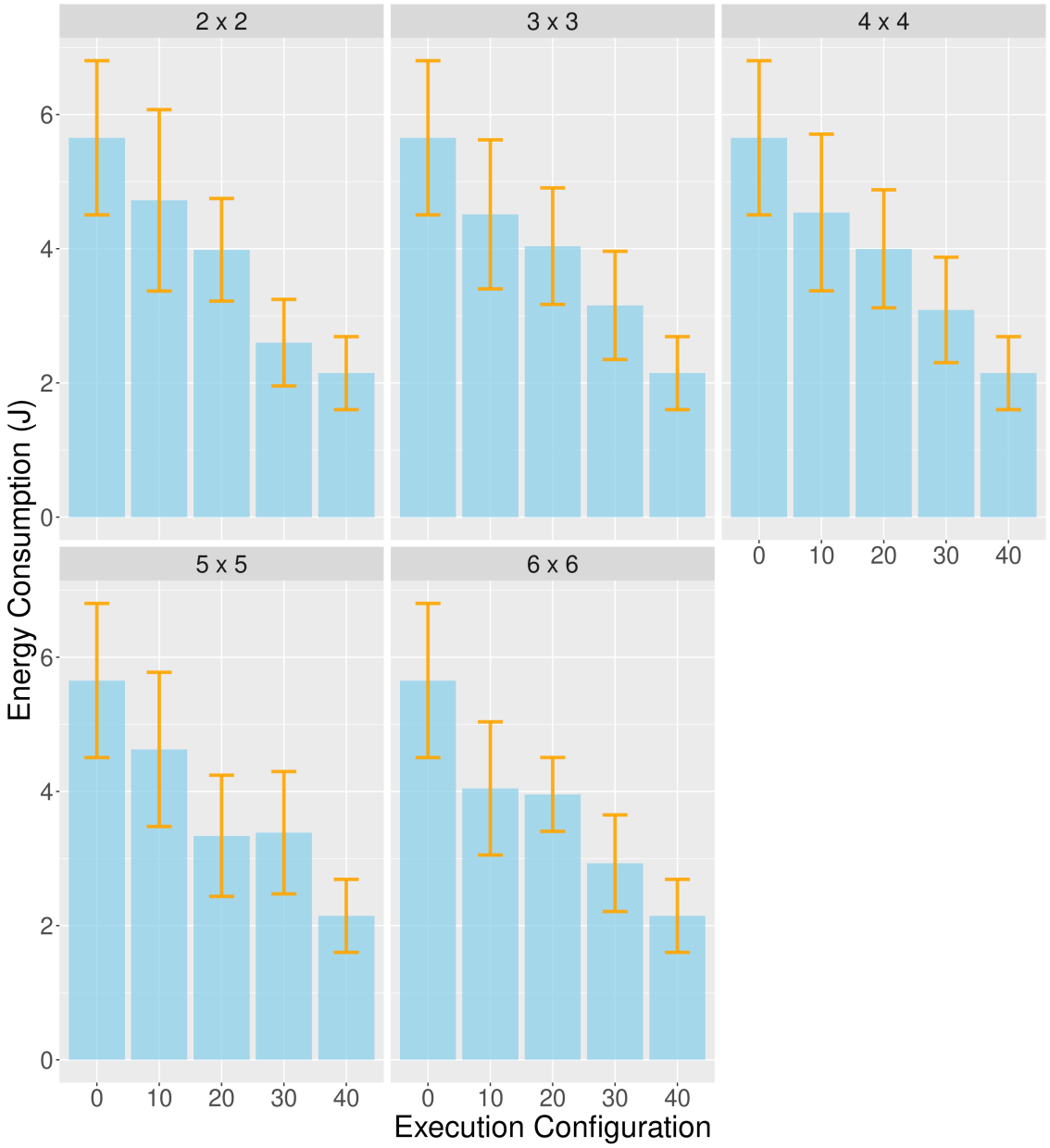
ble to observe the effect on the performance of executions of a more significant number of iterations as the loss threshold increases. While with a stricter threshold of 10% and, therefore, the possibility of executing a lower number of iterations in the faster and more efficient half version, the runtime is approximately 88 ms with an approximate consumption of 4.5 J. With a threshold of 20%, the runtime is approximately 76.5 ms with an approximate consumption of 4 J. With a threshold of 30%, the runtime is approximately 64 ms and consumption of approximately 3 J.

Figure 6.23 – HotSpot3D runtime using a problem size of 1024 and 1000 iterations for different accuracy loss thresholds.



Source: The Authors

Figure 6.24 – HotSpot3D energy consumption using a problem size of 1024 and 1000 iterations for different accuracy loss thresholds.



Source: The Authors

7 CONCLUSIONS

Approximate computing techniques, particularly those involving reduced and mixed precision, are widely studied in the literature as a means of accelerating application execution and reducing energy consumption. While many studies have analyzed the impact of these techniques on the performance and accuracy of a wide range of application domains, most focus on serial and non-iterative applications. Although some studies have explored various approximate computing techniques for iterative applications, one of the biggest challenges of using approximation in this context is its sensitivity to errors, particularly in the use of reduced-precision floating-point formats. Iterative applications rely on the results of previous computations, which are then used to perform subsequent computations on the application's dataset in the next iteration. Therefore, introducing precision errors in the execution of an iterative application can propagate and magnify significantly throughout the execution, making error control extremely important.

Iterative applications often operate on large datasets, which presents an additional challenge to approximate computing: monitoring the loss of execution accuracy. The most popular way to manage applications' accuracy loss is through runtime loss monitoring. While this is a simple and effective way to monitor accuracy loss in applications with small data domains, it becomes unfeasible in applications with a large volume of data, such as physical simulation applications where the data domains are usually multi-dimensional. Calculating the loss of accuracy at runtime is computationally expensive, and this problem is exacerbated in iterative applications where computations are typically repeated numerous times. Therefore, it is not feasible to monitor the loss of accuracy at each iteration of the execution without significantly increasing the computational cost.

One of the most popular techniques for approximate computing in GPUs is Precision Scaling. This technique involves changing the floating-point precision of a set of variables or arithmetic operations. Precision Scaling seeks to find the most suitable floating-point representation format for each variable or arithmetic operation based on a previously established limit for loss of accuracy. The goal is to reduce the size of the floating-point representation format for operations while ensuring that the error introduced by the reduction does not exceed the established limit for loss of accuracy.

Although Precision Scaling can significantly improve performance and energy efficiency in applications with a relatively small number of variables and arithmetic operations, it becomes increasingly difficult to optimize larger and more complex applications

due to the growing search space for optimizations. Furthermore, when optimizing iterative applications, the propagation and amplification of loss of accuracy between iterations makes it challenging to estimate the impact of changes in the floating-point representation on the variables and application operations. Using a new limit for loss of accuracy requires a new optimization of the application.

This thesis presents a methodology for sampling the loss of accuracy profile, extracting performance statistics and loss of accuracy, and generating interleaved execution configurations of multiple kernel versions of iterative applications on GPUs. To collect samples of the loss of accuracy profile of iterative applications throughout their execution, we divide the space represented by the number of iterations and the total loss of accuracy of the kernel version with the most significant loss into subsections. From these subsections, we collect samples of the loss of accuracy profile of multiple kernel versions in different execution configurations so that each version is included in the most extensive number of subsections. After collecting all the samples, we extract performance statistics and loss of accuracy for each kernel version at different stages of application execution through the subsections. From these statistics, we generate execution configurations that exploit the interleaved execution of multiple kernel versions according to a previously established limit for loss of accuracy.

We demonstrate that our methodology can generate interleaved execution configurations of kernel versions for different limits of accuracy loss in iterative applications in an offline manner. In addition, a single run of our methodology can collect samples of the accuracy loss profile of kernel versions at different stages of application execution. From these samples, performance and accuracy loss statistics can be extracted. This allows for generating numerous execution configurations with different limits of accuracy loss simply and practically, without requiring new measurements for the same kernel versions and problem size.

The experiments conducted in three iterative applications of physical simulation in three-dimensional data domains demonstrated the capability of the methodology to reduce runtime and energy consumption by interleaving kernel versions. While the method responsible for generating interleaved execution configurations of the kernel versions delivered an average performance, with some cases where it was unable to explore the interleaving of kernel versions to reduce runtime and potentially energy consumption, in the cases where it was successful, the results showed significant improvements, particularly in energy consumption: speedups up to 2 and up to 60% energy consumption reduction.

Moreover, although some results violated the loss of accuracy limits, only a minority significantly exceeded them.

Due to the various execution ramifications that occur during the collection of samples, our methodology requires a significant amount of GPU memory. As we only run our methodology on GPUs, we simplify our approach by disregarding cases where available memory is insufficient. In such cases, a simple solution would be to store the states of the data structures (which are otherwise duplicated) in the main system memory or on disk in case system memory is insufficient. This would eliminate the need to duplicate the structures and allow the same structures to be used. At the end of each phase, the states of the respective structures can be recovered from main memory or disk.

For simplicity, we only use two kernel versions - one exact and one approximate - in this work. Although different versions with a gradient of accuracy could be used, interleaving with a sequence of versions with increasing or decreasing accuracy, the purpose of this work is to demonstrate that it is possible to generate an execution configuration in a automated, simple and fast way based on the measurements of the loss of accuracy profile and the extraction of statistics from these measurements. Therefore, we decided to use only two versions: the exact version and an approximate, faster version. This approach shows that interleaving kernel versions with different levels of accuracy reduces not only runtime but also energy consumption.

As a future work, we suggest studying different optimization strategies for the method that generates interleaved execution configurations of kernel versions. Specifically, we recommend analyzing the use of neural networks and machine learning as an alternative to the strategy presented in this work for interleaving kernel versions. By using neural networks and machine learning, it would be possible to introduce a third intermediate kernel version or an arbitrary number of kernel versions with an accuracy gradient. This would allow leveraging the interleaving of different kernel versions, using versions with varying accuracy depending on the exact version's retraction conditions (or of versions with greater precision).

7.1 Publications

The following publications were made during the thesis:

- FREYTAG, Gabriel et al. Impact of Reduced and Mixed-Precision on the Efficiency

of a Multi-GPU Platform on CFD Applications. In: **International Conference on Computational Science and Its Applications**. Springer, Cham, 2022. p. 570-587.

- FREYTAG, Gabriel et al. Collaborative execution of fluid flow simulation using non-uniform decomposition on heterogeneous architectures. **Journal of Parallel and Distributed Computing**, v. 152, p. 11-20, 2021.
- FREYTAG, Gabriel et al. Impacto da Precisão Reduzida e Mista na Computação do Método de Lattice-Boltzmann em Multiplas GPUs. In: **Anais da XX Escola Regional de Alto Desempenho da Região Sul**. SBC, 2020. p. 177-178.
- FREYTAG, Gabriel et al. Non-uniform partitioning for collaborative execution on heterogeneous architectures. In: **2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. IEEE, 2019. p. 128-135.
- LIMA, João VF et al. A dynamic task-based d3q19 lattice-boltzmann method for heterogeneous architectures. In: **2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)**. IEEE, 2019. p. 108-115.
- FREYTAG, Gabriel et al. Non-uniform domain decomposition for heterogeneous accelerated processing units. In: **International Conference on Vector and Parallel Processing**. Springer, Cham, 2018. p. 105-118.

REFERENCES

BAEK, W.; CHILIMBI, T. M. Green: A framework for supporting energy-conscious programming using controlled approximation. **ACM SIGPLAN Notices**, v. 45, n. 6, p. 198–209, 2010. ISSN 15232867.

BOLZE, R. et al. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. **The International Journal of High Performance Computing Applications**, v. 20, n. 4, p. 481–494, 2006. Available from Internet: <https://doi.org/10.1177/1094342006070078>.

BU, Y. et al. Haloop: Efficient iterative data processing on large clusters. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 3, n. 1-2, p. 285–296, 2010.

CARSON, E.; STRAKOŠ, Z. On the cost of iterative computations. **Philosophical Transactions of the Royal Society A**, The Royal Society Publishing, v. 378, n. 2166, p. 20190050, 2020.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE. **2009 IEEE international symposium on workload characterization (IISWC)**. [S.l.], 2009. p. 44–54.

CHERUBIN, S. et al. TAFFO: Tuning Assistant for Floating to Fixed Point Optimization. **IEEE Embedded Systems Letters**, IEEE, v. 12, n. 1, p. 5–8, 2020. ISSN 19430671.

CHIANG, W.-F. et al. Rigorous floating-point mixed-precision tuning. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 52, n. 1, p. 300–315, 2017.

CHIPPA, V. K. et al. Analysis and characterization of inherent application resilience for approximate computing. In: **Proceedings of the 50th Annual Design Automation Conference**. New York, NY, USA: Association for Computing Machinery, 2013. (DAC '13), p. 1–9. ISBN 9781450320719. Available from Internet: <https://doi.org/10.1145/2463209.2488873>.

ESMAEILZADEH, H. et al. Neural acceleration for general-purpose approximate programs. In: IEEE. **2012 45th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2012. p. 449–460.

GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 23, n. 1, p. 5–48, 1991.

GRAILLAT, S. et al. Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic. **Journal of Computational Science**, v. 36, 2019. ISSN 18777503.

GUPTA, V. et al. Impact: Imprecise adders for low-power approximate computing. In: IEEE. **IEEE/ACM International Symposium on Low Power Electronics and Design**. [S.l.], 2011. p. 409–414.

HO, N.; WONG, W. Exploiting half precision arithmetic in nvidia gpus. In: **2017 IEEE High Performance Extreme Computing Conference (HPEC)**. [S.l.: s.n.], 2017. p. 1–7.

HO, N. M. et al. Efficient floating point precision tuning for approximate computing. **Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC**, IEEE, v. 0, p. 63–68, 2017.

HO, N. M.; SILVA, H. D.; WONG, W. F. GRAM: A Framework for Dynamically Mixing Precisions in GPU Applications. **ACM Transactions on Architecture and Code Optimization**, v. 18, n. 2, 2021. ISSN 15443973.

HOFFMANN, H. et al. Using code perforation to improve performance, reduce energy consumption, and respond to failures. 2009.

IEEE Standard for Binary Floating-Point Arithmetic. **ANSI/IEEE Std 754-1985**, p. 1–20, 1985.

IEEE Standard for Floating-Point Arithmetic. **IEEE Std 754-2008**, p. 1–70, 2008.

JIN, C. et al. A survey on software methods to improve the energy efficiency of parallel computing. **The International Journal of High Performance Computing Applications**, Sage Publications Sage UK: London, England, v. 31, n. 6, p. 517–549, 2017.

KERAMIDAS, G.; KOKKALA, C.; STAMOULIS, I. Clumsy value cache: An approximate memoization technique for mobile gpu fragment shaders. In: **Workshop on approximate computing (WAPCO'15)**. [S.l.: s.n.], 2015. p. 6.

KHUDIA, D. S. et al. Rumba: An online quality management system for approximate computing. In: **Proceedings - International Symposium on Computer Architecture**. New York, New York, USA: ACM Press, 2015. v. 13-17-June, p. 554–566. ISBN 9781450334020. ISSN 10636897. Available from Internet: <http://dl.acm.org/citation.cfm?doid=2749469.2750371>.

KIM, Y.; ZHANG, Y.; LI, P. An energy efficient approximate adder with carry skip for error resilient neuromorphic vlsi systems. In: IEEE. **2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.], 2013. p. 130–137.

KOTIPALLI, P. V. et al. AMPT-GA: Automatic mixed precision floating point tuning for GPU applications. **Proceedings of the International Conference on Supercomputing**, p. 160–170, 2019.

LAGUNA, I. et al. GPUMixer: Performance-driven floating-point tuning for GPU scientific applications. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 11501 LNCS, p. 227–246, 2019. ISSN 16113349.

LIU, C.; HAN, J.; LOMBARDI, F. A low-power, high-performance approximate multiplier with configurable partial error recovery. In: IEEE. **2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2014. p. 1–4.

MENON, H.; LAM, M. O. ADAPT: Algorithmic Differentiation for Floating-Point Precision Tuning. 2019.

MIGUEL, J. S.; BADR, M.; JERGER, N. E. Load value approximation. In: IEEE. **2014 47th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.], 2014. p. 127–139.

MITTAL, S. A survey of techniques for approximate computing. **ACM Computing Surveys**, v. 48, n. 4, 2016. ISSN 15577341.

PALFRAMAN, D. J.; KIM, N. S.; LIPASTI, M. H. Precision-aware soft error protection for gpus. In: IEEE. **2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.], 2014. p. 49–59.

RAHIMI, A.; BENINI, L.; GUPTA, R. K. Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures. **IEEE Transactions on Circuits and Systems II: Express Briefs**, IEEE, v. 60, n. 12, p. 847–851, 2013.

ROY, P. et al. ASAC: Automatic sensitivity analysis for approximate computing. **ACM SIGPLAN Notices**, v. 49, n. 5, p. 95–104, 2014. ISSN 15232867.

RUBIO-GONZ, C. et al. Precimonious: Tuning Assistant for Floating-Point Precision Categories and Subject Descriptors. **SC '13 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**, 2013.

RUBIO-GONZALEZ, C. et al. Floating-point precision tuning using blame analysis. **Proceedings - International Conference on Software Engineering**, v. 14-22-May-2016, p. 1074–1085, 2016. ISSN 02705257.

SAMADI, M. et al. Paraprox: pattern-based approximation for data parallel applications. **ACM SIGPLAN Notices**, v. 49, n. 4, p. 35–50, 2014. ISSN 1558-1160.

SAMADI, M. et al. Scaling performance via self-tuning approximation for graphics engines. **ACM Transactions on Computer Systems**, v. 32, n. 3, p. 1–29, 2014. ISSN 15577333.

SAMPSON, A. et al. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. **ACM SIGPLAN Notices**, v. 47, n. 6, p. 164, 2011. ISSN 03621340.

SAMPSON, A. et al. Approximate storage in solid-state memories. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 32, n. 3, p. 1–23, 2014.

SCHKUFZA, E.; SHARMA, R.; AIKEN, A. Stochastic optimization of floating-point programs with tunable precision. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 49, n. 6, p. 53–64, 2014.

SHALF, J. The future of computing beyond moore's law. **Philosophical Transactions of the Royal Society A**, The Royal Society Publishing, v. 378, n. 2166, p. 20190061, 2020.

SIDIROGLOU-DOUSKOS, S. et al. Managing performance vs. accuracy trade-offs with loop perforation. In: **Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering**. [S.l.: s.n.], 2011. p. 124–134.

SUTHERLAND, M.; MIGUEL, J. S.; JERGER, N. E. Texture cache approximation on gpus. In: **Workshop on approximate computing across the stack**. [S.l.: s.n.], 2015. p. 3.

THOMPSON, N. C. et al. The computational limits of deep learning. **arXiv preprint arXiv:2007.05558**, 2020.

VENKATARAMANI, S. et al. Quality programmable vector processors for approximate computing. In: IEEE. **2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.], 2013. p. 1–12.

XU, Q.; MYTKOWICZ, T.; KIM, N. S. Approximate computing: A survey. **IEEE Design & Test**, IEEE, v. 33, n. 1, p. 8–22, 2015.

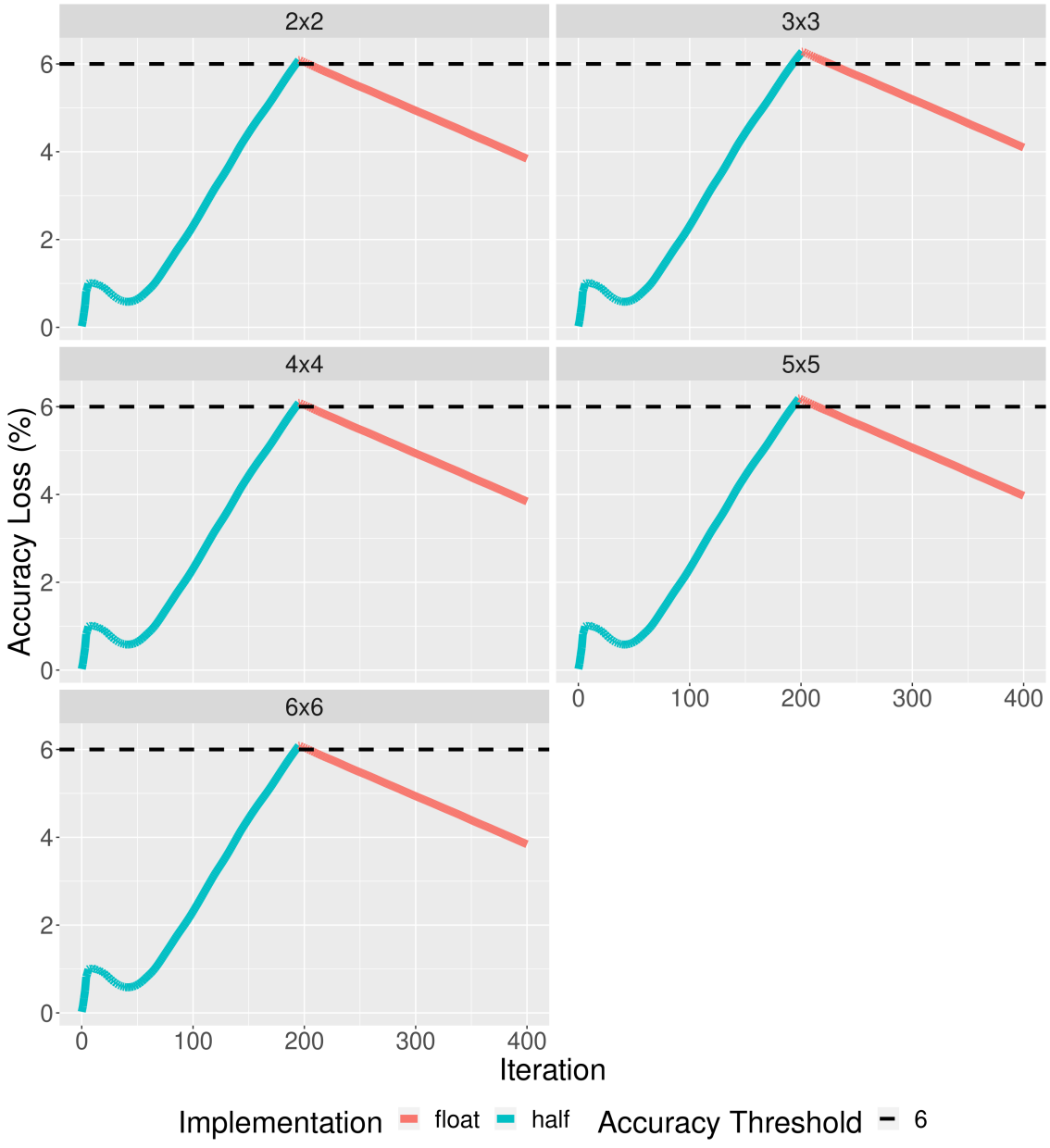
ZHANG, Q. et al. ApproxIt: An Approximate Computing Framework for Iterative Methods. **Proceedings of the 51st Annual Design Automation Conference**, p. 1–6, 2014.

APPENDIX A — EXPERIMENT RESULTS

A.1 Accuracy Loss Profiles

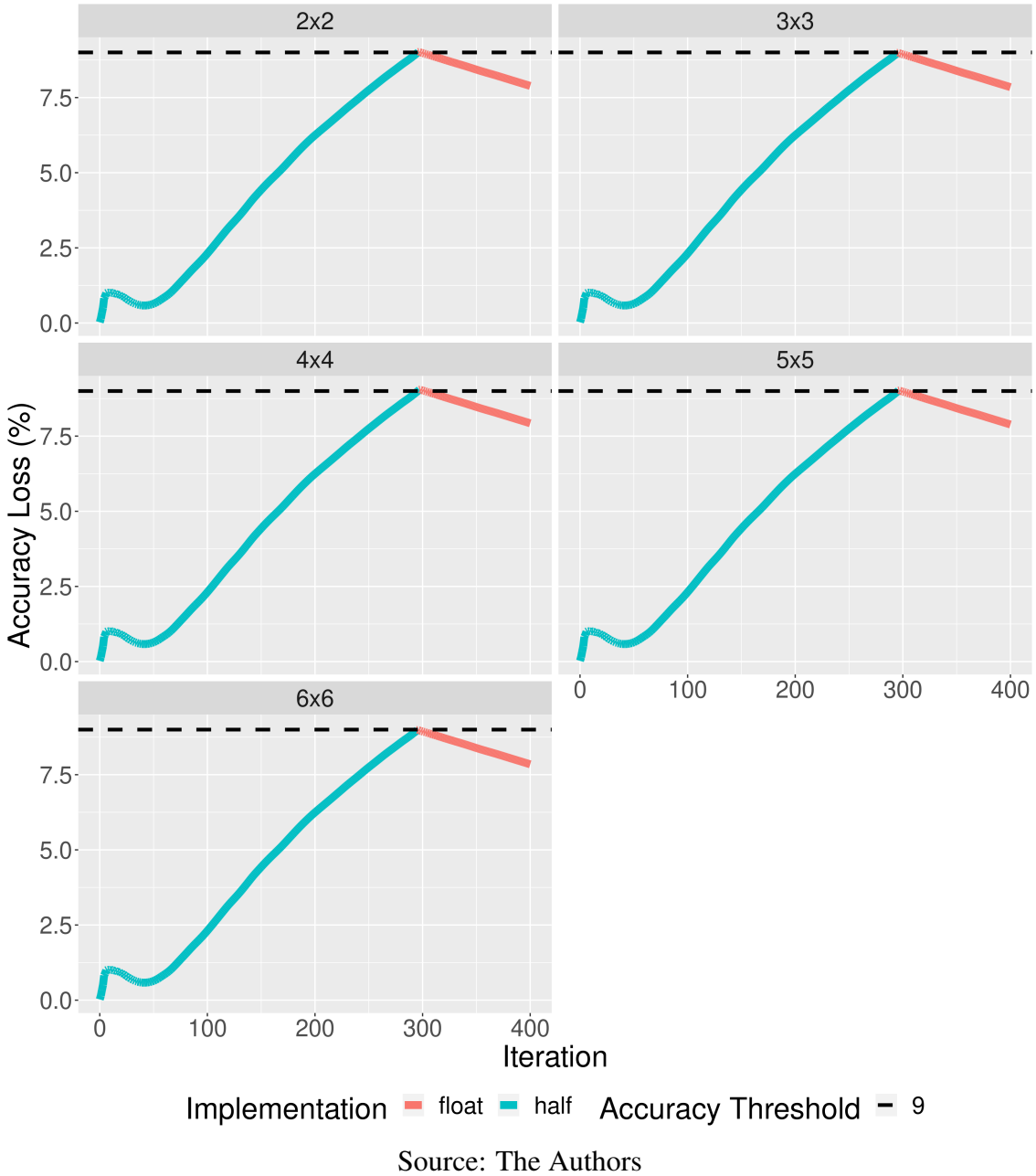
A.1.1 LBM3D

Figure A.1 – LBM3D accuracy loss profile using a problem size of 64, 400 iterations, and an accuracy loss threshold of 6%.



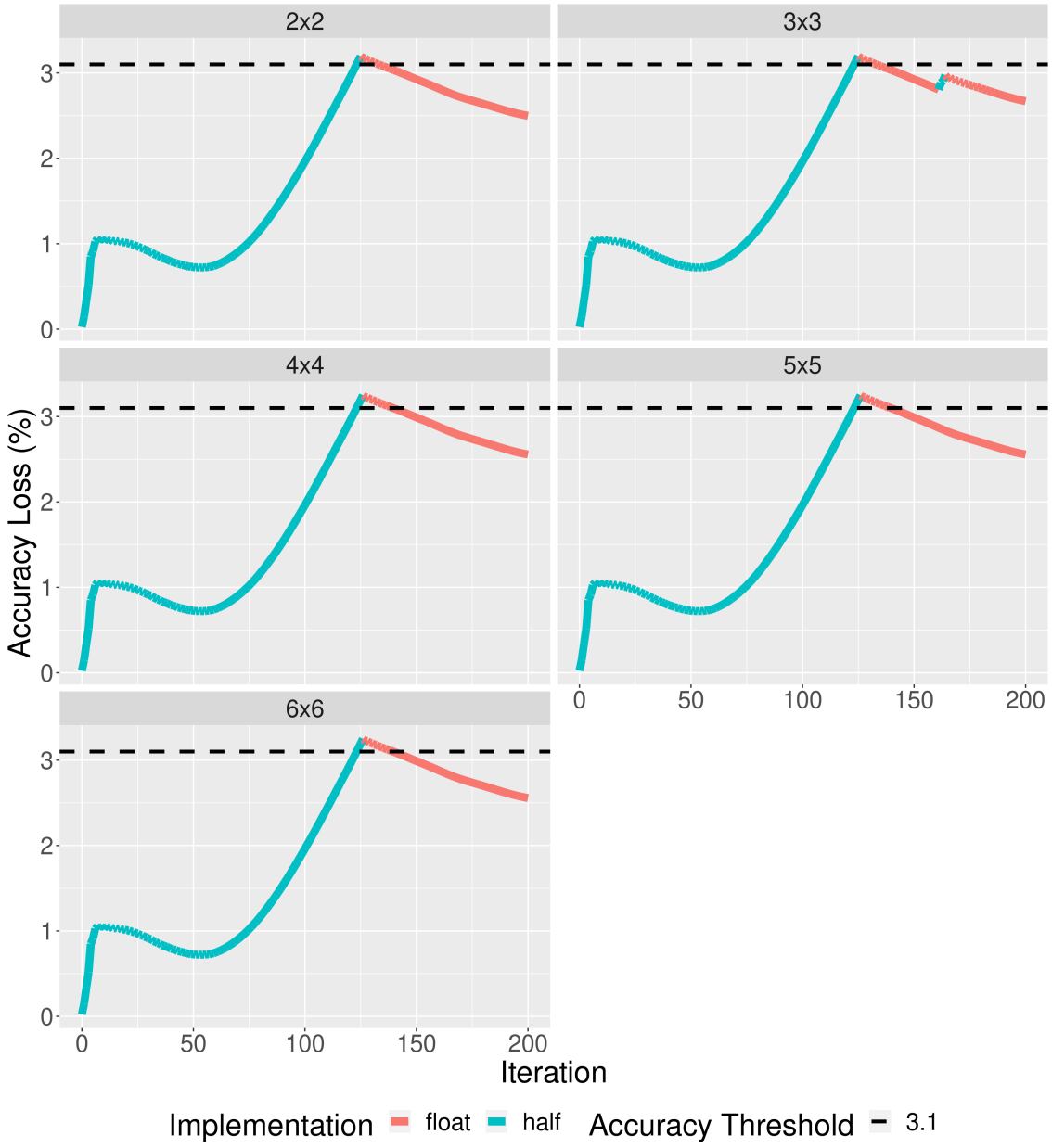
Source: The Authors

Figure A.2 – LBM3D accuracy loss profile using a problem size of 64, 400 iterations, and an accuracy loss threshold of 9%.



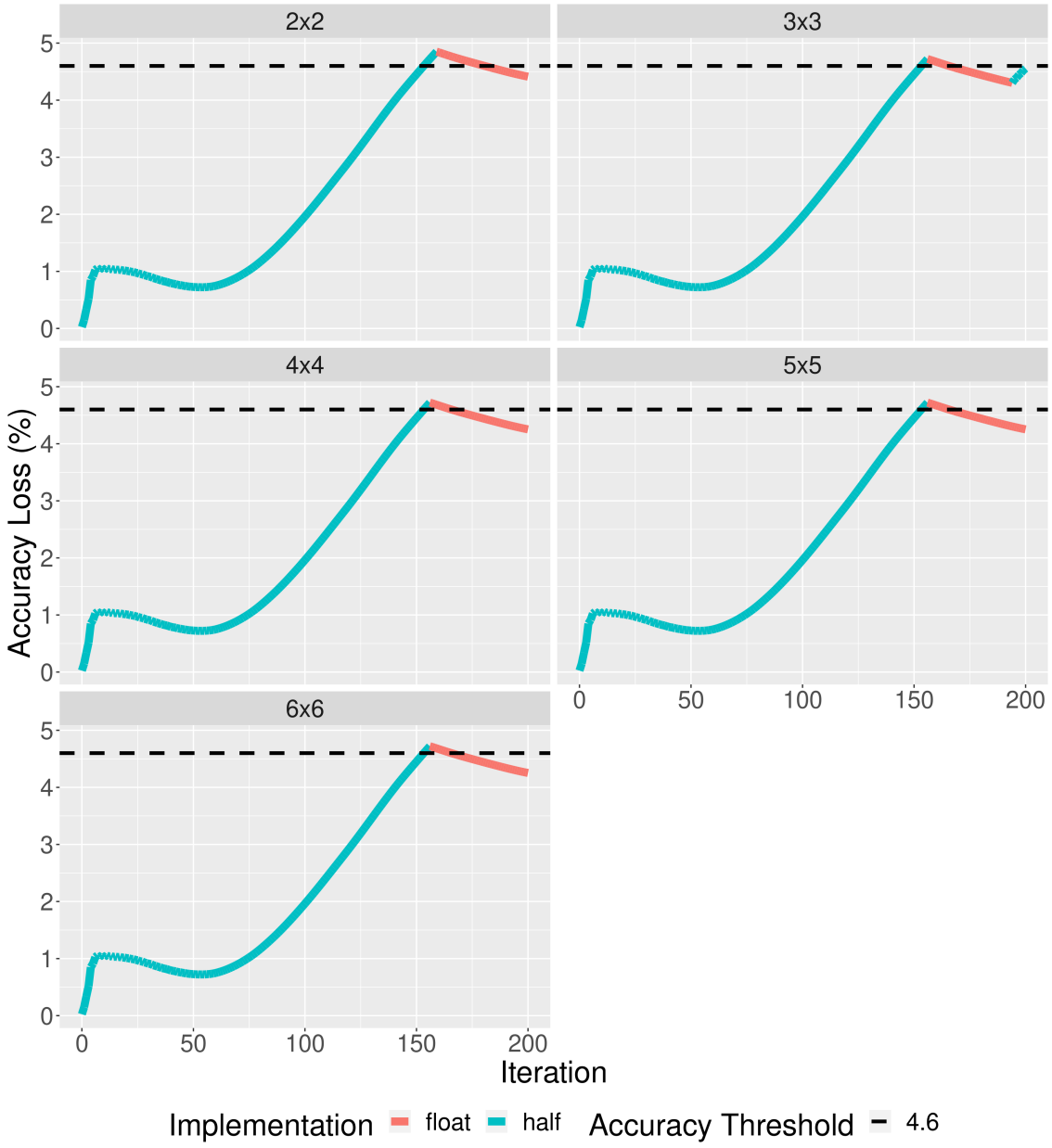
Source: The Authors

Figure A.3 – LBM3D accuracy loss profile using a problem size of 128, 200 iterations, and an accuracy loss threshold of 3.1%.



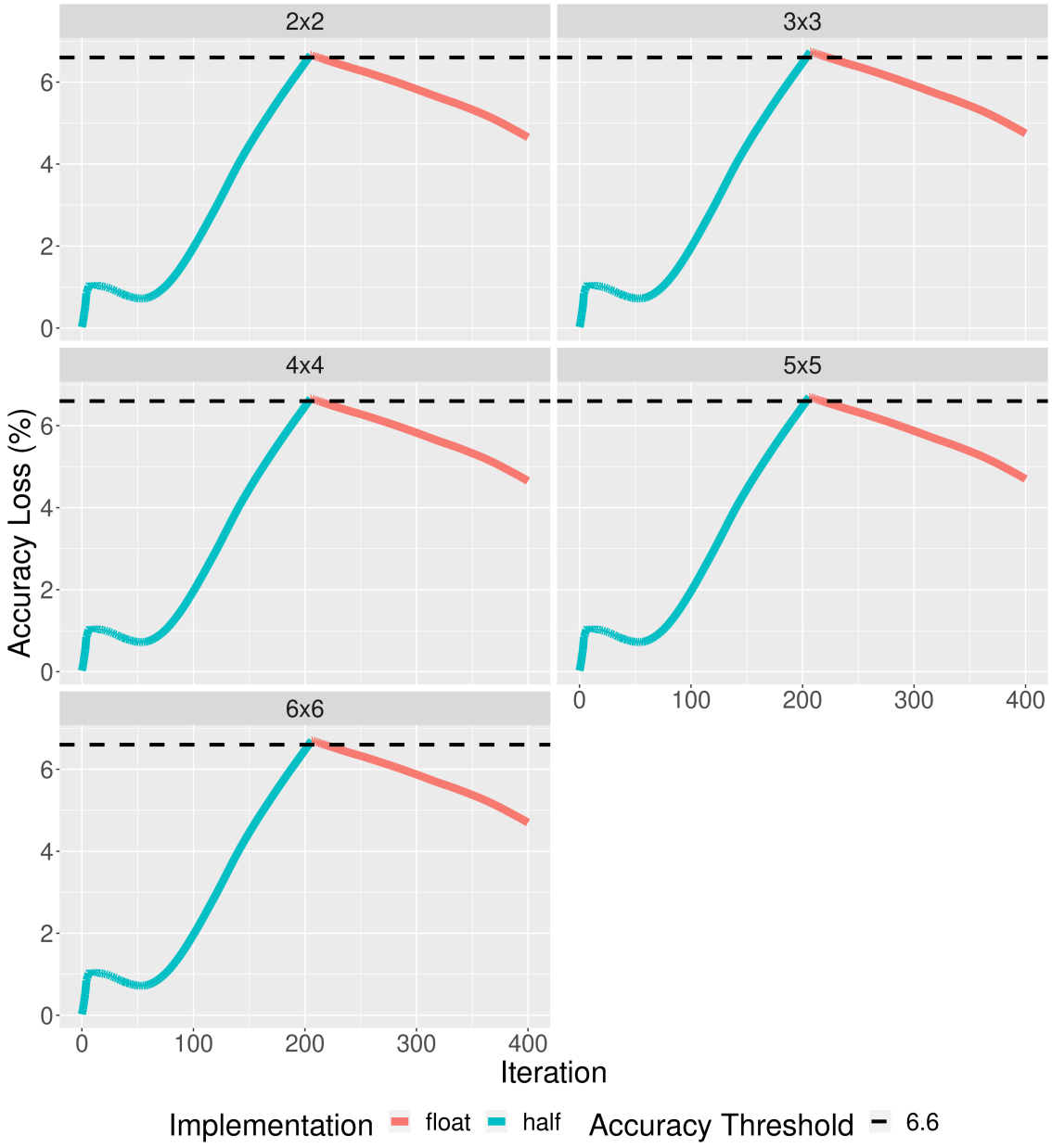
Source: The Authors

Figure A.4 – LBM3D accuracy loss profile using a problem size of 128, 200 iterations, and an accuracy loss threshold of 4.6%.



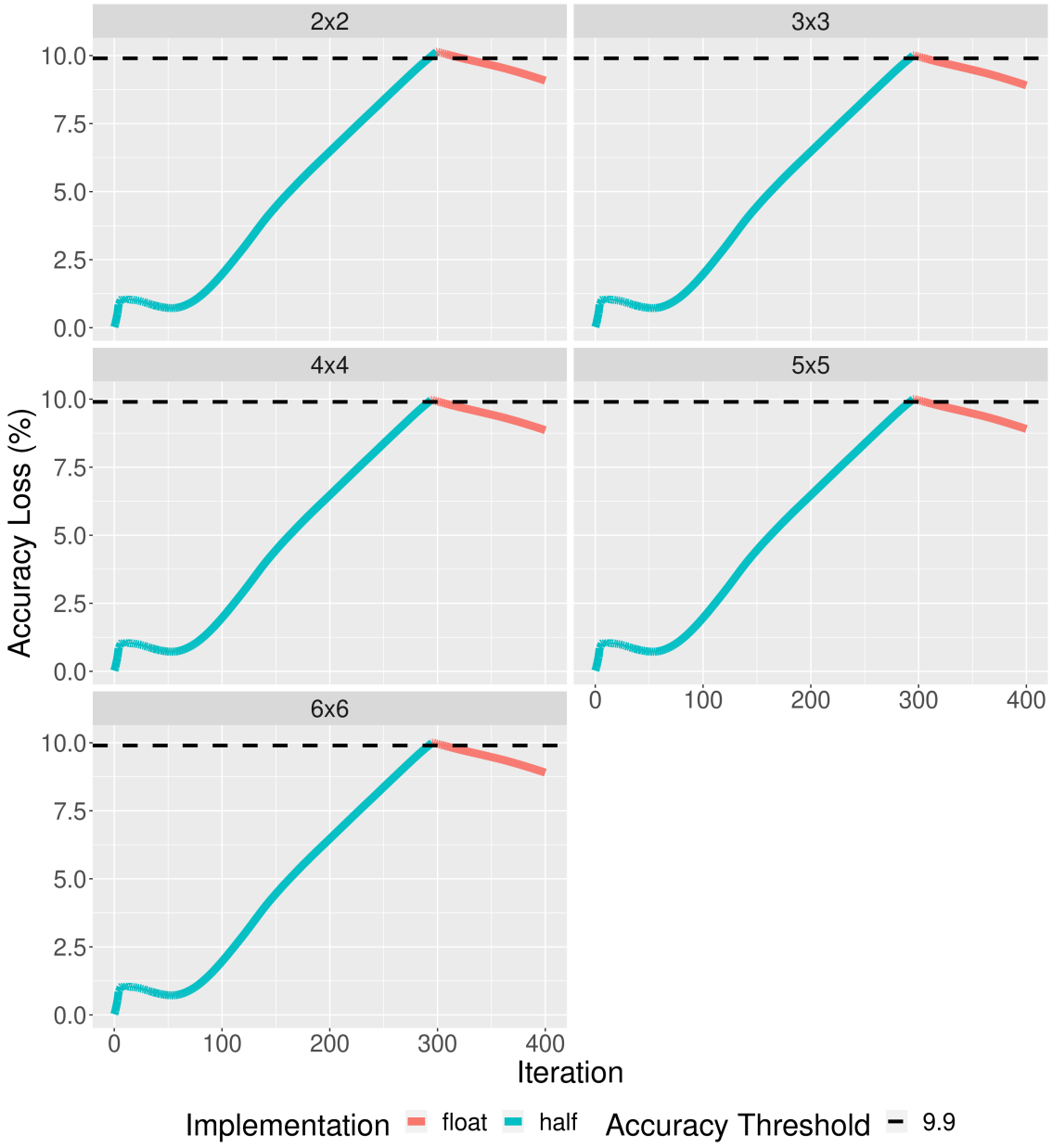
Source: The Authors

Figure A.5 – LBM3D accuracy loss profile using a problem size of 128, 400 iterations, and an accuracy loss threshold of 6.6%.



Source: The Authors

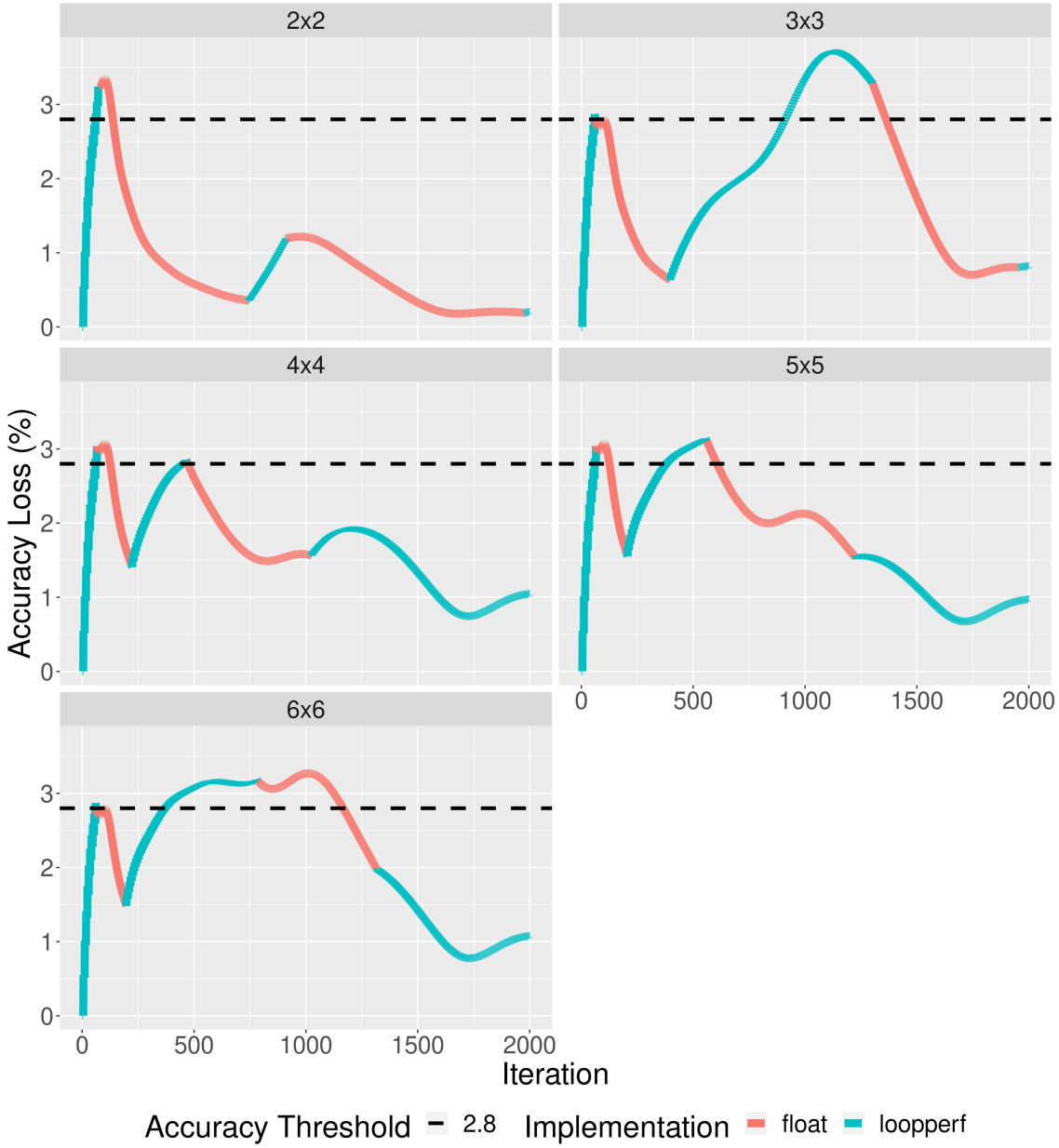
Figure A.6 – LBM3D accuracy loss profile using a problem size of 128, 400 iterations, and an accuracy loss threshold of 9.9%.



Source: The Authors

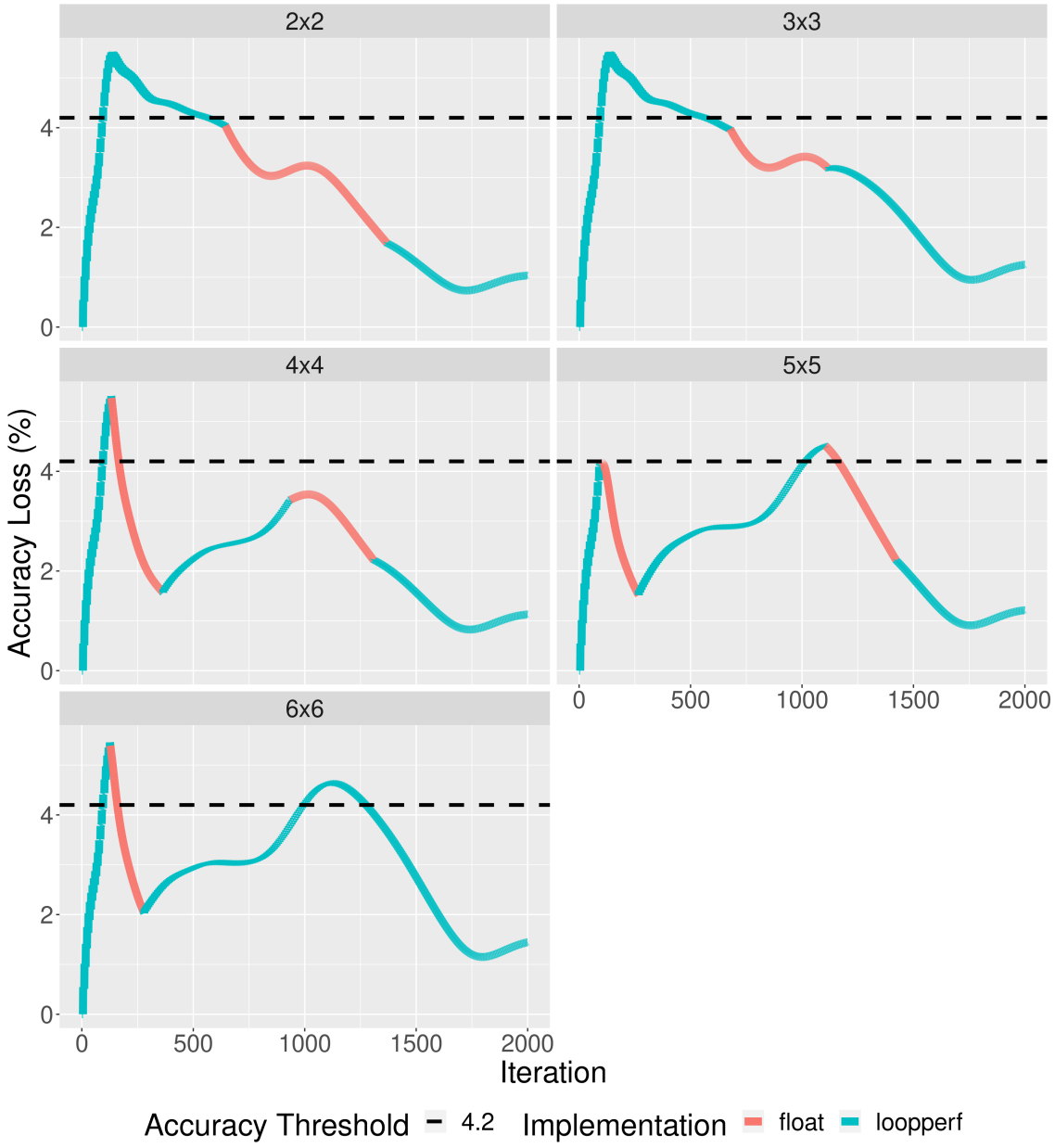
A.1.2 Euler3D

Figure A.7 – Euler3D accuracy loss profile using a problem size of 97152, 2000 iterations, and an accuracy loss threshold of 2.8%.



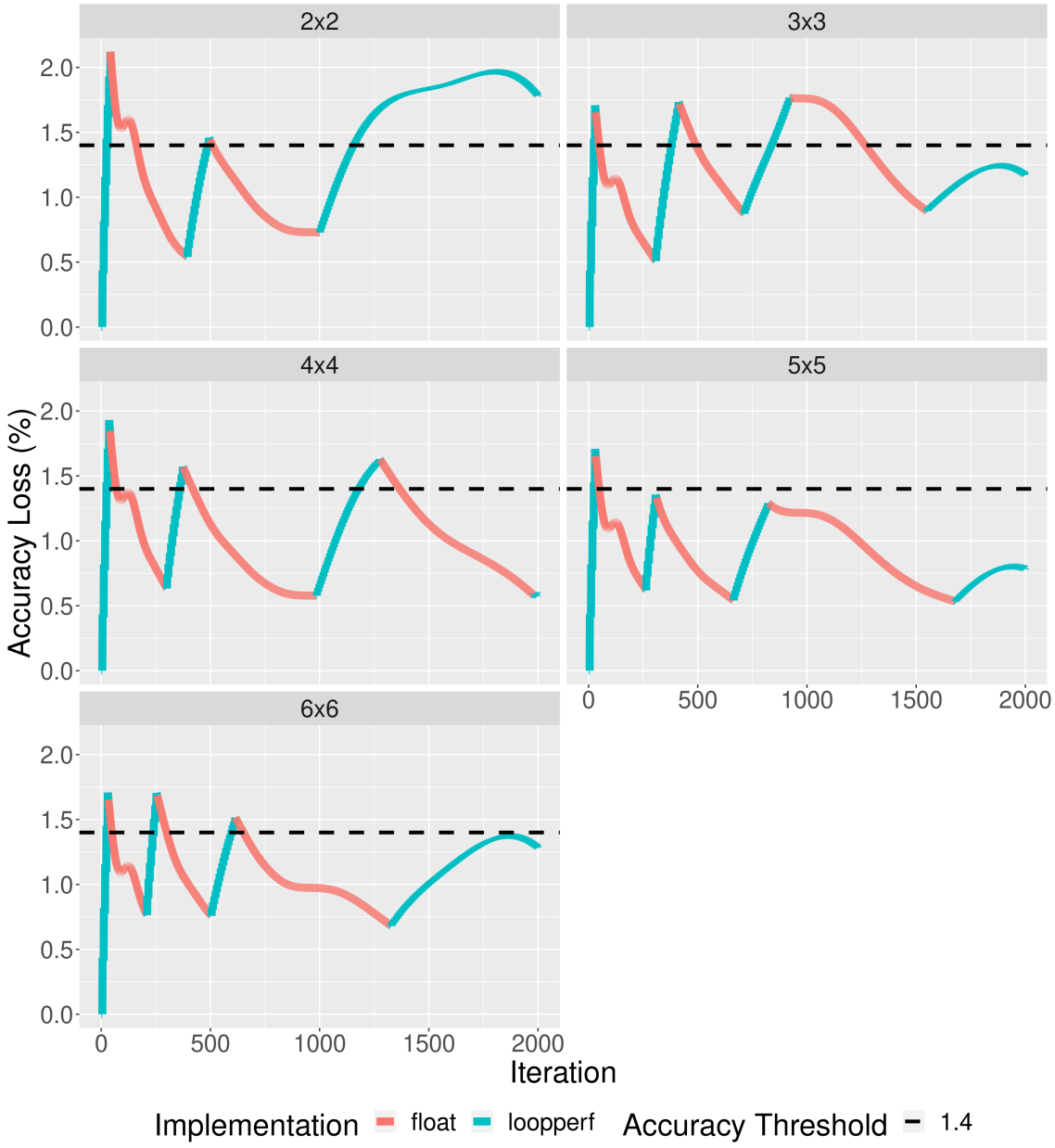
Source: The Authors

Figure A.8 – Euler3D accuracy loss profile using a problem size of 97152, 2000 iterations, and an accuracy loss threshold of 4.2%.



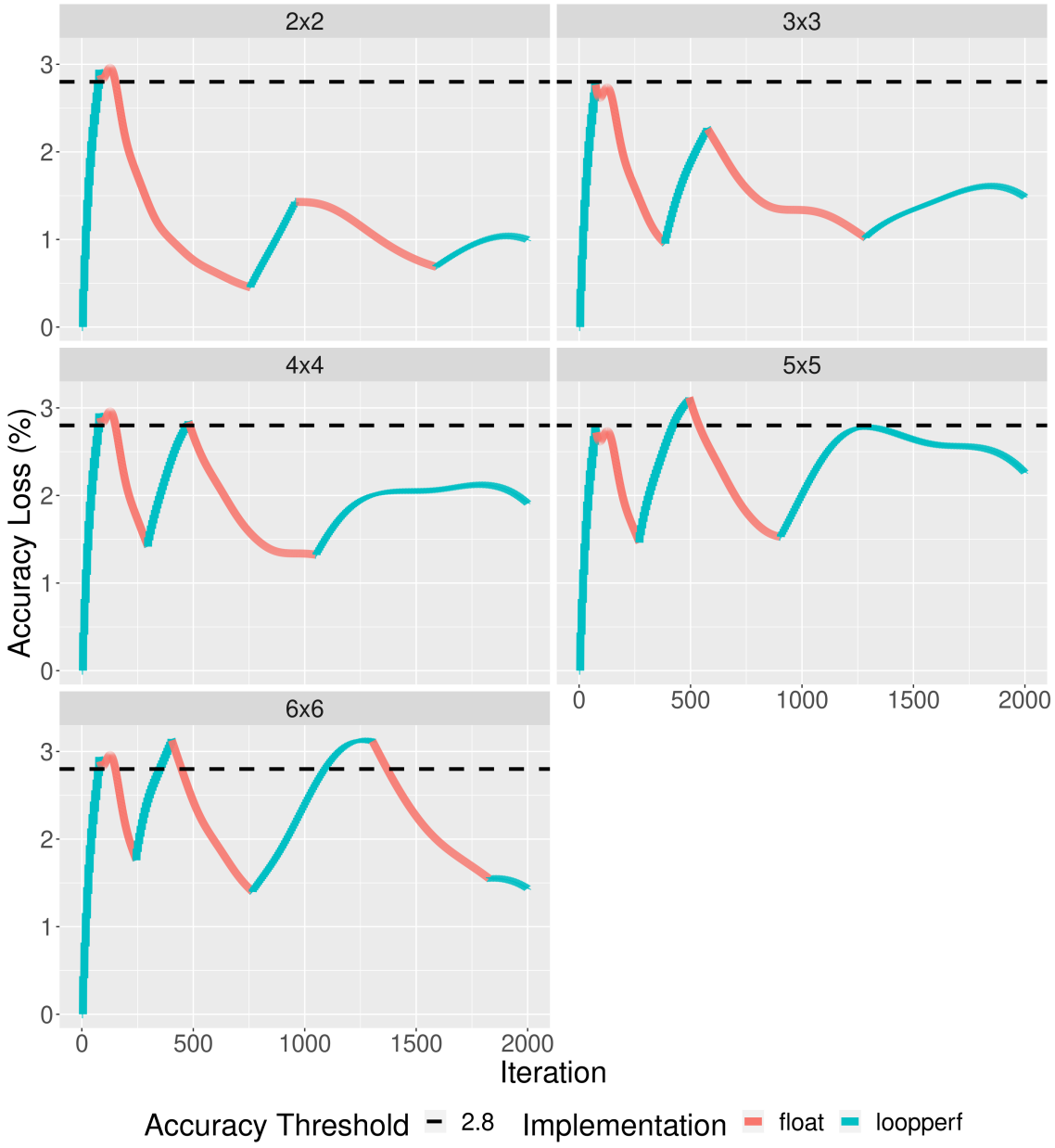
Source: The Authors

Figure A.9 – Euler3D accuracy loss profile using a problem size of 193536, 2000 iterations, and an accuracy loss threshold of 1.4%.



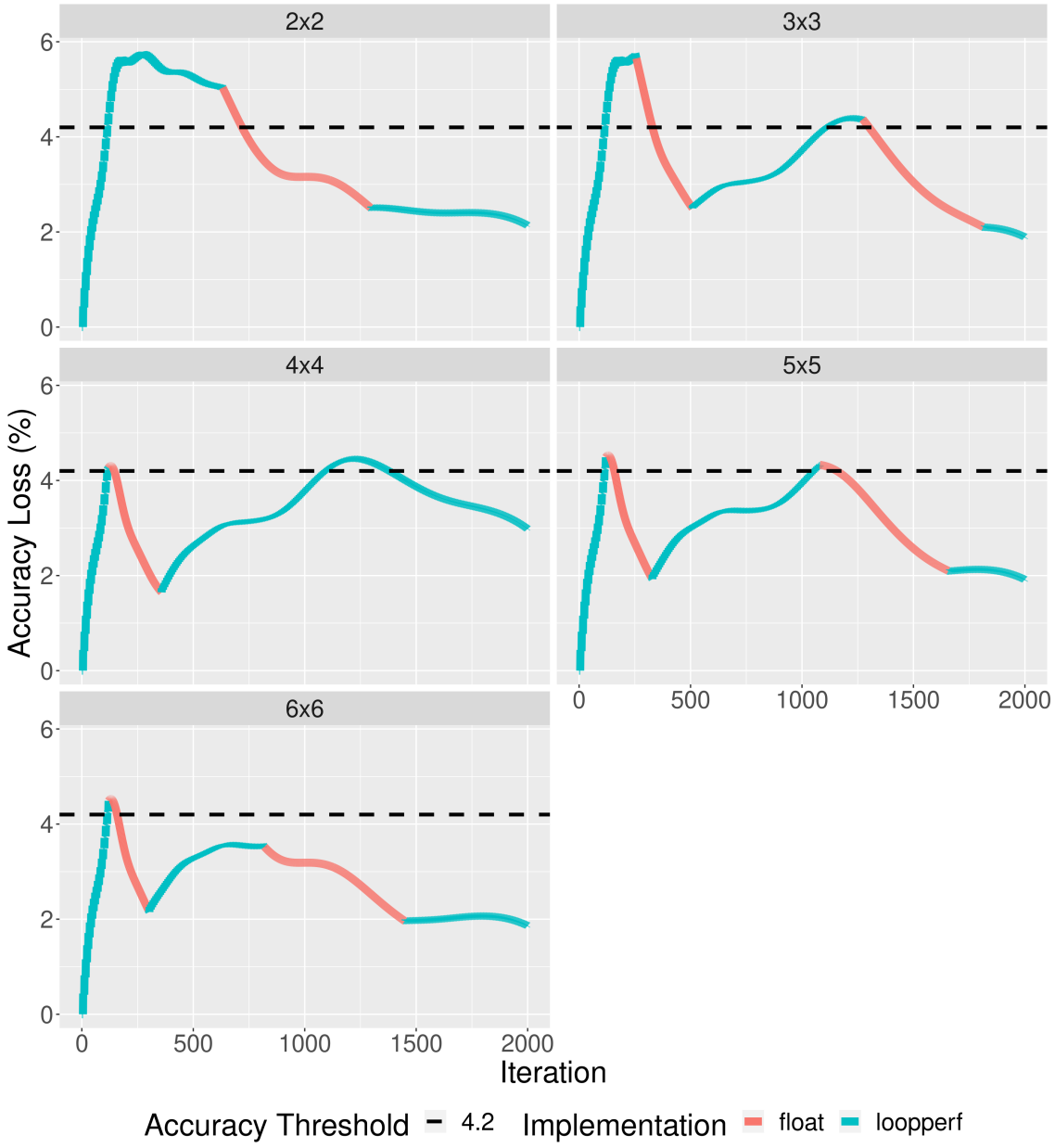
Source: The Authors

Figure A.10 – Euler3D accuracy loss profile using a problem size of 193536, 2000 iterations, and an accuracy loss threshold of 2.8%.



Source: The Authors

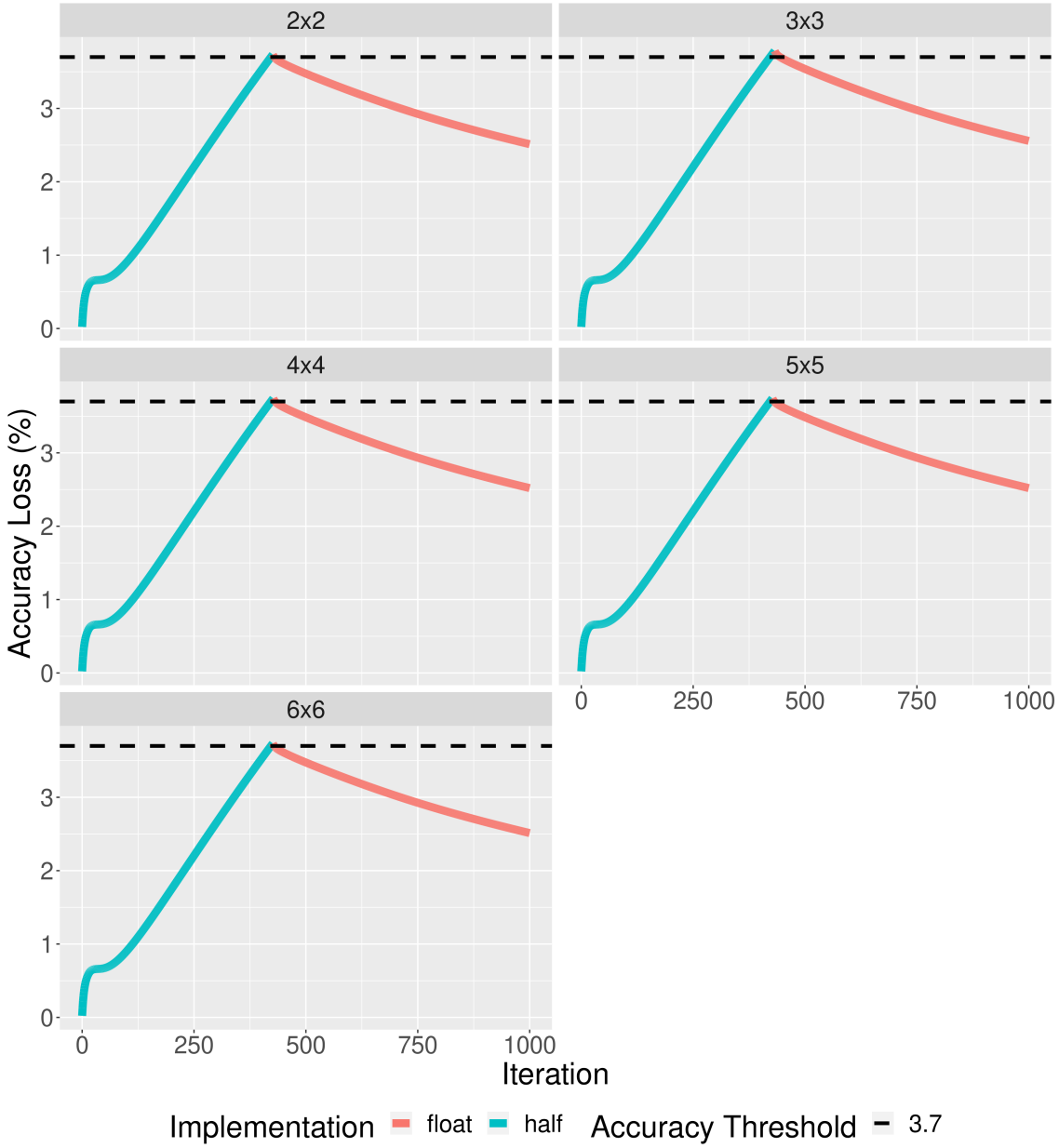
Figure A.11 – Euler3D accuracy loss profile using a problem size of 193536, 2000 iterations, and an accuracy loss threshold of 4.2%.



Source: The Authors

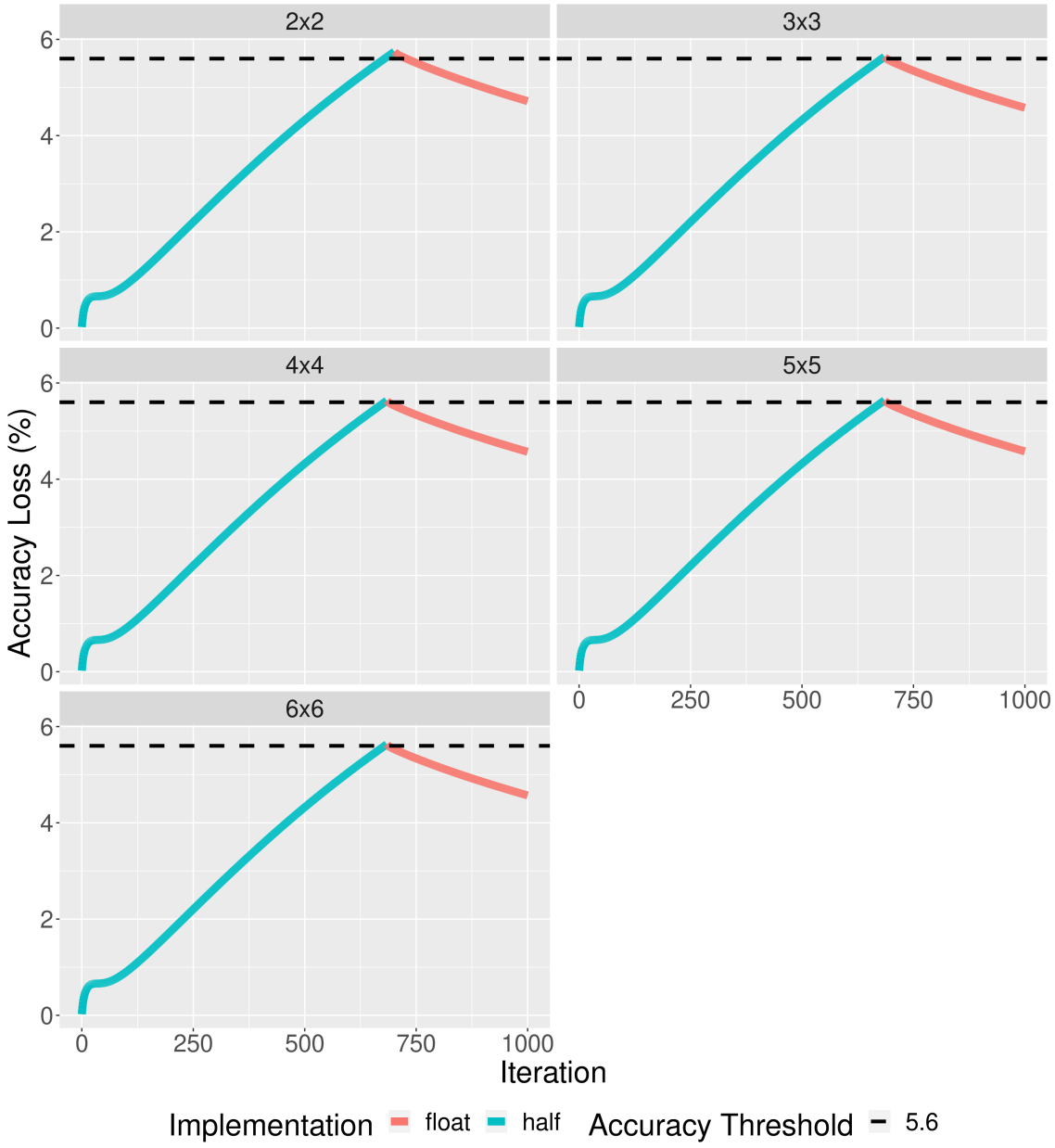
A.1.3 HotSpot3D

Figure A.12 – HotSpot3D accuracy loss profile using a problem size of 512, 1000 iterations, and an accuracy loss threshold of 3.7%.



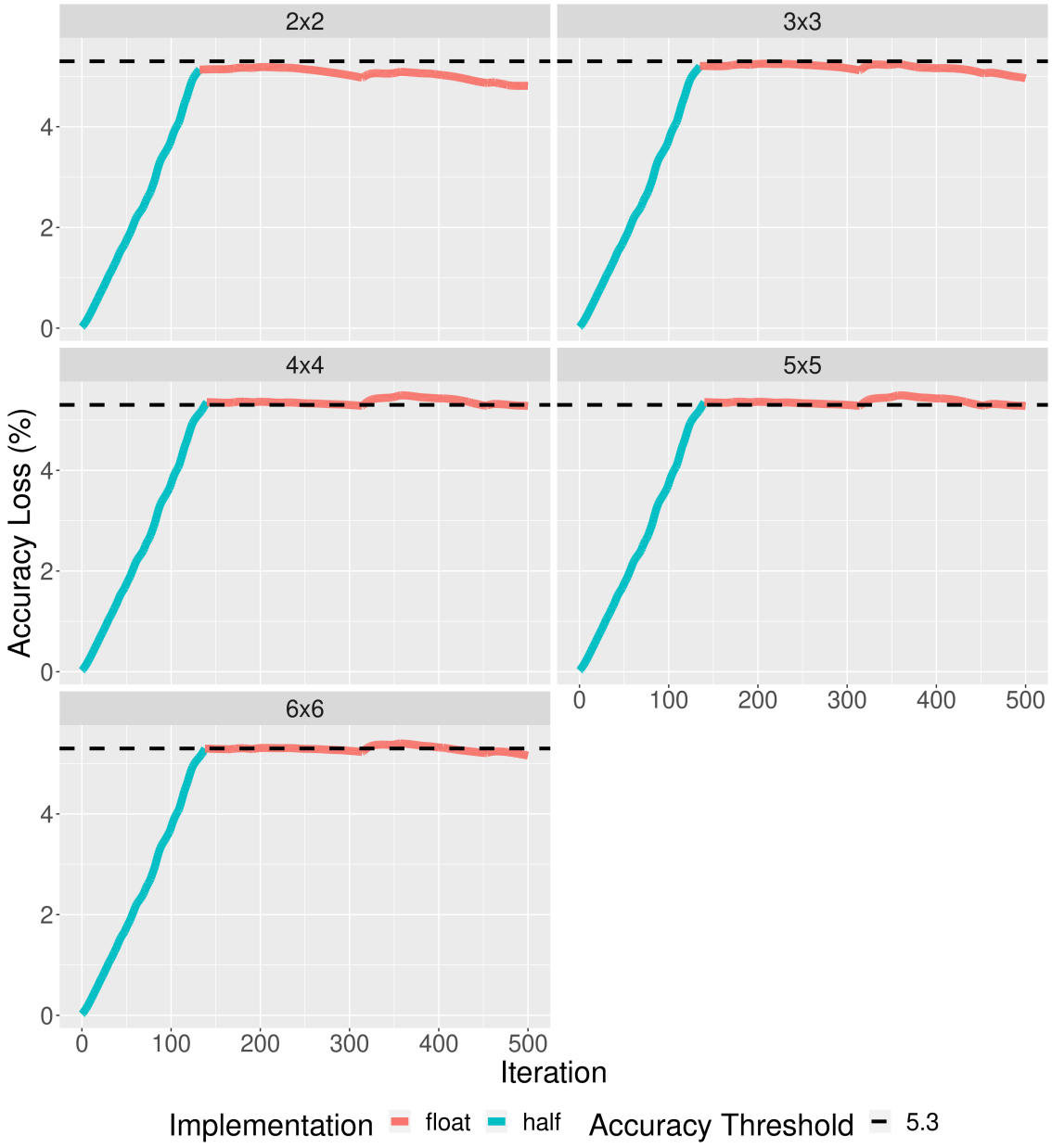
Source: The Authors

Figure A.13 – HotSpot3D accuracy loss profile using a problem size of 512, 1000 iterations, and an accuracy loss threshold of 5.6%.



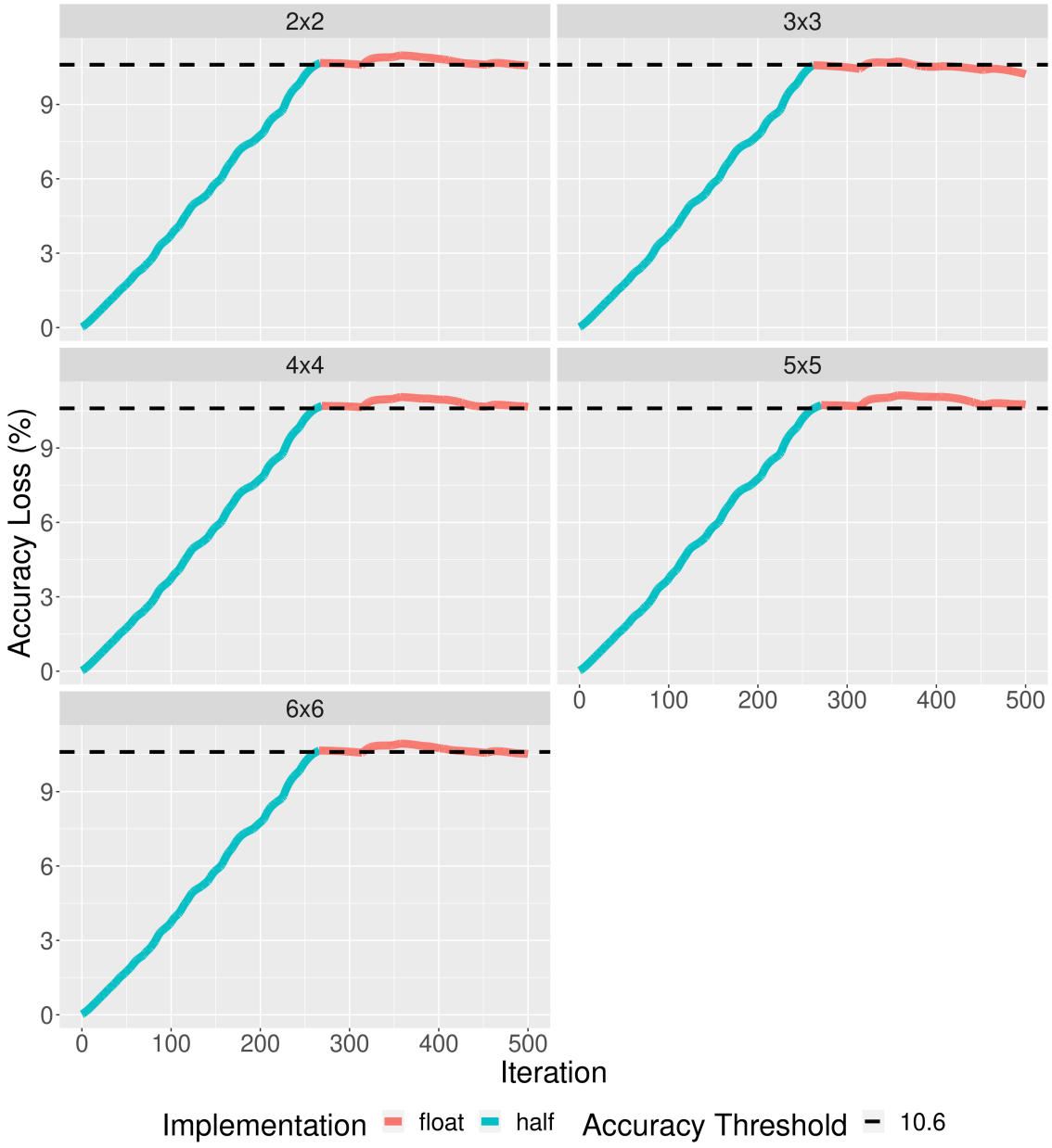
Source: The Authors

Figure A.14 – HotSpot3D accuracy loss profile using a problem size of 1024, 500 iterations, and an accuracy loss threshold of 5.3%.



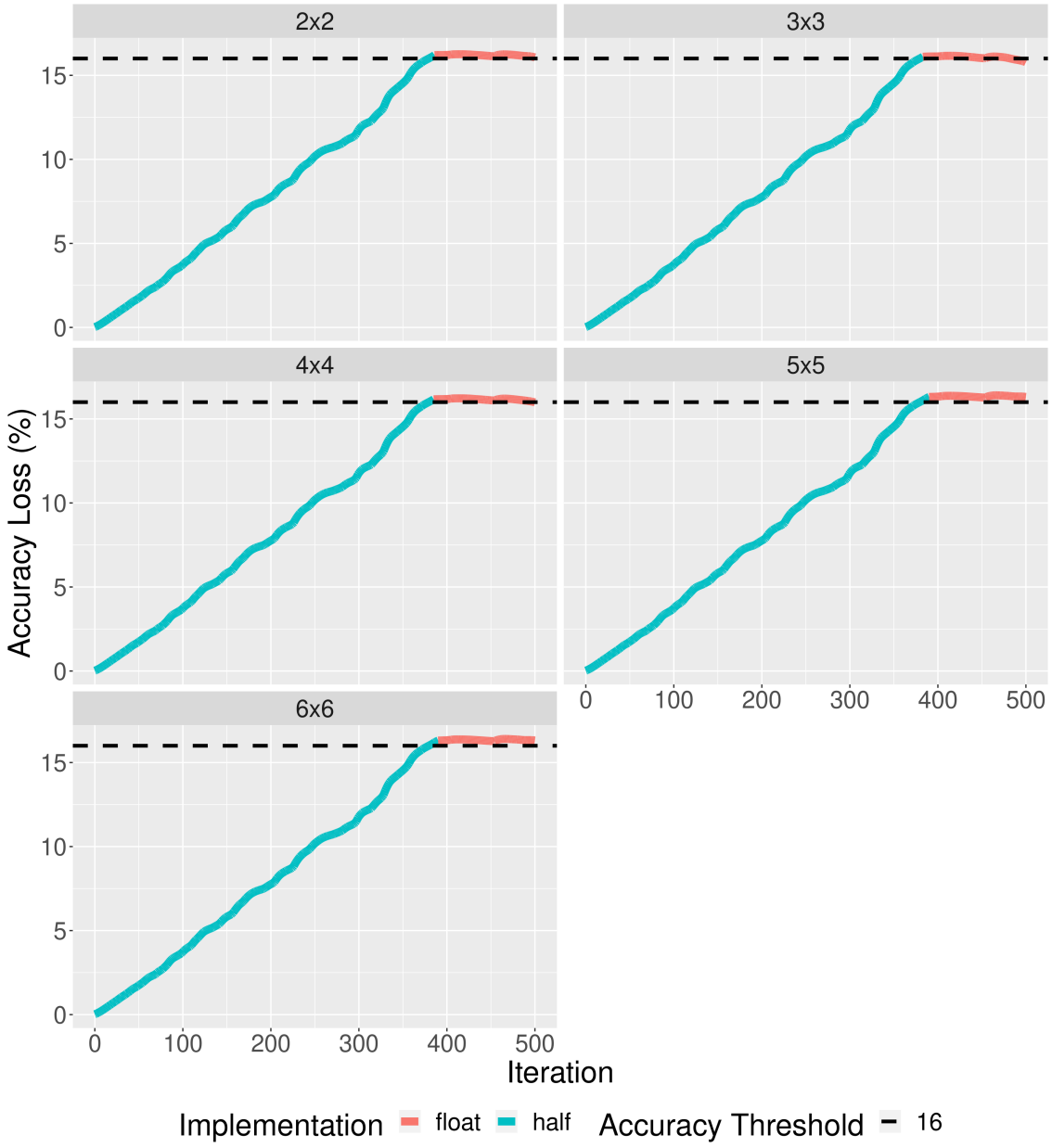
Source: The Authors

Figure A.15 – HotSpot3D accuracy loss profile using a problem size of 1024, 500 iterations, and an accuracy loss threshold of 10.6%.



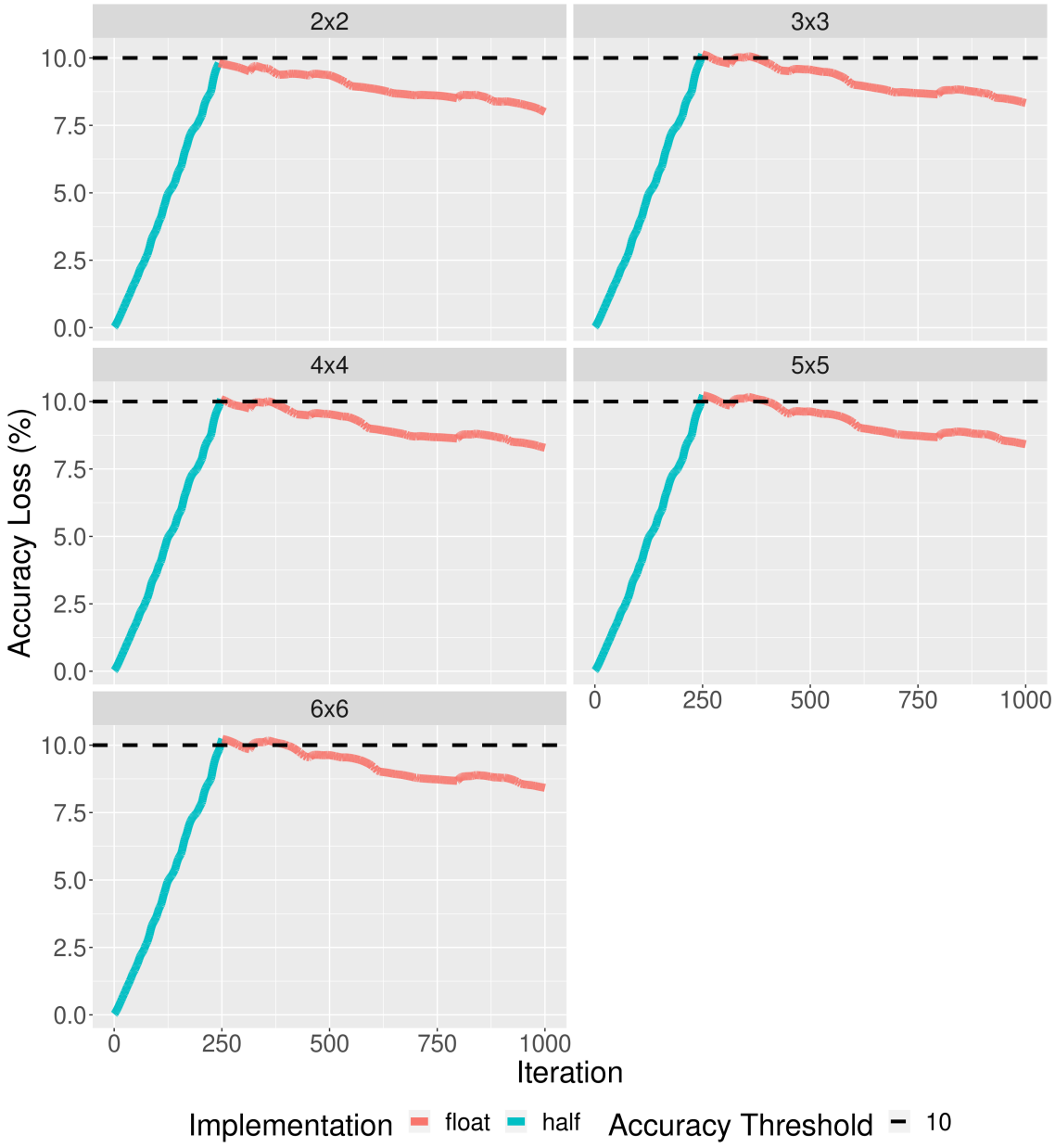
Source: The Authors

Figure A.16 – HotSpot3D accuracy loss profile using a problem size of 1024, 500 iterations, and an accuracy loss threshold of 16%.



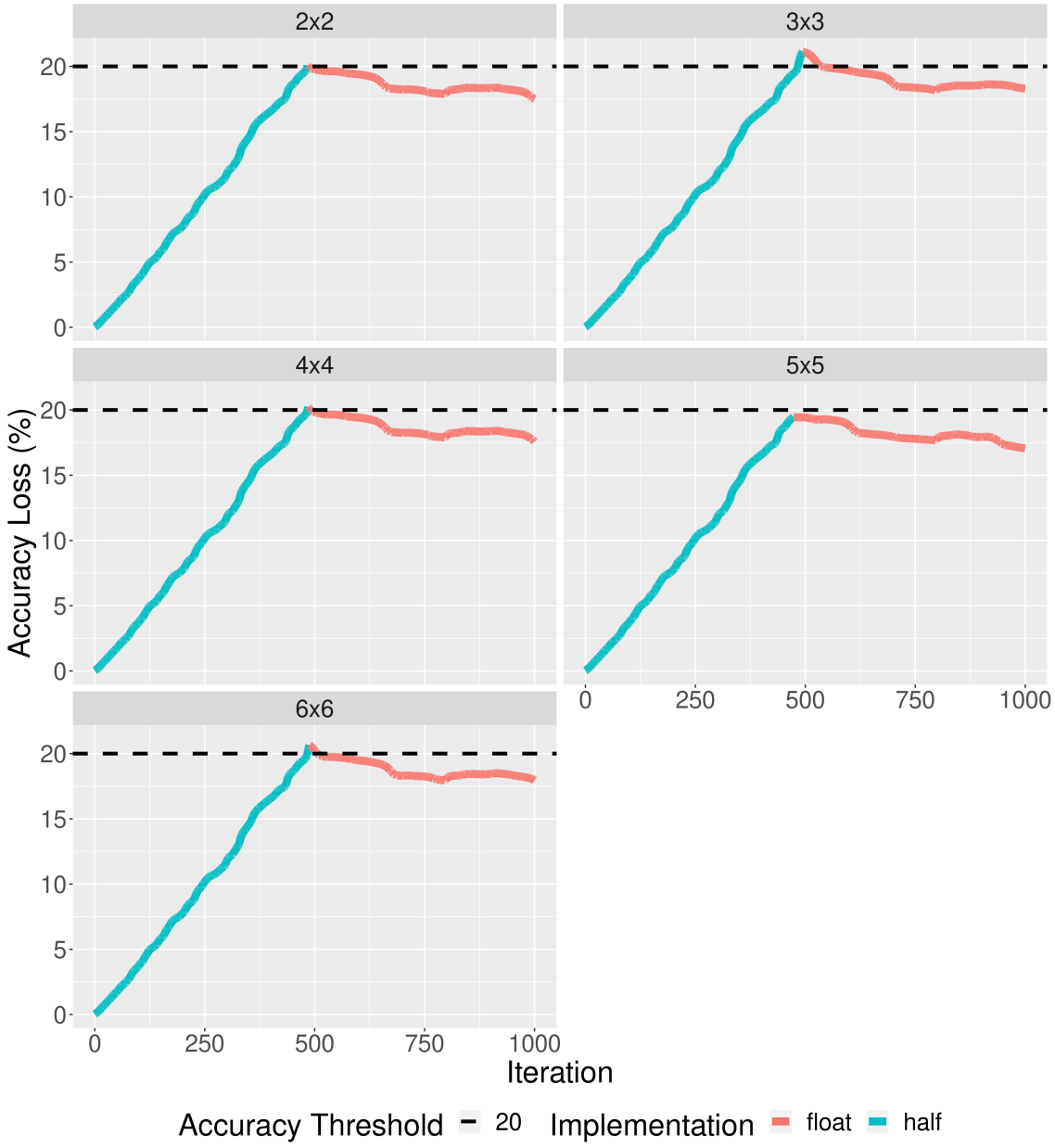
Source: The Authors

Figure A.17 – HotSpot3D accuracy loss profile using a problem size of 1024, 1000 iterations, and an accuracy loss threshold of 10%.



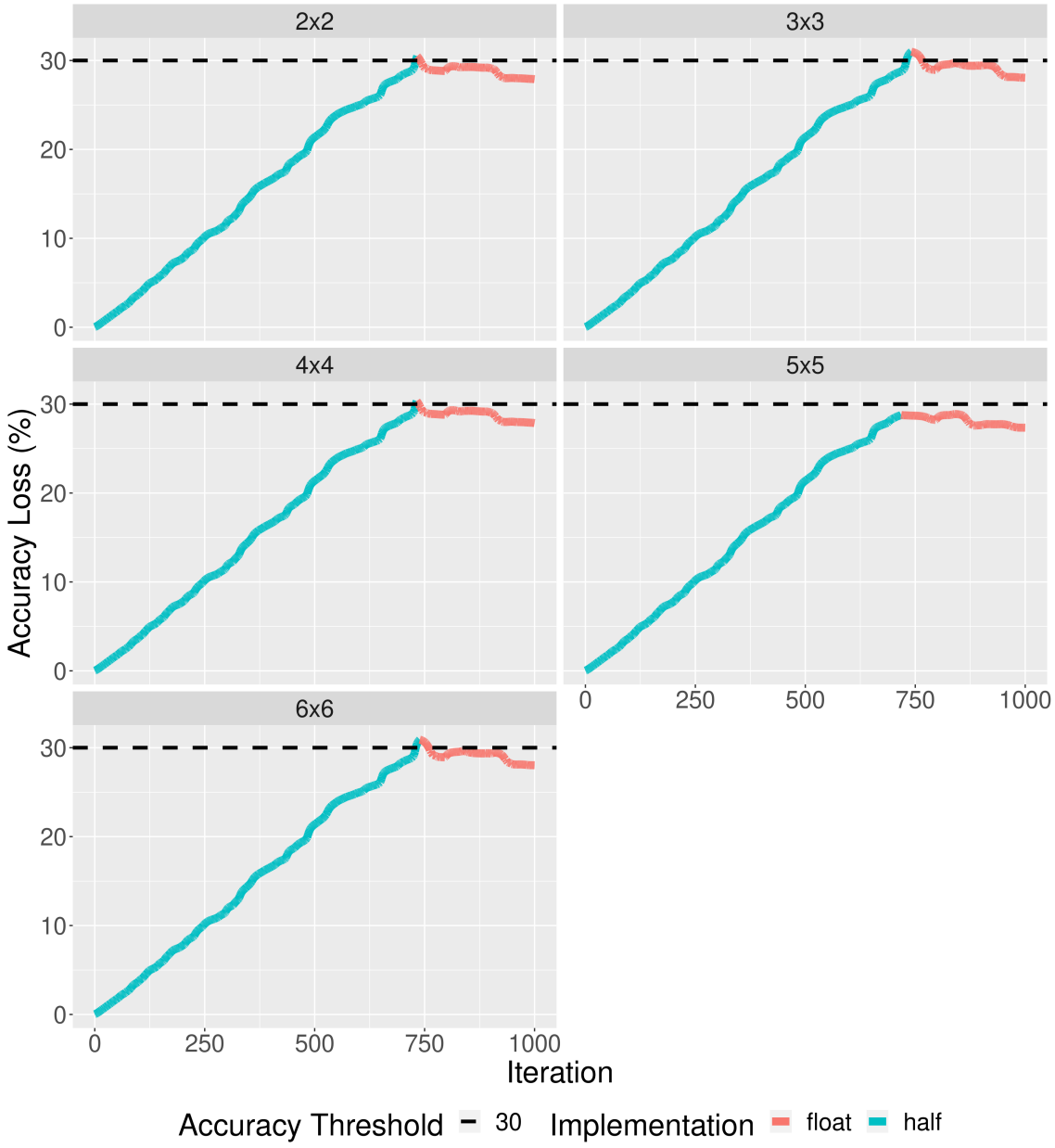
Source: The Authors

Figure A.18 – HotSpot3D accuracy loss profile using a problem size of 1024, 1000 iterations, and an accuracy loss threshold of 20%.



Source: The Authors

Figure A.19 – HotSpot3D accuracy loss profile using a problem size of 1024, 1000 iterations, and an accuracy loss threshold of 30%.



Source: The Authors

A.2 Performance Statistics

A.2.1 LBM3D

Table A.1 – LBM3D with a 3D problem size of 64 and 200 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	103.84	0.52	3.38	57.16	5.80
	6.1	61.28	0.05	2.19	48.98	2.91
2x2	1.5	80.65	0.14	1.81	62.66	5.02
	3.1	65.97	0.18	1.71	57.53	3.77
	4.6	65.14	0.03	0.75	52.60	3.30
3x3	1.5	76.31	0.10	1.33	60.04	4.51
	3.1	64.26	0.08	1.15	59.89	3.81
	4.6	65.00	0.03	1.53	51.88	3.33
4x4	1.5	69.43	0.09	1.46	57.76	4.01
	3.1	61.27	0.34	4.04	60.14	3.67
	4.6	67.71	0.06	1.36	50.52	3.37
5x5	1.5	79.98	0.00	1.17	51.00	4.05
	3.1	65.55	0.14	2.17	56.36	3.68
	4.6	60.18	0.07	1.45	51.45	3.10
6x6	1.5	75.09	0.07	2.07	53.53	3.98
	3.1	63.68	0.04	0.71	63.67	4.05
	4.6	60.89	0.07	0.83	53.84	3.22

Table A.2 – LBM3D with a 3D problem size of 64 and 400 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	165.59	1.39	6.86	89.84	14.88
	12	98.81	0.09	2.93	72.54	7.17
2x2	3	142.44	0.90	6.77	83.51	11.90
	6	132.44	0.41	4.80	83.60	11.07
	9	115.96	0.14	2.88	74.93	8.69
3x3	3	146.29	0.63	5.07	83.22	12.18
	6	131.32	0.30	5.45	81.44	10.70
	9	116.18	0.17	4.00	74.11	8.61
4x4	3	144.72	0.96	7.01	86.50	12.52
	6	132.36	0.30	3.08	79.37	10.51
	9	115.85	0.06	3.63	74.91	8.68
5x5	3	144.67	0.21	2.64	83.53	12.09
	6	131.87	0.28	6.20	84.25	11.11
	9	116.00	0.26	4.94	71.48	8.29
6x6	3	144.85	0.88	7.92	82.39	11.93
	6	132.40	0.68	5.89	79.32	10.50
	9	116.20	0.04	1.26	73.42	8.53

Table A.3 – LBM3D with a 3D problem size of 128 and 200 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	619.63	12.50	33.59	116.21	72.01
	6.1	404.88	6.00	28.51	103.35	41.58
2x2	1.5	529.06	9.04	31.11	109.19	57.77
	3.1	478.41	7.46	30.20	110.50	52.86
	4.6	441.00	6.93	29.98	109.57	48.32
3x3	1.5	519.19	8.99	32.12	111.05	57.65
	3.1	479.43	7.75	30.95	108.88	52.20
	4.6	439.39	6.75	29.30	109.09	47.93
4x4	1.5	518.04	8.64	30.97	111.66	57.84
	3.1	477.32	7.22	29.75	108.94	52.00
	4.6	444.32	6.80	29.49	109.44	48.63
5x5	1.5	515.82	8.59	31.06	111.17	57.35
	3.1	477.31	7.53	30.97	107.93	51.52
	4.6	444.27	6.66	29.10	108.81	48.34
6x6	1.5	515.87	8.48	30.78	111.11	57.32
	3.1	477.29	7.57	30.30	109.37	52.20
	4.6	444.31	6.70	29.57	108.14	48.05

Table A.4 – LBM3D with a 3D problem size of 128 and 400 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	1235.83	17.67	43.59	124.65	154.05
	13.2	787.84	9.74	39.61	116.02	91.38
2x2	3.3	1066.30	15.45	42.28	120.50	128.49
	6.6	1003.81	13.57	41.68	120.46	120.92
	9.9	900.09	11.19	40.80	118.27	106.45
3x3	3.3	1093.88	15.48	42.53	121.22	132.60
	6.6	1001.45	13.64	41.78	119.88	120.05
	9.9	904.59	11.30	40.82	118.50	107.19
4x4	3.3	1087.30	15.30	42.41	121.16	131.74
	6.6	1003.55	13.68	41.90	120.07	120.49
	9.9	905.61	11.38	41.04	118.86	107.64
5x5	3.3	1059.31	15.14	42.19	120.77	127.94
	6.6	1002.61	13.60	41.77	119.68	120.00
	9.9	904.48	11.30	40.91	118.52	107.20
6x6	3.3	1082.79	15.25	42.43	122.00	132.10
	6.6	1002.44	13.63	41.74	120.78	121.08
	9.9	904.59	11.39	41.12	119.56	108.16

A.2.2 Euler3D

Table A.5 – Euler3D with a 3D problem size of 97152 and 1000 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	118.40	0.00	3.06	45.78	5.41
	5.6	92.41	0.00	1.59	46.16	4.21
2x2	1.4	94.08	0.00	1.90	45.88	4.28
	2.8	108.70	0.00	1.75	46.60	4.98
	4.2	101.10	0.00	2.73	47.93	4.75
3x3	1.4	99.50	0.00	2.09	49.90	4.87
	2.8	101.23	0.00	2.55	49.52	4.94
	4.2	104.10	0.00	2.61	44.21	4.58
4x4	1.4	97.73	0.00	1.32	48.41	4.73
	2.8	92.96	0.00	2.95	46.72	4.26
	4.2	107.80	0.00	2.40	44.56	4.75
5x5	1.4	96.96	0.00	1.04	49.33	4.74
	2.8	93.20	0.00	1.39	44.57	4.09
	4.2	99.60	0.00	3.54	42.93	4.28
6x6	1.4	91.14	0.00	1.40	50.65	4.58
	2.8	97.31	0.00	1.62	46.91	4.53
	4.2	102.89	0.00	2.24	44.52	4.55

Table A.6 – Euler3D with a 3D problem size of 97152 and 2000 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum(J)
-	0	156.42	0.00	6.36	77.61	11.85
	5.6	140.82	0.00	3.28	68.07	9.20
2x2	1.4	157.67	0.00	4.83	72.25	10.99
	2.8	149.48	0.00	4.23	74.97	11.03
	4.2	129.41	0.00	3.06	79.07	10.23
3x3	1.4	138.61	0.00	3.05	83.14	11.52
	2.8	140.37	0.00	4.37	73.69	10.16
	4.2	134.47	0.00	5.53	71.17	9.40
4x4	1.4	147.46	0.00	3.02	75.61	10.75
	2.8	129.16	0.00	4.06	76.78	9.92
	4.2	144.77	0.00	6.49	72.35	10.12
5x5	1.4	142.55	0.00	2.28	76.05	10.65
	2.8	130.30	0.00	3.50	78.73	10.26
	4.2	147.87	0.00	5.75	69.32	9.88
6x6	1.4	132.65	0.00	4.77	80.51	10.68
	2.8	142.86	0.00	3.80	72.89	10.10
	4.2	122.43	0.00	3.57	71.65	8.77

Table A.7 – Euler3D with a 3D problem size of 193536 and 1000 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	200.46	0.30	8.55	54.19	10.86
	5.6	168.07	0.20	5.36	52.33	8.79
2x2	1.4	196.95	0.29	8.10	53.97	10.63
	2.8	190.69	0.32	8.91	53.72	10.25
	4.2	184.64	0.25	6.18	52.89	9.77
3x3	1.4	188.54	0.32	8.09	53.74	10.13
	2.8	197.10	0.24	6.37	53.15	10.49
	4.2	179.86	0.36	8.49	51.48	9.27
4x4	1.4	189.05	0.20	6.82	53.01	10.04
	2.8	185.90	0.15	4.38	52.75	9.81
	4.2	179.85	0.29	8.06	54.49	9.80
5x5	1.4	201.88	0.18	6.62	53.95	10.88
	2.8	184.37	0.23	7.11	54.28	10.01
	4.2	177.28	0.17	5.62	52.34	9.28
6x6	1.4	196.97	0.24	7.33	53.33	10.50
	2.8	191.58	0.25	7.46	54.37	10.40
	4.2	175.92	0.22	6.83	52.96	9.32

Table A.8 – Euler3D with a 3D problem size of 193536 and 2000 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	327.20	0.72	19.56	69.53	22.66
	5.6	271.24	0.57	14.76	69.27	18.35
2x2	1.4	303.99	0.61	18.23	66.82	20.23
	2.8	318.84	0.65	18.61	66.77	21.21
	4.2	300.72	0.63	17.30	65.93	19.75
3x3	1.4	316.69	0.65	18.33	65.70	20.75
	2.8	306.42	0.63	18.02	65.69	20.07
	4.2	310.40	0.65	18.22	63.98	19.79
4x4	1.4	340.26	0.77	20.06	62.80	21.36
	2.8	295.84	0.62	17.75	67.81	20.02
	4.2	286.28	0.64	17.29	66.29	18.90
5x5	1.4	315.84	0.67	18.31	66.71	21.01
	2.8	300.51	0.66	17.57	64.49	19.33
	4.2	310.04	0.66	18.08	64.36	19.89
6x6	1.4	319.53	0.72	19.11	65.31	20.78
	2.8	315.66	0.66	17.84	64.29	20.28
	4.2	298.42	0.69	17.58	68.53	20.34

A.2.3 HotSpot3D

Table A.9 – HotSpot3D with a 3D problem size of 512 and 500 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	12.05	0.00	0.80	34.41	0.41
	4.4	7.33	0.00	0.73	33.98	0.25
2x2	1.1	10.18	0.00	0.97	34.05	0.35
	2.2	9.69	0.00	0.73	34.73	0.34
	3.3	8.47	0.00	1.00	37.01	0.31
3x3	1.1	10.84	0.00	1.18	33.89	0.37
	2.2	9.68	0.00	1.67	34.21	0.33
	3.3	8.49	0.00	0.83	34.12	0.29
4x4	1.1	10.88	0.00	0.57	34.95	0.38
	2.2	9.67	0.00	0.30	34.04	0.33
	3.3	8.50	0.00	0.60	33.96	0.29
5x5	1.1	10.87	0.00	0.65	33.83	0.37
	2.2	9.68	0.00	1.22	34.94	0.34
	3.3	8.50	0.00	1.43	33.97	0.29
6x6	1.1	10.53	0.00	1.20	33.96	0.36
	2.2	9.70	0.00	1.20	34.56	0.34
	3.3	8.52	0.00	1.08	34.91	0.30

Table A.10 – HotSpot3D with a 3D problem size of 512 and 1000 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	24.52	0.00	1.13	37.53	0.92
	7.5	15.33	0.00	0.89	35.40	0.54
2x2	1.8	21.44	0.00	0.70	35.70	0.77
	3.7	20.67	0.00	1.17	35.53	0.73
	5.6	18.10	0.00	0.51	34.86	0.63
3x3	1.8	22.66	0.00	0.56	36.21	0.82
	3.7	20.63	0.00	1.07	35.54	0.73
	5.6	18.22	0.00	1.26	35.35	0.64
4x4	1.8	22.68	0.03	1.37	34.73	0.79
	3.7	20.66	0.00	1.24	35.63	0.74
	5.6	18.27	0.00	0.85	35.40	0.65
5x5	1.8	22.62	0.00	1.26	36.90	0.84
	3.7	20.67	0.00	1.25	35.47	0.73
	5.6	18.24	0.00	1.17	35.35	0.64
6x6	1.8	22.68	0.00	0.79	37.34	0.85
	3.7	20.65	0.00	1.70	36.32	0.75
	5.6	18.24	0.00	1.10	36.01	0.66

Table A.11 – HotSpot3D with a 3D problem size of 1024 and 500 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	48.98	0.44	1.84	47.27	2.32
	21.3	24.69	0.12	1.60	38.16	0.94
2x2	5.3	42.62	0.56	2.33	44.87	1.91
	10.6	36.09	0.20	2.63	40.80	1.47
	16	30.29	0.56	1.86	37.97	1.15
3x3	5.3	42.47	0.20	1.97	39.23	1.67
	10.6	36.30	0.05	1.49	36.25	1.32
	16	30.43	0.30	1.80	37.79	1.15
4x4	5.3	42.23	0.12	1.37	44.70	1.89
	10.6	35.96	0.00	1.19	37.57	1.35
	16	30.33	0.26	1.16	37.37	1.13
5x5	5.3	42.22	0.00	1.13	43.91	1.85
	10.6	35.88	0.33	1.54	38.98	1.40
	16	30.09	0.34	1.86	38.37	1.16
6x6	5.3	42.33	0.36	1.57	38.60	1.63
	10.6	36.11	0.00	1.18	41.95	1.51
	16	30.10	0.48	2.30	35.41	1.07

Table A.12 – HotSpot3D with a 3D problem size of 1024 and 1000 iterations.

Sections	Threshold (%)	Time (ms)	Memory Util (%)	GPU Util (%)	Power Draw (W)	Energy Consum (J)
-	0	100.65	1.87	3.36	56.17	5.65
	40	50.53	0.43	1.68	42.46	2.15
2x2	10	88.76	0.73	1.95	53.18	4.72
	20	76.61	0.57	1.69	52.01	3.98
	30	64.06	0.61	1.62	40.59	2.60
3x3	10	88.42	1.28	2.25	51.02	4.51
	20	76.10	0.53	1.91	53.06	4.04
	30	63.64	0.50	2.16	49.58	3.16
4x4	10	88.49	1.40	2.79	51.31	4.54
	20	76.56	0.82	1.82	52.22	4.00
	30	64.09	0.49	2.25	48.18	3.09
5x5	10	88.35	1.34	2.96	52.36	4.63
	20	77.17	1.10	2.38	43.26	3.34
	30	64.84	0.50	1.39	52.20	3.38
6x6	10	88.37	0.82	2.61	45.77	4.04
	20	76.43	0.86	2.39	51.75	3.96
	30	63.81	0.21	1.85	45.92	2.93

APPENDIX B — RESUMO EXPANDIDO

B.1 Melhorando o Desempenho de Aplicações Iterativas por meio da Execução Intercalada de Kernels CUDA Aproximados

Não é segredo que a demanda por poder computacional está aumentando rapidamente. À medida que a humanidade avança em direção à digitalização e sistemas inteligentes interconectados, a necessidade de armazenamento, processamento e análise de dados continua a crescer significativamente. Apesar de melhorias significativas no poder computacional e capacidade de armazenamento geração após geração, a demanda por capacidade de armazenamento e computação ainda excede em muito os recursos disponíveis, incluindo recursos de orçamento (MITTAL, 2016). Áreas de aplicação existentes e novas exigirão computação com eficiência energética ordens de magnitude mais alta do que o estado da arte atual (SHALF, 2020). Portanto, alcançar benchmarks de poder computacional mais altos economicamente exigirá hardware, algoritmos e métodos mais eficientes (THOMPSON et al., 2020).

Nos últimos anos, a computação aproximada se tornou uma abordagem popular e promissora para melhorar a eficiência e o desempenho dos sistemas de computador (XU; MYTKOWICZ; KIM, 2015). A ideia básica é trocar a precisão computacional por melhor desempenho do sistema e redução do consumo de energia (MITTAL, 2016). Projetando hardware e software para tolerar erros e permitir erros menores ou variações na saída de cálculos, a computação aproximada pode reduzir os requisitos de energia computacional. Essa abordagem pode alcançar tempos de computação mais rápidos e menor consumo de energia do que métodos tradicionais, resultando em economia de custos.

Precisão reduzida e precisão mista estão entre as técnicas de computação aproximada mais populares. Essas técnicas envolvem o uso de menos bits para representar valores numéricos de ponto flutuante. Por exemplo, em vez de usar 64 bits para representar um número, podemos usar apenas 32 bits. Ao fazer isso, podemos reduzir a quantidade de memória necessária para armazenar um valor e a quantidade de dados que precisa ser transferida e processada. Isso pode levar a reduções no tempo de computação e no consumo de energia (JIN et al., 2017).

Para evitar problemas de compatibilidade e imprecisões de cálculo, a IEEE padronizou a representação de números de ponto flutuante em binário em hardware e software

de computador. O padrão IEEE 754 (IEEE..., 1985) define um formato consistente e universalmente reconhecido para representar números de ponto flutuante. Isso permite cálculos precisos e confiáveis em diferentes sistemas e linguagens de programação. As revisões da IEEE 754-2008 (IEEE..., 2008) especificam vários formatos de ponto flutuante, incluindo os formatos de meia precisão, precisão única, precisão dupla e precisão quádrupla. Esses formatos usam 16, 32, 64 e 128 bits para representar números de ponto flutuante em binário.

A principal diferença entre os formatos de ponto flutuante é o número de bits usados para representar o número, o que influencia a precisão e o intervalo de números que podem ser representados. Por exemplo, o formato de precisão única usa 32 bits para representar um número de ponto flutuante. Tem uma precisão de cerca de sete dígitos decimais, enquanto o formato de precisão dupla usa 64 bits e tem uma precisão de cerca de 15 dígitos decimais. Quanto mais bits alocados para a mantissa, mais precisa pode ser a representação do número. No entanto, o valor decimal de 0,1 não pode ser representado com precisão em binário usando qualquer número finito de bits (GOLDBERG, 1991). Portanto, qualquer representação binária de 0,1 será uma aproximação. Por exemplo, no formato de precisão única, 0,1 é aproximado como 0,100000001490116119384765625, enquanto no formato de precisão dupla, é aproximado como 0,1000000000000000055511151231257827021181583404541015625.

Quando se trabalha com formatos de ponto flutuante de baixa precisão, como em técnicas de computação aproximada de precisão reduzida e mista, erros podem ocorrer devido à representação de números decimais em binário e à conversão entre diferentes formatos de ponto flutuante. A conversão de um número para um formato de precisão mais baixa pode resultar em arredondamento ou truncamento, levando à perda de alguns dígitos. Ao contrário, a conversão para um formato de precisão mais alta pode adicionar zeros adicionais, mas a precisão do número original permanece inalterada. Como resultado, a precisão da execução da aplicação é diretamente afetada pelo número de conversões entre diferentes formatos de ponto flutuante.

Outro fator que afeta a precisão dos cálculos de ponto flutuante é a diferença no erro introduzido por diferentes operações aritméticas. Por exemplo, subtrair dois números muito próximos em valor pode resultar em perda de precisão devido ao número limitado de dígitos significativos no formato de ponto flutuante. Multiplicar dois números com magnitudes muito diferentes também pode resultar em resultados imprecisos devido a overflow ou underflow. Em geral, adição e subtração são menos propensas a erros do

que multiplicação e divisão porque adição e subtração não amplificam erros na mesma extensão que multiplicação e divisão (GOLDBERG, 1991).

Portanto, controlar e limitar erros ao trabalhar com números de ponto flutuante e aproximações é crucial porque eles podem levar a resultados imprecisos e afetar significativamente a precisão geral da execução da aplicação. Consequentemente, é essencial considerar cuidadosamente os requisitos de precisão e precisão da aplicação específico e usar técnicas apropriadas para minimizar a introdução e acumulação de erros. Isso pode envolver o uso de formatos de precisão mais alta, operações aritméticas adequadas e técnicas de correção e compensação de erros. Fazê-lo torna possível melhorar a precisão e confiabilidade dos cálculos de ponto flutuante, o que é essencial para muitas aplicações em ciência, engenharia e outros domínios.

B.1.1 Motivação

Apesar de ser amplamente exploradas na literatura, as técnicas e ferramentas para ajustar a precisão das operações de ponto flutuante são principalmente focadas em aplicações não iterativas. Os principais domínios de aplicação são gráficos de computador, aprendizado de máquina, processamento de sinais, finanças e computação numérica (BAEK; CHILIMBI, 2010; RUBIO-GONZ et al., 2013; RUBIO-GONZALEZ et al., 2016; KHUDIA et al., 2015; CHERUBIN et al., 2020; ROY et al., 2014), robótica, compressão (KHUDIA et al., 2015), agrupamento e classificação, séries temporais, problemas de regressão (ZHANG et al., 2014). A seleção dessas aplicações é tipicamente baseada em sua representatividade da carga de trabalho do mundo real e sua notável resistência a erros de ponto flutuante (MITTAL, 2016).

Embora alguns trabalhos explorem aplicações científicas iterativas igualmente representativas, como os núcleos CG (Gradiente Conjugado), EP (Embarrassingly Parallel) e FP (Transformada de Fourier), além das pseudoaplicações SP (Penta-diagonal Escalar) e LU (Gauss-Seidel Inferior-Superior) da suíte de benchmarks paralelos NAS (RUBIO-GONZ et al., 2013; RUBIO-GONZALEZ et al., 2016; SAMPSON et al., 2011; GRAILLAT et al., 2019), jetEngine e turbine (CHIANG et al., 2017; MENON; LAM, 2019), e Método Lattice Boltzmann (LBM) (HO et al., 2017), as aplicações iterativas são significativamente mais sensíveis a erros de ponto flutuante, tornando muito mais difícil ajustar a precisão das operações de ponto flutuante.

As aplicações iterativas são algoritmos que repetem um conjunto específico de

instruções várias vezes até que uma condição específica seja atendida (BU et al., 2010). Essa técnica é usada para problemas que não podem ser resolvidos analiticamente ou quando a complexidade da solução de um problema torna impraticável calcular em um único passo (CARSON; STRAKOŠ, 2020). Nessa técnica, a saída de uma iteração se torna a entrada para a próxima. Embora isso torne as aplicações mais eficientes, também significa que qualquer aproximação feita em uma iteração é transportada para a próxima. Esses erros podem ser amplificados, levando a imprecisões no resultado final. Portanto, é essencial monitorar e validar cuidadosamente a saída das iterações para garantir a precisão do resultado final (ZHANG et al., 2014).

Garantir a qualidade da saída em aplicações iterativas pode ser desafiador devido à sua natureza repetitiva e ao uso típico de domínios de dados multidimensionais. Para controlar a qualidade de saída usando técnicas de aproximação, um método padrão é monitorar a perda de precisão durante a execução (MITTAL, 2016). No entanto, monitorar a perda de precisão durante a execução em aplicações iterativas pode ser impraticável devido ao volume de dados envolvido. Em aplicações de simulação científica, por exemplo, para calcular a perda de precisão em cada iteração em um domínio de dados 2D com 512 células no eixo x e no eixo y, 262144 valores devem ser comparados com os valores corretos. Usando um domínio de dados 3D com 512 células em cada lado, o número de células aumenta para 134217728. Como as aplicações científicas iterativas de simulação podem ter domínios de dados significativamente maiores, o custo computacional de verificar a perda de precisão durante a execução pode facilmente exceder o custo de execução da aplicação.

B.1.2 Objetivos

Nossa pesquisa concentra-se em desenvolver um método eficiente para acelerar a execução de aplicações usando técnicas de computação aproximada. Estamos interessados em aplicações iterativas, onde um conjunto dado de operações é aplicado repetidamente a um grande conjunto de dados. A execução de aplicações iterativas exige grande poder computacional devido ao número de operações e ao volume de dados nos quais as operações são realizadas (BU et al., 2010; CARSON; STRAKOŠ, 2020). No entanto, essas características tornam as GPUs ideais para acelerar a execução de aplicações iterativas, pois sua arquitetura permite a execução simultânea de operações em um extenso conjunto de dados. Portanto, nosso trabalho se concentrará em estudar ainda mais a acel-

eração de aplicações por meio de técnicas de computação aproximada em GPUs.

Em geral, as aproximações de aplicações em dispositivos GPU são alcançadas ajustando a precisão de ponto flutuante das operações ou usando métodos menos precisos específicos da arquitetura (SAMADI et al., 2014b; SAMADI et al., 2014a; LAGUNA et al., 2019). A agressividade das aproximações é relativa à Qualidade de Saída Alvo (TOQ) determinada pelo usuário. Para cada nova TOQ, uma nova análise é realizada para determinar quais métodos menos precisos ou precisões de ponto flutuante serão usados em cada operação para garantir que a TOQ seja respeitada. Embora essa abordagem permita melhorias significativas de desempenho, o espaço de busca de sintonia cresce significativamente à medida que aplicações maiores e mais complexos são usados. Uma maneira de evitar o custo de ajuste recorrente é usar várias versões de código com diferentes precisões, variando a seleção ou a ordem de execução das versões do kernel em tempo de execução de acordo com a proximidade da precisão de execução à TOQ.

Esta pesquisa explora a hipótese de que a execução intercalada de várias versões aproximadas de kernel, com base em seu perfil de perda de acurácia, pode melhorar o desempenho de aplicações iterativas. O perfil de perda de acurácia se refere à variação da perda ao longo da execução da aplicação. Tipicamente, várias versões de kernels com diferentes precisões são dimensionadas com base em medições de perda de precisão em tempo de execução ou calibrações de kernel realizadas antes ou durante a execução (LAGUNA et al., 2019; KOTIPALLI et al., 2019; HO; SILVA; WONG, 2021). No entanto, verificar a precisão em tempo de execução em aplicações maiores, como simulações científicas que usam domínios de dados 2D ou 3D, pode ser impraticável. Nossa solução proposta é analisar os perfis de perda de precisão da execução de várias versões de kernel CUDA com diferentes precisões para gerar uma configuração de execução que alterna a execução de diferentes versões de kernel. Essa configuração prioriza a execução da versão mais rápida, respeitando a TOQ definida pelo usuário.

As principais contribuições deste trabalho são:

1. É proposta uma nova metodologia para computação aproximada em aplicações iterativas em arquiteturas GPU. Com base no perfil de acurácia de várias versões de kernel aproximadas, uma configuração de execução intercalada dos kernels pode ser gerada inteiramente offline. Isso pode ser feito para um número arbitrário de TOQs se o conjunto de kernels e entradas de dados for o mesmo, sem medições adicionais.
2. A avaliação do desempenho e eficiência energética da metodologia proposta em

três aplicações iterativas bem conhecidas de simulação física em domínios tridimensionais. A principal conclusão da tese: a execução intercalada de múltiplas versões de kernel aproximadas com base em seu perfil de perda de acurácia pode melhorar o desempenho e a eficiência energética de aplicações iterativas.