Algorithm-Based Fault Tolerance for Matrix Operations on Graphics Processing Units: Analysis and Extension to Autonomous Operation

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
und dem Stuttgart Research Centre for Simulation Technology
der Universität Stuttgart
zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von

Claus Braun

aus Reutlingen

Hauptberichter:	Prof. Dr. Hans-Joachim Wunderlich
Mitberichter:	Prof. Matteo Sonza Reorda, PhD.

Tag der mündlichen Prüfung: 13. Juli 2015

Institut für Technische Informatik
der Universität Stuttgart

2015

To my parents and my brother.

Contents

List of Figures

Chapter 1

Chapter 2

Chapter 3

Chapter 5

Chapter 6

Chapter 7

Chapter 8

Appendix D

Appendix F

List of Tables

Chapter 7

Chapter 8

Appendix D

Appendix F

Acknowledgments

It is my pleasure to thank everyone who supported me during the past years.

I'm very grateful to my parents, Leni and Sigfried Braun, and my brother Axel, who constantly supported me throughout my education and studies. Without them, I would not have been able to even start with a work like this. I would like to thank Prof. Hans-Joachim Wunderlich for his professional supervision, his enduring support and for sharing many interesting ideas in countless discussions. I would also like to thank Prof. Matteo Sonza Reorda for his support and for accepting to be the second adviser of my thesis.

In addition, I would like to thank all my project partners within the SimTech Cluster of Excellence at the University of Stuttgart. In particular, I would like to thank Prof. Hans-Joachim Werner, Prof. Guido Schneider, Prof. Joachim Groß, and Dr. Markus Daub for interesting and fruitful interdisciplinary collaborations.

I very much enjoyed the work with colleagues and students who were at some time involved in my activities in Stuttgart. Sorry, I cannot list everyone here, you are not forgotten! I'd like to acknowledge those with whom I spend a lot of time in collaborations or personally: Melanie Elm, Michael Kochte, Stefan Holst, Eric Schneider, Alexander Schöll, Rafał Baranowski, Laura Rodríguez Gómez, Chang Liu, Michael Imhof, Marcus Wagner, Dominik Ull, Alejandro Cook, Atefe Dalirsani, Nadereh Hatami, Lars Bauer, Hongyan Zhang, Abdullah Mumtaz, Abdul-Wahid Hakmi, Anusha Kakarala, and Christian Zöllin.

A work like this is not possible without administrative and technical assistance: Thank you very much, Mirjam Breitling, Helmut Häfner, Lothar Hellmeier, and Wolfgang Moser(†).

Stuttgart, July 2015

Claus Braun

Abstract

Scientific computing and computer-based simulation technology evolved to indispensable tools that enable solutions for major challenges in science and engineering. Applications in these domains are often dominated by compute-intensive mathematical tasks like linear algebra matrix operations. The provision of correct and trustworthy computational results is an essential prerequisite since these applications can have direct impact on scientific, economic or political processes and decisions.

Graphics processing units (GPUs) are highly parallel many-core processor architectures that deliver tremendous floating-point compute performance at very low cost. This makes them particularly interesting for the substantial acceleration of complex applications in science and engineering. However, like most nano-scaled CMOS devices, GPUs are facing a growing number of threats that jeopardize their reliability. This makes the integration of fault tolerance measures mandatory.

Algorithm-Based Fault Tolerance (ABFT) allows the protection of essential mathematical operations, which are intensively used in scientific computing. It provides a high error coverage combined with a low computational overhead. However, the integration of ABFT into linear algebra matrix operations on GPUs is a non-trivial task, which requires a thorough balance between fault tolerance, architectural constraints and performance. Moreover, ABFT for operations carried out in floating-point arithmetic has to cope with a reduced error detection and localization efficacy due to inevitable rounding errors.

This work provides an in-depth analysis of Algorithm-Based Fault Tolerance for matrix operations on graphics processing units with respect to different types and combinations of weighted checksum codes, partitioned encoding schemes and architecture-related execution parameters. Moreover, a novel approach called A-Abft is introduced for the efficient online determination of rounding error bounds, which improves the error detection and localization capabilities of ABFT significantly.

Extensive experimental evaluations of the error detection capabilities, the quality of the determined rounding error bounds, as well as the achievable performance confirm that the proposed A-Abft method performs better than previous approaches. In addition, two case studies (QR decomposition and Linear Programming) emphasize the efficacy of A-Abft and its applicability to practical problems.

Zusammenfassung

Wissenschaftliches Rechnen und rechnergestützte Simulationstechnik haben sich zu unentbehrlichen Werkzeugen entwickelt, die Lösungen für wichtige Probleme in Wissenschaft und Technik ermöglichen. Anwendungen in diesen Bereichen werden häufig von rechenaufwändigen mathematischen Operationen, wie zum Beispiel Matrixoperationen aus der linearen Algebra, dominiert. Die Bereitstellung korrekter und vertrauenswürdiger Berechnungsergebnisse ist daher eine zentrale Grundvoraussetzung da die genannten Anwendungen direkten Einfluss auf Prozesse und Entscheidungen in Wissenschaft, Wirtschaft und Politik haben können.

Grafikprozessoren (GPUs) sind hochparallele Many-Core-Prozessorarchitekturen die eine außergewöhnlich hohe Gleitkommarechenleistung bei sehr niedrigen Kosten ermöglichen. Dies macht sie besonders interessant für die deutliche Beschleunigung von komplexen Anwendungen in Wissenschaft und Technik. Wie die meisten nanoskalierten CMOS-Schaltkreise sehen sich auch GPUs einer wachsenden Zahl von Störfaktoren gegenüber die ihre Zuverlässigkeit massiv beeinträchtigen. Dies macht die Integration von Fehlertoleranzmaßnahmen unabdingbar.

Algorithmenbasierte Fehlertoleranz (ABFT) erlaubt den Schutz wichtiger mathematischer Operationen die im wissenschaftlichen Rechnen zahlreiche Anwendung finden. ABFT bietet dabei eine hohe Fehlerabdeckung und verursacht nur einen geringen Mehraufwand bei der Berechnung. Die Integration von ABFT in Matrixoperationen auf Grafikprozessoren ist jedoch sehr anspruchsvoll da sie die Balance zwischen Fehlertoleranz, Prozessorarchitektur und Performanz erfordert. Darüber hinaus zeigt sich beim Einsatz von ABFT für Operationen die in Gleitkommaarithmetik ausgeführt werden häufig eine reduzierte Wirksamkeit der Fehlererkennung und -lokalisierung auf Grund von unvermeidlich auftretenden Rundungsfehlern.

Die vorliegende Arbeit stellt eine umfangreiche Analyse von ABFT für Matrixoperationen auf Grafikprozessoren unter den Gesichtspunkten verschiedener gewichteter Prüfsummenkodes, partitionierte Kodierungsschemata und ausführungsrelevanter Architekturparameter bereit. Darüber hinaus wird mit A-Abft eine neuartige Methode für die effiziente Bestimmung von Rundungsfehlerschranken zur Laufzeit vorgestellt, die die Fehlererkennung und -lokalisierung von ABFT deutlich verbessert.

Umfangreiche experimentelle Untersuchungen der Fehlererkennung, der bestimmten Rundungsfehlerschranken, sowie der erzielbaren Performanz bestätigen, dass die vorgeschlagene A-ABFT-Methode bessere Ergebnisse erzielt als bisherige Ansätze. Darüber hinaus wird die Anwendbarkeit und Effektivität von A-ABFT für praxisrelevante Probleme anhand zweier Fallstudien (QR-Zerlegung und Lineare Optimierung) gezeigt.

Chapter 1

Introduction

Scientific computing and computer-based simulation technology evolved to indispensable and highly valuable tools that impel solutions for major challenges in science and engineering. Dominated by compute-intensive mathematical operations, the high computational requirements of these applications often necessitate the use of high-performance computing (HPC) systems. This in turn limits the access to such applications and their benefits in many cases to national research laboratories, universities, governmental institutions and large corporations.

Graphics processing units (GPUs) developed within the last decade from highly specialized ASICs to versatile parallel many-core processor architectures that deliver tremendous floating-point compute performance at comparatively low cost. Contemporary GPUs enable the substantial acceleration of scientific applications and complex simulations outside of HPC compute centers, and open up an entirely new spectrum of applications. At the same time, graphics processing units are rapidly advancing into the field of embedded systems, where they provide parallel compute power for mobile devices and applications like realtime simulations or safety-critical image processing tasks in the automotive domain.

Besides high computational performance, reliability is of utmost importance for these applications to guarantee correct and trustworthy results, which can have considerable impact on scientific, economic and political processes and decisions.

Fault tolerance and its effective and efficient integration are therefore key challenges for allowing scientific computing and simulation technology to benefit in a reliable and sustainable way from many-core accelerator architectures like GPUs.

Fault tolerance enables systems to deliver their intended services even in presence of hardware failures or software errors. It can be applied at very different levels, *from the hardware up to the software and the algorithm.* The respective techniques at the different levels are thereby often guided by similar principles. The concept of redundancy, for instance, plays a fundamental role in most of them.

Hardware-based fault tolerance allows the effective protection against a variety of potential faults. The spectrum of techniques ranges from the hardening of single transistors or logic gates to hardware-implemented information, structural or temporal redundancy. Error detecting and correcting codes, for instance, can be implemented for communication channels and memories with a high gain in reliability and a comparatively low hardware overhead. Self-checking mechanisms or pure structural redundancy are effective measures for the protection of logic blocks such as ALUs. However, these techniques can be associated with significant cost in terms of area overhead, increased power consumption, and sometimes even loss of performance. Moreover, for highly parallel many-core graphics processing units, potential hardware-based fault tolerance measures may have to be implemented for structures that are placed a several hundred or thousand times on a single chip.

Software- and algorithm-based fault tolerance measures operate on top of the hardware. They can be integrated into firmware and hardware drivers, operating systems and middleware layers, as well as top level applications. Such techniques allow the effective and cost-efficient protection of processed data and the program control flow. In contrast to fixed hardware structures, software-based fault tolerance measures offer a high flexibility since they can be adapted to changing operating conditions in the field. As for hardware-based techniques, redundancy also plays an essential role for software- and algorithm-based fault tolerance. Redundant computations and encoding for the protection of data, and assertions or embedded signatures for the surveillance of the control flow are common examples of software-based fault tolerance measures. However, software- and algorithm-based fault tolerance measures can only tackle faults which propagate to the upper system layers and which become visible as error. Moreover, the integration of algorithm-based fault tolerance techniques often involves a significant redesign of existing algorithms.

The choice of an appropriate fault tolerance measure can therefore be a highly complex task, especially when a good balance between benefit and cost has to be found. Modern *cross-layer approaches to reliability* therefore try to combine different measures at different levels to achieve optimal fault tolerance.

This work addresses fault tolerance for essential mathematical matrix operations on graphics processing units at the algorithmic level. It provides a comprehensive analysis and assessment of *Algorithm-based Fault Tolerance* (ABFT) schemes for such operations with respect to their error detection, localization and correction capabilities, as well as their performance impact. Moreover, this work identifies the handling of rounding errors in state of the art ABFT schemes as a major weak point which hampers the widespread practical application of ABFT in scientific computing and simulation technology. To solve this problem, a novel, autonomously operating ABFT scheme for matrix operations is introduced, which utilizes a probabilistic approach for the efficient determination of rounding error bounds. This A-Abft scheme is tailored to the parallel nature of graphics processing units and operates online with low performance impact.

The remainder of this chapter briefly introduces the reader to modern scientific computing and simulation technology. It emphasizes their growing importance and the challenges in connection with GPU many-core processors. To motivate the application of fault tolerance, reliability requirements of scientific computations and simulations are discussed and an overview of potential reliability threats is given.

The chapter then turns to *Algorithm-based fault tolerance* as a particularly promising technique for improving the reliability of essential mathematical core operations like matrix multiplications in scientific applications. It shows that ABFT can be an excellent fit for modern cross-layer approaches to dependable scientific computing. Challenges and open research problems are presented regarding the integration of ABFT into linear algebra matrix operations on GPUs, which motivate the in-depth analysis and extension of ABFT for transparent and autonomous operation.

The chapter closes with the objectives and contributions of this work and outlines the structure of the remaining chapters it is composed of.

1.1 Modern Scientific Computing and Simulation

Scientific computing and computer-based simulation technology made extraordinary progress within the last decades. They fundamentally changed the way how scientific problems are tackled and how basic research is carried out today in many scientific domains [Winsb10]. Computational science is thereby an interdisciplinary endeavor of numerical mathematics, computer science and the research areas that provide the considered models and data.

Scientific computing complements and enhances the research in natural sciences and engineering by adding *numerical simulation* as third pillar to the existing two of *theory* and *experiment*. Computer-based simulations, so called *in-silico* experiments, are increasingly used to substitute practical experiments since they are often faster, cheaper, and—in many cases—more comprehensible. An important example for the large-scale application of scientific computing and simulation technology are the international research efforts in the area of global climate change and weather forecast [Gent11].

In engineering, the role of scientific computing and simulation technology changed from a supporting one to a leading one [Oberk10]. *Virtual prototyping* and *virtual testing* are just two synonyms for the application of both within development and evaluation processes of new products and systems. Well-known examples can be found in the aerospace and automotive industries [Benne97, Zorri03], as well as in nanoelectronics, where new products cannot be developed without massive use of scientific computations and simulations.

Applications in computational science are predominantly characterized by the following mathematical problem classes:

- Systems of linear algebraic equations,
- linear least square problems,
- eigenvalue and singular value problems,
- non-linear equations,
- optimization problems,
- interpolation and extrapolation,
- numerical integration and differentiation,
- ordinary and partial differential equations,
- (Fast) Fourier transform and spectral analysis.

The corresponding methods for the numerical solution of these problems are typically associated with high computational cost, which often grows rapidly with the problem size and the complexity of the considered model. The interested reader can find a comprehensive overview and discussion of such methods in [Heath97] and [Press07].

Linear algebra operations like the *matrix multiplication* play a central role in many solution algorithms for the above listed problem classes. As a consequence, dedicated software libraries such as *BLAS* [The N14a], *LINPACK* [The N14d] and *LAPACK* [The N14b] have grown over time, which allow the seamless integration of these operations in a wide variety of scientific applications. Their widespread use and the considerable computational effort they require, make linear algebra operations particularly interesting and attractive for acceleration on modern graphics processing units. However, because they play such a central role and due to the fact that the reliability requirements of the applications they will be used in can be very high, the simple parallelization of these operations is not sufficient. They have to be provided in a way that they are fast and fault tolerant. Selected examples of scientific applications which benefit from GPU-accelerated linear algebra operations, and the acceleration on GPUs in general, are presented in the next section.

1.2 From Graphics to GPU Computing

Development

The development of dedicated graphics hardware started in the early 1960s [Krull94, Suthe64]. *Graphics display controllers* (GDCs) and later *graphics processing units* (GPUs) evolved within thirty years to complex application-specific integrated circuits (ASICs) consisting of fixed-function pipelines. Their sole purpose was the transformation of internal, mathematical graphics representations into displayable pixel data [Hughe13].

With the spread of 3D graphics since the mid-1990s, the real-time performance requirements for complex, high resolution 3D graphics at interactive frame rates increased rapidly. GPU architectures therefore moved from fixed-function ASICs over microcoded processors to configurable and finally fully programmable, parallel many-core processors [Nicko10]. These *unified graphics architectures* [Lindh08,Nicko10] of modern GPUs differ fundamentally from conventional *central processing units* (CPUs).

Architecture

Advanced CPUs are designed for high single-thread performance through latency-optimized instruction processing. They devote significant fractions of their design to deep instruction pipelines, complex scheduling and control structures (e.g. out-of-order execution, branch prediction), as well as rather large cache memories for instructions and data [Henne12]. Multi-core CPU architectures follow the *multiple instruction stream, multiple data stream* (MIMD) paradigm [Flynn72]. They exploit internal parallelism at different levels, from the instruction level over the thread level to the core level. However, their *multi-core parallelism* increases comparatively slow (1→2→4→8→16→...) [Catan08a].

GPUs are designed for high computational throughput on data-parallel workloads. They follow the *single instruction stream, multiple data stream* (SIMD) paradigm [Flynn72] and consist of large numbers of uniform *stream processor cores* that are tightly grouped into *multiprocessor units*[1]. Stream processors share parts of the instruction issue and control logic, as well as registers and local memories within a multiprocessor unit. A special characteristic of GPU architectures is the waiver of overly complex instruction scheduling and control structures in favor of more arithmetic units, registers and embedded memories. Their memory system is explicitly exposed and comprises of several thousands of registers, shared memories and caches within the multiprocessors, accompanied by global graphics memories. The memory bandwidth of contemporary GPUs is more than ten times higher compared to CPUs[2].

Since the year 2000, the transistor count of GPUs closely followed *Moore's Law* [Moore65] and almost doubled every 18 months [Nicko10]. Within the same time, the number of stream processors also doubled nearly every 18 months, leading to rapidly growing *many-core parallelism* (128→240→512→1536→2880→...)[3] and significant performance improvements.

Implications and Challenges

The massively parallel nature of modern many-core GPU architectures has strong implications on their programmability, the design of algorithms and the integration of

[1] It should be noted at this point that the term *core* can be misleading for the comparison of CPU and GPU architectures. CPU cores are typically complete and almost independent processing units, which are placed in multiple copies on the same chip-die, sharing resources like higher level caches or bus interfaces. GPU stream processor cores are *floating-point processing pipelines* that require additional, shared infrastructure within a multiprocessor unit.

[2] A contemporary desktop CPU like the Intel Core i7-4770 offers up to 25.6GB/s memory bandwidth [Intel13], compared to up to 288GB/s of an NVIDIA GK110 GPU [NVIDI13]

[3] Exemplarily the increase of processing cores of subsequent generations of NVIDIA GPUs from 1997 (G80 architecture) to 2013 (GK110 architecture).

fault tolerance measures. GPUs require large numbers of concurrent threads (*many-threading*) to achieve high performance on data-parallel workloads. This implies that computational problems have to be decomposed in a way that they match this many-threading paradigm.

GPU memory architectures demand active management of available resources at each level. The per-thread register usage has to be controlled and kept low to allow high parallelism. Shared memories have to be used to enable data reuse and fast local thread communication. The limited capacity and bank-wise organization of shared memories on GPUs can lead to access conflicts and loss of performance. The global graphics memories offer large capacities associated with high access latencies. This favors aligned memory accesses, which can be combined by the hardware into single operations for improved bandwidth utilization, and necessitates the overlay of memory transfers and computations.

Threads on GPUs are executed in groups on multiprocessor units. The control flow within multiprocessor units operates at the granularity of such groups. This has the consequence that threads with a diverging control flow, e.g. caused by a differently evaluated comparison, lead to a serialized execution. Moreover, global synchronization of threads on GPUs for consistency reasons is highly expensive since all threads have to wait until the last finished.

The need for parallel workloads, the complex memory architecture and the control flow on GPUs, represent major challenges for the design and implementation of algorithms and they complicate the integration of software- and algorithm-based fault tolerance. Chapter 3 provides the reader with a detailed introduction to a contemporary GPU architecture and the associated programming model, which form the hardware basis of this work.

Spectrum and Significance of GPU Computing

The acceleration of complex, non-graphical scientific applications and the conducting of *in-silico experiments* on graphics processing units is rapidly gaining importance. This section provides the reader with selected references to exemplary applications. An extensive overview can be found in [Hwu11].

Computational biology, *bioinformatics* and the *life sciences* are among the domains which quickly adopted GPUs for the substantial acceleration of core applications.

Examples can be found in systems biology [Demat10], protein and genetic sequencing [Manav08, Vouzi11], as well as in the investigation of biological networks [Zhou11] and communication processes [Braun12a]. In the life sciences, medical imaging applications, in particular the reconstruction and analysis of images [Walte09, Agull11], are mapped to GPUs. In pharmacology, GPU-accelerated in-silico experiments are used to support and accelerate the design of new drugs [Sinko13].

Computational chemistry and related domains like *thermodynamics* and *process technology* exhibit numerous core methods with high potential for parallelization on GPUs. Particularly noteworthy are molecular dynamics (MD) [Yang07, Ander08, Stone10], molecular and quantum mechanics methods (MM/QM) [Ufimt08, Stone09], as well as Markov-chain molecular Monte-Carlo simulations [Braun12b]. In general, related methods such as parallel particle and n-body simulations on GPU are widely applied in other domains, for instance in astrophysics [Belle08, Hamad09].

The environmental and earth sciences such as *climate research and meteorology*, utilize GPUs for the accelerated evaluation of large-scale models of atmospheric and oceanographic systems [Shimo10, Mieli12], as well as the modeling and analysis of geophysical and seismic problems [Hollo05, Ehlig10, Abdel09, Okamo13].

In *electronic design automation* (EDA) both branches, the scientific research and the industrial engineering exhibit a multitude of highly compute-intensive tasks with strong demand for GPU acceleration [Croix13]. Design, validation and verification of digital circuits with billions of transistors benefit from GPU accelerated solvers, gate-level [Chatt09b, Chatt09a], register-transfer and system-level simulators [Nanju10]. In addition, key tasks like power analysis [Holst12] and fault simulation [Gulat08, Kocht10] are being adapted for GPUs.

In the classical engineering disciplines, GPUs are already being used in different fields, including *computational structural mechanics* (CSM) methods [Papad11] and *computational fluid dynamics* (CFD) [Liu04, Kolb05, Tolke08], dealing with Navier-Stokes models and Lattice Boltzman methods. In addition, multi-grid solvers [Goodn05] and finite-element methods (FEM) [Kiss12] are accelerated on graphics processing units.

A rather new domain with rapidly growing importance summarizes different methods and applications under the term *Big Data*. Here, techniques such as machine learning [Stein05, Catan08b], data mining and analytics [Ma09], searching [Soman10] and sorting [Sinto08, Satis09], as well as databases [Govin04, Govin06] are adapted and re-designed to utilize the compute power of GPUs. This domain also includes basic techniques like parallel random number generators [Sussm06, Langd09, Phill11] and map-reduce [He08, Stuar11].

The last application domain presented in this section is *computational finance* [Gaikw10, Pages12], where graphics processing units are increasingly used for the real-time analysis of financial markets, the assessment of risk, and for fast option pricing [Gaikw09, Surko10].

1.3 Reliability Requirements and Challenges

> *"Computational results can affect public policy, the well-being of entire industries, and the determination of legal liability in the event of loss of life or environmental damage."* [Oberk10]

High-consequence applications [Oberk10] where scientific computations and simulations are used to assess the reliability, robustness or safety of products and systems, or the risk of new technologies, pose extraordinary high reliability requirements. This is especially true for applications which can have direct influence on decision making processes, like for instance simulations of the underground deposition of nuclear waste [Bourg04], carbon sequestration [Hollo05, Ehlig10] or climatic change [Jones95, Morio11]. Such applications are often characterized by the fact that they do not allow the validation and verification of computational results by experiments due to high complexity, large timescales, high cost or risk.

The impact of scientific computing and simulation technology on research, economy and politics is directly related to the reliable provision of correct and trustworthy results. This in turn poses a hard challenge since modern scientific computing has to cope with *uncertainties* and *reliability threats*.

Uncertainty is a critical and often inevitable factor that affects scientific computing in mainly two different ways. First, uncertainty is introduced by the mathematical modeling when physical phenomena or other real-world processes are simplified and approximated. Second, the data is often associated with uncertainty, for instance due to low measurement precision or incompleteness. The handling and reduction of uncertainty, as well as the validation and verification of scientific computations are important subjects of current research [Oberk02, Roy11], but not in the scope of this work.

Like most nano-scaled CMOS semiconductor devices, modern graphics processing units are increasingly prone to a number of reliability threats which can have significant impact on the underlying computations of scientific applications. The spectrum of threats reaches from increasing variability in manufacturing processes over increasing susceptibility to transient events, to stress and aging-related effects. In the best case, the effects of these threats are mitigated by fault tolerance measures or, at least, become visible as errors. In the worst case, these effects remain undetected *silent data corruptions* and cause erroneous results with potentially far-reaching consequences. Since additional uncertainty in computed results due to such errors cannot be tolerated, the integration of appropriate fault tolerance measures has become mandatory. However, although GPUs have become highly attractive for parallel computing, their main purpose still is graphics processing and they are designed for the mass market. This puts tight limits to the integration of hardware-based fault tolerance due to the associated cost as well as the lower reliability requirements in this market. Section 1.4 provides an overview of the most important reliability threats for current semiconductor nano-electronics.

Challenges for the integration of algorithm-based fault tolerance arise from the hardware and the software. On the hardware side, GPUs and their implications on the design and implementation of algorithms (see Section 1.2) demand fault tolerance schemes which are closely tailored to their parallel architecture and the many-threading paradigm to prevent substantial loss of performance. On the software side, scientific applications and essential kernels like matrix operations are often highly optimized and tuned for maximum performance. This leaves only small room for the integration of fault tolerance measures. Only schemes with low computational overhead and low memory requirements can be considered for productive use. Moreover, the projected, short mean times between failures in upcoming generations of computing systems [Cappe09] demand fault tolerance schemes that operate online and with low latencies.

1.4 Reliability Threats

The unprecedented progress in *Complementary Metal-Oxide-Semiconductor* (CMOS) technology continues to drive the design and manufacturing of highly complex integrated circuits [ITRS13]. Semiconductor technology scaling [Moore65] enables latest generation CPUs and GPUs with reduced power consumption and improved performance, comprising of billions of transistors.

However, this development does not only have beneficial effects, it also emphasizes existing and introduces new threats to the reliability of sophisticated integrated semiconductor circuits [Bol09,Nassi10,ITRS13]. These threats can be roughly categorized into *manufacturing-related threats*, life cycle or *operation-related threats*, and external, *environmental threats*.

Manufacturing

CMOS manufacturing processes have to cope with physical and technological challenges among which *variability* [Borka05,Kuhn11] is one of the most critical. Random and systematic variations appear at different stages during the manufacturing process. Lithography variations [Owa04, Bensc08, ITRS13] cause amorphous structures and effects like increased line edge roughness (LER) [Aseno03]. Distortions of the silicon crystal structure, impurities and especially random dopant and implantation fluctuations (RDF) [Aseno98,Tang97] can lead to incomplete or defective structures, which cause threshold voltage fluctuations. The same applies to variations due to etching or chemical and mechanical polishing [Stine98]. Variability within the manufacturing process can cause altered electrical properties and potentially erroneous functional behavior. Production tests such as burn-in or packaging tests are used to filter out defective devices. However, test escapes are possible [Borka05] and more weak devices with latent defects may come into productive use with reduced reliability and an increased risk of transient and intermittent faults or even early-life failures in the field.

Operation

During operation and over the life cycle of CMOS devices, dynamic variations like temperature changes, crosstalk on signals or busses, and fluctuations in the power grid like power droop [Bowma09] and IR drop [Nithi10], can lead to erroneous behavior. These effects are accompanied by stress and aging mechanisms such as negative-biased temperature instability (NBTI) [Huard06, Alam05], hot carrier injections (HCI) [Segur04, Chen99], time-dependent dielectric breakdown (TDDB) [Ogawa03, Segur04], and electromigration [Arnau11]. Negative-biased temperature instability is a phenomenon where positive charges (Si^{+}) build up at Si/SiO_2 channel interfaces within

PMOS transistors, leading to increased threshold voltages. Hot carrier injections occur when carriers are accelerated along conducting channels and resulting collisions with substrate atoms cause electron/hole pairs (oxide damage). Time-dependent dielectric breakdown (TDDB) is a wear-out phenomenon where the insulating property of the device oxide is reduced. TDDB can lead to hard breakdowns when conductive channels are formed by accumulated charge in the oxide. Electromigration describes the movement of metal atoms caused by electron flux and high temperatures, which can lead to voids and cracks, as well as shorts and bridges.

Environment

Environmental reliability threats include different kinds of mechanical stress like vibrations [Sony 10], as well as high ambient temperatures. In addition, *single event effects* (SEE) can be caused by radiation. Such effects occur when ionizing particles like heavy ions, neutrons or alpha particles hit the substrate of a CMOS device and generate additional charges. If such charges are collected at junctions or contacts, they can change logic states within a circuit. In [Nicol11], Heijmen distinguishes six types of SEE based on their place of occurrence and impact. *Single-bit (SBU) and multi-cell upsets (MCU)* in memory cells and latches, *multi-bit upsets (MBU)* of two or more bits within a memory word, *single-event transients (SET)* causing voltage glitches that become bit errors when captured in a state element, and *single-event latchups (SEL)*, which can cause permanent damage in a circuit. Almost all parts of modern CMOS circuits are vulnerable to single event effects, from latches and flip-flops to static and dynamic RAM, as well as the logic. However, the actual impact of SEEs on the system behavior depends strongly on the used semiconductor material, the circuit design, the radiation intensity and potentially integrated fault tolerance measures.

All described mechanisms can cause performance degradations, reduced reliability during operation or even complete failure.

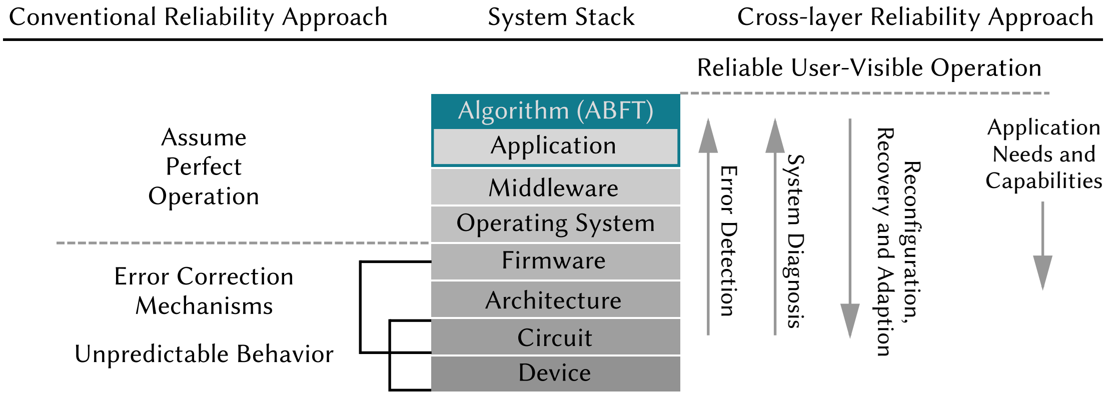
1.5 Algorithm-Based Fault Tolerance for Reliable GPU Computing

Algorithm-based fault tolerance (ABFT) is a fault tolerance technique introduced by Huang and Abraham in 1982 [Huang82, Huang84]. It is based on the idea of utilizing block codes for the encoding of data before it is processed by modified algorithms, which in turn produce encoded output data. The information redundancy introduced by the encoding is used to detect, localize and correct errors that occur during processing of the data. Algorithm-based fault tolerance can operate online and is characterized by a comparatively low overhead for the encoding and checking.

For certain application scenarios, algorithm-based fault tolerance can be less general or flexible compared to conventional fault tolerance techniques such as n-modular redundancy or redundant computations. The design and integration of ABFT can also be a non-trivial and challenging task depending on the target algorithm. However, algorithm-based fault tolerance allows the efficient and effective protection of essential mathematical operations, ranging from linear algebra matrix operations and decompositions to spectral analyses based on fast Fourier transforms. These operations represent extensively used core components within many scientific applications and can benefit substantially from acceleration on graphics processing units.

Cross-layer reliability approaches [DeHon10] leave the perspective of isolated fault tolerance measures at single, potentially lower levels and consider the whole system stack for decentralized, cooperative fault tolerance. Appropriate measures for the detection and mitigation of errors are implemented, matching the requirements of each layer optimally. Communication across the layers of the system stack allows the handling of errors at higher levels, which have not been detected or mitigated at lower levels. Figure 1.1 shows the different perspectives of isolated, conventional approaches and modern cross-layer reliability. Algorithm-based fault tolerance directly complements such cross-layer reliability approaches and can act at the top level of the system stack.

However, although ABFT is a mature technique on which a substantial amount of research has been conducted, several research questions and problems remain open and hamper the widespread, practical use of this technique: A broad spectrum of extensions and different encoding schemes has been proposed for ABFT over the years. These approaches unfold a considerable parameter space with a variety of options for their combination. The impact of individual methods and their potential combinations on the ability of ABFT to detect, localize and correct errors, as well as their impact on the performance of the target operation have not yet been adequately studied with respect to the application on GPUs. The implications of these architectures (see Section 1.2) make the selection and implementation of appropriate solutions difficult.


Conventional Reliability Approach
System Stack
Cross-layer Reliability Approach
Reliable User-Visible Operation
Assume Perfect Operation
Error Correction Mechanisms
Unpredictable Behavior
Algorithm (ABFT)
Application
Middleware
Operating System
Firmware
Architecture
Circuit
Device
Error Detection
System Diagnosis
Reconfiguration, Recovery and Adaption
Application Needs and Capabilities

▲ **Figure 1.1** — Algorithm-Based Fault Tolerance as part of a cross-layer reliability approach. Adapted from [Carte10].

Another problem represents the integration of algorithm-based fault tolerance into applications which rely on floating-point arithmetic as it is predominantly the case for scientific applications and simulations. Inevitable rounding errors complicate the encoding and especially the result checking procedures of ABFT. Significant errors with potentially malicious impact on the computed results have to be clearly distinguished from rounding errors due to limited precision. Such classifications require appropriate rounding error bounds. Although there exist different approaches for the determination of such bounds in ABFT, the state of the art solutions often lead to weak bounds, which increase the risk of significant errors slipping through the ABFT result check (cfg. Chapter 5).

The above-mentioned problems are contrary to a successful integration of ABFT into mathematical kernels and their practical use for scientific computations on GPUs. Such an integration requires far-reaching independence from the actual field of application, the processed data, their distributions and characteristics. The ABFT-protected kernels have to be provided in a way that they operate autonomously without requiring user knowledge about the processed data or for the selection of encodings and other parameters. Only this guarantees the flexible and successful use of ABFT.

1.6 Objectives and Contributions of this Work

This work targets algorithm-based fault tolerance for linear algebra matrix operations on graphics processing units. These operations can benefit substantially from parallel acceleration and form an essential foundation of many scientific applications. It provides a comprehensive analysis and assessment of the most important ABFT encoding approaches and the accompanying parameters with respect to error detection, localization, correction and performance impact on GPU architectures.

Furthermore, this work extends algorithm-based fault tolerance for matrix operations with a new method for the determination of rounding error bounds. This extension is based on a probabilistic model of floating-point arithmetic and enables ABFT-protected operations to determine rounding error bounds at runtime without requiring user knowledge or intervention. The developed method is tailored to modern GPU architectures and reduces the number of false-negative error detections while maintaining high performance of the target matrix operations. In addition, the method can be used to deliver detailed information on the rounding error that has to be expected for the performed operation. This allows the provision of ABFT-protected matrix operations on graphics processing units which act as accelerated, fault tolerant building blocks for scientific applications and simulations.

Besides an in-depth evaluation of the developed method with respect to the quality of the determined error bounds, the achievable error detection rates and the associated performance overhead, two case studies are presented which show the application of the method and the benefit of fault tolerant, accelerated matrix operations on GPU in scientific applications.

The remainder of this work comprises the following chapters: *Chapter 2* introduces the reader to the formal foundations on which subsequent chapters build upon. This includes definitions of essential terms and concepts from the fields of dependability, fault modeling, coding theory, and floating-point arithmetic.

Chapter 3 describes in detail the hardware architecture of a contemporary graphics processing units, which also formed the hardware platform for the experimental evaluation parts of this work. In addition to the GPU architecture, the software programming model is introduced to the reader.

Chapter 4 provides a concise summary of the state of the art for hardware-related fault tolerance measures with a special focus on fault tolerance for graphics processing units.

Chapter 5 introduces the reader in detail to the field of algorithm-based fault tolerance. A formal mathematical definition of ABFT for matrix operations is presented which provides the link to coding theory and which builds upon the foundations given in

Chapter 2. The chapter gives a brief overview of possible applications of ABFT and then focuses on relevant ABFT encoding techniques. The last part of the chapter presents the state of the art for the determination of error tolerances and rounding error bounds in ABFT.

Chapter 6 presents a novel ABFT approach called A-Abft, which operates autonomously and utilizes a probabilistic method for the online determination of rounding error bounds on graphics processing units. The reader is introduced to the mathematical background and the algorithmic steps that are tailored to GPU architectures.

Chapter 7 presents two case studies which show the application of the developed probabilistic A-Abft scheme for accelerated matrix multiplications on GPU. The first study considers a QR decomposition based on Householder reflections, which plays an important role in numerical mathematics for the solution of linear least squares problems and forms the basis of the QR algorithm for the solution of eigenvalue problems. The second case study considers an accelerated linear programming solver, which represents an essential tool for the solution of optimization problems.

Chapter 8 presents the analysis and assessment of the most relevant ABFT encoding schemes, their associated parameters, as well as the developed probabilistic method for the online determination of rounding error bounds. Error detection capabilities are evaluated and the impact on the achievable performance on graphics processing units investigated.

Chapter 9 summarizes the findings of this work, discusses the achieved results and points out starting points for further research.

Abbreviations, notations and supplemental material, as well as extended experimental results can be found in the appendices.

Chapter 2

Formal Foundations and Background

This chapter introduces the formal foundations of this work comprising important concepts, definitions and terminology on which subsequent chapters build upon. This includes the fields of dependability and fault modeling, coding theory, as well as floating-point computer arithmetic. In order to keep this introduction concise, potentially required basics from the fields of abstract and linear algebra, which can be found in literature, are omitted. Dependability-related mathematical definitions are given in Appendix B and fundamental definitions from coding theory in Appendix C. Where appropriate, references pointing to the corresponding literature are provided.

2.1 Dependability Concepts and Terminology

This section presents required definitions and concepts of dependability concerning this work. It closely follows the definitions introduced by Avižienis et al. in [Avizi04].

Defects, Faults, Errors and Failures

Starting point for the following definitions is the notion of a *system.* A system can be described as an entity which communicates and interacts with other systems. These other systems form the *environment* of the considered system, which is separated from this environment by its *boundaries.*

A system is determined by different properties among which its *functionality* is the most important one. The steps performed by the system to realize its specified functionality form its *behavior.* This behavior can be described as a sequence of states.

The system's behavior, as it can be observed by other systems in its environment, represents the *service* it provides. This service is delivered via dedicated points at the system's boundary, the so called *service interfaces.* The set of states that become visible at the service interfaces form the system's *external state.* The remaining states represent its *internal state.* The provided service can be described as a sequence of such states.

A *defect* is a distortion of the physical structure of the system's hardware. Various defect mechanisms, such as aging or radiation, can increase the probability of occurring defects.

A *fault* is an abstraction to model and represent a defect at the structural level. Faults can be categorized into *permanent*, *intermittent* and *transient* faults. A *permanent fault* occurs and remains in the system until it is repaired or replaced. An *intermittent fault* occurs repeatedly in regular or non-regular time intervals. A *transient fault* occurs once and typically disappears after a short period of time.

A correct service is delivered by a system as long as it provides its functionality according to the specifications. In cases where the service is not delivered or delivered deviating from the specifications, a *failure* has occurred. Such a failure implies that a single external state or a sequence of external states of the system differs from the correct sequence of states. This deviation is called an *error.* If the presence of such an error is signaled by the system, the error is referred to as *detected error.*

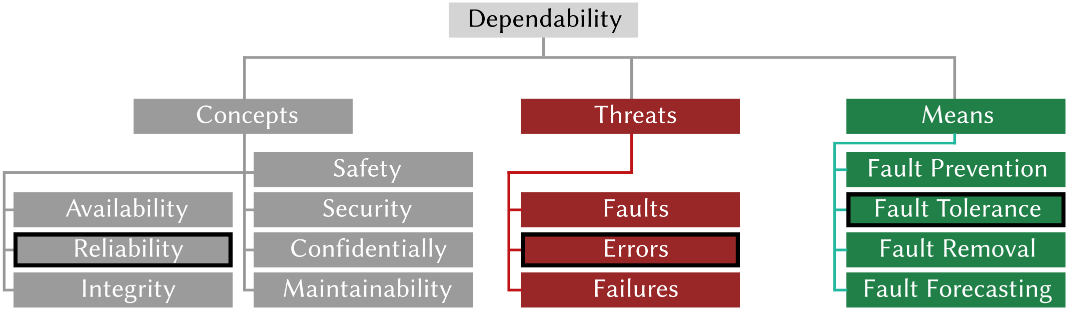
In the context of this work, a system consists of one or more graphics processing units and the matrix operations that are executed on these GPUs. The service interface is represented by software interfaces that allow to call software library routines.

Dependability, Reliability and Fault Tolerance

Dependability is not an isolated or stand-alone concept. It is a unification that integrates different concepts and it can be characterized by these concepts, the threats it faces and the means that can be applied to achieve it. The concepts comprise availability, reliability, safety, integrity and maintainability. Dependability is also often extended by security, which then adds confidentiality to the set of concepts. *Availability* describes the readiness of a system, which means the fraction of time the system delivers its correct service during a given interval of time. *Reliability* describes the probability that a system provides its correct service during a specified period of time (cfg. Definition B.4). *Safety*, in turn, implies that the operation of a system cannot have any dangerous or harmful effects on its environment. The concepts of *integrity* and *confidentiality* are also related to security and imply that a system cannot be altered in a malicious way and that information is not disclosed by a system unintentionally. The last concept, *maintainability*, describes the property that a system can be diagnosed and repaired in reasonable time. Mathematical definitions of availability and reliability are given in Appendix B.

The threats that potentially jeopardize the dependability of a system are *defects*, *faults*, *errors* and *failures*, as they have been introduced above. A dependable system tries to prevent and neutralize these threats by the application of the following groups of countermeasures: *Fault prevention* and *fault tolerance*, which try to prevent faults before they occur or try to ensure the correct service although faults have occurred. Fault prevention techniques target the design and implementation phase of a system, whereas fault tolerance techniques are intended to have effect during operation of the system. Fault tolerance includes the steps *error detection* and *error recovery*. *Fault removal* techniques are intended to reduce the total number of faults and the severity of their impact. Fault removal can be performed during the design phase, for instance by application of formal methods for validation of the design, as well as during operation. *Fault forecasting* techniques try to predict the occurrence and impact of future faults based on current fault information.

To contribute to the overall goal of *dependable* scientific computing on graphics processing units, this work is dedicated to improve the *reliability* of GPU matrix operations. This goal is achieved by applying means of *fault tolerance* that target *errors* which manifest themselves as faulty values. The dependability tree in Figure 2.1 summarizes the characteristics of dependability and highlights the parts involved in this thesis.



▲ **Figure 2.1** — Dependability tree with concepts, threats and means. Adapted from [Avizi04].

2.2 Fault Modeling

In complex circuits or systems the spectrum of potential defects and other disturbances is extremely wide. Even defects of the same kind can have significantly different effects depending on, for instance, the location of the defect. This makes essential tasks like the testing and diagnosis of such circuits, but also the development of appropriate fault tolerance measures, challenging or even impossible.

Fault models are an elegant way to deal with this complexity since they provide suitable abstractions which allow the bundling and collapsing of different sets of faults into single fault classes. However, a fault model typically covers only a fraction of all existing defects and the choice of a suitable model highly depends on the application.

Just like circuits, fault models can be specified at different levels, from the physical structure and the layout of a circuit up to the system architecture or the algorithm. With increasing abstraction at higher levels, a loss of resolution and degree of detail is inevitable. A classic example of a gate-level fault model is the *single stuck-at* fault model, which covers defects that cause signals to be stuck at a logic value 0 (stuck-at zero) or logic value 1 (stuck-at one). The reader can find a comprehensive overview of important electronic design fault models in [Bushn00].

In [Wunde10], the *conditional line flip* (CLF) model has been introduced, which allows a general description of faulty behavior at a single line or signal with the formulation $line \oplus condition$. The *line* statement represents an affected signal with its structural location and the *condition* refers to an arbitrary activation condition, for instance a boolean or time-related condition. The CLF model is able to express all traditional circuit fault models in a concise way.

Algorithm-based fault tolerance is a fault tolerance technique which acts at the level of algorithms and their respective implementations. Low level fault models such as the single stuck-at or the transition-delay fault model are therefore often of limited use and a higher level of abstraction is required. In [Huang84], a *module level fault model* has been introduced which considers single computational units or complete processors within a multiprocessor system as abstract modules. Modules are able to communicate with each other and they produce certain results that become visible at their service interfaces. Furthermore, the following assumptions apply for the module level fault model: Within a given period of time, at most one module produces erroneous results and latent failures are detected by periodic testing of the modules. Communication channels and memories are protected, for instance, by error detecting and correcting codes.

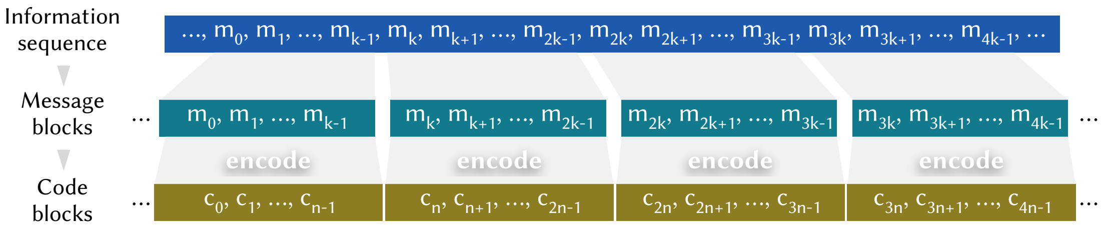
In this work, the module level fault model is refined and adapted for the application to architectures and execution paradigms of modern graphics processing units. Stream processing cores, which reside within multiprocessor units of GPUs (see Chapter 3), are considered as basic entities or modules in this model. Although the above-mentioned assumptions still apply, multiple modules are allowed to produce erroneous result values as long as they are not located within the same multiprocessor unit. This assumption is justified by the fact that modern graphics processing units consist of multiple multiprocessor units. In addition, it is assumed that the algorithm-based fault tolerance scheme is always able to execute its basic phases, which comprise the encoding of input data, the payload computation of the target algorithm, and the decoding and checking phase. This assumption, in turn, is justified by the fact that an ABFT scheme can be combined with additional software-based fault tolerance techniques for the protection of the control flow, as well as hardware-based fault tolerance measures like dedicated watchdog processors.

2.3 Coding Theory

Algorithm-based fault tolerance can be formulated and defined in terms of *linear block codes*. In ABFT, such linear block codes are applied at the word level instead of the classic bit level. This section provides the reader with a short introduction to linear block codes. Foundations and basic definitions from coding theory can be found in Appendix C.

Linear Block Codes

Linear block codes form the theoretical foundation of algorithm-based fault tolerance. A q-nary *block code* $\mathcal{C} : \Sigma^k \rightarrow \Sigma^n$, denoted by $\mathcal{C}(n,k)$, partitions a sequence of input information symbols m_i into *message blocks* of length k. These blocks are independently encoded into *code blocks* or *codewords* of equal length n with $k \leq n$ (see Figure 2.2). For a block code $\mathcal{C}(n,k)$ with the minimum Hamming distance d, the number of detectable errors is given as $d-1$ and the number of correctable errors is given as $\lfloor \frac{d-1}{2} \rfloor$. Such block codes are denote as $\mathcal{C}(n,k,d)$.


Information sequence
$\dots, m_0, m_1, \dots, m_{k-1}, m_k, m_{k+1}, \dots, m_{2k-1}, m_{2k}, m_{2k+1}, \dots, m_{3k-1}, m_{3k}, m_{3k+1}, \dots, m_{4k-1}, \dots$
Message blocks
…
$m_0, m_1, \dots, m_{k-1}$
$m_k, m_{k+1}, \dots, m_{2k-1}$
$m_{2k}, m_{2k+1}, \dots, m_{3k-1}$
$m_{3k}, m_{3k+1}, \dots, m_{4k-1}$
…
encode
encode
encode
encode
Code blocks
…
$c_0, c_1, \dots, c_{n-1}$
$c_n, c_{n+1}, \dots, c_{2n-1}$
$c_{2n}, c_{2n+1}, \dots, c_{3n-1}$
$c_{3n}, c_{3n+1}, \dots, c_{4n-1}$
…

▲ **Figure 2.2** — Basic principle of block codes.

For the definition of linear block codes and subsequent considerations, it is beneficial to formulate these codes by algebraic means: A q-nary (n,k) block code $\mathcal{C}(n,k,d)$ over the finite field $\mathbb{F}_q$ is called a *linear block code*, if and only if $\mathcal{C}(n,k,d)$ is a subspace of the vector space $\mathbb{F}_q^n$ with dimension k. The linearity implies that an arbitrary superposition of valid codewords yields a valid codeword.

The encoding and decoding of information using linear block codes can be elegantly expressed through the the *generator matrix* G and the the *parity-check matrix* H associated with the code (see Definitions C.8 and C.10). An information sequence which has been partitioned into message blocks m is encoded into corresponding code blocks c by simply multiplying each message block with the generator matrix:

$$c := m \cdot G.$$

For the decoding and detection of errors e, the syndrome s is computed using the parity-check matrix H:

$$s^T := H \cdot e^T. \tag{2.1}$$

If the syndrome s is non-zero, the error e is detected. All non-zero errors e that fulfill $H \cdot e^T = 0$ lead to a valid codeword and cannot be detected.

In chapter 5, the formal definition of algorithm-based fault tolerance for matrix operations will be introduced using the concepts described above.

2.4 Floating-Point Number Systems and Arithmetic

Being an infinite, continuous set of values, the real numbers pose a challenge for digital computer systems, since these are limited to finite sets of binary values for the representation of information. This fact urged the development of suitable approximations which allow the mapping of at least a portion of the real numbers to this finite, discrete quantity of machine-representable values. A comprehensive overview of different number formats and representations that have been proposed over time, can be found in [Mulle10]. Today, real numbers in scientific computing and engineering are predominantly represented by floating-point numbers according to the IEEE Standard 754™ for floating-point arithmetic [IEEE 08]. The following introduction is based on this standard using base 2.

Floating-Point Numbers

The design of floating-point systems is a non-trivial task which requires the balancing of different, in parts conflicting, constraints like accuracy, performance, value ranges, portability and implementation complexity [Mulle10]. Floating-point arithmetic also exhibits certain characteristics and properties which have to be considered thoroughly by the user and whose defiance or careless treatment can lead to serious errors.

A floating-point number is a compound consisting of a *sign*, a *significand* and an *exponent*. This compound is accompanied by a set of parameters like the *radix* or *base* upon which the number is defined, the *precision* and the *value range of the exponent* [Mulle10]:

- The **radix** or **base** $\boldsymbol{\beta}$ is an integer value with $\boldsymbol{\beta} \geq 2$. Unless indicated otherwise, radix $\boldsymbol{\beta} = 2$ is used throughout this work.
- The **precision** $\boldsymbol{p}$ is an integer value with $\boldsymbol{p} \geq 2$. It defines the number of digits d_i in the significand.
- The **sign** $\boldsymbol{s}$ is an integer value with $\boldsymbol{s} \in \{0, 1\}$.
- The **exponent limits** $\boldsymbol{e_{min}}$ and $\boldsymbol{e_{max}}$ are integer values with $\boldsymbol{e_{min}} \leq 0 \leq \boldsymbol{e_{max}}$. These values represent the minimum and the maximum possible value of the exponent.

- The **exponent** $\boldsymbol{e}$ is an integer value with $\boldsymbol{e_{min}} \leq \boldsymbol{e} \leq \boldsymbol{e_{max}}$.
- The **normalized significand**[1] $\boldsymbol{m}$ is a number which is represented by a digit string of the form $d_0 \bullet d_1 d_2 \ldots d_{p-1}$, where d_i is an integer digit with $0 \leq d_i < 2$ and therefore $0 \leq \boldsymbol{m} < 2$.

This allows the formulation of a floating-point number a in the form

$$a = (-1)^s \times m \times 2^e. \tag{2.2}$$

With radix $\boldsymbol{\beta} = 2$, normalization does not only guarantee the unique representation of floating-point numbers, it also allows the implicit storage of the digit on the left of the radix point, which is always one. This yields an additional bit of precision for the significand. However, with a normalized floating-point number format, zero can no longer be directly represented, which makes a separate encoding necessary (see Appendix D).

Floating-point numbers whose exponent is $\boldsymbol{e} = \boldsymbol{e_{min}}$ and whose significand is $|m| < 1$ do not fulfill the normalization condition and are called *subnormal floating-point numbers*[2]. Although subnormal numbers complicate the implementation of floating-point arithmetic, they do fill the gap between the smallest representable, normal floating-point number and zero, which allows for so called *graceful underflow*.

Rounding and Sources of Error

Floating-point numbers are a finite set of rational numbers which have a terminating expansion with respect to the radix $\boldsymbol{\beta}$. In case of radix $\boldsymbol{\beta} = 2$, rational numbers whose denominator is a power of 2 can be represented exactly. Numbers without a terminating expansion have to be adjusted, which means that they are mapped to an appropriate floating-point number nearby. This adjustment is called *rounding* and introduces a so-called *rounding error*.

The mapping of numbers without a terminating expansion to a floating-point number is not always unambiguous. In cases where a candidate is located exactly in the middle between two numbers, distinct tie-breaking rules are required. Such rules, or *rounding*

[1] In this work, the significand of floating-point numbers is denoted by M or m, which is derived from the term *mantissa* and which is used to avoid confusion with the *sign* s of such numbers. The term significand was originally introduced by Forsythe and Moler [Forsy67].

[2] Subnormal floating-point numbers are also often called *denormal*.

modes, have to be applied in a consistent manner to achieve reproducible and predictable results.

The IEEE Standard 754™ for floating-point arithmetic (see Appendix D) provides detailed guidelines and definitions of rounding algorithms for arithmetic operations such as addition, multiplication or square root [Goldb91,Mulle10,IEEE 08]. The *round-to-nearest* rounding mode maps a real number a to $\boldsymbol{RN(a)}$, which is the floating-point number closest to a. In case that a is located exactly in the middle between two floating-point numbers, a so called *tie-breaking rule* is required. In IEEE-754(-2008), *round-to-nearest-even* is the default, which maps a to the floating-point number whose integral significand is even. Another option for tie-breaking is *round-ties-away*, where the floating-point number whose magnitude is the larger one is chosen. The *round-towards-zero* rounding mode maps a to $\boldsymbol{RZ(a)}$, which is the floating-point number closest to a, but not greater in magnitude than a. The *round-towards-*$+\infty$ rounding mode maps a to $\boldsymbol{RU(a)}$, which is the smallest floating-point number greater than or equal to a, possibly $+\infty$. The *round-towards-*$-\infty$ rounding mode maps a to $\boldsymbol{RD(a)}$, which is the largest floating-point number less than or equal to a, possibly $-\infty$.

For the quantification of the rounding error that occurs in a floating-point computation, an appropriate measure can be beneficial. For any floating-point number system and accompanying rounding procedure, the *machine epsilon* ϵ_M is therefore defined as the difference between 1.0 and the next-nearest number that can be represented in the given floating-point number format[3,4]:

$$\epsilon_M := \beta^{-(p-1)} \tag{2.3}$$

In case of radix $\beta = 2$ and for the single and double precision IEEE 754 number formats, the machine epsilon is hence defined as

$$\epsilon_M \quad := \quad 2^{-23} \approx 1,1921 \cdot 10^{-7} \quad \text{(single precision)} \tag{2.4}$$

$$\epsilon_M \quad := \quad 2^{-52} \approx 2,2205 \cdot 10^{-16} \quad \text{(double precision).} \tag{2.5}$$

Besides the rounding error, floating-point arithmetic exhibits certain characteristics which can lead to considerable errors, if they are not treated carefully. This is especially

[3] It should be noted that the IEEE Standard 754™ for floating-point arithmetic [IEEE 08] does not provide a definition for the term *machine epsilon*.

[4] The term *unit roundoff* $\boldsymbol{u}$ can also be found in literature and is often used interchangeably with the term *machine epsilon*.

true when software is designed with properties of the real numbers in mind that do not apply to floating-point numbers. In general, the addition and multiplication of floating-point numbers are neither associative nor distributive, which makes an important difference to the real numbers. Moreover, the phenomenon of *subtractive cancellation* can lead to a loss of accuracy if floating-point numbers are subtracted which are almost equal. Additions and subtractions of floating-point numbers which differ significantly in their magnitude can lead to a phenomenon called *absorption*, where the contribution of the smaller operand is lost. Similar problems arise with *underflows* in floating-point computations.

Rounding errors and phenomena such as those described above can be regarded as major drawbacks of floating-point arithmetic. However, in this work, the opposite perspective is taken and these phenomena are considered as inevitable properties of floating-point arithmetic which even provide a certain degree of inherent fault tolerance. This has particular implications for the design and implementation of fault tolerance techniques such as ABFT. In the context of floating-point arithmetic, such fault tolerance techniques must never be more accurate than the level of accuracy provided by the arithmetic system itself. This effectively means that errors can be tolerated as long as they are in the order of magnitude of the performed operation's expected rounding error. This in turn makes the derivation of rounding error information and the determination of appropriate rounding error bounds an essential task.

CHAPTER 3

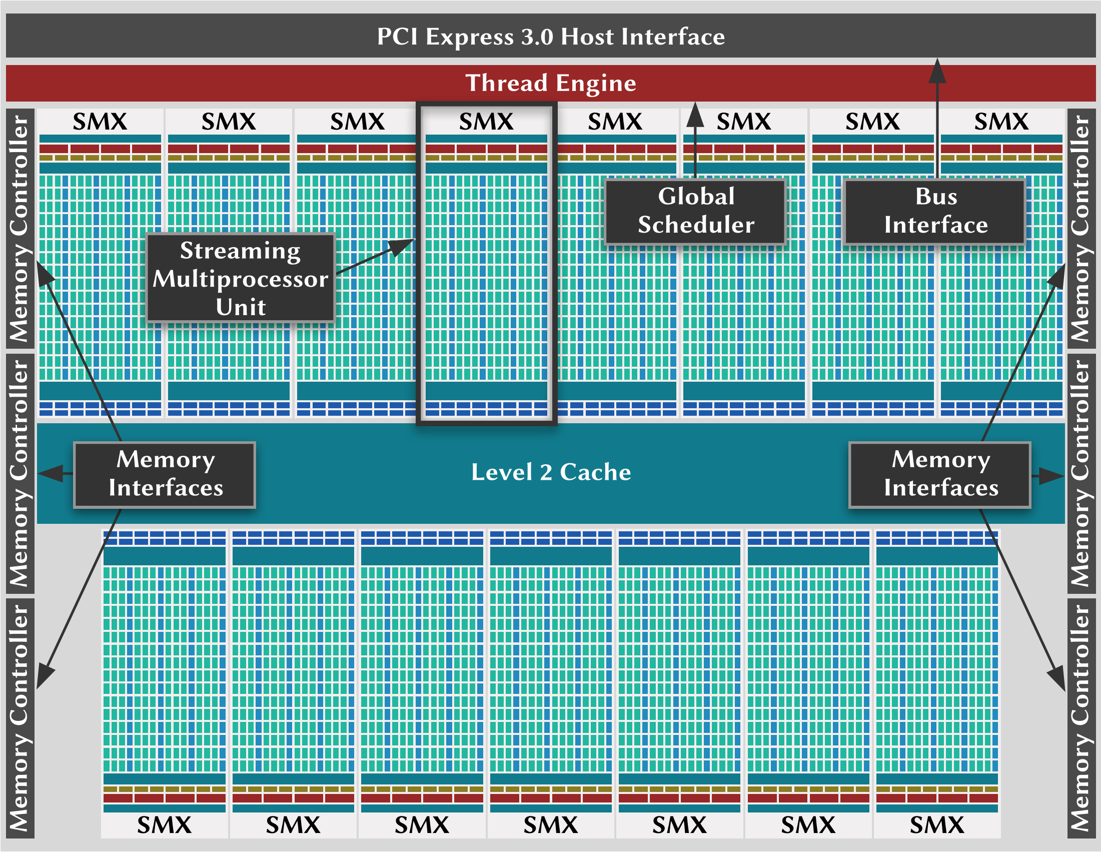
GPU Architectures and Programming Models

This chapter introduces modern graphics processing architectures and the accompanying programming models in detail. The basic principles are explained using the example of the NVIDIA GK 110 GPU architecture [NVIDI12] and the *Compute Unified Device Architecture* (CUDA) programming model. Since modern graphics processing units and programming models exhibit strong similarities, the described principles are transferable to other contemporary architectures. The largest currently available GPU based on the described architecture consists of $7.1 \cdot 10^9$ transistors which reside on a 533 mm^2 silicon die, manufactured in 28 nm CMOS technology. It comprises 2880 processing cores and delivers a theoretical peak performance of 4.29 TFLOPS single precision and 1.43 TFLOPS double precision.

3.1 Global GPU Architecture

Advanced graphics processing units possess many-core stream architectures, which follow the *Single Instruction stream, Multiple Data stream* (SIMD) [Flynn72] paradigm. In principle, these architectures execute a single instruction stream, coming from a single GPU program, by a large number of parallel threads on different data elements.

However, latest generation GPUs are capable of executing multiple of such GPU programs simultaneously and thus execute multiple different instruction streams in parallel. This allows a continuous and higher utilization of available compute resources and improves the throughput. Such GPU architectures are hierarchically organized with clusters of processing cores that share parts of the infrastructure like registers, shared memories, as well as parts of the scheduling and control logic. The GK 110 architecture is subdivided into 15 multiprocessor units[1] of which each hosts 192 processing cores. Besides the multiprocessor units, the GPU contains a global workload scheduler, a level-2 data cache and six 64-bit memory controllers. The communication with host systems is realized through a PCI Express 3.0 bus interface. Figure 3.1 shows the top level block diagram of the architecture.

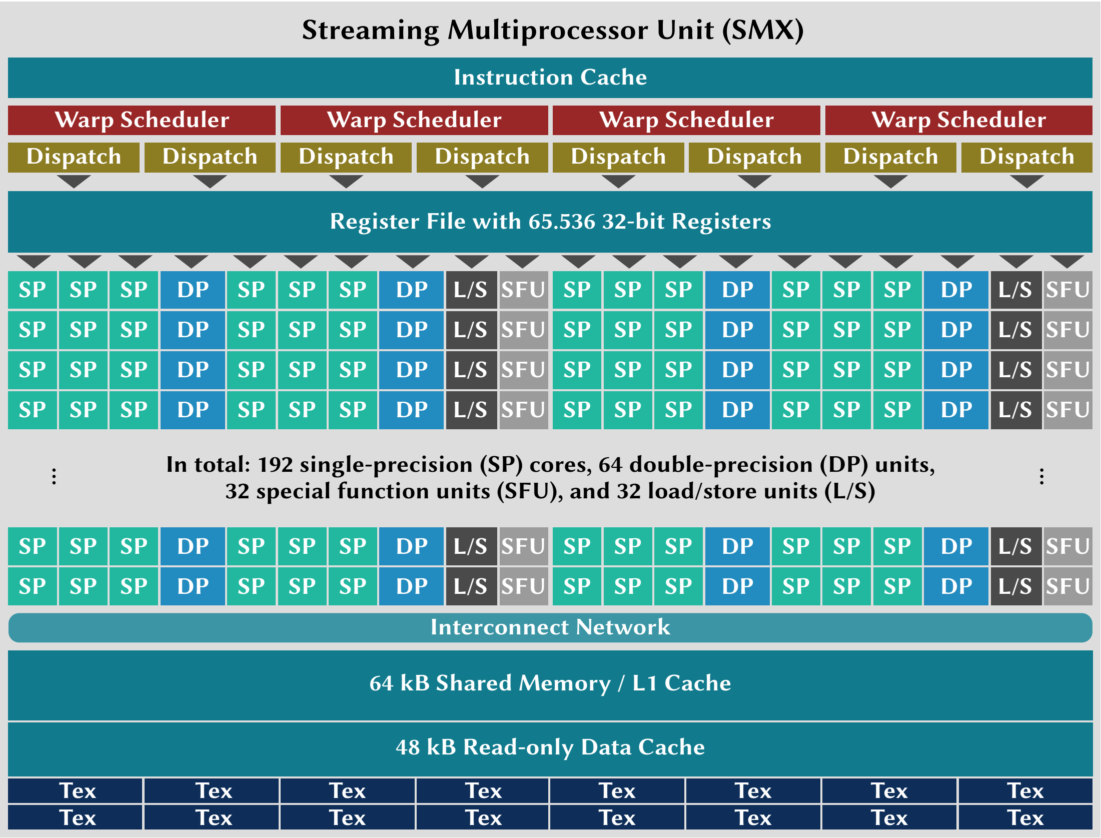


▲ **Figure 3.1** — NVIDIA GK 110 "Kepler" GPU architecture top-level block diagram. Adapted from [NVIDI12].

[1] *Streaming multiprocessors* or *SMX*.

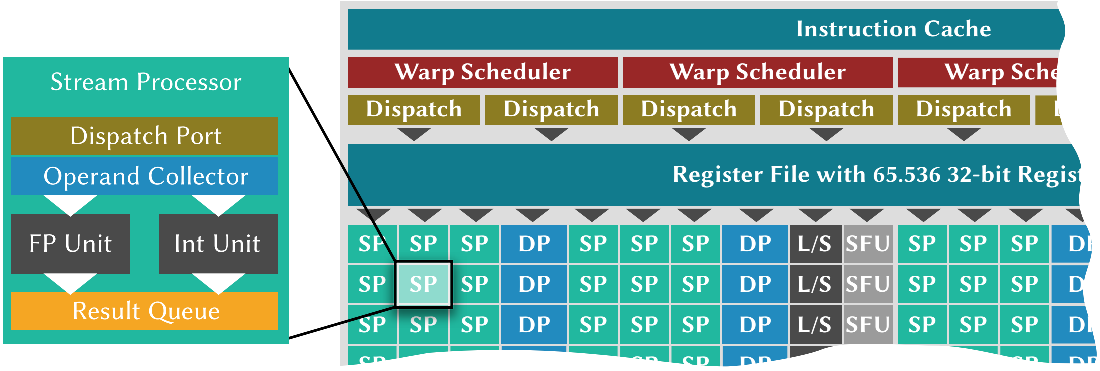
3.2 Streaming Multiprocessor Units

Multiprocessor units form the foundation of modern GPUs in which the arithmetic work is done. The grouping of processing cores and the required infrastructure into multiprocessors allows seamless scaling of GPU architectures and thus a broad range of support for embedded and mobile GPUs up to fully equipped GPUs for use in HPC applications. The streaming multiprocessor units (SMX) of the GK 110 architecture are dominated by 192 processing cores, each with a single precision floating-point (SP) and an integer arithmetic logic unit, as well as 64 cores with double precision floating-point (DP) and integer arithmetic units. These processing cores are accompanied by 32 special-function units (SFU) which provide fast approximate transcendental operations (e.g. $\sin(x)$), and 32 load-store units (L/S). Figure 3.2 depicts the block digram of a streaming multiprocessor unit.


Streaming Multiprocessor Unit (SMX)
Instruction Cache
Warp Scheduler
Warp Scheduler
Warp Scheduler
Warp Scheduler
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Dispatch
Register File with 65.536 32-bit Registers
SP SP SP DP SP SP SP DP L/S SFU SP SP SP DP SP SP SP DP L/S SFU
SP SP SP DP SP SP SP DP L/S SFU SP SP SP DP SP SP SP DP L/S SFU
SP SP SP DP SP SP SP DP L/S SFU SP SP SP DP SP SP SP DP L/S SFU
SP SP SP DP SP SP SP DP L/S SFU SP SP SP DP SP SP SP DP L/S SFU
⋮
In total: 192 single-precision (SP) cores, 64 double-precision (DP) units, 32 special function units (SFU), and 32 load/store units (L/S)
⋮
SP SP SP DP SP SP SP DP L/S SFU SP SP SP DP SP SP SP DP L/S SFU
SP SP SP DP SP SP SP DP L/S SFU SP SP SP DP SP SP SP DP L/S SFU
Interconnect Network
64 kB Shared Memory / L1 Cache
48 kB Read-only Data Cache
Tex Tex Tex Tex Tex Tex Tex Tex
Tex Tex Tex Tex Tex Tex Tex Tex

▲ **Figure 3.2** — NVIDIA GK 110 streaming multiprocessor unit (SMX) block diagram. Adapted from [NVIDI12].

The stream processing cores (SP/DP) consist of a dispatch port which receives instructions from the multiprocessor's instruction dispatchers and an operand collector which prepares the required input data elements. Each processor core is equipped with a floating-point processing unit and an integer processing unit. The floating-point units of SP cores work with 32-bit single precision arithmetic, the ones of the DP cores with 64-bit double precision arithmetic. Figure 3.3 shows the block diagram of an SP streaming processor core.



▲ **Figure 3.3** — NVIDIA GK 110 streaming processor unit (SP) block diagram.

The GK 110 GPU architecture is compliant with the IEEE Standard 754™ for floating-point arithmetic [IEEE 08, NVIDI14c]. It provides a fast fused multiply-add operation (FMAD), which combines the multiplication and addition of numbers in the form $a \cdot b + c$ into a single operation. Besides higher performance, the advantage of fused multiply-add operations is the improved accuracy of the computed results since only the final result $\circ(a \cdot b + c)$ is rounded instead of the twice rounded result $\circ(\circ(a \cdot b) + c)$ when standard multiplication and addition operations are used.

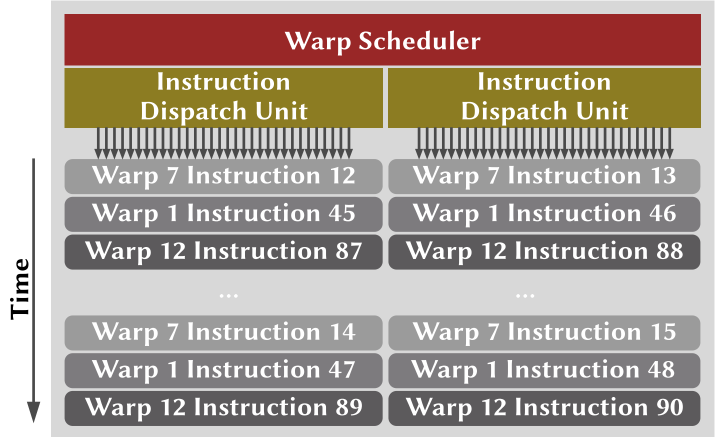
For the handling of data, the processing cores within a multiprocessor unit share 65.536 32-bit registers, a 64 kB shared memory and a 48 kB read-only cache memory for constant data. Fast communication among the hardware resources is realized by an interconnection network.

3.3 Thread Organization and Scheduling

Graphics processing units gain high performance through massive on-chip parallelism and hence large numbers of parallel executing threads. In contrast to contemporary multi-core CPUs, which typically handle less than 100 threads in parallel[2], this implies that several thousands of threads have to be organized and controlled by the GPU hardware. The thread control is therefore decentralized in GPU architectures and large parts are transferred to the multiprocessor units.

The GK 110 architecture partitions the control logic within a multiprocessor unit into four schedulers for groups of threads, so called *warps* [NVIDI12]. Each of these warp schedulers has two instruction issue units. This allows the concurrent issue and execution of four warps with up to two independent instructions dispatched per warp. Current implementations of the GK 110 architecture use a warp size of 32 threads and support a maximum of 64 warps per multiprocessor unit. With such a configuration, 2048 threads can be active per multiprocessor and a total of 30.720 threads per GPU. The schedulers use register score-boarding for long latency operations and predetermined latency information from the compiler to avoid data hazards within the data path.

Since accesses to lower-level memories, such as the global memory (see Section 3.4), are very expensive in terms of latency, the schedulers suspend waiting warps and choose execution-ready candidates among the remaining warps. Within a fixed sequence, always two instructions from the individual warps are sent to the processor cores. Figure 3.4 shows a single warp scheduler and its two instruction issue units with such a sequence of issued instructions. The organization and control of threads from the software perspective (programming model) is introduced in Section 3.5.



▲ **Figure 3.4** — Scheduling of thread groups (warps) within multiprocessors. Adapted from [NVIDI12].

[2] The IBM POWER8 architecture comprises 12 processor cores each with support for 8 threads [Fluhr14].

3.4 Memory Hierarchy

Modern graphics processing architectures feature rich memory hierarchies with different levels of embedded memories, which provide different capacities, access latencies and access paths for threads. Recent GPU architectures, which target applications in high-performance computing, offer single-error correction, double-error detection (SECDED) ECC-protection for register files, shared memories and caches, as well as for the global device memories. In addition, single-error correction through parity checks is available for certain caches.

The memory access granularity depends on the level of the memory resource. Registers can be individually and directly accessed by threads. Accesses of multiple threads to lower level caches and global device memories are typically bundled by the hardware into single memory operations.

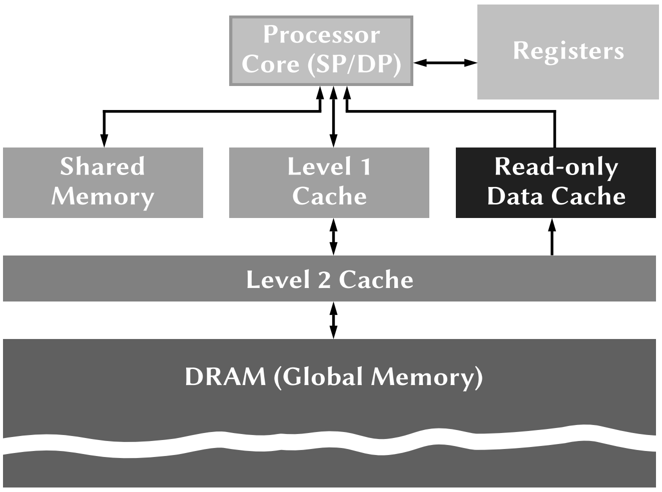
Within the GK 110 GPU architecture, each multiprocessor (SMX) comprises an ECC-protected register file with 65.536 32-bit registers. A single thread can address up to 255 registers as private local storage. The registers are managed by the compiler.

The next lower level within the memory hierarchy is a local memory of 64 kB capacity within each multiprocessor unit. This memory can be partitioned into a shared memory, which is then fully managed by the software, and a level-1 data cache, which is managed by the hardware. The partitioning is possible in shared memory to cache ratios of 32 kB:32 kB, 16 kB:48 kB and 48 kB:16 kB. The shared memory partition is accessible by all threads on a multiprocessor. The bandwidth of this memory for 64 bit and larger load operations is 256 B per core clock cycle, which yields up to 2.5 TB/s. This high bandwidth is achieved by internal partitioning of the shared memory into so called *banks* that can be accessed simultaneously. Memory read or write accesses to the shared memory with subsequent addresses that fall into different banks can therefore be processed in parallel. However, if addresses refer to the same bank, a *bank conflict* arises and the accesses are serialized, leading to a reduced throughput.

Besides the shared memory and level-1 cache, each multiprocessor is equipped with a parity-protected read-only cache of 48 kB capacity. This cache can be either managed automatically by the hardware or explicitly by the software.

In addition to the described memory resources, the GK 110 architecture provides a level-2 data cache of 1536 kB capacity. This cache is also ECC-protected and transparently managed by the hardware. Its main purpose is the unification and communication of data among the multiprocessor units. All accesses to the global device memory are routed through this cache, including accesses to the host system's main memory and the device memory of other GPUs within the system.

The lowest level within the GK 110 memory hierarchy is formed by the ECC-protected GDDR5 global device memory, which provides capacities of up to several gigabytes. The global device memory is accessible by all threads on the GPU, the host system's CPU, as well as other GPUs in the system. Thread accesses are bundled and the corresponding memory operations are issued per warp. The bandwidth of the global device memory reaches up to 250 GB/s, the typical access latency is between 400 and 800 core clock cycles. Figure 3.5 shows the global memory hierarchy of the GK 110 architecture, direct access paths are indicated by arrows.



▲ **Figure 3.5** — Memory hierarchy of the GK 100 GPU architecture with access paths of the threads. Adapted from [NVIDI12].

3.5 CUDA Programming Model

Parallel programming of computer systems has always been among the most challenging tasks in software development. The problem at hand needs to be partitioned in a way that separate workloads can be generated to utilize available parallel computer resources. Communication, memory access and synchronization among processes or threads have to be thoroughly managed to achieve high performance without race conditions or other conflicts.

With graphics processing units, this challenge has been taken to a new level. Thousands of processing cores are now available on a single chip to execute even larger numbers of threads in parallel. Exposed memory systems with many different storage options demand intelligent use to achieve high performance. This new level of the parallel programming challenge can only be mastered by suitable abstractions and programming models.

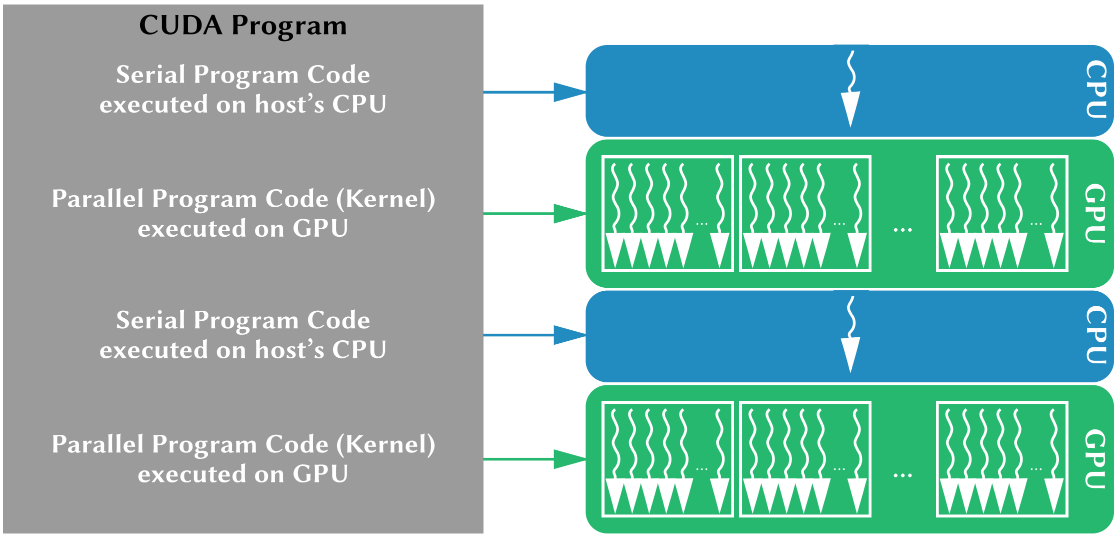
Modern programming models for graphics processing units like CUDA [NVIDI14a] and OpenCL [Khron14] provide the required abstractions to enable the efficient parallel programming of GPUs. The abstractions and basic concepts are thereby very similar. In the following, the Compute Unified Device Architecture (CUDA) is introduced, which is the natural choice for the Nvidia GK 110 GPU architecture. Moreover, CUDA forms the software platform for all implementation parts of this work.

The CUDA computing platform and parallel programming model was first introduced by Nvidia in November 2006. CUDA comprises a set of key abstractions for the handling of threads, memory accesses, as well as communication and synchronization. It introduces extensions to the C programming language to utilize these abstractions for the formulation of parallel programs. In addition, CUDA provides a full-featured software development ecosystem with compilers, debuggers and libraries for GPUs.

A CUDA C/C++ program consists of different code segments, which are either executed on the host system's CPU or on the GPU. Code segments which execute on the GPU are called *kernels*. Figure 3.6 depicts the structure of such a program consisting of host code and GPU kernels. Wavy arrows represent threads in the diagram.

Kernels

A CUDA GPU kernel is represented by a C function that starts with a *__global__* declaration specifier. The kernel code itself is written in standard C code using a small set of additional keywords, e.g. for the declaration of variables in shared memory or for the synchronization of threads. The number of parallel threads that are used to execute the kernel code on the GPU can be specified by the software developer using an *execution configuration syntax* which is described below. A CUDA program can consist of multiple different kernels.


CUDA Program
Serial Program Code executed on host's CPU
Parallel Program Code (Kernel) executed on GPU
Serial Program Code executed on host's CPU
Parallel Program Code (Kernel) executed on GPU
CPU
GPU
CPU
GPU

▲ **Figure 3.6** — Structure of a CUDA C/C++ program. Adapted from [NVIDI12].

Threads

The thread abstraction is one of the most important concepts in CUDA. It gives the software developer full control about all threads that execute kernel code. In CUDA, each thread has a unique ID which is accessible for the software developer through the built-in *threadIdx* variable. The threadIdx variable is a 3-component vector type which allows to identify threads in one-dimensional, two-dimensional and three-dimensional arrangements. This provides a flexible way for the software developer to adapt the thread arrangement to the problem at hand, for instance a two-dimensional thread formation for certain matrix operations.

As indicated by the threadIdx variable, threads are grouped into *thread blocks*, which can be one-dimensional, two-dimensional or three-dimensional. A thread block is executed on a single streaming multiprocessor unit (SMX). Contemporary graphics processing units support thread blocks with up to 1024 threads. Threads within a thread block can cooperate via the shared memory of the multiprocessor unit and synchronize their accesses to memory. Explicit synchronization points can be specified by the software developer using the *__syncthreads()* directive. In this case, all threads of the block have to wait until the last thread finishes (lightweight barrier synchronization).

In the one-dimensional case, the addressing of threads is done directly, the value of the threadIdx variable corresponds to the thread's index x. In the two-dimensional case with thread blocks of dimensions (D_x, D_y), the value of the threadIdx variable for a thread with index (x, y) is $(x + y \cdot D_x)$. Accordingly, in the three-dimensional case with thread blocks of dimensions (D_x, D_y, D_z), the thread ID is determined by $(x + y \cdot D_x + z \cdot D_x \cdot D_y)$. The dimensions of a thread block can be retrieved via the built-in *blockDim* variable.

Grids

To facilitate the organization of thread blocks, CUDA provides the abstraction of *grids of thread blocks.* A grid can have one, two or three dimensions. Each kernel is represented by a single grid of thread blocks, which in turn consists of all the threads that execute the kernel on the GPU.

A thread block within a grid can be uniquely identified by its thread block ID which is accessible through the built-in *blockIdx* variable. A general requirement for thread blocks is that they must be independent of each other and executable in any arbitrary order. This requirement is fundamental to allow the hardware a flexible scheduling of the blocks.

The execution of a GPU kernel requires a proper configuration which specifies the dimensions of the thread blocks and the shape of the grid. CUDA provides software developers with a *execution configuration syntax* of the form *<<<NumberOfBlocks, ThreadsPerBlock>>>* to define such execution configurations.

Memory

To ease the handling and management of the available GPU memory resources for the software developer, CUDA provides a set of variable type qualifiers as extensions to the standard data types of the C programming language. Variables that are declared without such qualifiers are typically mapped to registers by the compiler[3]

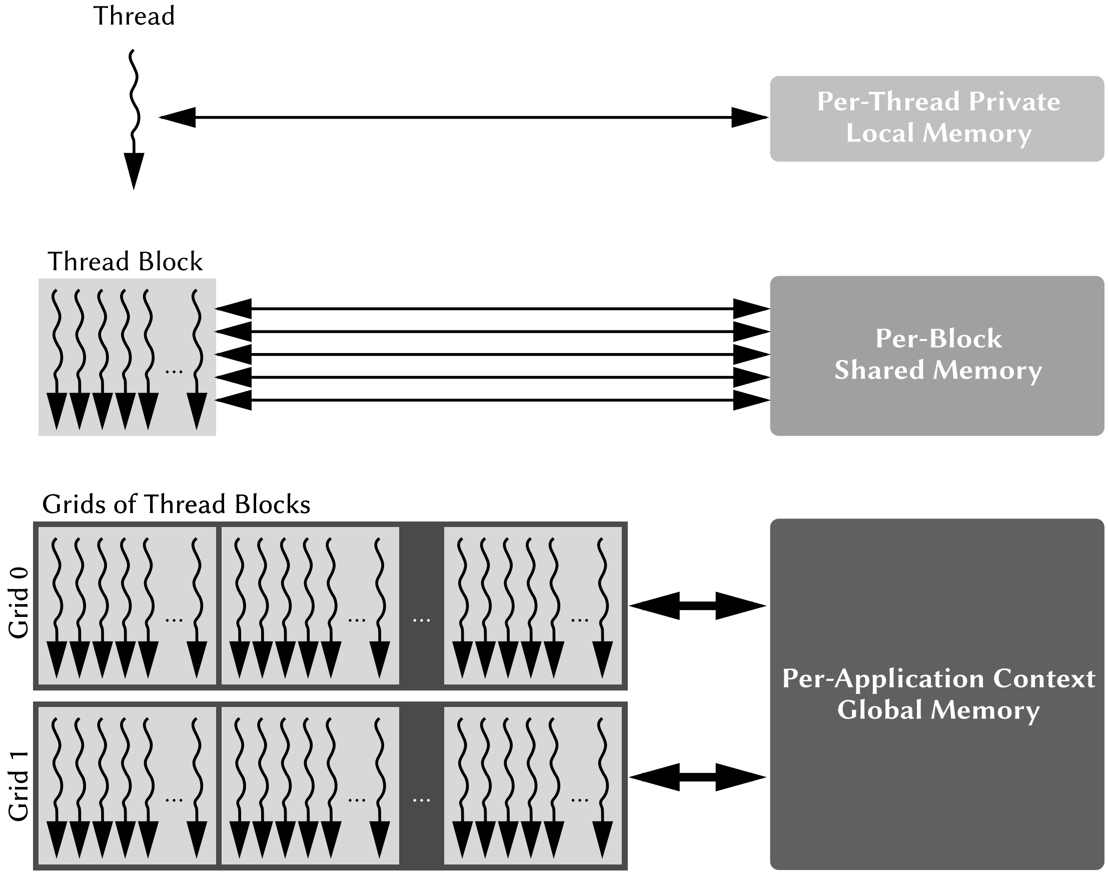
The *__device__* variable qualifier declares variables that are stored in the global device memory. Such variables are accessible by all threads that execute the kernel and have the same lifecycle as the application.

[3] This is true as long as enough free registers are available for a thread. In cases where the maximum number of registers is reached, the affected variables are outsourced to the device memory. This mechanism is called *register spilling*.

The __*constant*__ variable qualifier declares variables that are stored in the constant memory space which is a cached part of the global device memory. Variables in constant memory can be accessed by all threads and share the lifecycle of the application.

The __*shared*__ variable qualifier declares variables that reside within the shared memory of a multiprocessor unit. These variables are only accessible by the threads within the same thread block and they share the lifecycle of this block.

Figure 3.7 shows a diagram with the thread organization and the access to different levels of the memory hierarchy. For more details on the memory management, the reader is referred to the CUDA documentation in [NVIDI14a] and [NVIDI14b].



▲ **Figure 3.7** — Thread organization and memory access hierarchy. Adapted from [NVIDI12].

Chapter 4

Fault Tolerance for Graphics Processing Units

Modern graphics processing units do not only represent some of the most complex and most sophisticated contemporary processor architectures, they also belong to the class of nano-scaled devices, which are typically manufactured in latest CMOS technology. As introduced in Chapter 1.3, graphics processing units are therefore increasingly confronted with a rapidly growing spectrum of very different factors that threaten their reliability. This in turn makes the integration of fault tolerance mandatory.

Fault tolerance is a mature research area, which provides hardware and software solutions to exploit and manage redundancy in order to improve the reliability of systems. For the sake of conciseness, a detailed introduction to fault tolerance is omitted at this point. Profound introductions to the design and the most important concepts and methods of self-checking and fault tolerant systems can be found in [Koren07], [Lala01] and [Goess08].

The selection and combination of appropriate fault tolerance measures is often a non-trivial task, which requires detailed information about the vulnerability of the target system. This chapter therefore starts with a brief overview of common methods for the vulnerability assessment of GPUs and presents recent results, which strongly motivate the integration of fault tolerance measures. The chapter then puts its focus to fault tolerance measures that have been proposed for GPUs.

4.1 Vulnerability Assessment

Mainly three different options exist to assess the vulnerability of modern graphics processing units, for instance to transient effects. The first option are analytical methods such as the *Architectural Vulnerability Factor* (AVF) [Mukhe08], which allows to identify and distinguish components within a design by their susceptibility to soft errors. The second option are *simulation-based evaluations*, which range from gate-level or RT-level logic simulations combined with fault injections, up to system-level simulations. The third option are *physical experiments* such as the exposure of a circuit-under-test to different kinds of radiation.

Since floating-point processing is a core task of graphics processing units, a general structural hardware model and an integrated multi-level simulation environment have been developed in course of the preparations of this thesis. The hardware model is independent of vendor-specific hardware information, it provides transferable soft error data, and it offers large flexibility for the adaption to new hardware structures. The simulation environment utilizes gate-level logic simulation and fault injections to gain detailed soft error data. The hardware basis for the model is a synthesized netlist of an industry-grade floating-point unit (FPU), which complies with the IEEE Standard 754™ for floating-point arithmetic. Fault injections are performed exhaustively, which means that signal-line-flips or stuck-at-faults are injected into each gate of the FPU separately and not only into the state elements. Given that the structural model of the FPU consists of ca. 22.000 logic gates, 22.000 fault injections are performed for the characterization of a single operation.

Although hardware structures for floating-point processing typically exhibit a certain inherent fault tolerance, soft error characterizations of the FPU's four basic arithmetic operations FADD, FSUB, FMUL and FDIV showed that between 27.5% (FDIV) and 36.8% (FMUL) of all injected faults[1] caused an error in the FPU's internal state. Between 14.5% (FADD, FSUB) and 24.4% (FMUL) of such internal state errors lead to erroneous computational results. Additional results of these experiments have been presented in [Wunde13] and emphasize the vulnerability of floating-point processing logic.

In [Tan11], Tan et al. introduced *GPGPU-SODA*, a software framework for the soft error characterization of graphics processing units based on the GPU architecture

[1] For the referenced soft error characterizations, the FPU processed random input data between 10^{-3} and 10^{3}.

simulator *GPGPU-Sim* [Bakho09]. The framework allows the computation of Architectural Vulnerability Factors for components of the GPU microarchitecture like the warp scheduler, the processing cores, as well as the register file. The authors performed exhaustive analyses across different workloads based on examples from the Nvidia CUDA SDK [NVIDI15], the Rodinia Benchmark [Che09], and the Parboil Benchmark [Strat12]. The experimental results showed that the AVF of the warp scheduler, the register file and the processing cores achieved up to 76%, 95%, and 75%, respectively. Tan et al. concluded that the *"majority structures in GPGPUs (e.g. warp scheduler, streaming processors, and register file) are highly vulnerable"* and that *"a comprehensive technique is needed to protect them against soft error attacks"*.

In [Tselo13], Tselonis et al. evaluated the vulnerability of modern GPUs to permanent faults in the registers. Based on the cycle-accurate GPU architecture simulator *GPGPU-Sim* [Bakho09], the authors performed extensive fault injection experiments, where they injected stuck-at-zero and stuck-at-one faults at different positions of randomly chosen registers. A vector addition (vadd) and a matrix multiplication with and without shared memory utilization (mmult-sm/mmult) formed the benchmarks. Across all experiments, 33.98% (vadd), 34.98% (mmult-sm) and 39.94% (mmult) of the injected faults did not cause any error, which shows a rather high tolerance against permanent faults in registers. However, 65.89% (vadd), 61.48% (mmult-sm) and 43.99% (mmult) of the injections caused erroneous computational results. Tselonis et al. also investigated the impact of permanent faults in registers on the overall performance of the GPU. Their results showed that, even with significantly increasing numbers of affected registers in different streaming multiprocessors, the GPU performance degraded only moderately due to high thread-level redundancy.

With *GPU-Qin* [Fang14], Fang et al. introduced a method for the vulnerability assessment of GPU applications based on fault injections. The approach utilizes a GPU debugger and performs the fault injections at assembly-language level of GPU applications. It allows to model faults at the granularity of individual instructions. The authors selected 15 benchmark programs from the Nvidia CUDA SDK [NVIDI15] and the Rodinia Benchmark [Che09], which also included the matrix multiplication and the matrix transpose. The experimental results showed crash rates, i.e. the application did not finish execution, between 31% (matrix multiplication) and around 40% (matrix transpose), as well as silent data corruption (SDC) rates between 22% (matrix transpose) and 25% (matrix multiplication).

In recent years, several works have been published on radiation-based experiments for the vulnerability assessment of GPUs and GPU applications. In [Rech13] Rech et al. exposed graphics processing units (Nvidia GTX480) to different levels of neutron flux between $4 \cdot 10^4 \frac{n}{cm^2 \cdot s}$ and $1 \cdot 10^6 \frac{n}{cm^2 \cdot s}$, and evaluated the behavior of a matrix multiplication and a Fast-Fourier-Transform kernel. The authors reported cross sections[2], which, in case of the matrix multiplication, represent the probability for a neutron that hits the GPU during the matrix multiplication to cause an output error. A FIT rate (failures in time) of $2.81 \cdot 10^4$ was reported for the matrix multiplication and $5.17 \cdot 10^5$ for the FFT. For the matrix multiplication, a majority of 51.51% of the erroneous results was affected by multiple errors. Although the FIT rates appear low at first, the authors pointed out that such rates are intolerable for scientific calculations or even safety-critical applications. Extended experimental results for the Fast-Fourier-Transform can be found in [Saben14b].

In [Saben14a], Sabena et al. used a similar experimental setup to evaluate the vulnerability of GPU caches and shared memories. The target hardware for the experiments was an Nvidia Tegra 3 quad-core ARM9 CPU, which was equipped with an Nvidia Quadro 1000M GPU. The applied neutron flux was about $3.4 \cdot 10^5 \frac{n}{cm^2 \cdot s}$. The authors reported soft error cross sections ($\frac{cm^2}{device}$) of $1.42 \cdot 10^{-9}$, $2.48 \cdot 10^{-9}$ and $1.35 \cdot 10^{-8}$ for the L1 cache, the shared memory and the L2 cache, respectively.

In [Olive14], Oliveira et al. carried out radiation-based experiments to assess the vulnerability of the GPU main memory and the efficacy of error detecting and correcting codes (ECC) (see Section 4.2). The devices-under-test in these experiments were an Nvidia Tesla C2050 and an Nvidia K20 GPU, the neutron flux was about $1.0 \cdot 10^6 \frac{n}{cm^2 \cdot s}$ and $4 \cdot 10^4 \frac{n}{cm^2 \cdot s}$. The authors used a matrix multiplication, a Fast-Fourier-Transform and Hotspot [Che09] as benchmarks for the evaluations. The experimental results showed silent data corruption (SDC) FIT rates of $3.46 \cdot 10^4$, $4.8 \cdot 10^4$ and $1.62 \cdot 10^3$ for the matrix multiplications, the FFT and Hotspot, respectively. The authors also summarized all cases in which the GPU stopped responding and had to be reset under the term *functional interruption* (FI). The reported FI FIT rates were $1.49 \cdot 10^4$ (matrix multiplication), $1.17 \cdot 10^4$ (FFT) and $8.48 \cdot 10^2$ (Hotspot).

In summary, all vulnerability assessments that have been published so far, either being based on analytical methods, simulations or physical experiments, show that GPUs are

[2] The cross section for the matrix multiplication was obtained by dividing the number of observed errors per time unit by the flux.

highly vulnerable to soft errors, but also to permanent faults. The integration of fault tolerance measures is therefore of utmost importance in order to ensure the reliable use of graphics processing units in science and engineering, as well as safety-critical applications.

4.2 Memory, ECC and ABFT

Memory plays an essential role in modern GPU architectures, which typically incorporate larger main or device memories, L2 caches, L1 caches and shared memories, as well as large register files with several thousands of registers (cfg. Chapter 3). With exception of the main memory, all memories are directly distributed across the GPU chip. Since GPU memories are susceptible to soft errors, different fault tolerance measures have been proposed for their protection.

In [Maruy10], Maruyama et al. introduced a software framework that combines data coding for the detection of bit-flip errors in the main memory and checkpointing for the recovery. The approach targets GPUs without ECC-protected memories. For each 128-byte data block in memory, Maruyama et al. propose the application of a modified cross-parity code. Two words of parity bits from 32 words of protected data are computed. One word for the data in the given order and one word for the rotated words of the same data. The first parity word allows the detection of one single-bit error in a word, whereas the second word ensures that double-bit errors within a block are detectable. The encoding and checking of data words is intertwined with the GPU's standard load and store operations. To minimize the cost of error correction, the authors proposed to use a checkpointing scheme, which allows quick rollbacks in cases where errors are detected. The reported experimental results showed that the proposed approach reached up to 86% of the target GPU's unprotected data throughput. Like most encoding-based protection mechanisms for memories, the method proposed by Maruyama et al. is not able to detect errors that occur during computations. This is a significant difference to ABFT schemes like the A-Abft method proposed in this thesis. Moreover, with increasing hardware support for ECC-protected memories in modern GPUs, the potential benefits of this software-based ECC mechanism are diminishing.

Errors that occur in the computational logic of graphics processing units are typically not detected by memory protection mechanisms such as error detecting and correcting codes. This increases the risk of silent data corruptions (SDCs). In [Yim11], Yim et al.

introduced *HAUBERK*, a lightweight method for the detection of SDCs in GPU programs. The authors propose two types of error detectors, namely, duplication and checksum variables for non-loop portions of GPU kernels, and accumulation-based value range checking for the protection of loop portions. The authors report an average performance overhead of 15.3% for different benchmark GPU kernels equipped with *HAUBERK*. For the error detection coverage, 86.8% are reported, leaving a fraction of 13.2% of injected faults leading to SDCs. The error detectors proposed by Yim et al. are orthogonal to the ABFT method proposed in this thesis. However, the *HAUBERK* mechanisms can be integrated into such an ABFT scheme to protect the control part and to ensure that the different ABFT phases (e.g. encoding, rounding error determination, result checking, etc.) are executed correctly.

In recent generations of professional graphics processing units targeting the high-performance computing (HPC) market, vendors like Nvidia started to integrate single error correction, double error detection (SECDED) memory protection. According to [NVIDI12], ECC-protection can be activated for register files, shared memories, L1 and L2 caches, as well as the GDDR main memory. Read-only caches are reported to be protected by single error correction parity codes. Although no detailed technical information on the ECC-protection is disclosed by the vendors, it is highly likely that (72,64)-Hamming SECDED codes [Koren07, Lala01] are implemented in these GPUs, which use 8 additional check bits for the protection of 64 data bits. In typical hardware implementations, memory lines are extended to hold 72 bits and the 8 check bits are transparently computed by the memory controller. In [NVIDI13], Nvidia reports that the activation of the SECDED ECC-protection will cause a reduced overall performance and a reduction of the available main memory by about 6.25%, which suggests a hybrid hardware/software implementation of the memory protection.

Memory protection mechanisms based on SECDED ECC and Algorithm-Based Fault Tolerance operate at very different layers of the system stack. ECC-protection for memories is typically implemented at lower hardware layers and operates transparently for the user, independent of upper system layers. Algorithm-Based Fault Tolerance techniques like the proposed A-Abft, in contrast, operate at the top layer of the system stack. Nonetheless, both techniques share the basic principle of utilizing information redundancy by encoding the data they protect. Data integrity is their joint goal. However, ECC-protection schemes for memories can only detect and to a certain extent correct errors that occur within the memory, for instance caused by a transient event. Errors that occur in the computational logic and propagate to the memory cannot be detected. Since ABFT schemes like A-Abft have more knowledge on the semantics of the data they protect, they are able to detect and to certain extent correct computational errors, as well as memory errors[3].

ECC-protection for memories is therefore no replacement for Algorithm-Based Fault Tolerance schemes and vice versa. In contrast, both techniques have to be applied in conjunction to achieve a maximum level of protection. Latest experimental results from radiation-beam tests in [Olive14] clearly showed, that ECC-protection for GPU memories is very effective and reduces the silent data corruption (SDC) rate by an order of magnitude. The same experimental results also showed, that the integration of Algorithm-Based Fault Tolerance schemes, on top of ECC, enables a further reduction of the SDC rate by another order of magnitude. Especially the matrix multiplication that has been equipped with ABFT showed an SDC rate of less then 2% of the unprotected version.

First works such as [Li13], present cooperative approaches, which closely integrate ECC-protection and Algorithm-Based Fault Tolerance, and which move into the direction of modern cross- or multi-layer approaches to dependability.

To enable a seamless realization of hybrid schemes using ECC-protection and ABFT, the used ABFT schemes have to be designed to operate autonomously and in a transparent manner for the user—just like hardware-based ECC memory protection. The A-Abft method introduced in this thesis fulfills this requirement and is hence very well suited for such hybrid approaches.

4.3 Checkpointing

Checkpointing is a classic fault tolerance technique [Koren07], which is widely used, for instance, in scientific high-performance computing, to protect complex and long-running applications. The basic idea of checkpointing is to reduce the error correction overhead by taking snapshots of the system or application state in regular intervals and storing them on a stable storage medium such as hard disk drives. In case of errors, an application does not have to be restarted from the beginning, it suffices to load the last checkpoint and continue execution from there. Depending on the application,

[3] Under the assumption that the encoding and checking procedures of an ABFT scheme are executed correctly.

checkpoints can be rather large, since they have to record all relevant information that is required to restart from the saved state.

In [Takiz09], Takizawa et al. introduced *CheCUDA*, a software framework for Nvidia GPUs, which utilizes driver API calls in order to generate checkpoints from the GPU state. *CheCUDA* copies all user data from the GPU main memory into the host's (CPU) main memory. It then completely deletes all existing memory entries on the GPU. In a subsequent step, *CheCUDA* writes the current state of the GPU application, including the part running on the host, into a checkpoint file on the system's hard disk drive. After the checkpoint has been written, *CheCUDA* re-initializes the GPU and recreates the context saved in the checkpoint by copying all data back to the GPU. By storing the complete GPU application state in the checkpoint file, *CheCUDA* does not only allow classic checkpointing and restart, it also allows the migration of complete GPU tasks to other GPUs. Takizawa et al. implemented *CheCUDA* as add-on to the Berkeley Labs Checkpoint Restart (BLCR) package [Hargr06]. The provided experimental evaluations of *CheCUDA* showed runtime overheads of up to almost 100%, especially in cases where many memory resources were allocated on the GPU.

Xu et al. proposed a hierarchical application-level checkpointing method called *HiAL-Ckpt* [Xu10], which is tailored for hybrid CPU-GPU systems. The proposed method performs classic checkpointing hierarchically and separately for the CPU and the GPU. The state of the CPU application is regularly recorded into checkpoints that are stored on the system's hard disk drive. The state of the GPU application is saved in shorter intervals into checkpoints within the CPU main memory. *HiAL-Ckpt* provides compiler directives for the checkpointing that have to be placed manually by the developer, who also has to set the checkpointing intervals. Xu et al. provided an experimental evaluation of *HiAL-Ckpt* based on a single benchmark (SWIM) from the Spec200 benchmark suite. *HiAL-Ckpt* checkpointing directives have been placed into loops within the benchmark's source code. GPU checkpoints have been recorded every 100 loop iterations, whereas CPU checkpoints have been recorded every 500 iterations. The authors reported 2.5% execution time overhead for the checkpointing and 0.19% and 0.1% execution time overhead for the recovery after a CPU fault or a GPU fault, respectively.

In [Laoso11], Laosooksathit et al. proposed a two-phase checkpointing protocol for graphics processing units, which utilizes latency-hiding features of modern GPUs, like Nvidia CUDA Streams [NVIDI14a]. CUDA Streams allows the overlapping of memory transfers and computations, and hence reduces idle times. The basic principle of the proposed checkpointing protocol is similar to *HiAL-Ckpt* [Xu10], with the difference that the GPU continues with the kernel execution, while the checkpoint information is transferred to the host. Checkpoint data from the GPU is first kept in the host's main memory and after completion of the GPU-to-CPU memory transfer written into a checkpoint file on the system's hard disk drive. Failures that occur during kernel execution compromise and destroy the application context and the application data on the GPU. To restore the application, the proposed protocol has to recreate the application context and to copy back the data from the checkpoint file. Laosooksathit et al. used a matrix addition for the experimental evaluation of their checkpointing protocol, and showed that it induces lower execution time overhead than a variant without overlapping memory transfers. The overhead was especially for large problem sizes smaller. In comparison to the non-streamed checkpointing, the proposed protocol required between 0.2 ms and 44.0 ms less execution time using four streams, and 2.0 ms to 48.5 ms less execution time using eight streams.

The applicability of the proposed GPU checkpointing techniques is more general than the applicability of Algorithm-Based Fault Tolerance schemes like A-Abft, since modifications of the target algorithm are not required. However, with sophisticated scientific applications, the amount of data that has to be stored for each checkpoint can quickly grow very large. This can be particularly important on GPUs, where computations are typically favored over memory transfers, which means that the re-computation or correction of an erroneous result (e.g. ABFT) is often more attractive than a complete rollback that involves substantial memory transfers. However, for applications that consist of ABFT-protectable and non-ABFT-protectable parts, combinations of checkpointing and ABFT schemes like A-Abft can be very attractive [Bosil15].

4.4 Software-Based Self-Test

The detection of errors is an essential prerequisite for all fault tolerance measures in order to be effective. As summarized in Chapter 1.3, the reliability of modern graphics processing units is not only threatened by transient events that cause soft errors, but also by defects, latent faults, damages caused by the environment, and wear-out effects.

Hardware test methods [Bushn00] are therefore extensively used during and directly after the manufacturing process to identify defective and faulty parts. These test methods typically apply sets of stimuli (test patterns) to a circuit-under-test and compare the circuit's test responses to a golden reference. The test of modern semiconductor devices is a complex and challenging task, which requires appropriate test infrastructure on the chip, as well as expensive external test hardware such as automated test equipment (ATE). In addition, hard constraints like a maximum fault coverage, a minimum test power and a minimum test time often have to be met.

Software-Based Self-Test (SBST) [Shen98, Chen01, Chen03, Corno03, Gizop04, Bayra06, Singh06, Gurum06, Wen06, Zhou06, Psara10] represents a class of test methods, which utilize the instruction set architecture (ISA) and other processor resources, such as instructions, functional units and memories, to test microprocessors and associated components. The basic principle behind SBST is to map and transform test patterns into executable software programs, which then apply these test patterns and check the test responses. SBST is non-intrusive and it can be applied without requiring additional hardware test infrastructure. SBST can be applied at-speed and it allows the periodic online testing of hardware in the field [Pasch05, Sanch05, Krani06].

Software-based self-test methods evolved within the last decades from the test of single integer arithmetic units [Pasch05] and floating-point units [Xenou09], to the test of complex systems-on-chip (SoC) and parallel multi-core processors [Corno03]. SBST is therefore particularly interesting for GPU fault tolerance measures since it allows to gain required information on the health status of the GPU hardware. This information, in turn, can then be utilized by the fault tolerance measures.

In [Kalio14], Kaliorakis et al. performed in-depth analyses on the parallelization and application of SBST programs on massively parallel many-core architectures. Since every processor core within such an architecture has to execute the test program and since there is no communication required between the cores during the test, the parallelization of such SBST programs appears trivial. However, the experimental results showed that a straight-forward parallelization of the test programs did not lead to satisfactory speedups. The authors explained the poor performance with an overextension of the architecture's memory system and on-chip interconnection network, since the test programs are highly memory-intensive. Based on their experimental results, Kaliorakis et al. proposed a new method for the parallelization of SBST programs on many-core architectures, which focuses on the test preparation phase and the utilization of fast inter-core communication channels. The method partitions the overall test pattern set into as many subsets as processor cores exist. Each processor core stores its fraction of test patterns in a fast local memory and starts the execution of the SBST program with these patterns. In a subsequent step, the remaining parts of the test pattern set are loaded via a fast inter-core communication channel from the other cores. Kaliorakis et al. reported speedups of up to 47.6% for SBST programs parallelized with the proposed method.

Di Carlo et al. [Di Ca13a] utilized software-based self-tests for the detection and localization of hardware faults in Nvidia Fermi GPUs. The proposed method allows a fine-grained fault detection by exploiting state-of-the-art and custom SBST methods for all components within a streaming multiprocessor (SMX) of the GPU. The SBST programs are realized as GPU kernels that are executed by parallel instances on the GPU. Each component within the streaming multiprocessor unit (e.g. integer units, floating-point units, special-function units, etc.) is covered by a single kernel. For each instance of a test kernel, the test responses are computed on the GPU and each executing thread writes its unique ID into the test response. The computed test responses are then transferred back to the CPU, where they are compared to precomputed golden references. In cases of a mismatch, a hardware fault is detected and subsequently localized by the thread ID and the ID of the streaming multiprocessor unit. For the test of the integer units, Di Carlo et al. use an SBST method proposed by Paschalis et al. [Pasch01], the floating-point units are covered by an SBST method proposed by Xenoulis et al. [Xenou09]. The special-functional units that are used to compute transcendental mathematical operations, are tested by a custom kernel which exercises all supported operations on pseudo-random input patterns. Di Carlo et al. performed experimental evaluations of their SBST method based on fault injections. The reported percentages of detected faults range from 99.9% for the integer unit and 99.8% for the floating-point unit to 93.16% for the special-function units.

Algorithm-Based Fault Tolerance methods like A-Abft and Software-Based Self Test methods are orthogonal and not directly related. However, combinations of both techniques can be used to further improve the reliability of GPU-accelerated scientific computations. SBST-kernels that are scheduled regularly on the GPU in-between ABFT-protected operations can, for instance, help to monitor the health state of the hardware.

4.5 Redundant Hardware and Scheduling

The exploitation of redundancy is a key characteristic of fault tolerance measures. In [Sheaf07], Sheaffer et al. investigated the scope of hardware redundancy that is required to enable fault tolerant computations on GPUs. The authors proposed hardware modifications, including comparison logic and new data-paths for the signaling of errors, which allow a dual-issue of thread blocks, exploiting temporal locality and data cache effects. The approach replicates incoming data elements and schedules thread blocks twice. The computational results of both runs are compared afterwards.

In [Dimit09], Dimitrov et al. evaluated different options for application-level redundant scheduling. The first technique (*R-Naïve*) uses duplication with comparison, and schedules GPU kernels twice, before the computational results are compared. The second proposed scheduling (*R-Scatter*) tries to exploit unused instruction-level parallelism within the GPU kernel code, whereas the third technique (*R-Thread*) utilizes available thread-level parallelism. While the *R-Naïve* scheduling can be applied without restrictions on all GPU architectures, the proposed *R-Scatter* method exploits free instruction slots due to data dependencies and places redundant instructions within these slots. This approach showed higher benefit on VLIW-like GPU architectures as they are found in older AMD GPUs. With the *R-Thread* scheduling, each GPU thread does the same amount of work as in the original GPU kernel. However, the number of allocated thread blocks per kernel is doubled. The extra thread blocks perform the redundant computations and are scheduled together with the thread blocks that execute the original code. If a GPU application does not fully utilize the GPU, the redundantly scheduled thread blocks cause less overhead compared to the duplication approach *R-Naïve*. Dimitrov et al. experimentally evaluated the three scheduling approaches and reported execution times for *R-Naïve*, which are 199% of the original execution times. *R-Scatter* caused between 40% and 110% execution time overhead, and *R-Thread* caused 97% overhead on average.

Inspired by simultaneous multi-threading, Backer and Karri [Backe12] proposed a scheduling scheme called *Partial Warp Redundancy* (PWR), which replicates threads and assigns them to different stream processors of the GPU to be executed in lockstep. The method targets transient and permanent faults. Threads within a warp are replaced by redundant threads from other thread blocks. In order to achieve a high coverage, the authors proposed to replace half of the threads within a warp. To reduce the performance penalty, Backer and Karri introduced two different versions of their scheduling, E-PWR and C-PWR, which use different time intervals after which the computed results are compared. Experimental evaluations based on examples from the Nvidia CUDA SDK [NVIDI15] have been performed using a GPU simulator. For the fault injection experiments, the authors reported 63.91% detected faults for PWR with constant comparison of the results. For E-PWR, 33.92% and for C-PWR 44.95% detected faults are reported. Regarding the performance, overheads of up to 44.65% are reported.

Jeon et al. [Jeon12] investigated the utilization of threads within a warp for different GPU benchmarks from the Nvidia CUDA SDK [NVIDI15], the Parboil Benchmark [Strat12] and the ERCBench benchmark suite [Chang10]. For the majority of evaluated GPU programs, the authors reported that not all threads within a warp are fully utilized over the runtime. Based on these findings, Jeon et al. proposed *Warped-DMR*, a lightweight error detection method for graphics processing units, which opportunistically exploits spatial and temporal redundancy. *Warped-DMR* comprises two different techniques, Intra-Warp DMR and Inter-Warp DMR, which enable the detection of errors within and between warps. The Intra-Warp DMR approach uses inactive threads within a warp to verify the computations of the active threads of the same warp. The technique imposes only small performance overhead and requires only small hardware modifications to enable the comparison of the duplicated computational results. The second proposed technique, Inter-Warp DMR, is used when all threads within a warp are fully utilized and a redundant execution within the warp is not possible. In this case, dual-modular temporal redundancy is exploited by duplicate scheduling of each fully utilized warp. A hardware queue is used to buffer warps until they are scheduled. Thread shuffling is used to change the thread-to-core assignment, so that each redundant thread is executed by a different processor core, compared to the original thread. Jeon et al. evaluated the proposed *Warped-DMR* method with a GPU simulator and reported up to 96.43% error coverage and a worst case performance overhead of 16%.

In [Dweik14], Dweik et al. proposed *Warped-Shield*, an approach for tolerating permanent hardware faults in streaming multiprocessor units of GPUs. The approach comprises two techniques, *thread shuffling* and *dynamic warp deformation*, which are motivated by potential under-utilizations of streaming multiprocessors. Based on a fault map, which indicates faulty stream processors within a multiprocessor unit, and which has to be determined beforehand by an appropriate error detection technique, thread shuffling reroutes threads, which are scheduled on faulty stream processors, to healthy processors. After execution, the threads are rerouted to the original processor cores in order to write their results to the correct register file banks. The proposed thread shuffling requires modifications of the hardware in form of two multiplexer stages. The first stage, called shuffling-MUXes, is introduced right after the instruction issue unit and before the stream processors. The second stage, called reshuffling-MUXes, is introduced between the last execution stage and the write-back stage. Thread shuffling operates intra-warp and relies on idle stream processors. In cases where the GPU program fully utilizes the warps, thread shuffling is not possible. In these cases, the dynamic warp formation technique is applied, which splits a given fully utilized warp into a number of sub-warps with a reduced number of active threads. The sub-warps are scheduled consecutively. After the last sub-warp finished execution, the original warp is reassembled and the results for all active threads are written back to the register file. The dynamic warp deformation requires a modified warp scheduler, which utilizes the fault map information to generate the sub-warps. Dweik et al. evaluated the proposed *Warped-Shield* method using the GPU architecture simulator *GPGPU Sim* and benchmarks from ISPASS [Bakho09], the Parboil Benchmark [Strat12] and the Rodinia Benchmark [Che09]. In worst case expriments, where 75% of the stream processors were faulty, the proposed technique guaranteed the correct execution of the benchmarks with an average performance overhead of 35.5%. In another scenario with 50% faulty stream processors, an average performance overhead of 7% is reported.

In [Di Ca13b], Di Carlo et al. introduced a software-level fault mitigation strategy for Nvidia GPUs, which targets permanent faults in the streaming multiprocessor units. The technique relies on fault information (fault maps), which indicate faulty multiprocessors. The required fault map information is assumed to be gained by periodically executed functional tests [Di Ca13a]. GPU kernels that are instrumented with the proposed method exploit the fault map to identify thread blocks that will be scheduled to faulty multiprocessors. The execution of these thread blocks is then cancelled. The execution of a GPU kernel is considered to be correct, if every thread block has been scheduled to a healthy multiprocessor unit. If there remains at least one thread block whose execution has been cancelled, the method creates several copies of this thread block and re-schedules the copies. With more than one copy per thread block, the method guarantees that each of the thread blocks is executed on at least one healthy multiprocessor. The process of re-scheduling copies is repeated until no thread blocks are left. The authors evaluated the proposed mitigation strategy for two GPU kernels from the Nvidia CUDA SDK [NVIDI15] (matrix multiplication and histogram computation) with different numbers of faulty multiprocessor units. The experimental results showed that the instrumented GPU kernels exhibited no performance overhead in case of zero faulty multiprocessors. Even with increasing numbers of up to 7 faulty multiprocessor units, the instrumented kernels delivered correct computational results and still outperformed a reference implementation on the CPU.

4.6 Discussion and Differentiation from ABFT

This chapter briefly reviewed methods for the vulnerability assessment of modern graphics processing units and presented recent experimental results, which strongly emphasize the need for efficient and effective fault tolerance measures for GPUs. The chapter then introduced the most prominent non-ABFT fault tolerance techniques that have been proposed for graphics processing units so far.

With the growing support of ECC-protection mechanisms for GPU memories, one of the most critical vulnerability issues of contemporary GPUs is increasingly addressed by the major vendors. This renders techniques like the software-based ECC in [Maruy10] more and more obsolete, although computing systems built upon consumer-grade graphics processing units without ECC might still benefit to a certain extent from them. As discussed in Section 4.2, ECC-based memory protection and the A-Abft method proposed in this thesis do not conflict, in contrast, they supplement each other and can be efficiently combined to achieve a stronger protection of data in memory. Fault tolerance techniques such as *HAUBERK* [Yim11], which try to protect the computational portions of GPU kernels in order to reduce silent data corruptions can also be directly combined with ABFT measures like A-Abft in a cooperative way.

Checkpointing techniques are typically realized at system- or application-level and operate with another granularity than A-Abft. They are also often associated with a significantly higher performance overhead. While ABFT is intended to protect single mathematical tasks, checkpointing is often used to cover multiple such operations. Nonetheless, combinations of both techniques can be attractive to cover complete applications, which might comprise non-ABFT-protectable parts.

Software-Based Self-Test methods are particularly important for the fault tolerance of graphics processing units since they can provide a link to the health status of the hardware for ABFT in cross- or multi-layer approaches to dependability. Information on faulty functional units or memories can be incorporated by ABFT schemes, for instance to adapt encoding and checking procedures.

Approaches based on redundant scheduling typically induce a significantly higher performance overhead compared to ABFT-based solutions like A-Abft. In addition, techniques for the utilization of unused instruction-level parallelism, as proposed in [Dimit09], are becoming increasingly less attractive with current generation graphics processing units. These GPU architectures offer sophisticated thread scheduling mechanisms that support the concurrent execution of multiple different kernels to achieve a significantly higher utilization of the parallel hardware. However, more recent redundant scheduling approaches, which exploit the underutilization of warps, can be an attractive companion to A-Abft, especially for the encoding and checking procedures, where typically not all threads on the GPU are fully utilized.

Chapter 5

Algorithm-Based Fault Tolerance

Algorithm-based Fault Tolerance (ABFT) is an algorithm-level fault tolerance technique originally introduced by Huang and Abraham in [Huang82] and [Huang84]. The basic idea of ABFT comprises three major aspects:

- First, the input data for the target algorithm is encoded before it is processed.
- Second, the target algorithm is modified to work on encoded input data and to produce encoded output data.
- Third, the computational steps within the target algorithm are distributed among the available compute resources.

The encoding of input data is achieved by application of linear block codes. However, in contrast to the conventional application of such codes at the word- or bit-level, the encoding in ABFT takes place at the higher level of the target algorithm's complete input data set.

Although not as generally applicable as conventional fault tolerance techniques such as triple modular redundancy (TMR), Algorithm-based Fault Tolerance is able to significantly improve the reliability of essential mathematical operations with a comparatively low overhead. This makes ABFT particularly attractive for the protection of linear algebra matrix operations on graphics processing units.

This chapter introduces the reader to the field of Algorithm-based Fault Tolerance. Due to the abundance of published research on this topic, the chapter focuses on the important basic principles and the areas of the state of the art most relevant for this work.

5.1 Introduction to ABFT for Matrix Operations

This section introduces the basic principles and definitions of Algorithm-based Fault Tolerance for linear algebra matrix operations according to [Huang82] and [Huang84], followed by a more general formulation of the method which provides the link to the coding theory foundations in Chapter 2.

Algorithm-based Fault Tolerance for matrix operations performs the encoding of input data by computing checksums for the elements within the rows and columns of input matrices. The computed checksums are stored in additional columns and rows. The following definitions introduce the notions of *column-*, *row-* and *full-checksum matrices.*

Definition 5.1 (Column-checksum matrix) *Given an $(m \times n)$-matrix $\boldsymbol{A}$ consisting of m rows and n columns. The corresponding* ***column-checksum matrix $\boldsymbol{A_{cc}}$*** *is an $(m + 1 \times n)$-matrix which consists of the matrix $\boldsymbol{A}$ in the first m rows and a* ***column summation vector*** *or* ***column-checksum vector*** *in the $(m + 1)$-th row:*

$$\boldsymbol{A_{cc}} := \left[\begin{array}{ccc} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \\ \hline \boldsymbol{a_{m+1,1}} & \cdots & \boldsymbol{a_{m+1,n}} \end{array}\right]. \tag{5.1}$$

The ***checksum elements*** *of the column-checksum vector are computed as*

$$\boldsymbol{a_{m+1,j}} := \sum_{i=1}^{m} a_{i,j}, \quad \textit{for } 1 \le j \le n. \tag{5.2}$$

In literature, column-checksum matrices are often represented as

$$\boldsymbol{A_{cc}} := \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{e}^T \boldsymbol{A} \end{bmatrix}, \tag{5.3}$$

where $\boldsymbol{e}^T$ *is the* $(1 \times m)$ *unit vector* $[1, 1, 1, ..., 1]$ *and* $\boldsymbol{e}^T \boldsymbol{A}$ *is the column-checksum vector.*

Definition 5.2 (Row-checksum matrix) *Given an* $(n \times q)$*-matrix* $\boldsymbol{A}$ *consisting of* n *rows and* q *columns. The corresponding* ***row-checksum matrix*** $\boldsymbol{A}_{rc}$ *is an* $(n \times q + 1)$*-matrix which consists of the matrix* $\boldsymbol{A}$ *in the first* q *columns and a* ***row summation vector*** *or* ***row-checksum vector*** *in the* $(q+1)$*-th column:*

$$\boldsymbol{A}_{rc} := \left[\begin{array}{ccc|c} a_{1,1} & \cdots & a_{1,q} & \boldsymbol{a}_{1,q+1} \\ \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,q} & \boldsymbol{a}_{n,q+1} \end{array}\right]. \tag{5.4}$$

The ***checksum elements*** *of the row-checksum vector are computed as*

$$\boldsymbol{a}_{j,q+1} := \sum_{j=1}^{q} a_{i,j}, \quad \text{for } 1 \leq i \leq q. \tag{5.5}$$

Row-checksum matrices can also be represented as

$$\boldsymbol{A}_{rc} := \begin{bmatrix} \boldsymbol{A} & \boldsymbol{A}\boldsymbol{e} \end{bmatrix}, \tag{5.6}$$

where $\boldsymbol{e}$ *is the* $(q \times 1)$ *unit vector* $[1, 1, 1, ..., 1]$ *and* $\boldsymbol{Ae}$ *is the row-checksum vector.*

Definition 5.3 (Full-checksum matrix) *Given an* $(m \times q)$*-matrix* $\boldsymbol{A}$ *consisting of* m *rows and* q *columns. The corresponding* ***full-checksum matrix*** A_{fc} *is an* $(m + 1 \times q + 1)$*-matrix which consists of the matrix* $\boldsymbol{A}$ *in the first* m *rows and first* q *columns, and which has a* ***column-checksum vector*** *in the* $(m+1)$*-th row and a* ***row-checksum vector*** *in the* $(q+1)$*-th column:*

$$\boldsymbol{A}_{fc} := \left[\begin{array}{ccc|c} a_{1,1} & \cdots & a_{1,q} & \boldsymbol{a}_{1,q+1} \\ \vdots & \ddots & \vdots & \vdots \\ a_{m,1} & \cdots & a_{m,q} & \boldsymbol{a}_{m,q+1} \\ \hline \boldsymbol{a}_{m+1,1} & \cdots & \boldsymbol{a}_{m+1,q} & \boldsymbol{a}_{m+1,q+1} \end{array}\right]. \tag{5.7}$$

The ***checksum elements*** *of the checksum vectors are computed according to Equation 5.2 and Equation 5.5, respectively.*

The full-checksum matrix can also be represented as

$$A_{fc} = \begin{bmatrix} A & Ae \\ e^T A & e^T Ae \end{bmatrix}. \tag{5.8}$$

A great advantage of Algorithm-based Fault Tolerance is the continuous accessibility of the original input data matrices even in the encoded state, which can be easily seen by the abbreviated representations of the checksum matrices in Equations 5.3, 5.6 and 5.8. Typical applications of Algorithm-based Fault Tolerance store matrices in the full-checksum format and manipulate them accordingly in the full-, column-, or row-checksum format.

Huang and Abraham introduced Algorithm-based Fault Tolerance for five matrix operations [Huang84], namely the multiplication of two matrices, the multiplication of a matrix and a scalar, the addition of two matrices, the matrix transpose, and the LU decomposition of a matrix. Each of these operations preserves the checksum property of the encoding.

Over time, a multitude of Algorithm-based Fault Tolerance schemes has been proposed for essential mathematical operations. A selection of the most important operations includes the Gaussian Elimination [Baner88, Luk88, Baner90, Roy C93, Du11], the LU decomposition [Huang84, Luk86, Luk88, Turmo00b, Davie11, Du11], the QR factorization [Jou86, Luk88, Reddy90, Roy C93], Singular Value decomposition [Chen86, Turmo00b], iterative Conjugate Gradient solvers [Aykan87, Obori11, Shant12], as well as the Fast Fourier Transfom [Jou85, Malek85, Baner88, Sha94, Wang94, Turmo00b].

Error Detection, Localization and Correction

To introduce the error detection, localization and correction procedures of ABFT, the example of the matrix multiplication is considered by means of the following theorem from [Huang84]:

Theorem 5.1 (ABFT matrix multiplication) *Given an $(m \times n)$-matrix A and an $(n \times q)$-matrix B, which are encoded to the $(m + 1 \times n)$ column-checksum matrix A_{cc}*

and the $(n \times q+1)$ *row-checksum matrix* $\boldsymbol{B_{rc}}$.

1. *The result of a column-checksum matrix* $\boldsymbol{A_{cc}}$ *multiplied by a row-checksum matrix* $\boldsymbol{B_{rc}}$ *is a full-checksum matrix* $\boldsymbol{C_{fc}}$:

$$\boldsymbol{C_{fc}} := \boldsymbol{A_{cc}} \cdot \boldsymbol{B_{rc}} \tag{5.9}$$

$$\begin{bmatrix} \boldsymbol{C} & \boldsymbol{Ce} \\ \boldsymbol{e}^T\boldsymbol{C} & \boldsymbol{e}^T\boldsymbol{Ce} \end{bmatrix} := \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{e}^T\boldsymbol{A} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{B} & \boldsymbol{Be} \end{bmatrix}. \tag{5.10}$$

2. *The corresponding information matrices* $\boldsymbol{A}$, $\boldsymbol{B}$, *and* $\boldsymbol{C}$ *have the following relation:*

$$\boldsymbol{C} := \boldsymbol{A} \cdot \boldsymbol{B}. \tag{5.11}$$

The interested reader can find the proof of Theorem 5.1 in [Huang84].

Definition 5.4 (Reference checksum) *Given an* $(m+1 \times q+1)$ *full-checksum matrix* $\boldsymbol{C_{fc}}$ *which comprises of the* $(m \times q)$ *information matrix* $\boldsymbol{C}$ *in the first* m *rows and the first* q *columns, and which has a* ***column-checksum vector*** *in the* $(m+1)$*-th row and a* ***row-checksum vector*** *in the* $(q+1)$*-th column.*

Reference column-checksums $c^*_{m+1,j}$ *are computed for the elements of the information matrix* $\boldsymbol{C}$ *by*

$$c^*_{m+1,j} := \sum_{i=1}^{m} c_{i,j} \quad \textit{for } 1 \leq j \leq q \tag{5.12}$$

and ***reference row-checksums*** $c^*_{i,q+1}$ *are computed by*

$$c^*_{i,q+1} := \sum_{j=1}^{q} c_{i,j} \quad \textit{for } 1 \leq i \leq m. \tag{5.13}$$

Reference checksums are computed after an operation like the matrix multiplication has been performed, or when the correctness of the full-checksum matrix has to be verified, for instance after a memory transfer. Reference checksums form an essential part of the ABFT error detection and localization procedure since they are required for the computation of *column-* and *row-syndromes*, which are defined as follows:

Definition 5.5 (Column and row syndromes) *Given an* $(m+1 \times q+1)$ *full-checksum matrix* $\boldsymbol{C_{fc}}$ *with checksum elements in the column- and row-checksum vectors, as well as the corresponding reference column- and row-checksums.*

The ***column syndrome*** s_{c_j} *is defined as the difference between the column-checksum* $c_{m+1,j}$ *and the corresponding reference column-checksum* $c^*_{m+1,j}$*:*

$$s_{c_j} := c_{m+1,j} - c^*_{m+1,j}. \tag{5.14}$$

The ***row syndrome*** s_{r_i} *is analog defined as the difference between the row-checksum* $c_{i,q+1}$ *and the corresponding reference row-checksum* $c^*_{i,q+1}$*:*

$$s_{r_i} := c_{i,q+1} - c^*_{i,q+1}. \tag{5.15}$$

Non-zero syndromes indicate the occurrence of errors.

The *ABFT error detection procedure* computes all column and row syndromes for a full-checksum matrix $\boldsymbol{C_{fc}}$. Syndromes that are non-zero indicate the occurrence of an error within the corresponding column or row of the full-checksum matrix. In [Huang84], the authors take a *single error assumption* for the proposed Algorithm-based Fault Tolerance scheme. The scheme is able to *detect*, *localize* and *correct* a single erroneous element in a full-checksum matrix $\boldsymbol{C_{fc}}$. Since such an error can occur within the elements of the information matrix $\boldsymbol{C}$, as well as within the elements of the column- and row-checksum vectors, the following cases have to be checked:

$$If \quad \exists j : s_{c_j} \neq 0 \quad \wedge \quad \forall i : s_{r_i} = 0 \quad \rightarrow \quad \text{error in column checksum } c_{m+1,j} \tag{5.16}$$

$$If \quad \exists i : s_{r_i} \neq 0 \quad \wedge \quad \forall j : s_{c_j} = 0 \quad \rightarrow \quad \text{error in row checksum } c_{i,q+1} \tag{5.17}$$

$$If \quad \exists i : s_{r_i} \neq 0 \quad \wedge \quad \exists j : s_{c_j} \neq 0 \quad \rightarrow \quad \text{error in matrix element } c_{i,j}. \tag{5.18}$$

The *localization* and *correction* of an erroneous matrix element depends on the outcomes of the syndrome checks given by Equations 5.16- 5.18:

Error in column-checksum In case that the conditions of Equation 5.16 are met, the error occurred within the column-checksum element $c_{m+1,j}$, whose index j is determined by the index of the indicating, non-zero column syndrome s_{c_j}.

The erroneous column-checksum element is corrected by replacing it with its corresponding reference column-checksum $c^*_{m+1,j}$.

Error in row-checksum In case that the conditions of Equation 5.17 are met, the error occurred within the row-checksum element $c_{i,q+1}$, whose index i is given by the index of the indicating, non-zero row syndrome s_{r_i}. The erroneous column-checksum element is corrected by replacing it with its corresponding reference row-checksum $c^*_{i,q+1}$.

Error in matrix element In case that the conditions of Equation 5.18 are met, the error occurred within the element $c^{err}_{i,j}$ of the information matrix $\boldsymbol{C}$. The row index i of the erroneous element is determined by the index of the indicating, non-zero row syndrome s_{r_i}. The column index j of the erroneous element is analog determined by the index of the indicating, non-zero column syndrome s_{c_j}. The erroneous matrix element $c^{err}_{i,j}$ is corrected by adding the row or column syndrome:

$$c^{corr}_{i,j} := c^{err}_{i,j} + s_{r_i} \quad \text{or} \quad c^{corr}_{i,j} := c^{err}_{i,j} + s_{c_j}. \tag{5.19}$$

In the course of this work, it will be shown that the above conditions for the detection of errors—the comparison of row and column syndromes with zero—poses a severe challenge if Algorithm-based Fault Tolerance schemes are applied in the context of floating-point arithmetic. The impact of rounding errors, in particular within the checksums, renders the direct comparison of checksums impractical and demands more advanced ABFT error detection procedures that are able to cope with rounding errors.

Another drawback of the basic Algorithm-based Fault Tolerance scheme introduced above is the limited number of errors that can be localized and corrected. If the set $S_{p,r}$ of all unique $(p \times r)$ full-checksum matrices is considered, the minimum matrix distance of this set, i.e. the minimum matrix distance between all possible pairs of full-checksum matrices in $S_{p,r}$, is 4. Hence, a single erroneous element within a full-checksum matrix can be corrected. In cases where more than one column and row syndrome are unequal to zero, the erroneous matrix elements can no longer be localized and hence the complete matrix has to be recomputed to provide a correct result. This weakness of the initial approach triggered the development of Algorithm-based Fault Tolerance schemes with *Weighted Checksum Codes*, which enable the detection, localization and correction of multiple errors.

5.2 Weighted Checksum Codes and the Coding Theoretical Foundations of ABFT

Weighted Checksum Codes are a superset of the checksum code introduced in [Huang84], which improve the *error detection*, *localization* and *correction* capabilities of Algorithm-based Fault Tolerance for matrix operations. The concept has originally been introduced by Jou and Abraham in [Jou86] and was later brought into a formal coding theoretical framework by Anfinson and Luk in [Anfin88].

The basic idea behind weighted checksum codes is to change the impact that each addend (data element) has on the final value of the checksum. The impact is determined by a *weight (factor)* the addend is multiplied with.

Preliminary remark: For the sake of simplicity, the following definitions are introduced for square $(n \times n)$-matrices. The definitions, however, can be easily transferred to non-square matrices without loss of generality.

Definition 5.6 (Weight, weight vector and weight matrix) *Given an $(n \times n)$-matrix $\boldsymbol{A}$ with data elements $a_{i,j}$ for $1 \leq i, j \leq n$.*

*A **weight** $w_k \in \mathbb{R}$ with $1 \leq k \leq n$ is a coefficient (factor) a data element $a_{i,j}$ is multiplied with before it is added during the computation of a (weighted) checksum.*

*A **weight vector** $\boldsymbol{w}$ is an $(1 \times n)$-vector, which consists of weights w_i with $1 \leq i \leq n$.*

$$\boldsymbol{w} := \begin{bmatrix} w_1 & w_2 & \cdots & w_n \end{bmatrix}. \tag{5.20}$$

*A **weight matrix** $\boldsymbol{W}$ is a $(d \times n)$-matrix which consists of d weight vectors $\boldsymbol{w_t}$ with $1 \leq t \leq d$, and which generates d weighted checksums for a column- or row-vector of matrix $\boldsymbol{A}$.*

$$\boldsymbol{W} := \begin{bmatrix} \boldsymbol{w}_1 \\ \boldsymbol{w}_2 \\ \vdots \\ \boldsymbol{w_d} \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \ddots & \vdots & \vdots \\ w_{d,1} & w_{d,2} & \cdots & w_{d,n} \end{bmatrix}. \tag{5.21}$$

To enable the detection of up to d errors and the correction of up to $\lfloor \frac{d}{2} \rfloor$ errors, d weighted checksums are computed for a given information matrix $\boldsymbol{A}$. The definitions of the *weighted column-*, *weighted-row* and *weighted full-checksum matrix* are as follows:

Definition 5.7 (Weighted column-checksum matrix) *Given an* $(n \times n)$*-matrix* A *consisting of* n *rows and* n *columns, and a* $(d \times n)$ *weight matrix* W*. The corresponding* ***weighted column-checksum matrix*** A_{wcc} *is an* $(n + d \times n)$*-matrix which consists of the matrix* A *in the first* n *rows and* d ***weighted column-checksum vectors*** *in the remaining lower rows:*

$$A_{wcc} := \begin{bmatrix} A \\ W \cdot A \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ wcs_{1,1} & wcs_{1,2} & \cdots & wcs_{1,n} \\ wcs_{2,1} & wcs_{2,2} & \cdots & wcs_{2,n} \\ \vdots & \ddots & \vdots & \vdots \\ wcs_{d,1} & wcs_{d,2} & \cdots & wcs_{d,n} \end{bmatrix} \tag{5.22}$$

with the ***weighted column checksums*** $wcs_{i,j}$.

Definition 5.8 (Weighted row-checksum matrix) *Given an* $(n \times n)$*-matrix* A *consisting of* n *rows and* n *columns, and a* $(d \times n)$ *weight matrix* W*. The corresponding* ***weighted row-checksum matrix*** A_{wrc} *is an* $(n \times n + d)$*-matrix which consists of the matrix* A *in the first* n *columns and* d ***weighted row-checksum vectors*** *in the remaining columns:*

$$A_{wrc} := \begin{bmatrix} A & A \cdot W^T \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & wcs_{1,1} & wcs_{1,2} & \cdots & wcs_{1,d} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & wcs_{2,1} & wcs_{2,2} & \cdots & wcs_{2,d} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & wcs_{n,1} & wcs_{n,2} & \cdots & wcs_{n,d} \end{bmatrix} \tag{5.23}$$

with the ***weighted row checksums*** $wcs_{i,j}$.

Definition 5.9 (Weighted full-checksum matrix) *Given an* $(n \times n)$*-matrix* A *consisting of* n *rows and* n *columns, and a* $(d \times n)$ *weight matrix* W*. The corresponding* ***weighted full-checksum matrix*** A_{wfc} *is an* $(n + d \times n + d)$*-matrix which consists* *of the matrix $\boldsymbol{A}$ in the first n rows and first n columns, and d* ***weighted column-checksum vectors*** *and d* ***weighted row-checksum vectors*** *in the remaining rows and columns:*

$$\boldsymbol{A}_{wfc} := \begin{bmatrix} \boldsymbol{A} & \boldsymbol{A} \cdot \boldsymbol{W}^T \\ \boldsymbol{W} \cdot \boldsymbol{A} & \boldsymbol{W} \cdot \boldsymbol{A} \cdot \boldsymbol{W}^T \end{bmatrix}. \tag{5.24}$$

In [Anfin88], Anfinson and Luk provided the required definitions and theorems to embed Algorithm-based Fault Tolerance into a proper coding theoretical framework. Starting point is the $(d \times n + d)$ *weighted checksum matrix* $\boldsymbol{H}$ of a weighted checksum code C, which is defined analog to the parity check matrix in Chapter 2.3. The weighted checksum matrix $\boldsymbol{H}$ is essential for the error detection procedure since it is required for the computation of the syndromes:

Definition 5.10 (Weighted checksum matrix) *For a given* ***weighted checksum code*** *C, the $(d \times n + d)$-**weighted checksum matrix** $\boldsymbol{H}$ is defined as*

$$\boldsymbol{H} := \begin{bmatrix} w_{1,1} & w_{2,1} & w_{3,1} & \cdots & w_{n,1} & -1 & 0 & \cdots 0 \\ w_{2,1} & w_{2,2} & w_{2,3} & \cdots & w_{2,n} & 0 & -1 & \cdots 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{d,1} & w_{d,2} & w_{d,3} & \cdots & w_{d,n} & 0 & 0 & \cdots -1 \end{bmatrix}. \tag{5.25}$$

The weighted checksum matrix $\boldsymbol{H}$ comprises weights $w_{i,j}$ in the first d rows and first n columns, and a negative $(d \times d)$-identity matrix in the remaining columns.

Any $(n + d)$-vector $\boldsymbol{x}$ that fulfills the condition

$$\boldsymbol{H} \cdot \boldsymbol{x} = 0 \tag{5.26}$$

is an ***error-free vector*** *and lies within the* ***nullspace***

$$N(\boldsymbol{H}) := \{\boldsymbol{x} | \boldsymbol{H}\boldsymbol{x} = 0\} \tag{5.27}$$

of the weighted checksum matrix $\boldsymbol{H}$.

A vector $\boldsymbol{x} \in N(\boldsymbol{H})$ is called a ***code vector****. Vectors which lie in the domain of $\boldsymbol{H}$ but not within $N(\boldsymbol{H})$ are called* ***non-code vectors****. The nullspace $N(\boldsymbol{H})$ defines the* ***code space*** *or* ***code*** *C.*

The weights $w_{i,j}$ play an essential role for a weighted checksum code C, since they have direct impact on the code's distance and hence on the number of detectable and correctable errors. A weighted checksum code C with distance d is able to detect up to $d-1$ errors and correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors. To create a distance d code, the weights $w_{i,j}$ in the weighted checksum matrix $\boldsymbol{H}$ have to be chosen in a way that in any combination of $d-1$ columns of $\boldsymbol{H}$ the vectors are linearly independent. For the coding theoretical definitions of, for instance, the distance of a code, the reader is referred to Appendix C or [Shu04].

Analog to [Huang84], syndromes or syndrome vectors are required in order to detect errors within encoded matrices and vectors.

Definition 5.11 (Syndrome vector and correction vector) *Given a vector $\boldsymbol{x}$ and a weighted checksum matrix $\boldsymbol{H}$, which defines the weighted checksum code C. For any potentially erroneous versions $\tilde{\boldsymbol{x}}$ of vector $\boldsymbol{x}$, the corresponding* ***syndrome vector $\boldsymbol{s}$*** *is defined as:*

$$\boldsymbol{s} := \boldsymbol{H} \cdot \tilde{\boldsymbol{x}}. \tag{5.28}$$

A non-zero syndrome vector $\boldsymbol{s}$ indicates and error within vector $\tilde{\boldsymbol{x}}$ (cfg. Eq. 5.26).

The corresponding ***correction vector $\boldsymbol{c}$*** *is defined as:*

$$\boldsymbol{c} := \tilde{\boldsymbol{x}} - \boldsymbol{x} \tag{5.29}$$

and hence $\boldsymbol{H} \cdot \boldsymbol{c} = \boldsymbol{s}$.

The correction procedure is a non-trivial task which involves the solution of a typically overdetermined system of (non-)linear equations. Consider the $(n + d \times n)$ weighted column-checksum matrix $\boldsymbol{A}_{wcc}$ which has d additional rows of weighted checksums

$$\boldsymbol{A}_{wcc} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \\ wcs_{1,1} & wcs_{1,2} & \cdots & wcs_{1,n} \\ wcs_{2,1} & wcs_{2,2} & \cdots & wcs_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ wcs_{d,1} & wcs_{d,2} & \cdots & wcs_{d,n} \end{bmatrix}. \tag{5.30}$$

After a target operation has been performed on the matrix, the syndrome vectors are computed. In the following, the $(d \times 1)$-syndrome vector $\boldsymbol{s}_j$ for the j-th column of matrix $\boldsymbol{A}_{wcc}$ is considered:

$$\boldsymbol{s}_j = \begin{bmatrix} wcs_{1,j} - wcs^*_{1,j} \\ wcs_{2,j} - wcs^*_{2,j} \\ \vdots \\ wcs_{d,j} - wcs^*_{d,j} \end{bmatrix}, \tag{5.31}$$

with the weighted reference column-checksums $wcs^*_{i,j}$.

For the correction of erroneous values in the j-th column vector, the $(n + d \times 1)$ correction vector $\boldsymbol{c}_j$ has to be determined. By definition [Anfin88], this correction vector $\boldsymbol{c}_j$ can have only α non-zero entries, which implies that a maximum of α erroneous positions $i_1, \ldots, i_\alpha$ and the corresponding correction values $c_{i_1}, \ldots, c_{i_\alpha}$ have to be determined. To do so, the following system of equations has to be solved:

$$\begin{aligned} w_{1,i_1} \cdot c_{i_1} + w_{1,i_2} \cdot c_{i_2} + \cdots + w_{1,i_\alpha} \cdot c_{i_\alpha} &= s_{1,j} && (5.32) \\ w_{2,i_1} \cdot c_{i_1} + w_{2,i_2} \cdot c_{i_2} + \cdots + w_{2,i_\alpha} \cdot c_{i_\alpha} &= s_{2,j} && (5.33) \\ &\vdots && (5.34) \\ w_{d,i_1} \cdot c_{i_1} + w_{d,i_2} \cdot c_{i_2} + \cdots + w_{d,i_\alpha} \cdot c_{i_\alpha} &= s_{d,j} && (5.35) \end{aligned}$$

This system comprises of $\alpha \cdot d$ unknowns in d equations and only the values $s_{i,j}$ with $1 \leq i \leq d$ from the syndrome vector are known. It therefore can only be solved if the weights fulfill the condition that they generate d linearly independent columns in matrix $\boldsymbol{H}$.

Although the general encoding scheme of Algorithm-based Fault Tolerance is independent of the input data that is processed, the applied checksum codes can cause severe numerical problems such as underflow, overflow and accumulated rounding errors in checksums, when applied in floating-point arithmetic. This in turn can reduce the error detection and correction capabilities of the ABFT scheme. A considerable number of codes has therefore been proposed over time to improve the numerical properties of ABFT and in particular to reduce the impact of floating-point rounding errors during the encoding process. This work focuses in this section on the most relevant codes that are applicable and appropriate for linear algebra matrix operations.

In [Jou86], Jou and Abraham proposed *exponential weighted checksum codes* with weights of the form $w_j^{(i)} = \beta^{(j-1)(i-1)}$ for $j = 1, \ldots, n$ and $i = 1, \ldots, d$. For $\beta = 2$ such weights can be efficiently computed through shift operations. However, the proposed weights grow rapidly for large problem sizes and can lead to overflow problems. To mitigate overflows, Luk proposed in [Luk86] the use of *linear weights* $w_j = j$ for $j = 1, ..., n$. Nair and Abraham introduced a set of *general linear codes* in [Nair88], which cause minimal numerical error during the encoding process. They also introduced in this work the notion of *code reflectivity*, which describes the ratio of change in a code value (checksum) due to the change in the data value that causes the change in the checksum. General linear codes should be designed to achieve a high ratio to ensure that small errors in the information elements cause large changes in the checksums. Depending on the application and the characteristics of the input data sets, Nair and Abraham proposed *average* and *weighted average checksum codes* with encoder vectors of the form $\boldsymbol{p}^T = [\frac{1}{n}, \frac{1}{n}, \frac{1}{n}, \cdots, \frac{1}{n}]$. Here, the average value rather than the sum of the column or row elements are stored for the encoding to avoid the possibility of very large checksum values. The proposed *weighted average checksums* are similar to the standard weighted checksums with the difference that the code vectors are formed by taking weighted averages of the information elements. The authors in [Nair88] also investigated *periodic encoder vectors* such as $\boldsymbol{p}^T = [1, -1, 1, -1, ...]$, which cause less memory overhead since only few weights have to be stored and which reduce overflows during the encoding. In addition, Nair and Abraham investigated *normalized encoder vectors* of the form $\boldsymbol{p}^T = [\frac{c}{|x|}, \frac{c}{|x|}, \frac{c}{|x|}, ...]$, where c is a user-defined constant value and $|x|$ is the norm of the column or row of the matrix to be encoded. With this encoding, the entries of the encoder vector change according to the processed input data, and occurrences of underflows are minimized. In [Bliss88], Bliss et al. investigated the numerical properties of ABFT checksum encodings with respect to their application in floating-point arithmetic. The authors performed a forward worst-case rounding error analysis and proposed a *minimum dynamic range (MDR)* SEC code of the form $a_{n+1} = \sum_{i=0}^{n-1} \left(\frac{(1+i)}{n}\right) \cdot a_i$.

All of the above introduced weighted checksum codes can be directly implemented for ABFT-protected matrix operations on graphics processing units without any restrictions. However, although such a significant number of checksum codes has been introduced for Algorithm-based Fault Tolerance schemes, most of these encodings have been investigated and evaluated separate of each other. In Chapter 8, the error detection and correction capabilities of standard, exponential, linear, MDR and periodic encodings are therefore evaluated for matrix operations on graphics processing units. Besides the pure error detection and correction capabilities, the performance impact of the encodings is examined. In addition, the fault tolerance potential of different combinations of such codes like normal together with exponential checksums, or normal checksums together with MDR encodings are evaluated.

5.3 Partitioned Encoding

The use of floating-point arithmetic can have severe impact on Algorithm-based Fault Tolerance schemes based on weighted checksum codes. The inevitable rounding error that occurs during the encoding procedure affects the computed checksums and can hence reduce the error detection and correction capabilities of such schemes. This significantly limits the *scalability to larger problem sizes*, which in turn, are often of most interest for parallelization on modern graphics processing units.

Given an $(n \times 1)$-weighted encoder vector $\boldsymbol{w}$ and an $(n \times 1)$-vector $\boldsymbol{a}$, which has to be encoded. In [Nair88], Nair and Abraham pointed out that the rounding error ε_{enc} that affects the final checksum is bounded by:

$$\varepsilon_{enc} \leq \epsilon_M \cdot m \cdot \|\boldsymbol{w}^T\|_2 \cdot \|\boldsymbol{a}\|_2, \tag{5.36}$$

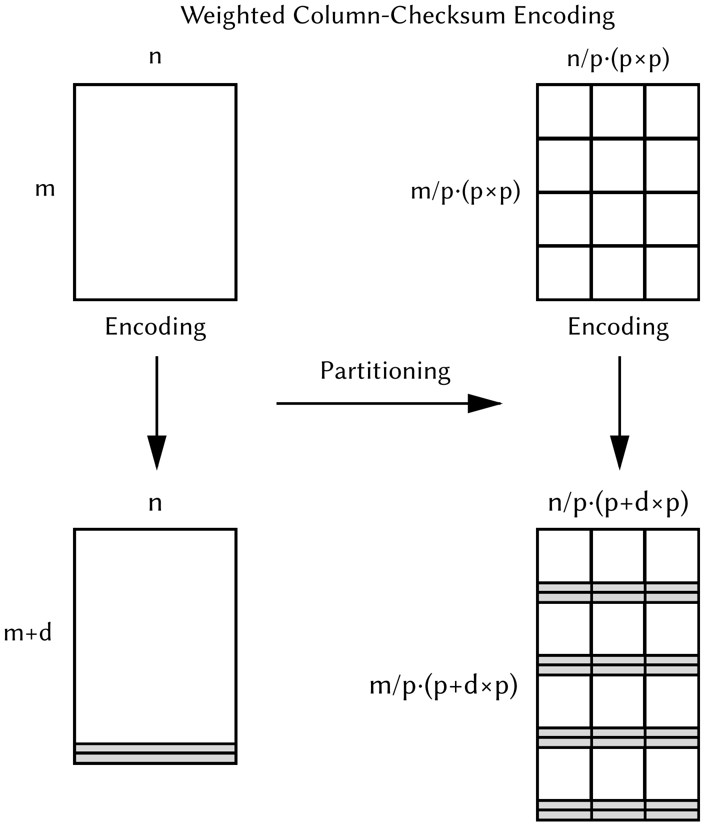
where ϵ_M is the machine epsilon (cf. Equation 2.4) and m is a parameter that represents the number of bits in the significand [Cohen73]. Since ϵ_M and m are defined by the machine and hence are non-alterable, the upper bound ε_{enc} on the encoding rounding error only depends on the *Euclidean norms* of the weighted encoder vector $\boldsymbol{w}$ and vector $\boldsymbol{a}$. With increasing vector sizes, the weighted checksum code exhibits increasingly poorer numerical properties.

In [Rexfo92], Rexford and Jha proposed *partitioned encoding* for Algorithm-based Fault Tolerance schemes with weighted checksum codes to improve the numerical properties of the encoding procedure and to enable scalability to large problem sizes[1].

Given a matrix operation that applies ABFT with some weighted checksum code using d encoder vectors, which can handle matrices up to dimensions $(p \times p)$ without suffering from the negative impact of rounding errors. Then any $(m \times m)$-matrix with $m > p$ and m a multiple of p, can be partitioned into $\left(\frac{m}{p}\right)^2$ $(p \times p)$ matrices. Each of these

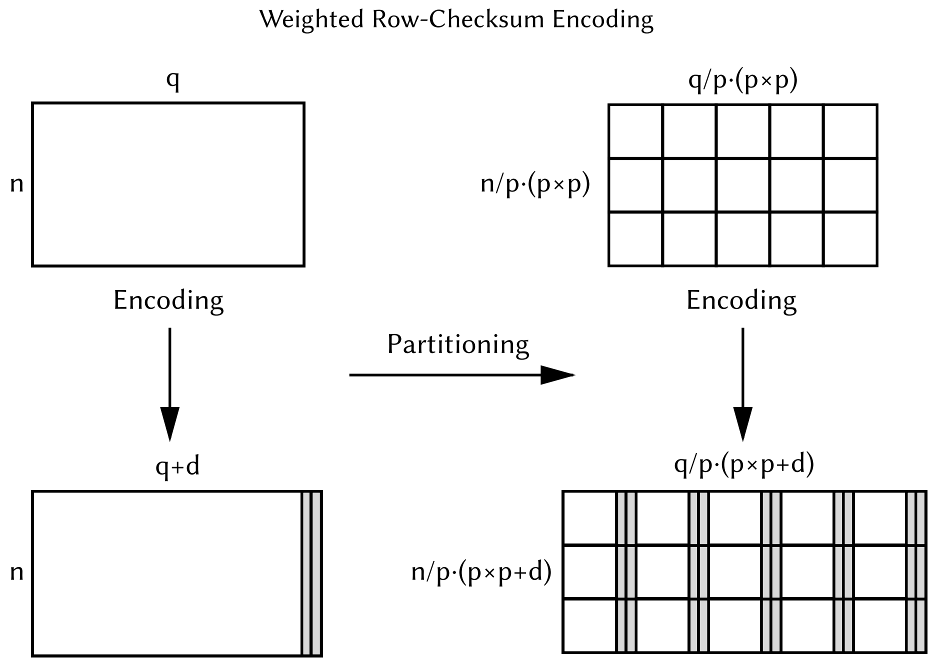
[1] In [Huang84], Huang and Abraham already described the use of partitioned encoding schemes for block-based algorithms and the distribution of computations in parallel systems. However, they did not further discuss the benefits of partitioned encoding for the improvement of numerical properties.

sub-matrices can be encoded into a weighted column-, row-, or full-checksum matrix of dimensions $(p + d \times p)$, $(p \times p + d)$, or $(p + d \times p + d)$, respectively. In case of the weighted full-checksum matrix, the partitioned encoding approach requires $(\frac{m}{p} \cdot (p + d))^2$ matrix elements in contrast to the non-partitioned encoding with $(m + d)^2$ elements. Figure 5.1 and Figure 5.2 depict the basic principle of partitioned encoding for ABFT with weighted checksum codes for weighted column- and row checksum matrices.



▲ **Figure 5.1** — Partitioned encoding for weighted column-checksum matrices.

Partitioned encoding enables the detection and correction of more errors in the complete matrix, as long as these errors are distributed across different sub-matrices. Furthermore, the partitioned encoding significantly improves the numerical properties of the ABFT encoding procedure, as shown in [Rexfo92]. According to Rexford and Jha [Rexfo92], the upper bound on the rounding error due to the encoding process is proportional to the Euclidean norm of the encoder vector, and the improvement of the numerical



▲ Figure 5.2 — Partitioned encoding for weighted row-checksum matrices.

properties can be quantified by the factor

$$\gamma := \frac{\|\boldsymbol{w}_p^T\|_2}{\|\boldsymbol{w}_m^T\|_2}. \tag{5.37}$$

The partitioned encoding checksum property is preserved by matrix operations like the matrix multiplication, the matrix addition, the multiplication of a matrix with a scalar, the matrix transpose, as well as the LU decomposition. Further matrix operations can be adapted to support ABFT with partitioned encoding.

Moreover, partitioned encoding matches classic block-based matrix algorithms, for instance, for the multiplication of matrices. It also provides a perfect match for the block-based execution paradigm (thread blocks) on modern graphics processing units. In this context, the impact of the sub-matrix dimensions ($p \times p$) on the encoding and hence on the error detection and correction capabilities of ABFT schemes, as well as the potential impact on the GPU performance has not yet been investigated. Since p represents an important design parameter for fault tolerant linear algebra matrix operations on GPUs, Chapter 8 provides an in-depth analysis of relevant encoding block sizes on GPUs.

5.4 Handling of Rounding Errors

In floating-point arithmetic systems, Algorithm-based Fault Tolerance schemes have to cope with rounding errors during different phases. Rounding errors are introduced during the initial encoding and during the computation of reference checksums in the result checking procedure. Rounding errors are, of course, also introduced during the computations performed by the target algorithm. In Section 5.2, different codes have been introduced which improve the numerical properties of the encoding. In Section 5.3, partitioned encoding was introduced as a technique which also improves the numerical quality of the encoding, and which enables the seamless scaling of ABFT schemes to large problem sizes without affecting the encoding. Nevertheless, the rounding errors that affect the checksum comparison during the ABFT result checking procedure remain a hard challenge, since they render the direct comparison for equality of checksums useless.

Appropriate *rounding error bounds* (tolerances) are therefore required for the reliable online classification of errors into inevitable rounding errors due to limited numerical precision, tolerable compute errors in the magnitude of such rounding errors, and into critical errors that are larger than those and not tolerable.

Up to date, different approaches have been proposed for the determination of rounding error bounds, including *experimental calibration-based* approaches, schemes based on *mantissa checks* and approaches based on *analytical rounding error bounds.*

In [Baner88] and [Baner90], Banerjee et al. determined rounding error bounds based on *experimental calibrations.* The authors expressed the norm of the error ε_{comp} in computations due to rounding as

$$\|\varepsilon_{comp}\| := K \cdot F(N) \cdot 2^{-2t}, \tag{5.38}$$

where K is an *application-dependent* constant, $F(N)$ is an *application-dependent function of the problem size*, and t is the size of the significand in the used floating-point representation. The authors did not disclose any further details on the determination of the application-specific constant K or the function $F(N)$. They only mentioned that for the Fast Fourier Transform $F(N) = log(N)$. In fault injection experiments performed by the authors, the problem size was kept fix and the norm of the error due to rounding was substituted by a constant K'. This constant was chosen through calibration in a way that the error coverage was maximized while keeping the number of false alarms equal to zero. To achieve a suitable calibration, a large number of different data sets was considered. The data sets were all of the same size and span approximately the same range. The target computation was executed repeatedly on the data sets, starting with a rounding error bound equal to zero. Every time the fault tolerance scheme detected an error (false alarm), the bound was increased. This procedure has been repeated until no more false alarms occurred on the data sets.

The experimental calibration has massive drawbacks which make it impractical for the online determination of rounding error bounds. First, the calibration requires several runs before the intended target computation can be performed. Second, the determined rounding error bound is highly likely to produces false-positive and, even worse, false-negative detections when the characteristics of the input data sets change. Third, the problem size is kept fix for the calibration and hence, every time the problem size changes, the calibration has to be repeated.

Another group of approaches considers mantissa checks for floating-point results. In [Assad92], so called *integer mantissa checks* (IMC) have been proposed for matrix multiplications. These checks treat the significands of floating-point elements in the input matrices and the product matrix as integers and use the integer ALU to test for errors in the results. The original approach was only able to apply integer mantissa checks for floating-point multiplications, the authors therefore proposed an additional check for the floating-point additions and called the scheme a hybrid checksum test. In [Zhang94], *extended mantissa checksums* (EMC) have been introduced, which also split the significands of elements in the product matrix into two parts, which are then extended to the maximum length allowed by the system's integer precision. Errors are detected and corrected in both parts before the parts are merged in the end to yield a full-checksum result matrix. In [Dutt96], the authors extended their previously proposed integer mantissa checks and applied the hybrid checksum test to the LU decomposition. They also described possible extensions of the hardware to support integer mantissa checks with less time overhead.

Approaches based on mantissa checks exhibit several issues which hinder their application. First, the checks are hybrid and only errors in floating-point multiplications can be detected by these checks. Errors in the floating-point addition part can not be detected since this requires the alignment of significand products and hence a loss of precision. A second, non-mantissa check is therefore required for floating-point additions. Another problem are potential overflows in the product of integer matrices, although this has been addressed in parts by [Zhang94]. The extraction of the significands and the required computation of the significand product introduce additional computational and memory overhead.

The third group of proposed approaches for the determination of rounding error bounds is based on classic *analytical rounding error bounds*. Inter alia, such analytical bounds can be found for numerous linear algebra matrix operations in [Higha02] and [Golub12].

In [Turmo00a, Turmo00b] and [Turmo03], Turmon et al. derived and evaluated sets of checksum tests for a number of ABFT-protected linear algebra operations, including the matrix multiplication, to distinguish between intolerable errors and inevitable rounding errors. The checksum tests are derived from classic analytical rounding error bounds, for instance for the multiplication of matrices $\boldsymbol{A} \in \mathbb{R}^{m \times k}$ and $\boldsymbol{B} \in \mathbb{R}^{k \times n}$:

$$\boldsymbol{E} \quad := \quad \tilde{\boldsymbol{P}} - \boldsymbol{A} \cdot \boldsymbol{B} \tag{5.39}$$

$$||\boldsymbol{E}||_\infty \quad \leq \quad \max(m, n, k) \cdot ||\boldsymbol{A}||_\infty \cdot ||\boldsymbol{B}||_\infty \cdot \varepsilon_M, \tag{5.40}$$

where $\boldsymbol{E}$ is the error matrix, $\tilde{\boldsymbol{P}}$ is the machine-computed matrix product and ϵ_M is the machine epsilon (cf. Equation 2.4). The authors proposed an input-independent checksum test of the form

$$d := \tilde{\boldsymbol{P}} \cdot \boldsymbol{w} - \boldsymbol{A} \cdot \boldsymbol{B} \cdot \boldsymbol{w}. \tag{5.41}$$

$$\frac{||d||_\infty}{||\boldsymbol{A}||_\infty \cdot ||\boldsymbol{B}||_\infty \cdot ||\boldsymbol{w}||_\infty} \quad > \quad \tau \cdot \epsilon_M \Longrightarrow \textbf{error} \tag{5.42}$$

$$\frac{||d||_\infty}{||\boldsymbol{A}||_\infty \cdot ||\boldsymbol{B}||_\infty \cdot ||\boldsymbol{w}||_\infty} \quad \leq \quad \tau \cdot \epsilon_M \Longrightarrow \textbf{rounding error}, \tag{5.43}$$

where $\boldsymbol{w}$ is an ABFT encoder vector[2] and τ is defined as an input-independent threshold value. If the discrepancy in Equation 5.42 is larger than $\tau \cdot \epsilon_M$ the approach signals an error. Otherwise, the divergence is classified as rounding error (Equation 5.43).

The matrix norms required for the proposed checksum test are associated with considerable computational overhead. Turmon et al. therefore derived simplified tests. Instead of $||\boldsymbol{A}||_\infty \cdot ||\boldsymbol{B}||_\infty$, they proposed to compute $||\tilde{\boldsymbol{P}}||_\infty$ which equals $||\boldsymbol{A} \cdot \boldsymbol{B}||_\infty$ in the error-free case. Since $||\boldsymbol{A} \cdot \boldsymbol{B}||_\infty \leq ||\boldsymbol{A}||_\infty \cdot ||\boldsymbol{B}||_\infty$, this bound underestimates the upper bound for the rounding error and increases the risk of false-positive detections. To mitigate this problem, the authors suggested the application of an additional scaling factor which can be absorbed in the factor τ. Following the same idea, $||\tilde{\boldsymbol{P}} \cdot \boldsymbol{w}||_\infty$ could

[2] Turmon et al. used the basic encoder vector $\boldsymbol{w} = [1, 1, ..., 1]$.

be used as replacement for $||\boldsymbol{A}||_\infty \cdot ||\boldsymbol{B}||_\infty \cdot ||\boldsymbol{w}||_\infty$. However, this vector norm intensifies the risk of false alarms even more. Turmon et al. therefore suggested to compute $\lambda \cdot ||\boldsymbol{w}||_\infty + ||\tilde{\boldsymbol{P}}\boldsymbol{w}||_\infty$, using an *application-dependent factor* λ. In conclusion, the above introduced substitutions yielded a set of checksum tests with different trade-offs between error detection capability and computational overhead. The tests all follow from the indicated difference matrix $\boldsymbol{\Delta}$, which is never computed explicitly, by computation of the norm $\delta = ||\boldsymbol{\Delta} \cdot \boldsymbol{w}||_\infty$, which is then scaled by a factor defined by the above substitutions. The so called *ideal test* is then defined as

$$\frac{\delta}{||\boldsymbol{A}||_\infty \cdot ||\boldsymbol{B}||_\infty \cdot ||\boldsymbol{w}||_\infty} > \tau \cdot \epsilon_M \Longrightarrow \textbf{error}, \tag{5.44}$$

the so called *matrix test* as

$$\frac{\delta}{||\tilde{\boldsymbol{P}}||_\infty \cdot ||\boldsymbol{w}||_\infty} > \tau \cdot \epsilon_M \Longrightarrow \textbf{error} \tag{5.45}$$

and the so called *vector test* as

$$\frac{\delta}{\lambda \cdot ||\boldsymbol{w}||_\infty + ||\tilde{\boldsymbol{P}}\boldsymbol{w}||_\infty} > \tau \cdot \epsilon_M \Longrightarrow \textbf{error}. \tag{5.46}$$

In [Gunne01], Gunnels et al. followed the general idea of the above introduced approach and proposed norm-based tests for the matrix multiplication, which allow the detection of errors within the result matrix $\boldsymbol{C}$, as well as in the two input matrices $\boldsymbol{A} \in \mathbb{R}^{(m \times k)}$ and $\boldsymbol{B} \in \mathbb{R}^{(k \times n)}$. They considered the norm of the difference vectors $\boldsymbol{d}$ and $\boldsymbol{e}$ for row- and column-checksums:

$$\begin{aligned} \boldsymbol{d} &:= \tilde{\boldsymbol{C}}\boldsymbol{w} - \boldsymbol{C}\boldsymbol{w} \qquad (5.47) \\ \boldsymbol{e} &:= \boldsymbol{v}^T\tilde{\boldsymbol{C}} - \boldsymbol{v}^T\boldsymbol{C}, \qquad (5.48) \end{aligned}$$

where $\boldsymbol{w}$ and $\boldsymbol{v}^T$ are ABFT encoder vectors and $\tilde{\boldsymbol{C}}$ is the machine-computed matrix product. The corresponding checksum tests were of the form

$$\begin{aligned} ||\boldsymbol{d}||_\infty &> \tau \cdot ||\boldsymbol{A}||_\infty \cdot ||\boldsymbol{B}||_\infty \qquad (5.49) \\ ||\boldsymbol{e}^T||_\infty &> \tau \cdot ||\boldsymbol{A}||_\infty \cdot ||\boldsymbol{B}||_\infty, \qquad (5.50) \end{aligned}$$

where $\tau = \max(m, n, k) \cdot \varepsilon_M$ with the machine epsilon ϵ_M (cf. Equation 2.4).

The norm-based approaches based on analytical rounding error bounds introduce, depending on the chosen checksum test, a considerable computational overhead. Moreover, the different error detection and overhead trade-offs require user interaction for the selection of the appropriate test. In [Turmo03], Turmon et al. summarize: *"The key question for a fault tolerance practitioner is to set the threshold to achieve the right tradeoff between correct fault detections and false alarms. Because of the imprecision inherent in the error bounds, theoretical results can only give an indication of how expected error scales with algorithm inputs; the precise constants for best performance must in general be determined empirically for a given algorithm implementation."* This renders these approaches unattractive for Algorithm-based Fault Tolerance schemes which are intended to operate in an autonomous manner, independent of the input data and user-defined thresholds. Another point of criticism of the above described approaches, which is worth to be considered, is the fact that these methods determine rounding error bounds for complete (sub-)matrices and not for individual checksums. Hence, the resolution or sharpness of these bounds can be rather low.

In [Roy C93], Roy-Chowdhury and Banerjee introduced a novel approach for the determination of rounding error bounds for individual checksums. Since their approach is based on the principles of classic analytical rounding error bounds, it is called *Simplified Error Analysis* (SEA). The basic idea of this method is to keep track of the accumulated rounding error in variables after each update of the variables.

Let $\tilde{v}^k$ be the value of a variable v after the k-th update, and let v^k be the exact, error-free value of the variable. Roy-Chowdhury and Banerjee use $EB(v^k)$ as a bound on the magnitude of the rounding error accumulation in the computation of $\tilde{v}^k$, where the following relation holds between $\tilde{v}^k$, v^k and $EB(v^k)$:

$$|\tilde{v}^k| \leq |v^k| + EB(v^k). \tag{5.51}$$

The set of variables which are involved in the computations between the k-th and $(k+1)$-th update of v is denoted by V^k and the bound on the magnitude of the accumulated rounding error associated with these variables is denoted by $EB^k(V)$. The bound $EB^k(V)$ of the accumulated rounding error is updated according to

$$EB^{k+1}(v) \longleftarrow F^k(V^k, EB(V^k), \epsilon_M), \tag{5.52}$$

where $\epsilon_M = 2^{-t}$ is the machine epsilon and t is the number of mantissa bits.

If the computations that take place during the k-th and $(k+1)$-th update are known, the function F^k can be derived by application of classic rounding error analysis for the performed basic floating-point operations. However, this approach of analyzing and tracking the accumulated rounding error is associated with considerable memory and time overhead. Roy-Chowdhury and Banerjee therefore proposed the following three simplifications:

1. The update function F^k is typically given as polynomial in ε_M. For a reduction of the computational overhead, the higher order error terms ($O(\varepsilon_M^2)$) are omitted.
2. The introduction and tracking of a rounding error bound for each variable involved in a computation is very expensive. The *Simplified Error Analysis* therefore derives upper bounds on the total rounding error for groups of variables, for instance the elements within a row or column of a matrix.
3. For certain algorithms the derived rounding error bound can grow exponentially with the execution time. This is reflected by an additional constant factor $K \cdot EB(V^k)$, $K > 1$, in the update function F^k. In the *Simplified Error Analysis*, the constant factor K is set to 1, which prevents exponential growth of the bound.

In [Roy C93], the *Simplified Error Analysis* (SEA) is introduced for the QR decomposition, the Gaussian Elimination and the matrix multiplication, which serves as example for the following introduction of SEA.

Given two $(n \times n)$-matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ to be multiplied on a parallel computer system with p processors. The computation of the matrix multiplication is distributed among the p processors by sending an $(m \times n)$ row strip of matrix $\boldsymbol{A}$ and a full copy of matrix $\boldsymbol{B}$ to each of the processors. The ABFT checksum encoding is done on these $(m \times n)$-submatrices of $\boldsymbol{A}$. Due to the described distribution of the computations, Roy-Chowdhury and Banerjee considered the matrix-vector multiplication $\boldsymbol{A}^{(m \times n)} \cdot \boldsymbol{b} = \boldsymbol{c}$, instead of the full matrix multiplication, to introduce the SEA rounding error bound for the ABFT checksum comparison procedure. The rounding error bound consists of a combination of basic analytical rounding error bounds for the product and sum of floating-point numbers, as well as the inner product of vectors. These bounds are briefly introduced in the following:

Definition 5.12 (Analytical rounding error bound for products) *Given a set of n real numbers $x_i \in \mathbb{R}$. The product $fl(\prod_{i=1}^{n} x_i)$ of the x_i computed in floating-point* *arithmetic is defined as*

$$fl(\prod_{i=1}^{n} x_i) := (1+E)\prod_{i=1}^{n} x_i, \tag{5.53}$$

where E is the upper bound on the rounding error with

$$|E| < (n-1)\cdot\epsilon_M, \tag{5.54}$$

and with the machine epsilon ϵ_M.

Definition 5.13 (Analytical rounding error bound for summations) *Given a set of n real numbers $x_i \in \mathbb{R}$. The sum $fl(\sum_{i=1}^{n} x_i)$ of the x_i computed in floating-point arithmetic is defined as*

$$fl(\sum_{i=1}^{n} x_i) := \sum_{i=1}^{n} x_i(1+\delta_i), \tag{5.55}$$

where δ_i is the upper bound on the rounding error with

$$\begin{aligned} |\delta_1| &< (n-1)\cdot\epsilon_M & (5.56)\\ |\delta_i| &< (n+1-i)\cdot\epsilon_M, for\ i = 2,\dots,n, & (5.57) \end{aligned}$$

and with the machine epsilon ϵ_M.

Definition 5.14 (Analytical rounding error bound for inner products) *Given two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ with real number elements $x_i, y_i \in \mathbb{R}$. The inner product of the two vectors computed in floating-point arithmetic is defined as*

$$fl(\boldsymbol{x}^T\boldsymbol{y}) := fl(\sum_{i=1}^{n} x_i y_i) = \sum_{i=1}^{n} x_i y_i(1+\delta_i), \tag{5.58}$$

where δ_i is the upper bound on the rounding error with

$$\begin{aligned} |\delta_1| &< n\cdot\epsilon_M & (5.59)\\ |\delta_i| &< (n+2-i)\cdot\epsilon_M, for\ i = 2,\dots,n, & (5.60) \end{aligned}$$

and with the machine epsilon ϵ_M.

The rounding error bound can also be expressed as

$$|fl(\boldsymbol{x}^T\boldsymbol{y}) - \boldsymbol{x}^T\boldsymbol{y}| \leq n \cdot \epsilon_M \cdot \|\boldsymbol{x}\|_2\| \cdot \boldsymbol{y}\|_2. \tag{5.61}$$

Based on these basic rounding error bounds, Roy-Chowdhury and Banerjee introduced the *rounding error bound for the original checksum elements c_i*: Given the original checksum element $c \in \mathbb{R}$ and the accompanying original checksum element $fl(c)$ computed in floating-point arithmetic. The rounding error that affects the original checksum element is bounded by

$$|fl(c) - c| < (m-1) \cdot \epsilon_M \cdot \sum_{i=1}^{m} \|\boldsymbol{a}_i\|_2 \cdot \|\boldsymbol{b}\|_2 + n \cdot \epsilon_M \|\boldsymbol{s}\|_2 \cdot \|\boldsymbol{b}\|_2, \tag{5.62}$$

where $\boldsymbol{b}$ is an $(n \times 1)$ column vector from matrix $\boldsymbol{B}$, $\boldsymbol{a}_i$ is the i-th row vector of the $(m \times n)$-sub-matrix from matrix $\boldsymbol{A}$, and $\boldsymbol{s}$ is the row vector consisting of the column checksums of this sub-matrix.

Analogously, they specified the *rounding error bound for the reference checksums*: Given the reference checksum element $c^* \in \mathbb{R}$ and the accompanying original checksum element $fl(c^*)$ computed in floating-point arithmetic. The rounding error that affects the reference checksum element is bounded by

$$|fl(c^*) - c^*| < (n+m-1) \cdot \epsilon_M \cdot \|\boldsymbol{b}\|_2 \cdot \sum_{i=1}^{m} \|\boldsymbol{a_i}\|_2, \tag{5.63}$$

where $\boldsymbol{a}_i$ is the i-th row vector of the $(m \times n)$-sub-matrix from matrix $\boldsymbol{A}$ and $\boldsymbol{s}$ is the row vector consisting of the column checksums of this sub-matrix.

The final SEA rounding error bound for the matrix-vector multiplication using ABFT column checksums is then given the difference between these two rounding error bounds:

$$\begin{aligned} |fl(c_{n+1}) - fl(c^*_{n+1})| &< ((n+2m-2) \cdot \|b\|_2 \cdot \sum_{i=1}^{m} \|a_i\|_2 \\ &+ n \cdot \|a_{m+1}\|_2 \cdot \|b\|_2) \cdot \epsilon_M. \end{aligned}$$

The SEA rounding error bound for row checksums is determined in an analogous manner.

The *Simplified Error Analysis* represents a major advance in the direction of autonomously operating ABFT schemes, which do not require the setting of threshold values or any other user interaction. The required vector norms can be efficiently precomputed on graphics processing units, which reduces the overall performance overhead. However, since the determined SEA rounding error bounds strongly depend on these vector norms, a dependence on the problem size is introduced which can lead to rounding error bounds that are too loose and therefore increase the risk of false-negative error detections.

5.5 Applicability to Graphics Processing Units

This chapter provided a detailed introduction to Algorithm-Based Fault Tolerance for matrix operations, including weighted checksum codes and partitioned encoding schemes. In addition, the most important methods for the handling of rounding errors in ABFT schemes have been reviewed.

The weighted checksum codes, which have been introduced in Section 5.2, are all well suited for the application in ABFT schemes for the protection of matrix operations on graphics processing units. The mathematical and numerical properties they exhibit, allow the efficient computation of these codes without any restrictions. The efficacy of the different weighted checksum codes is evaluated in-depth in Chapter 8.

Partitioned encoding schemes, as they were discussed in Section 5.3, not only improve the numerical properties of the ABFT checksum encoding, they also match classic block-based matrix algorithms, for instance, for the multiplication of matrices. Moreover, the partitioned encoding matches the block-based organization of threads on modern graphics processing units. The application of partitioned encoding schemes is therefore particularly attractive for ABFT schemes on GPUs.

As motivated in Section 5.4, floating-point rounding errors can have severe impact on the error detection and localization efficacy of Algorithm-Based Fault Tolerance schemes. Calibration-based approaches that have been proposed for the determination of appropriate rounding error bounds, are inefficient and cause significant overhead, especially if the problem sizes and the characteristics of the input data sets vary. Schemes that utilize the separate encoding and checking of the mantissa- or significand-part of floating-point numbers are highly counter-intuitive for ABFT schemes on massively parallel floating-point processors. In addition, the mantissa-preserving methods often have to be realized as hybrid schemes with different tests for different arithmetic operations. The checks for input and result matrices, which have been proposed by Turmon et al., as well as by Gunnels et al., are rather coarse-grained, but showed a good efficiency. The methods are in large parts based on norms, which can be efficiently computed on graphics processing units. However, the input-dependent threshold value τ, which has to be set by the developer or the user, renders these approaches less attractive for implementations of autonomously operating ABFT schemes on GPUs. Analytical methods for the determination of rounding error bounds such as the *Simplified Error Analysis*, allow the realization of autonomously operating ABFT schemes on graphics processing units. The required vector and matrix norms can be efficiently precomputed on graphics processing units, which reduces the overall performance overhead. The quality of the determined rounding error bound, however, may not be always good enough to reliably distinguish between significant and insignificant errors.

CHAPTER 6

A-ABFT: AUTONOMOUS ALGORITHM-BASED FAULT TOLERANCE

The computation and comparison of checksums are essential steps in ABFT-protected operations (see Chapter 5, equations 5.1, 5.4, 5.7). For ABFT matrix operations carried out in integer arithmetic, conventional ABFT schemes use the direct comparison of checksums to detect errors in computed results. Diverging checksums lead to failing comparisons for equality and indicate errors. In case of large checksum values, potential overflows are often prevented by the application of modulo arithmetic.

In scientific computing and computer-based simulation technology, a vast majority of applications relies on floating-point arithmetic and computed results are therefore often prone to rounding errors. This creates a major challenge, because the impact of such rounding errors renders the direct comparison for equality of checksums impractical. As a consequence, the check conditions for the column and row syndromes (cf. Definition 5.5):

$$If \quad \exists j : s_{c_j} \neq 0 \quad \wedge \quad \forall i : s_{r_i} = 0 \quad \rightarrow \quad \text{error in column checksum } c_{m+1,j} \tag{6.1}$$

$$If \quad \exists i : s_{r_i} \neq 0 \quad \wedge \quad \forall j : s_{c_j} = 0 \quad \rightarrow \quad \text{error in row checksum } c_{i,q+1} \tag{6.2}$$

$$If \quad \exists i : s_{r_i} \neq 0 \quad \wedge \quad \exists j : s_{c_j} \neq 0 \quad \rightarrow \quad \text{error in matrix element } c_{i,j}. \tag{6.3}$$

now become

$$\begin{aligned}
&If \quad \exists j : s_{c_j} > \epsilon_{c_j} \quad \wedge \quad \forall i : s_{r_i} \leq \epsilon_{r_i} \quad \rightarrow \quad \text{error in column checksum } c_{m+1,j} && (6.4)\\
&If \quad \exists i : s_{r_i} > \epsilon_{r_i} \quad \wedge \quad \forall j : s_{c_j} \leq \epsilon_{c_j} \quad \rightarrow \quad \text{error in row checksum } c_{i,q+1} && (6.5)\\
&If \quad \exists i : s_{r_i} > \epsilon_{r_i} \quad \wedge \quad \exists j : s_{c_j} > \epsilon_{c_j} \quad \rightarrow \quad \text{error in matrix element } c_{i,j}, && (6.6)
\end{aligned}$$

where ϵ_{c_j} and ϵ_{r_i} are rounding error bounds for the column and row checksums.

The application of this new kind of check conditions requires knowledge about the rounding errors that occur during the computation of the respective checksums and it requires the determination of appropriate rounding error bounds ϵ_{c_j} and ϵ_{r_i}. The precise determination of such bounds is essential, because bounds that are too large and hence too far away from the actual rounding error, increase the risk of *false-negatives.* In these cases, errors which can affect the final result negatively, may slip undetected through the ABFT result checking procedure. Bounds that are too small, which means that they are in the magnitude of the actual rounding error or below, may increase the rate of *false-positives.* Although false-positives do not affect the final result, they do have a negative impact on the performance of the ABFT-protected operation since they trigger unnecessary corrections.

In this work, the term *quality of a rounding error bound* is used to qualitatively describe the properties of determined rounding error bounds:

Definition 6.1 (Quality of a Rounding Error Bound) *The* ***quality*** *of a rounding error bound ϵ_{c_j} or ϵ_{r_i} for a column or row checksum, is defined as the reciprocal of the* ***distance*** *between the determined rounding error bound and the absolute rounding error ε_{abs} that affects the checksum:*

$$q := \frac{1}{|\varepsilon_{abs} - \epsilon_{c_j}|} \qquad or \qquad q := \frac{1}{|\varepsilon_{abs} - \epsilon_{r_i}|}. \tag{6.7}$$

The smaller the distance between the determined rounding error bound and the actual rounding error, the higher the quality of the bound.

The problem of rounding errors in ABFT checksums has been tackled before in different ways (see Chapter 5.4). However, state of the art approaches often generate non-negligible overhead or they yield rounding error bounds of minor quality and hence favor *false-negatives.* In addition, some of the approaches proposed so far depend on the problem size or the characteristics of the input data. This often leads to situations where user knowledge about the performed operation and user interaction are required. This hampers the autonomous operation of ABFT schemes, which is an essential prerequisite for the successful application of ABFT, for instance in fault tolerant matrix operations on GPU.

With A-Abft, this work contributes a new method for *autonomously operating Algorithm-Based Fault Tolerance*, which enables the *efficient online determination of rounding error bounds for ABFT matrix operations.* The method is based on a probabilistic rounding error analysis and designed for parallel ABFT matrix operations on GPUs. The method supports partitioned encoding schemes with different block sizes and works independently of the chosen checksum encoding algorithm. It induces a low performance overhead and yields rounding error bounds which are up to two orders of magnitude closer to the actual rounding error than comparable state of the art techniques. The application of the method is shown for parallel ABFT-protected matrix multiplications on graphics processing units.

6.1 Overview of the Method

The goal of A-Abft is the *efficient online determination of rounding error bounds for the ABFT result checking procedure.* This goal is achieved by a probabilistic modeling and analysis of rounding errors that appear in floating-point computations and an efficient algorithm for the computation of rounding error bounds on graphics processing units [Braun14].

A central task of the A-Abft method comprises the determination of an appropriate confidence interval I_{ε_c} for the rounding error ε_c that affects an ABFT checksum c. The confidence interval is defined as follows:

Definition 6.2 (Confidence Interval for Rounding Error) *Given a column or row checksum c, which is affected by the rounding error ε_c. The* ***confidence interval for the rounding error*** *ε_c that affects the checksum c is defined as*

$$I_{\varepsilon_c} := [\mathtt{E}(\varepsilon_c) - \omega \cdot \sigma(\varepsilon_c),\ \mathtt{E}(\varepsilon_c) + \omega \cdot \sigma(\varepsilon_c)], \tag{6.8}$$

where $\mathtt{E}(\varepsilon_c)$ *is the* ***expectation value*** *of the rounding error and* $\sigma(\varepsilon_c)$ *is the* ***standard deviation*** *of the rounding error, which is defined as*

$$\sigma(\varepsilon_c) := \sqrt{\mathtt{Var}(\varepsilon_c)}, \tag{6.9}$$

where $\mathtt{Var}(\varepsilon_c)$ *is the* ***variance*** *of the rounding error, and* ω *is a scaling factor.*

To enable the computation of these terms, a probability distribution for the significands of the results of floating-point operations is required. As will be shown in this chapter, a *reciprocal distribution* can be utilized for this task. Based on this reciprocal distribution, a probability distribution for the *significand's error* will be deduced. This in turn allows the computation of the expectation value and the variance of the significand's error with respect to different floating-point operations. In a subsequent step, these terms are then used for the computation of the expectation value $\mathtt{E}(\varepsilon_c)$ and the variance $\mathtt{Var}(\varepsilon_c)$ of the A-Abft checksum's rounding error ε_c.

Section 6.2 introduces the probabilistic modeling and analysis of rounding errors in floating-point arithmetic and the assumptions that are made. Section 6.3 provides the empirical and theoretical justification for the reciprocal probability distribution. The formulas for the distribution, the expectation value and the variance of the significand's error are introduced in Section 6.4. Section 6.6 then introduces the formulas for the expectation value and the variance of the rounding error, and presents their application to the computation of inner products. The algorithm for the parallel computation of the rounding error bounds for A-Abft checksums, and its integration into an ABFT matrix multiplication on graphics processing units, are described in Section 6.7.

The formal foundations are first introduced for a general system of floating-point arithmetic based on a radix β. Where appropriate, formulas are then given for the binary case with radix $\beta = 2$ which matches the IEEE Standard 754™-2008 for floating-point arithmetic used by modern graphics processing units. The introduced A-Abft method can be directly applied to computations carried out in this floating-point standard.

6.2 Probabilistic Rounding Error Analysis

The probabilistic modeling and analysis of rounding errors in floating-point arithmetic has been introduced in [Barlo85a] and is described below. Basic arithmetic floating-point operations are considered at first to motivate and introduce the use of a reciprocal probability distribution for the significands of the results of such operations. A rounding algorithm with symmetric tie-breaking is assumed, which complies with the IEEE Standard 754™ for floating-point arithmetic. Other rounding algorithms can be applied as well and will lead to a second order difference in the computed expectation value of the rounding error.

Given a floating-point arithmetic system with radix β and two normalized, t-digit floating-point numbers a and b of the form

$$a := m_a \cdot \beta^{e_a} \quad \text{and} \quad b := m_b \cdot \beta^{e_b}. \tag{6.10}$$

Let s denote the *exact* result of a floating-point operation $\odot \in \{+, -, \cdot, /\}$ and s^* the t-digit *machine-computed* result. Then

$$s \equiv a \odot b \equiv s^* + \varepsilon, \tag{6.11}$$

where

$$\varepsilon := s - s^* \tag{6.12}$$

is the rounding error that occurred during the computation. The results s and s^* are assumed to have the same exponent. Cases where the exponents differ, arise with a probability of only $\mathcal{O}(\beta^{-t})$ and are not considered here. Hence, if $s = m_s \cdot \beta^e$ and $s^* = m_s^* \cdot \beta^e$, then m_s, $m_s^* \in [\frac{1}{\beta}, 1)$ are the significands and e is the exponent. The rounding error ε can now be expressed as

$$\varepsilon := s - s^* = m_s \cdot \beta^e - m_s^* \cdot \beta^e = \underbrace{(m_s - m_s^*)}_{=\varrho} \cdot \beta^e = \varrho \cdot \beta^e, \tag{6.13}$$

where ϱ denotes the *significand's error*.

In order to deal with the significand's error and in order to be able to perform a probabilistic rounding error analysis, an appropriate probability distribution is required. From the perspective of the integer numbers, a uniform distribution can be assumed, since these numbers can be seen as equally spaced points on a homogeneous, straight line. Floating-point numbers, however, are not equally spaced. In a floating-point number system, the distance between the two largest representable numbers is very large, whereas the distance between the two smallest representable positive numbers is comparatively small. A uniform probability distribution is therefore not suitable.

According to [Hammi70] and [Barlo85a], a reciprocal probability distribution over the interval $[\frac{1}{\beta}, 1)$ is therefore assumed for the approximation of the significands' distribution of results of floating-point operations $\odot \in \{+, -, \cdot, /\}$:

$$r(m_s) = \frac{1}{m_s \cdot \ln \beta} \quad \text{with } m_s \in [\frac{1}{\beta}, 1). \tag{6.14}$$

Based on this distribution $r(m_s)$, the probability distribution for the significand's error ϱ and the formulas for its expectation value $\mathtt{E}(\varrho)$ and variance $\mathtt{Var}(\varrho)$ with respect to different floating-point operations will be deduced in Section 6.4. However, the application of the reciprocal probability distribution has to be justified first.

6.3 Reciprocal Probability Distribution

In 1881, Newcomb observed during his studies, that the first pages of books with tables of logarithms were worn out much more than the pages in the rear parts. In [Newco81], he stated "*that the ten digits do not occur with equal frequency (...). The first significant figure is oftener 1 than any other digit, and the frequency diminishes up to 9.*" Newcomb further concluded: "*The law of probability of the occurrence of numbers is such that all mantissæ of their logarithms are equally probable.*"

In 1938, Benford described the same observation [Benfo38][1] and presented additional empirical evidence for the phenomenon which he gained through the examination of more than 20.000 first digits from various sources. He formulated the probability for a first decimal digit D_1, which became known as *Benford's Law*:

Definition 6.3 (Probability of First Digit - Benford's Law) *A given set of numbers satisfies Benford's Law, when the first leading digit D_1 appears with probability*

$$Prob(D_1 = d_1) = log_{10}\left(1 + \frac{1}{d_1}\right), \quad for\ all\ d_1 = 1, 2, ..., 9. \tag{6.15}$$

In 1995, Hill provided a more complete and more general form of Benford's Law as a statement about the joint distribution of all digits (*General Significant-Digit Law*) [Hill95a]

[1] It seems that Benford was not aware of Newcomb's work since he did not give any reference to [Newco81].

for an arbitrary base $\beta \geq 2$:

$$Prob\left((D_1^{(\beta)}, D_2^{(\beta)}, ..., D_m^{(\beta)},) = (d_1, d_2, ..., d_m)\right) = log_\beta \left(1 + \left(\sum_{j=1}^{m} \beta^{m-j} \cdot d_j\right)^{-1}\right), \tag{6.16}$$

where $D_1^{(\beta)}, D_2^{(\beta)}, ..., D_m^{(\beta)}$ are the first, second, etc. significant digits in base β, and where therefore d_1 is an integer in $\{1, 2, ..., \beta - 1\}$, and for $j \geq 2$, d_j is an integer in $\{0, 1, 2, ..., \beta - 1\}$. log_β denotes the logarithm to the base β, and m is a positive integer.

Benford's Law finds application in numerous areas, e.g. for the evaluation and analysis of errors due to overflow and underflow [Felds82, Felds86] or rounding in floating-point computations [Barlo81, Knuth81, Barlo83, Barlo85b]. Moreover, its application has been proposed for the design and evaluation of floating-point processing architectures [Hammi70, Knuth81, Schat81]. In the recent past, methods for the detection of manipulations and fraud in statistical data, as well as forensic financial accounting for the analysis of corporate balances [Nigri97, Nigri12], have become an emerging field of application.

Although not all possible sets of data show Benford characteristics, solid empirical evidence for the appearance of Benford's Law can be found in many different areas. In [Berge11], Berger and Hill summarize its appearance in the set of physical constants [Knuth81, Burke91], the results of scientific computations [Hammi70, Knuth81], geographical data [Benfo38], election results and census data, as well as financial accounting data [Nigri97, Nigri12]. Moreover, the authors mention the appearance of Benford' Law in solutions of ordinary and partial differential equations, products of independent, identically distributed random variables, in mixtures of random samples, and stochastic models like geometric Brownian motions, order statistics, random matrices, Lévy processes and Bayesian models [Leemi00, Engel03, Berge05, Mille08, Schur08].

Over time, numerous attempts have been made to prove Benford's Law. Benford himself tried to prove the law as some "built-in characteristic of our number system" [Hill98] by considering the natural numbers $\mathbb{N}$ and their frequencies. Benford chose an integration approach and studied the non-continuous but directly connected intervals 10.000 – 20.000, 20.000 – 99.999 and 99.999 – 100.000. He then stated that "*the area under the curve will be very closely 0.30103, where the entire area of the frame of coordinates has an area 1.*"

Flehinger [Flehi66] followed a similar approach and applied a re-iterated-averaging technique (Cesàro summation) to define a generalized density which assigns the correct Benford value $\log_{10}(2)$ to $\{D_1 = 1\}$ [Hill95a].

A major contribution to the explanation of Benford's Law has been made by Pinkham, who introduced an invariance principle for the law's characterization. In [Pinkh61], he reasoned that if there exists something like a *universal law* for the distribution of first digits, such a law has to be independent of units. Pinkham showed that the logarithmic law is scale-invariant and that it is the only law with this property. The basic idea of invariance prepared the ground for further, more comprehensive explanations of Benford's Law.

Today, the most complete explanation of Benford's Law is given by Hill who provides an extension of the invariance reasoning from scale- to base-invariance [Hill95b], and who puts Benford's Law into a proper, countably additive probability framework [Hill95a]. The concept of base-invariance refers to the requirement that a universal law has to be independent of the base used for the representation of numbers and the general idea of Hill's approach is to think of data as being a mixture of random samples from different probability distributions.

An excellent introduction to Benford's Law, the different approaches for its characterization and explanations, as well as different applications can be found in [Berge11].

6.4 Distribution, Expectation Value and Variance of the Significand's Error

This section introduces the expectation value $\mathtt{E}(\varrho)$ and the variance $\mathtt{Var}(\varrho)$ of the significand's error ϱ (cf. Equation 6.13) with respect to the four basic arithmetic floating-point operations $\odot \in \{+, -, \cdot, /\}$.

As pointed out in Section 6.1, the derivation of a probability distribution for the significand's error is based on the assumption of a reciprocal distribution of floating-point significands (Equation 6.14), which has been justified by Benford's Law in Section 6.3. To derive the distribution of the significand's error ϱ and to compute its expectation value and variance, answers to the following questions are required: How are the probability distributions of floating-point numbers combined and transformed, when the four basic arithmetic operations carried out? How do the intermediate and trailing digits of floating-point numbers behave under these operations? The second question is particularly important because these digits are the parts of floating-point numbers that are affected by the significand's error and hence the rounding error.

In [Hammi70], Hamming showed for the first time how the distributions of floating-point numbers are combined when operations $\odot \in \{+, -, \cdot, /\}$ are performed. He proved that the reciprocal distribution persists under multiplication and division, which means that, if one of the factors in these operations comes from the reciprocal distribution and regardless of the distribution of the other factors, the result's probability distribution will be the reciprocal distribution. Turner [Turne82] also showed that for the results of arithmetic operations, for instance resulting from successive multiplications, the leading significant digits are not uniformly distributed, but follow the logarithmic distribution. In [Turne84], he extended his results by showing, without any assumption of scale-invariance, that the logarithmic distribution remains invariant under all further arithmetic operations–both multiplicative and additive. Barlow and Bareiss [Barlo85a] summarized the existing conditions and statements about the reciprocal distribution of floating-point significands, and they introduced a theorem for the justification of this distribution, which explicitly considers the impact of long sequences of multiplications and divisions on the distribution.

Feldstein and Goodman [Felds76] investigated the distribution of trailing digits in positive floating-point numbers for arbitrary bases β and their analysis showed that the n-th digit ($n \geq 2$) is approximately uniformly distributed. They also showed that the approximation depends on the base β and n, and that it becomes better for $n \to \infty$. Bustoz et al. [Busto79] extended these investigations and presented additional results on the distribution of trailing digits for logarithmically distributed numbers and for errors in floating-point multiplications. Based on this prior work and under the assumption that the significands m have a probability distribution whose density is Lipschitz continuous, Barlow and Bareiss derived a distribution for the significand's error $\varrho = m - m^*$, the k discarded digits after the application of the four basic arithmetic operations $\odot \in \{+, -, \cdot, /\}$:

Theorem 6.1 (Probability Distribution of the Significand's Error) *Let $m \cdot \beta^e$ be a real number where $m \in [\frac{1}{\beta}, 1)$ follows a probability distribution F with continuous* *density $f(m) = F^{'}(m)$ satisfying*

$$|f(m) - f(y)| \leq K \cdot |m - y| \quad \textit{for all } m \in [\frac{1}{\beta}, 1) \tag{6.17}$$

for a fixed constant K. Let $m^* \cdot \beta^e$ be $m \cdot \beta^e$ truncated to t digits and k be the number of discarded digits, which are represented below by $[.m_{t+1}...m_{t+k}]$. Define

$$a := \lfloor (m - m^*) \cdot \beta^{-t+k} \rfloor \cdot \beta^{-k} = [.m_{t+1}...m_{t+k}] \tag{6.18}$$

and let $Q_k^t(A)$ be the probability distribution of $A = \varrho \cdot \beta^t$. Then

$$\lim_{k \to \infty} Q_k^t(A) = U(A) + \mathcal{O}(\beta^{-t}) \textit{ and} \tag{6.19}$$

$$\lim_{t \to \infty} Q_k^t(A) = U(A) + \mathcal{O}(\beta^{-k}), \tag{6.20}$$

where

$$U(A) = \begin{cases} 0 \textit{ if } A < 0 \\ A \textit{ if } A \in [0, 1) \\ 1 \textit{ if } A \geq 1. \end{cases} \tag{6.21}$$

The proof of this theorem is omitted and can be found in [Barlo85a] (pages 333-335). Since Barlow and Bareiss approximated the density function f by the reciprocal distribution, they only considered problems involving the values of k and t. For the modeled problems, such as the inner product in Section 6.6, k is the number of digits discarded in the operational error. In multiplications $k = t$ and in divisions $k = \infty$. For addition and subtraction the number k of discarded digits is between 0 and the largest representable floating-point number, but is frequently less than t. Thus, Barlow and Bareiss approximate $Q_k^t(A)$ by $U(A)$ when dealing with multiplications or divisions, but approximate Q_k^t by $\lfloor A \cdot \beta^{-k} \rfloor \cdot \beta^k$ when dealing with errors from additions and subtractions.

Expectation Value and Variance of the Significand's Error for Multiplications and Divisions

Based on the distribution of the significand's error ϱ and the assumption of a symmetric rounding algorithm, the expectation value $\mathtt{E}(\varrho)$ of the significand's error for multiplications and divisions is derived as [Barlo85a] (page 336):

Derivation 6.1 (Expectation Value of Significand's Error (Mul./Div.)) *Given a floating-point number system with radix β and t-digit floating-point numbers. The* ***expectation value of the significand's error in multiplications and divisions*** *is derived as*

$$\mathtt{E}(\varrho) = \frac{\beta^{-2t}}{48}\left(12 \cdot (1+\beta) - \frac{(\beta+1)(\beta^2+1)}{ln\beta}\right) + \mathcal{O}(\beta^{-3t}), \tag{6.22}$$

and in the case of radix $\beta = 2$ as

$$\begin{aligned}\mathtt{E}(\varrho) &= \frac{2^{-2t}}{48}\left(12(1+2) - \frac{(2+1)(4+1)}{\ln 2}\right) + \mathcal{O}(2^{-3t}) &(6.23)\\ &\leq \frac{1}{3}\cdot 2^{-2t} + \mathcal{O}(2^{-3t}). &(6.24)\end{aligned}$$

In the course of this work, the higher order term is omitted and the expectation value $\mathtt{E}(\varrho)$ in case of multiplications and divisions is computed by

$$\mathtt{E}(\varrho) = \frac{1}{3}\cdot 2^{-2t}. \tag{6.25}$$

The variance $\mathtt{Var}(\varrho)$ for multiplications and divisions is derived as

Derivation 6.2 (Variance of Significand's Error (Mul./Div.)) *Given a floating-point number system with radix β and t-digit floating-point numbers. The* ***variance of the significand's error in multiplications and divisions*** *is*

$$\mathtt{Var}(\varrho) = \frac{1}{12}\cdot \beta^{-2t} + \mathcal{O}(\beta^{-3t}), \tag{6.26}$$

and in the case of base $\beta = 2$

$$\mathtt{Var}(\varrho) = \frac{1}{12}\cdot 2^{-2t} + \mathcal{O}(\beta^{-3t}). \tag{6.27}$$

Again, the higher order term is omitted in the following and the variance is computed by

$$\mathtt{Var}(\varrho) = \frac{1}{12}\cdot 2^{-2t}. \tag{6.28}$$

Expectation Value and Variance of the Significand's Error for Additions and Subtractions

Since the number k of discarded digits varies, the distribution of the significand's error ϱ for additions and subtractions is more complicated to be determined. Barlow and Bareiss circumvent this problem by bounding the expectation value $\mathtt{E}(\varrho)$ and the variance $\mathtt{Var}(\varrho)$. Hence, for the addition and subtraction of two floating point numbers the expectation value $\mathtt{E}(\varrho)$ is derived as [Barlo85a] (page 339):

Derivation 6.3 (Expectation Value of the Significand's Error (Add./Sub.)) *Given a floating-point number system with radix β and t-digit floating-point numbers. The **expectation value of the significand's error in additions and subtractions** is*

$$\mathtt{E}(\varrho) = 0. \tag{6.29}$$

The variance $\mathtt{Var}(\varrho)$ is derived as [Barlo85a] (page 339):

Derivation 6.4 (Variance of the Significand's Error (Add./Sub.)) *Given a floating-point number system with radix β and t-digit floating-point numbers. The **variance of the significand's error in additions and subtractions** is*

$$\mathtt{Var}(\varrho) \leq \beta^{-2t} \cdot \left(\frac{1}{12} + \frac{1}{6 \cdot \beta^2} \right). \tag{6.30}$$

In the case of base $\beta = 2$ as

$$\mathtt{Var}(\varrho) \leq \frac{1}{8} \cdot 2^{-2t}. \tag{6.31}$$

6.5 Expectation Value and Variance of Rounding Errors

Based on the established distribution and the equations of the expectation value and the variance of the significand's error, the respective equations for the actual rounding error ε in the result of a floating-point operation can be introduced according to [Barlo85b] and [Barlo85a] (page 340).

Derivation 6.5 (Expectation Value of the Rounding Error) *Given a floating-point number system with radix β and exponent e. Let r be the result of a floating-point operation, which is affected by the rounding error ε_r. The* ***expectation value*** $\mathtt{E}(\varepsilon_r)$ *of the rounding error ε_r is*

$$\mathtt{E}(\varepsilon_r) = sgn(r) \cdot \beta^e \cdot \mathtt{E}(\varrho), \tag{6.32}$$

where the exponent is $e = \lceil \log_\beta |r| \rceil$ and $\mathtt{E}(\varrho)$ is the expectation value of the significand's error with respect to the performed arithmetic operation $\odot \in \{+,-,\cdot,/\}$.

In case of base $\beta = 2$ the expectation value is

$$\mathtt{E}(\varepsilon_r) = sgn(r) \cdot 2^e \cdot \mathtt{E}(\varrho). \tag{6.33}$$

Derivation 6.6 (Variance of the Rounding Error) *Given a floating-point number system with radix β and exponent e. Let r be the result of a floating-point operation, which is affected by the rounding error ε_r. The* ***variance*** $\mathtt{Var}(\varepsilon_r)$ *of the rounding error ε_r is*

$$\mathtt{Var}(\varepsilon_r) = \beta^{2e} \cdot \mathtt{Var}(\varrho), \tag{6.34}$$

where the exponent is $e = \lceil \log_\beta |r| \rceil$ and $\mathtt{Var}(\varrho)$ is the variance of the significand's error with respect to the performed arithmetic operation $\odot \in \{+,-,\cdot,/\}$.

In case of base $\beta = 2$ the variance is

$$\mathtt{Var}(\varepsilon_r) = 2^{2e} \cdot \mathtt{Var}(\varrho). \tag{6.35}$$

For a floating-point number s which is the result of a sequence of n independent basic arithmetic operations, the expectation value, the variance and the standard deviation of the rounding error ε_s are given by

$$\mathtt{E}(\varepsilon_s) = \sum_{i=1}^{n} \mathtt{E}(\varepsilon_{r_i}) \tag{6.36}$$

$$\mathtt{Var}(\varepsilon_s) = \sum_{i=1}^{n} \mathtt{Var}(\varepsilon_{r_i}) \tag{6.37}$$

$$\sigma(\varepsilon_s) = \sqrt{\mathtt{Var}(\varepsilon_s)}. \tag{6.38}$$

6.6 Rounding Error Bounds for A-Abft Checksums

The method for the determination of rounding error bounds for A-Abft checksums builds upon the formal foundations that have been introduced in the previous sections. The method is described using the example of an A-Abft-protected matrix multiplication on the GPU. Nonetheless, with the equations for the expectation values and variances of the four basic arithmetic operations given by [Barlo85a], this method can be used to derive rounding error bounds for other A-Abft-protected operations.

Given the example of an A-Abft matrix multiplication $\boldsymbol{C_{fc}} = \boldsymbol{A_{cc}} \cdot \boldsymbol{B_{rc}}$ on the GPU (cf. Theorem 5.1), each checksum element $c_{i,j}$ in the resulting full-checksum matrix $\boldsymbol{C_{fc}}$ is formed as an inner product of a row vector from the column-checksum matrix $\boldsymbol{A_{cc}}$ and a column vector from the row-checksum matrix $\boldsymbol{B_{rc}}$. Such an inner product is a combination of multiplications (intermediate products $\hat{c}_r$) and their summation:

$$c_{i,j} = \sum_{r=1}^{n} a_{i,r} \cdot b_{r,j} = \sum_{r=1}^{n} \hat{c}_r. \tag{6.39}$$

The determination of the rounding error bound for the checksum element $c_{i,j}$ therefore requires the consideration of the combination of rounding errors from the multiplications and the summation.

For the summation s_n^* of n intermediate products $\hat{c}_1^*, \ldots, \hat{c}_n^*$, the following recursion is derived for the GPU-computed result and the rounding error ε_k:

Derivation 6.7 (Recursion for Summation of Intermediate Products) *Given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ for which the inner product*

$$c = \sum_{r=1}^{n} a_{i,r} \cdot b_{r,j} = \sum_{r=1}^{n} \hat{c}_r, \tag{6.40}$$

has to be computed with the intermediate products $\hat{c}_r$. The summation of the intermediate products on the GPU can be expressed by the following recursion:

$$s_{k+1}^* + \varepsilon_{k+1} = s_k^* + \hat{c}_{k+1}, \quad \text{for } k = 1, \ldots, n-1, \tag{6.41}$$

where the ε_k are the involved rounding errors.

Derivation 6.8 (Recursion for the Rounding Errors) *The recursion for the difference*

$$\Delta s_k = s_k - s_k^* \tag{6.42}$$

between the exact result and the GPU-computed result, which represents the rounding error, is

$$\Delta s_{k+1} = \Delta s_k + \varepsilon_{k+1} \tag{6.43}$$

and can be expressed as

$$\Delta s_n = \sum_{k=2}^{n} \varepsilon_k. \tag{6.44}$$

For the probabilistic determination of the rounding error within the sum s_n, the confidence interval comprising the expectation value $\mathtt{E}_\Sigma(\Delta s_n)$ and the standard deviation $\sigma_\Sigma = \sqrt{\mathtt{Var}_\Sigma(\Delta s_n)}$ has to be formed. According to Equations 6.36

$$\begin{aligned} \mathtt{E}_\Sigma(\Delta s_n) &= \sum_{k=2}^{n} \mathtt{E}(\varepsilon_k) && (6.45) \\ \mathtt{Var}_\Sigma(\Delta s_n) &= \sum_{k=2}^{n} \mathtt{Var}(\varepsilon_k). && (6.46) \end{aligned}$$

With Equations 6.29 and 6.31 for the addition and subtraction of two floating-point numbers, the application of the Equation 6.32 yields

$$\mathtt{E}_\Sigma(\Delta s_n) = 0 \tag{6.47}$$

and

$$\begin{aligned} \mathtt{Var}_\Sigma(\Delta s_n) &= \sum_{k=2}^{n} \mathtt{Var}(\varepsilon_k) && (6.48) \\ &\leq \sum_{k=2}^{n} 2^{2e_k} \cdot \mathtt{Var}(\varrho) && (6.49) \\ &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} 2^{2e_k} && (6.50) \end{aligned}$$

where e_k denotes the exponent of the intermediate result s_k^* after the addition of the k-th addend. Since the terms in the sum are independent the variance is bound by

$$\mathtt{Var}_\Sigma(\Delta s_n) \leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} (s_k^*)^2. \tag{6.51}$$

For an upper bound y with $s_k^* \leq k \cdot y$, Equation 6.51 can be bounded by

$$\begin{aligned} \texttt{Var}_\Sigma(\Delta s_n) &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} (k \cdot y)^2 && (6.52) \\ &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \left(\frac{n \cdot (n+1) \cdot (2n+1)}{6} \right) \cdot y^2. && (6.53) \end{aligned}$$

With the standard deviation $\sigma_\Sigma(\Delta s_n)$, the confidence interval is determined by

$$\texttt{E}_\Sigma(\Delta s_n) = 0 \tag{6.54}$$

and

$$\sigma_\Sigma(\Delta s_n) \leq \sqrt{\frac{n \cdot (n+1) \cdot (2n+1)}{48}} \cdot y \cdot 2^{-t} \tag{6.55}$$

To complete the derivation of the confidence interval for the rounding error in the inner product, the above considerations have to be extended with respect to the rounding error α_k caused by the multiplications $\hat{c}_k = a_{i,k} \cdot b_{k,j}$, which is

$$\hat{c}_k^* + \alpha_k = \hat{c}_k. \tag{6.56}$$

The recursion for the rounding error in the summation within the inner product therefore changes to

$$\Delta s_{k+1} = \Delta s_k + \varepsilon_k + \alpha_k, \tag{6.57}$$

which can be expressed as

$$\Delta s_n = \sum_{k=2}^{n} \varepsilon_k + \sum_{k=1}^{n} \alpha_k. \tag{6.58}$$

Analog to the Equations 6.45, the expectation value and the variance can be derived as

$$\begin{aligned} \texttt{E}_{\Sigma\Pi}(\Delta s_n) &= \texttt{E}_\Sigma(\Delta s_n) + \texttt{E}_\Pi(\Delta s_n) && (6.59) \\ &= \sum_{k=2}^{n} \texttt{E}(\varepsilon_k) + \sum_{k=1}^{n} \texttt{E}(\alpha_k) && (6.60) \end{aligned}$$

and

$$\begin{aligned} \texttt{Var}_{\Sigma\Pi}(\Delta s_n) &= \texttt{Var}_\Sigma(\Delta s_n) + \texttt{Var}_\Pi(\Delta s_n) && (6.61) \\ &= \sum_{k=2}^{n} \texttt{Var}(\varepsilon_k) + \sum_{k=1}^{n} \texttt{Var}(\alpha_k). && (6.62) \end{aligned}$$

With the Equations 6.25 and 6.28 the contribution of the rounding error variance after k multiplications in the sum of all partial variances is

$$\texttt{Var}_\Pi(\Delta s_n) = \sum_{k=1}^{n} \texttt{Var}(\alpha_k). \tag{6.63}$$

For $y = a_d \cdot b_d$ with $d \in \{1, \ldots, n\}$, where the variance $\mathtt{Var}(\alpha_d)$ is maximal, Equation 6.63 can be bounded by

$$\mathtt{Var}_{\Pi}(\Delta s_n) \leq n \cdot \mathtt{Var}(\alpha_d). \tag{6.64}$$

The variance is maximal if the exponent e of the multiplication's result is maximal

$$\mathtt{Var}(\alpha_d) = 2^{2e_d} \cdot \mathtt{Var}(\varrho) \geq 2^{2e_k} \cdot \mathtt{Var}(\varrho) = \mathtt{Var}(\alpha_k). \tag{6.65}$$

Using y as upper bound, the variance of the rounding error from the multiplications is

$$\begin{aligned} \mathtt{Var}_{\Pi}(\Delta s_n) &\leq n \cdot \mathtt{Var}(\alpha_y) && (6.66) \\ &\leq n \cdot y^2 \cdot \mathtt{Var}(\varrho) && (6.67) \\ &= \frac{1}{12} \cdot n \cdot 2^{-2t} \cdot y^2. && (6.68) \end{aligned}$$

Analog, the expectation value is derived as

$$\begin{aligned} \mathtt{E}_{\Pi}(\Delta s_n) &= n \cdot y \cdot \mathtt{E}(\varrho) && (6.69) \\ &\leq \frac{1}{3} \cdot n \cdot 2^{-2t} \cdot y. && (6.70) \end{aligned}$$

With the expectation values and the variances from the summation and the multiplications, the confidence interval for the rounding error in the final inner product can be formed using the standard deviation

$$\begin{aligned} \sigma_{\Sigma\Pi}(\Delta s_n) &= \sqrt{\mathtt{Var}_{\Sigma\Pi}(\Delta s_n)} && (6.71) \\ &= \sqrt{\mathtt{Var}_{\Sigma}(\Delta s_n) + \mathtt{Var}_{\Pi}(\Delta s_n)} && (6.72) \\ &\leq \sqrt{\frac{n \cdot (n+1) \cdot (n + \frac{1}{2}) + 2n}{24}} \cdot 2^{-t} \cdot y. && (6.73) \end{aligned}$$

6.7 Parallel Computation of Rounding Error Bounds

This section introduces into to the parallel determination of probabilistic rounding error bounds on GPU. Taking the example of an A-ABFT-protected matrix multiplication that uses a partitioned encoding scheme, the typical problem sizes of interest for execution on GPUs imply that large numbers of checksums have to be processed during the A-ABFT checking procedure. In order to keep the performance overhead low, the proposed method performs this step in parallel for all checksum elements.

As it has been shown in the foundations presented above, an appropriate upper bound y has to be identified before the respective equations for the expectation value, the variance and the standard deviation can be evaluated. This upper bound is essential, because it has direct influence on the quality of the final rounding error bound. The smaller the upper bound y can be chosen, the closer the rounding error bound will be to the actual rounding error. An algorithm for the identification of such upper bounds y is therefore introduced before the complete algorithm for the efficient online determination of rounding error bounds for A-ABFT checksums is presented.

Identification of Upper Bounds

For the A-ABFT-protected matrix multiplication $\boldsymbol{A_{cc}} \cdot \boldsymbol{B_{rc}} = \boldsymbol{C_{fc}}$, every element in the full-checksum result matrix is formed as an inner product of a row vector $\boldsymbol{a}_i$ from the column-checksum matrix $\boldsymbol{A_{cc}}$ and a column vector $\boldsymbol{b}_j$ of the row-checksum matrix $\boldsymbol{B_{rc}}$. A trivial approach for the identification of an upper bound y would select the element with the largest absolute value from each of the vectors $\boldsymbol{a}_i$ and $\boldsymbol{b}_j$, and then take the product of these two elements as y. Although this procedure always yields a valid upper bound, this bound will be a *worst-case bound*, which is often too pessimistic. This is due to the fact that it is not guaranteed that exactly the two maximal elements are actually combined during the formation of the inner product.

The identification of an improved upper bound y depends on the position or, more precisely, the indices i and j of the vector elements. The proposed algorithm determines two sets of candidate elements from both involved vectors and checks whether combinations of elements from these two sets exist.

Definition 6.4 (Candidate Sets) *Given a row vector $\boldsymbol{a}_i$ from a column-checksum matrix $\boldsymbol{A_{cc}}$, and column vector $\boldsymbol{b}_j$ form a row-checksum matrix $\boldsymbol{B_{rc}}$.*

The ***candidate set*** *A_{idx} of vector $\boldsymbol{a}_i$ consists of the p vector elements from $\boldsymbol{a}_i$ with the largest absolute values and their index positions within the vector.*

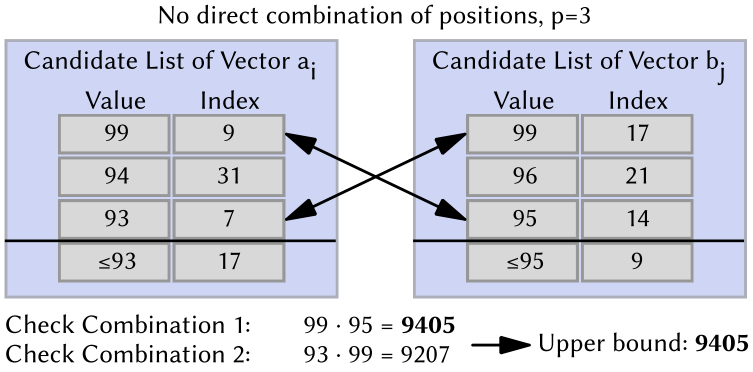
The ***candidate set*** *B_{idx} of vector $\boldsymbol{b}_j$ consists of the p vector elements from $\boldsymbol{b}_j$ with the largest absolute values and their index positions within the vector.*

The candidate sets A_{idx} and B_{idx} are generated during a scan operation in which the absolute value of each vector element is computed. An element is added to the candidate set if its absolute value is larger than the absolute value of the currently smallest element within the set. Besides the absolute value of the element, its index is stored. The candidate sets can contain up to p elements with largest absolute values. After the candidate sets are generated, the algorithm cross-checks the indices in the two sets for direct combinations.

For any indices i and j the upper bound $y_{i,j}$ is then determined as the ***maximum** of the following three cases*:

- $S = A_{idx} \cap B_{idx} \neq \varnothing$ and large elements from vector a_i and vector b_j are multiplied $\Rightarrow max\{|a_s \cdot b_s|\}$ with $s \in S$.
- $S = A_{idx} \cap B_{idx} = \varnothing$, but the maximum of the p absolute values from vector a_i is multiplied with any of the elements from vector $b_j \Rightarrow max\{|a_r|\} \cdot min\{|b_s|\}$, with $r \in A_{idx}$ and $s \in B_{idx}$.
- $S = A_{idx} \cap B_{idx} = \varnothing$, but the maximum of the p absolute values from vector b_j is multiplied with any of the elements from vector $a_i \Rightarrow max\{|b_s|\} \cdot min\{|a_r|\}$, with $r \in A_{idx}$ and $s \in B_{idx}$.

Figures 6.1 and 6.2 depict the two situations that can occur during the check of the candidate set. In Figure 6.1 the two candidate sets have been determined for $p = 3$ and the indices of the elements show that there are no direct combinations. In this case, the maximum element from A_{idx} is multiplied with the minimum element from B_{idx} and vice versa. The maximum of these two products is used as upper bound y.



Candidate List of Vector a_i

Value	Index
99	9
94	31
93	7
≤93	17

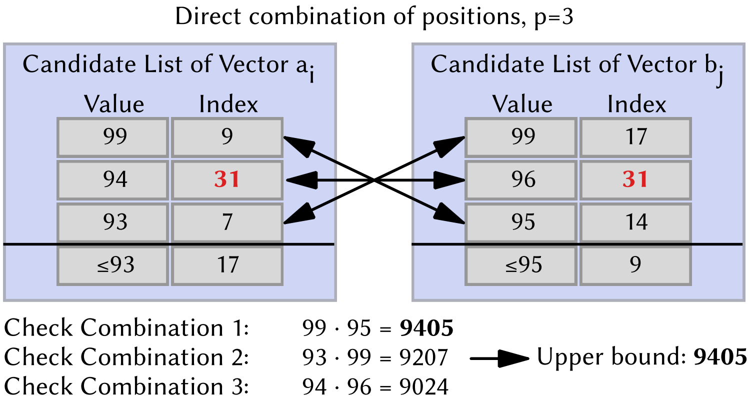
Candidate List of Vector b_j

Value	Index
99	17
96	21
95	14
≤95	9

Check Combination 1: 99 · 95 = **9405**
Check Combination 2: 93 · 99 = 9207
Upper bound: **9405**

▲ **Figure 6.1** — Check of candidate sets, p=3, no direct combinations.

Figure 6.2 depicts the case where a direct combination between two elements from vector $\boldsymbol{a}_i$ and vector $\boldsymbol{b}_j$ occur. In this case, the product of this direct combination has also to be considered. The maximum of the three products is taken as upper bound y.



▲ **Figure 6.2** — Check of candidate sets, p=3, direct combinations.

The quality of the error bound can be improved by increasing the number p of considered largest absolute values from both vectors. However, an increasing number p also increases the computational overhead. Extensive experimental evaluations of the impact of p on the quality of the rounding error bound, as well as the induced computational overhead are presented in Chapter 8.

Parallel Computation of Rounding Error Bounds on GPU

In the following, the integration of the developed method for the online determination of rounding error bounds is demonstrated using the example of an A-ABFT-protected matrix multiplication on GPU [Braun14]. The multiplication utilizes a partitioned encoding scheme with blocks of dimension $bs \times bs$, as it has been introduced in Chapter 5. Terms like *thread* and *thread block* are used in this section in compliance with the GPU programming model described in Chapter 3.

The basic algorithm for the A-ABFT matrix multiplication on GPU comprises of three major phases:

1. The input data are encoded with checksums,
2. the matrix multiplication is performed,
3. the A-ABFT checking procedure is executed on the computed matrix product.

The determination of the rounding error bounds is integrated into this scheme by separating the *generation of candidate sets for the identification of upper bounds* and the *computation of the rounding error bounds.* The generation of the candidate sets becomes part of the encoding phase (1) and the determination of the upper bounds y, as well as the computation of the error bounds, becomes part of the result checking procedure in phase (3). This leads to the following algorithmic steps, which are encapsulated into GPU kernels that are executed by thread blocks of dimension $bs \times 1$:

1. **GPU kernel for checksum encoding and candidate set generation.**
   a) Computation of the column checksums for the $bs \times bs$ sub-matrix $\boldsymbol{A_i}$ of $\boldsymbol{A_{cc}}$.
   b) Generation of the candidate set $A_{idx,i}$ with the p largest absolute value elements in each sub-matrix $\boldsymbol{A_i}$ of matrix $\boldsymbol{A_{cc}}$.
   c) Computation of the row checksums for the $bs \times bs$ sub-matrix $\boldsymbol{B_i}$ of $\boldsymbol{B_{rc}}$.
   d) Generation of the candidate sets $B_{idx,i}$ with the p largest absolute value elements in each sub-matrix $\boldsymbol{B_i}$ of matrix $\boldsymbol{B_{rc}}$.
2. **Computation of the matrix product.**
3. **GPU kernel for the reduction of the block-wise determined candidate sets $A_{idx,i}$ and $B_{idx,i}$ into the candidate sets A_{idx} and B_{idx} for the p global elements with the largest absolute values per row/column vector.**
4. **GPU kernel for the determination of upper bounds, computation of rounding error bounds and A-ABFT checking procedure.**
   a) Computation of the rounding error bounds for the row and column checksum elements of each $bs \times bs$ result sub-matrix $\boldsymbol{C_i}$ of the full checksum matrix $\boldsymbol{C_{fc}}$.
   b) Computation of the reference row and column checksums for each $bs \times bs$ result sub-matrix $\boldsymbol{C_i}$ of the full checksum matrix $\boldsymbol{C_{fc}}$.
   c) Comparison of the original and the reference checksums using the computed rounding error bounds.

Kernel for checksum encoding and candidate set generation

This GPU kernel is launched in step 1) with thread blocks of dimension $bs \times 1$, where each row or column of the sub-matrix is processes by a single thread. It combines the computation of A-ABFT row or column checksums with the generation of the candidate sets for the p elements with the largest absolute values. The kernel reduces expensive accesses to the global GPU device memory by loading the elements of the processed sub-matrix into the shared memory. In a first step, the threads iterate over the elements of a sub-matrix from top to bottom (column checksum case) or from left to right (row checksum case) and accumulate the elements into the respective checksum elements. After an element has been added, it is replaced in shared memory by its absolute value. In a subsequent step, the threads re-iterate over the sub-matrix elements to identify the p elements with the largest absolute values and store the indices (positions) of these elements. Since the p elements with the largest absolute values are determined within each sub-matrix, a total of $\frac{m}{bs} \cdot p$ such elements has been found at the end, where $\frac{m}{bs}$ is the number of sub-matrices. Hence, an additional reduction kernel (step 3) is executed in parallel to the matrix multiplication (step 2) to reduce the candidate sets to the p global elements with the largest absolute values for each vector. Algorithm 1 in Appendix E describes all performed steps in detail.

Kernel for the determination of upper bounds, computation of rounding error bounds and A-Abft checking procedure

The A-Abft-protected matrix multiplication is finalized by the invocation of this last kernel which performs a sequence of four major tasks. First, the upper bounds y are determined based on the candidate sets which have been generated in step 1) and have been reduced to global candidate sets for the vectors in parallel to the matrix multiplication. The kernel checks these sets for index combinations and computes the upper bounds according to the rules introduced above. The upper bounds y are then used in the second task for the evaluation of the equations of the expectation value and the standard deviation of the rounding error to determine the rounding error bound ε. In a subsequent step, the reference checksums are computed and the comparison for the A-Abft check is performed. In the error-free case the result of the matrix multiplication is returned. In case of errors, the corresponding correction procedures are invoked. Algorithm 2 in Appendix E shows the different steps for the column checksum case in detail. A first efficient implementation of the proposed method [Braun14] has been introduced in [Halde14].

Chapter 7

Case Studies on the Application of A-ABFT

In addition to the detailed experimental evaluation of the introduced A-Abft method in Chapter 8, two case studies on the application of A-Abft in real world examples of high practical relevance are presented in this chapter. First, a QR decomposition based on Householder transformations is realized using A-Abft-protected matrix multiplications on a GPU. Second, a linear programming solver based on the Revised Simplex algorithm is presented, which also utilizes A-Abft-protected GPU matrix multiplications.

7.1 A GPU-Accelerated QR Decomposition

The factorization or *decomposition* of matrices belongs to the most fundamental operations in linear algebra. The objective of a matrix decomposition is to represent a given matrix $\boldsymbol{A}$ as product of two other matrices with some special characteristics. A very common application of matrix decompositions is the solution of systems of linear equations. Depending on the actual application and the intended target decomposition, different factorization methods can be applied. Among the most important methods are the *LU decomposition*, the *Cholesky decomposition*, as well as the *QR decomposition*.

The interested reader can find a comprehensive introduction to matrix factorizations in [Golub12].

In this first case study, a QR decomposition based on Householder transformations has been implemented, which utilizes ABFT-protected GPU matrix operations. The QR decomposition serves as test bench for the further evaluation and comparison of the proposed A-Abft method and the SEA-ABFT approach. QR decompositions across different matrix dimensions have been combined with fault injection experiments to investigate the efficiency and efficacy of both ABFT methods.

7.1.1 QR Decomposition of Matrices

The goal of a *QR decomposition* is to represent a given matrix $\boldsymbol{A}$ as product of two matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$:

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}, \tag{7.1}$$

where $\boldsymbol{Q}$ is an orthogonal matrix[1] with $\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}$ and $\boldsymbol{R}$ is an upper right triangular matrix. The above decomposition is given for the square case of $\boldsymbol{A} \in \mathbb{R}^{n \times n}$. The QR decomposition can also be defined for rectangular matrices $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ with $m \geq n$ as

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R} = \boldsymbol{Q}\begin{bmatrix} \boldsymbol{R}_1 \\ 0 \end{bmatrix} = [\boldsymbol{Q}_1\ \boldsymbol{Q}_2]\begin{bmatrix} \boldsymbol{R}_1 \\ 0 \end{bmatrix} = \boldsymbol{Q}_1\boldsymbol{R}_1, \tag{7.2}$$

where $\boldsymbol{R}_1$ is a $n \times n$ upper right triangular matrix which is padded at the bottom with a $m - n \times n$ matrix 0 consisting of zeros. Corresponding to the partitioning, $\boldsymbol{Q}_1$ is a $m \times n$ and $\boldsymbol{Q}_2$ a $m \times m - n$ matrix. The partitioned representation of the QR decomposition in the rectangular case is often called *reduced* or *thin QR decomposition* [Golub12].

In general, three options exist for the computation of a QR decomposition, including the *Gram-Schmidt method*, *Givens rotations* and *Householder transformations*. Since the QR decomposition with Householder transformations can be expressed in terms of matrix multiplications, it is particularly attractive for the integration of ABFT-protected GPU matrix operations.

7.1.2 Householder Transformations

Householder transformations are linear transformations, introduced 1958 by Alston Scott Householder [House58], which reflect a given vector on a hyperplane through the

[1] In the complex case, $\boldsymbol{Q}$ is an unitary matrix with $\boldsymbol{Q}^*\boldsymbol{Q} = \boldsymbol{I}$

coordinate origin. Due to this reflection property, these linear transformations are also often called *Householder reflections*. The main feature of this transformation is that all components of the vector, with the exception of one, are reduced to zero. Householder transformations can be expressed as so called *Householder matrices* and provide an elegant way for the computation of QR decompositions.

The reflection plane can be described by a normal vector $\boldsymbol{v}$, which is orthogonal to this plane. This yields the definition of the Householder matrix $\boldsymbol{H}$ as

$$\boldsymbol{H} := \boldsymbol{I} - \frac{2\boldsymbol{v}\boldsymbol{v}^T}{\boldsymbol{v}^T\boldsymbol{v}}, \tag{7.3}$$

where $\boldsymbol{I}$ is the identity matrix. Householder matrices are orthogonal ($\boldsymbol{H}^T = \boldsymbol{H}^{-1}$), symmetric ($\boldsymbol{H} = \boldsymbol{H}^T$), and hence involutory ($\boldsymbol{H}^2 = \boldsymbol{I}$). In addition, Householder matrices possess the eigenvalue -1 with multiplicity 1 and the eigenvalue 1 with multiplicity $n-1$.

7.1.3 QR Decomposition with Householder Transformations

The QR decomposition of a given matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ can be computed by repeated application of Householder transformations which map column vectors of $\boldsymbol{A}$ to the first unit vector. With each iteration, the Householder transformation is applied to a reduced sub-matrix of $\boldsymbol{A}$ until a sequence of Householder matrices is determined which computes the QR decomposition. To illustrate this procedure, the following example from [Golub12] is considered with the orthogonal matrix $\boldsymbol{Q} \in \mathbb{R}^{m \times m}$ and the upper triangular matrix $\boldsymbol{R} \in \mathbb{R}^{m \times n}$ ($m \geq n$). Let $m = 6$ and $n = 5$ and suppose that the Householder matrices $\boldsymbol{H}_1$ and $\boldsymbol{H}_2$ have been computed such that

$$\boldsymbol{H}_2 \cdot \boldsymbol{H}_1 \cdot \boldsymbol{A} = \begin{bmatrix} \square & \square & \square & \square & \square \\ 0 & \square & \square & \square & \square \\ 0 & 0 & \blacksquare & \square & \square \\ 0 & 0 & \blacksquare & \square & \square \\ 0 & 0 & \blacksquare & \square & \square \\ 0 & 0 & \blacksquare & \square & \square \end{bmatrix} \tag{7.4}$$

Considering the highlighted matrix elements ■ in the above matrix, a Householder matrix $\tilde{\boldsymbol{H}}_3 \in \mathbb{R}^{4\times 4}$ is determined such that

$$\tilde{\boldsymbol{H}}_3 \cdot \begin{bmatrix} \blacksquare \\ \blacksquare \\ \blacksquare \\ \blacksquare \end{bmatrix} = \begin{bmatrix} \square \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{7.5}$$

If $\boldsymbol{H}_3 = diag(\boldsymbol{I}_2, \tilde{\boldsymbol{H}}_3)$, then

$$\boldsymbol{H}_3 \cdot \boldsymbol{H}_2 \cdot \boldsymbol{H}_1 \cdot \boldsymbol{A} = \begin{bmatrix} \square & \square & \square & \square & \square \\ 0 & \square & \square & \square & \square \\ 0 & 0 & \square & \square & \square \\ 0 & 0 & 0 & \square & \square \\ 0 & 0 & 0 & \square & \square \\ 0 & 0 & 0 & \square & \square \end{bmatrix}. \tag{7.6}$$

After n such steps an upper triangular matrix $\boldsymbol{H}_n\boldsymbol{H}_{n-1}...\boldsymbol{H}_3\boldsymbol{H}_2\boldsymbol{H}_1\boldsymbol{A} = \boldsymbol{R}$ is computed and by setting $\boldsymbol{H}_n\boldsymbol{H}_{n-1}...\boldsymbol{H}_3\boldsymbol{H}_2\boldsymbol{H}_1 = \boldsymbol{Q} \Longrightarrow \boldsymbol{A} = \boldsymbol{QR}$.

Algorithm 3 in Appendix E presents the pseudocode of the implemented QR decomposition with the major steps in detail.

7.1.4 Experimental Evaluation

To evaluate the effectiveness of the A-ABFT method and the probabilistic determination of rounding error bounds, the matrix operations within the QR decomposition have been replaced by GPU-accelerated and A-ABFT-protected variants. The application of these A-ABFT-protected matrix operations on the GPU lead to *speedups of up to* $20x$ compared to the unprotected CPU version, which utilized highly optimized BLAS operations[2].

Faults have been injected into every matrix multiplication that has been executed during the computation of the QR decomposition. A random bit-flip has been injected into a randomly selected matrix element. After the computation of the matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ the matrix $\boldsymbol{A}_{new} = \boldsymbol{Q} \cdot \boldsymbol{R}$ has been computed and compared against a reference matrix $\boldsymbol{A}_{ref}$ that has been computed without fault injection. For the simplified error

[2] Intel Math Kernel Library (MKL) BLAS routines.

analysis SEA and the proposed probabilistic approach PEA, the difference matrices $\boldsymbol{D_{SEA}} = \boldsymbol{A_{ref}} - \boldsymbol{A_{SEA,new}}$ and $\boldsymbol{D_{PEA2}} = \boldsymbol{A_{ref}} - \boldsymbol{A_{PEA2,new}}$ have then been computed and compared by their Frobenius and Maximum norm.

$$\|D_x\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{i,j}|^2} \quad \text{and} \quad \|D_x\|_\infty = \max_{i=1,\ldots,m}\sum_{j=1}^{n}|a_{i,j}|. \tag{7.7}$$

The experiments have been performed over the test data set T_{ortho}. The results of the experimental evaluation have been generated in collaboration with Halder and have also been reported in [Halde14].

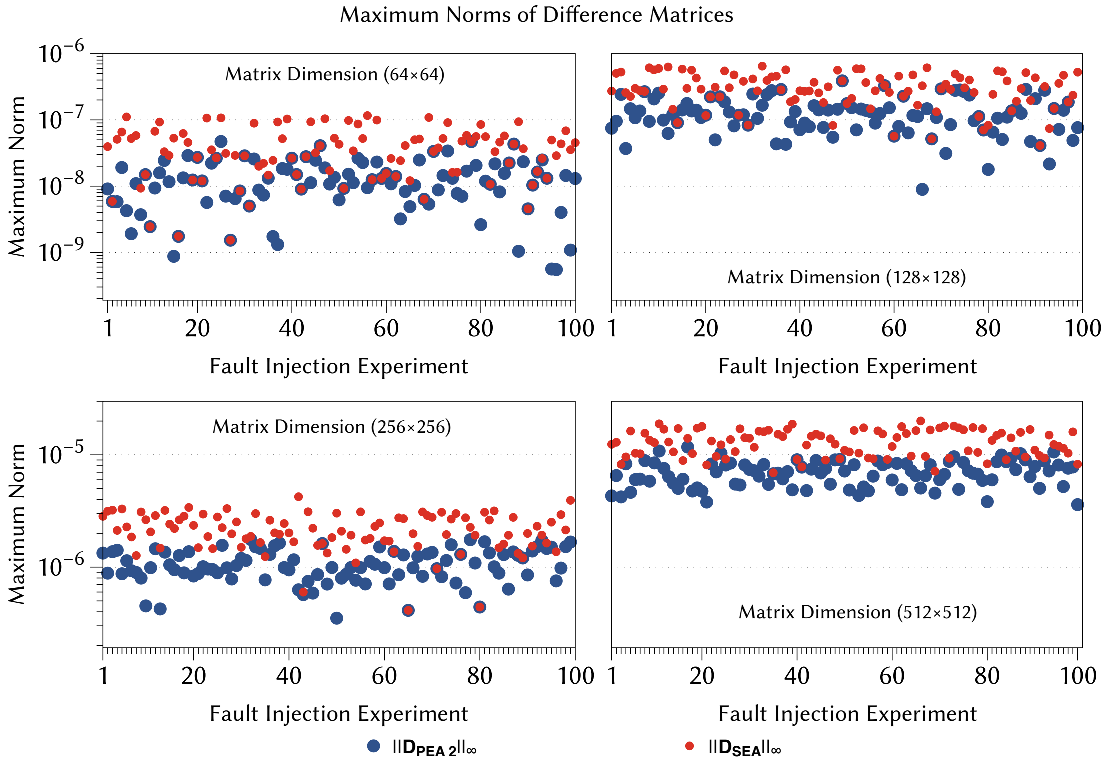
The results of the fault injection experiments show that the application of A-ABFT leads to a significant reduction of errors in the final results. The analysis of the Frobenius and the Maximum norms for the difference matrices revealed errors that are smaller by factors from 10^{10} to 10^{12}, in comparison to the unprotected matrix multiplication. On average, the simplified error analysis SEA showed errors that were almost twice as large as those reported for the proposed probabilistic approach PEA. Table 7.1 summarizes the Frobenius and Maximum norms of the difference matrices for the unprotected and the two A-ABFT-protected matrix multiplications.

▼ **Table 7.1** — Average Frobenius and Maximum norms of the difference matrices $\boldsymbol{D_{NoABFT}}$, $\boldsymbol{D_{SEA}}$ and $\boldsymbol{D_{PEA}}$.

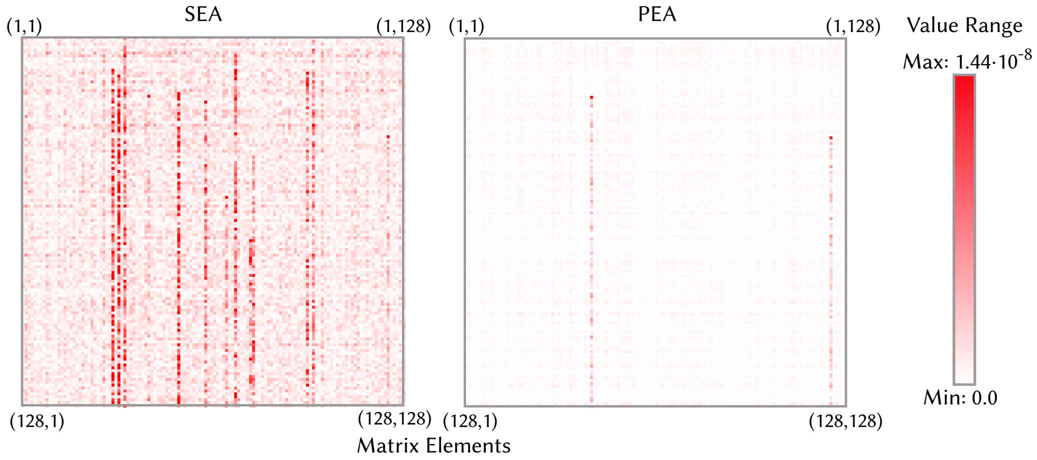
Matrix $[n \times n]$	$\|\boldsymbol{D_{NoABFT}}\|_F$	$\|\boldsymbol{D_{SEA}}\|_F$	$\|\boldsymbol{D_{PEA}}\|_F$	$\|\boldsymbol{D_{NoABFT}}\|_\infty$	$\|\boldsymbol{D_{SEA}}\|_\infty$	$\|\boldsymbol{D_{PEA}}\|_\infty$
64	$1.99 \cdot 10^{3}$	$1.14 \cdot 10^{-8}$	$3.74 \cdot 10^{-9}$	$8.48 \cdot 10^{3}$	$4.26 \cdot 10^{-8}$	$1.41 \cdot 10^{-8}$
128	$3.94 \cdot 10^{3}$	$1.39 \cdot 10^{-7}$	$3.15 \cdot 10^{-8}$	$2.06 \cdot 10^{4}$	$6.39 \cdot 10^{-7}$	$1.36 \cdot 10^{-7}$
256	$8.39 \cdot 10^{3}$	$4.17 \cdot 10^{-7}$	$2.01 \cdot 10^{-7}$	$6.06 \cdot 10^{4}$	$2.21 \cdot 10^{-6}$	$1.08 \cdot 10^{-6}$
512	$1.77 \cdot 10^{4}$	$2.27 \cdot 10^{-6}$	$1.16 \cdot 10^{-6}$	$1.59 \cdot 10^{5}$	$1.32 \cdot 10^{-5}$	$7.02 \cdot 10^{-6}$
1024	$3.61 \cdot 10^{4}$	$2.51 \cdot 10^{-5}$	$9.43 \cdot 10^{-6}$	$3.91 \cdot 10^{5}$	$1.68 \cdot 10^{-4}$	$6.93 \cdot 10^{-5}$

Figure 7.1 presents the Maximum norms of the difference matrices for 100 fault injection experiments. The results show that the PEA2 variant of A-ABFT delivers consistently smaller errors across almost all experiments and problem sizes. With increasing problem sizes, a factor of $2x$ becomes apparent between SEA and PEA.

Figure 7.2 depicts the measured error in each matrix element for a QR decomposition with problem size 128 × 128. Each plotted pixel corresponds to a single matrix element



▲ **Figure 7.1** — Maximum norms of the difference matrices $\boldsymbol{D}_{PEA2}$ and $\boldsymbol{D}_{SEA}$ for 100 fault injection experiments per matrix dimension.



▲ **Figure 7.2** — Comparison of impact of injected faults during QR decomposition for simplified error analysis SEA and probabilistic rounding error determination PEA.

and the shading indicates the error of this element. The darker the shading, the larger the error that affects the matrix element. The plot shows the higher effectiveness of the probabilistic method for the determination of rounding error bounds and it also shows the propagation of errors, which is significantly higher for the simplified error analysis SEA.

This first case study on the application of A-Abft shows the benefits of the introduced method with respect to the required effort for its integration into existing applications, the achievable acceleration on GPUs, as well as the fault tolerance improvement. Algorithm 3 in Appendix E shows that no adaptation or changes are required for the integration, A-Abft-protected matrix operations can be directly used as drop-in replacement for standard BLAS library calls. The user also does not require any knowledge about the fault tolerance scheme—it operates autonomously and transparently. The results further show that the fault tolerant acceleration of the matrix multiplication on the GPU yields speedups of up to 20x compared to a highly optimized vendor BLAS-library, which utilizes all CPU cores. Finally, the experimental results show that the rounding error bounds determined by A-Abft are much closer to the actual rounding error than those determined with the state of the art SEA approach, which enables computational results with no or only significantly smaller errors.

7.2 A GPU-Accelerated Linear Programming Solver

This chapter presents a second case study which demonstrates the application of the A-Abft-protected GPU matrix multiplication for the acceleration of a *Linear Programming* solver. The A-Abft method proposed in this thesis is compared against the SEA ABFT approach (see Chapter 5) on the basis of fault injection experiments. The performance is evaluated for an unprotected version of the solver and the two ABFT-protected versions.

7.2.1 Linear Programming

Linear Programming is a key technique for the optimization of linear objective functions that are subject to linear (in-)equality constraints. The method has originally been developed by Leonid Kantorovich in 1939 and gained increasing attention with the introduction of the *Simplex* algorithm by George Bernard Dantzig in 1947. Since then, Linear Programming emerged to an indispensable tool in *Operations Research* where it is used for the optimization of business processes, production planning and scheduling tasks, as well as the allocation of resources.

The optimization problem, called the linear program, is represented by the *linear objective function* $\boldsymbol{c} : \mathbb{R}^n \rightarrow \mathbb{R}$, a set of n *variables* $x_1, ..., x_n$ with $x_i \in \mathbb{R}$ and $x_i \geq 0$, as well as m *linear constraints* $\boldsymbol{a} : \mathbb{R}^n \rightarrow \mathbb{R}$ defined by the inequalities[3]

$$a_{1,1} \cdot x_1 + a_{1,2} \cdot x_2 + \cdots + a_{1,n} \cdot x_n \leq b_1 \tag{7.8}$$

$$\vdots \tag{7.9}$$

$$a_{m,1} \cdot x_1 + a_{m,2} \cdot x_2 + \cdots + a_{m,n} \cdot x_n \leq b_m \tag{7.10}$$

with $b_i \in \mathbb{R}$. Formulated as matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and the two vectors $\boldsymbol{b} \in \mathbb{R}^m$ and $\boldsymbol{c} \in \mathbb{R}^n$, the goal is to determine a vector $\boldsymbol{x} \in \mathbb{R}^n$ whose elements are non-negative and represent a feasible solution for

$$\boldsymbol{A} \cdot \boldsymbol{x} \leq \boldsymbol{b} \tag{7.11}$$

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \leq \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \tag{7.12}$$

such that the inner product

$$\boldsymbol{c}^T \boldsymbol{x} = c_1 \cdot x_1 + c_2 \cdot x_2 + \cdots + c_n \cdot x_n \tag{7.13}$$

becomes maximal:

$$\max\{\boldsymbol{c}^T \boldsymbol{x} \mid \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \quad x_i \geq 0\}. \tag{7.14}$$

7.2.2 Simplex and Revised Simplex Algorithm

The Simplex algorithm is a method for the solution of linear programs which has been introduced by George Bernard Dantzig [Dantz63] in 1947. The Simplex algorithm is able to either determine an exact solution for the linear program in finite time, or to prove that the linear program has no solution. It has been shown that the Simplex algorithm and all its derivates exhibit an exponential *worst-case runtime* [Klee70]. However, in most practical applications, Simplex algorithms show a much better runtime behavior and often outperform alternative methods like interior points algorithms [Karma84,Neste94] or the ellipsoid method [Khach80].

[3] Constraints can also be expressed as equations "=" and inequalities with "≥". Transformations into the "≤" representation exist.

The Simplex algorithm operates in two phases. In the first phase, an initial feasible basic solution is searched for the linear program. This is done by consideration of an auxiliary linear program based on the original program with additionally introduced *artificial variables*. The objective of the auxiliary linear program is to minimize the sum of the artificial variables. If such an optimal solution can be determined and if the auxiliary linear program also possesses a solution where all artificial variables are equal to zero, this latter solution can be taken as initial feasible solution for the original program. If no such solution exits, the Simplex algorithm aborts.

In phase two, the Simplex algorithm starts with the initial feasible basic solution computed in phase one, and iteratively tries to improve the value of the objective function. Within each step, the current basis is changed and a new basic solution is computed. The change of the basis is achieved by pivot operations, where a current basic variable is selected and replaced with a non-basic variable. The basic variable is called the *leaving variable*, whereas the non-basic variable is called the *entering variable*. Different rules and criteria for the selection procedure have been proposed over time. One possibility will be to choose the non-basic variable which yields the maximum marginal improvement of the objective function's value. The column vector corresponding to the selected entering variable is called the *pivot column*. In the next step, the leaving variable has to be determined. Therefore, the quotient of the pivot column's elements and the corresponding entries on the right-hand side of the tableau are computed. The minimum resulting quotient designates the leaving variable and the so called *pivot row*. Further details on the selection of entering and leaving variables, the remaining computational steps for the change of the basis, as well as an extensive introduction to Linear Programming and available solution methods can be found in [Luenb08]. In conclusion, the Simplex algorithm continues to change the basis until no further improvement can be achieved or the linear program is proved to be unbounded.

The LP solver developed for this case study uses a so called *Revised Simplex algorithm* [Dantz53], which provides a higher computational and higher memory efficiency. A common characteristic of revised simplex algorithms is that they do not store the full Simplex tableau. Instead, only a representation of the basis $\boldsymbol{B}$ of the matrix representing the constraints is stored. These matrix-based methods, which are mathematically equivalent to the original Simplex algorithm, map the steps of the original algorithm to sequences of linear algebra operations. This makes them particularly attractive for acceleration on graphics processing units and hence for the integration of ABFT-protected GPU matrix operations.

As described above, Revised Simplex algorithms do not store the complete Simplex tableau (dictionary). The core of these methods is formed by the basis matrix $\boldsymbol{B}$. The entering variables are determined by computation and checking the potential contribution of each non-basic variable x_j by

$$\boldsymbol{z}_j - \boldsymbol{c}_j = \boldsymbol{c}_B \boldsymbol{B}^{-1} \boldsymbol{A}_j - \boldsymbol{c}_j, \tag{7.15}$$

where $\boldsymbol{A}_j$ denotes the column in the matrix $\boldsymbol{A}$ corresponding to the considered variable x_j. The non-basic variable with the smallest negative difference is then selected:

$$\text{Index } p = \{j \mid \tilde{\boldsymbol{c}}_j = \min_t \{\tilde{\boldsymbol{c}}_t\}, \tilde{\boldsymbol{c}}_j < 0\}. \tag{7.16}$$

If no entering variable can be determined in this procedure, the current solution is the optimal solution. For the determination of the leaving variable the basic solution $\boldsymbol{x}_B$ has to be computed. With all non-basic variables set to zero the equations can be expressed as

$$\boldsymbol{B}\boldsymbol{x}_B = \boldsymbol{b}, \tag{7.17}$$

which allows the computation of $\boldsymbol{x}_B$ through

$$\boldsymbol{x}_B = \boldsymbol{B}^{-1}\boldsymbol{b}. \tag{7.18}$$

This computation is performed for each variable using

$$\alpha = \boldsymbol{B}^{-1}\boldsymbol{A}_p \tag{7.19}$$

The leaving variable is the one with the smalles quotient Θ_j

$$\Theta_j = \frac{x_{B_j}}{\alpha_j} \tag{7.20}$$

$$\text{Index } q = \{j \mid \Theta_j = \min_t \{\Theta_t\}, \alpha_j > 0\}. \tag{7.21}$$

If $\alpha_i \leq 0$ the linear program is unbounded and the algorithm terminates. In the last step of each iteration, the basis $\boldsymbol{B}$ has to be updated. A detailed overview of the algorithmic steps performed by the LP-solver developed for this case study is given in Algorithm 4, which can be found in Appendix E.

7.2.3 Implementation

In [Spamp09b], a Linear Programming solver has been presented which utilizes the parallel compute power of GPUs for the acceleration of matrix-vector and matrix-matrix operations. Based on this work and further details on the implementation presented in [Spamp09a], a Revised Simplex LP solver has been implemented for this case study. Several modifications have been applied to the originally proposed code to adapt the solver to contemporary GPUs and to reduce the number of aborts when real-world LP problems are treated[4]. To ease the experimental evaluation, a parser for the *Mathematical Programming System (MPS)* file format has been integrated into the solver. MPS is a widespread ASCII-based file format to represent and archive linear programming and mixed integer programming problems. Detailed information on the structure of this file format can be found in [LPSOL14]. In addition, the LP solver has been instrumented to allow fault injections during the Simplex iterations for the experimental evaluation. Algorithm 4 in Appendix E shows the pseudocode of the algorithm.

The implemented LP solver performs the two basic execution phases according to the original Simplex algorithm. In phase one the solver tries to determine a feasible basic solution for the auxiliary LP problem generated through the introduction of the artificial variables. If such a solution is found, the solver enters phase two of the Simplex algorithm and iterates until it finds an optimal solution or the proof that the linear program is unbounded.

7.2.4 Experimental Evaluation

The experimental evaluation of the developed Revised Simplex LP solver has been performed on the basis of benchmark problems taken from *The Netlib Linear Programming Test Problems suite* [The N14c]. All problems are given in MPS format and have been classified in the NetLib suite according to the CUTE classification system [Bonga95]. The Linear Programming problems themselves have been selected rather randomly under the aspects of covering a wider range of problem sizes, and keeping the simulation times for the fault injection experiments within a reasonable timeframe. Table 7.2 summarizes the considered Linear Programming problems and lists the number of constraints and variables, as well as the optimal value of each problem.

[4] N.B.: The original solver in [Spamp09b] has only been evaluated for randomly generated LP problems and was not always able to solve the LP problems considered in this case study (see Section 7.2.4)

▼ **Table 7.2** — Evaluated Linear Programming Problems from NetLib.

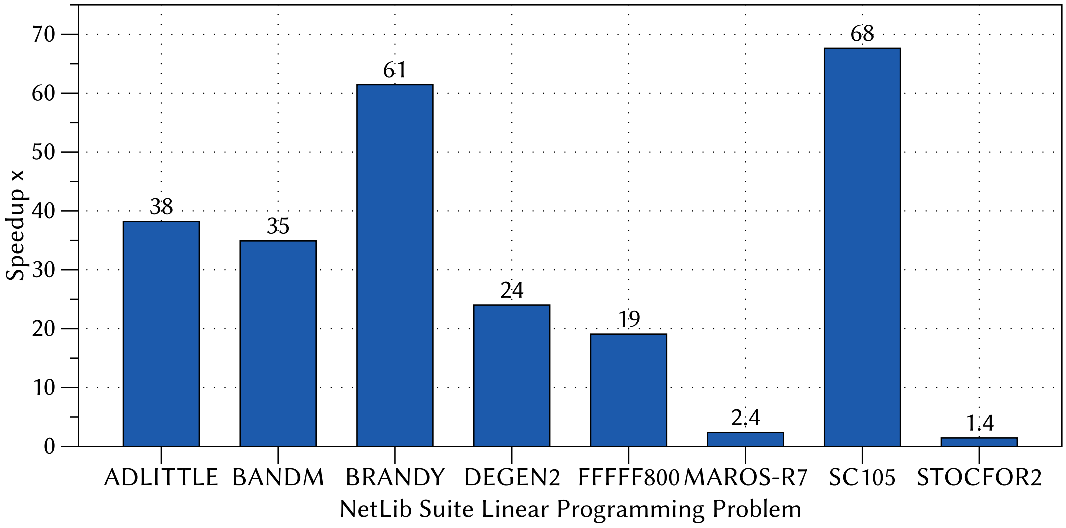
LP Problem	Constraints	Variables	Non-Zeros	Optimum
ADLITTLE	57	97	465	$2.2549496316 \cdot 10^5$
BANDM	306	472	2659	$-1.5862801845 \cdot 10^2$
BRANDY	221	249	2150	$1.5185098965 \cdot 10^3$
DEGEN2	445	534	4449	$-1.4351780000 \cdot 10^3$
FFFFF800	525	854	6235	$5.5567961165 \cdot 10^5$
MAROS-R7	3137	9408	$1.51 \cdot 10^5$	$1.4971851665 \cdot 10^6$
SC105	106	103	281	$-5.2202061212 \cdot 10^1$
STOCFOR2	2158	2031	9492	$-3.9024408538 \cdot 10^4$

All experiments have been conducted on compute server *CPU I* and the Nvdiai Tesla K40c GPU accelerator. Within the LP solver, all matrix-vector and matrix-matrix operations have been replaced by GPU-accelerated kernels. The experiments have been performed with two different versions of the solver, one without A-ABFT-protection and one with A-ABFT-protected matrix operations. The version with A-ABFT-protected matrix operations provided support for the proposed probabilistic method for the determination of rounding error bounds PEA, as well as support for the simplified error analysis SEA. For both methods, different combinations of checksum encodings have been evaluated.

The two phase Simplex algorithm has been executed and single-bit flips have been injected into the significands of randomly chosen matrix elements during the basis update steps ($\boldsymbol{B}^{-1} = \boldsymbol{E}\boldsymbol{B}^{-1}$, see Algorithm 4), which is the only matrix-matrix multiplication in the algorithm. Figure 7.3 reports the achieved speedups of the A-ABFT-protected LP solver in comparison to the unprotected CPU variant, which utilized highly optimized BLAS library routines[5].

Significant speedups of up to $68x$ could be achieved by the fault tolerant A-ABFT-protected LP solver on the GPU in most cases. The problems *MAROS-R7* and *STOCFOR2* lead only to smaller speedups up to $2.4x$. It has to be noted at this point that the implementation of the LP solver used in this case study is by no means at the level of highly optimized commercial LP solvers. It was only intended for demonstration purposes, far reaching algorithmic optimizations were beyond the scope of this case study.

[5] Intel Math Kernel Library (MKL) BLAS routines.

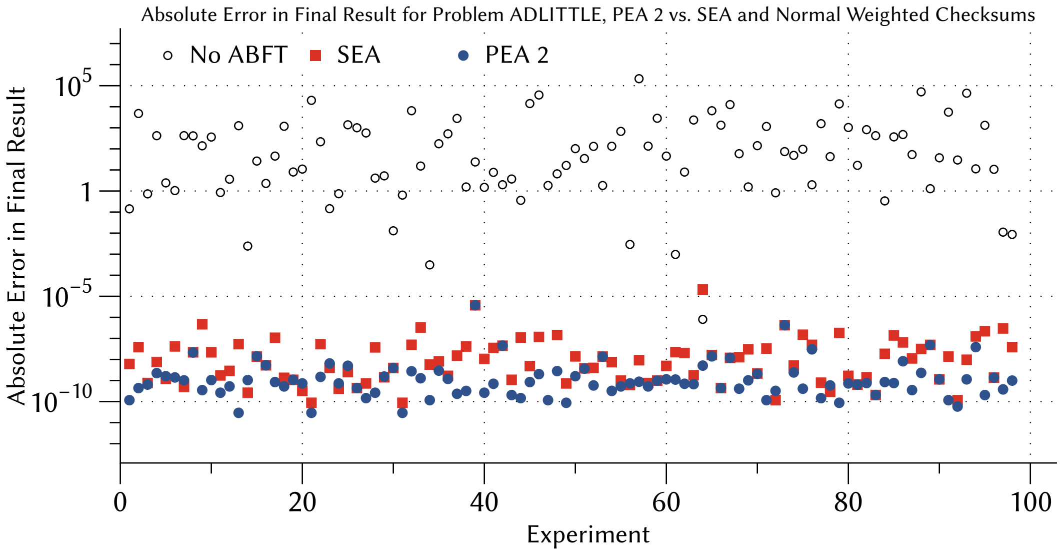


▲ **Figure 7.3** — Comparison of LP solver performance of an unprotected multi-core CPU implementation versus an A-Abft-protected GPU implementation.

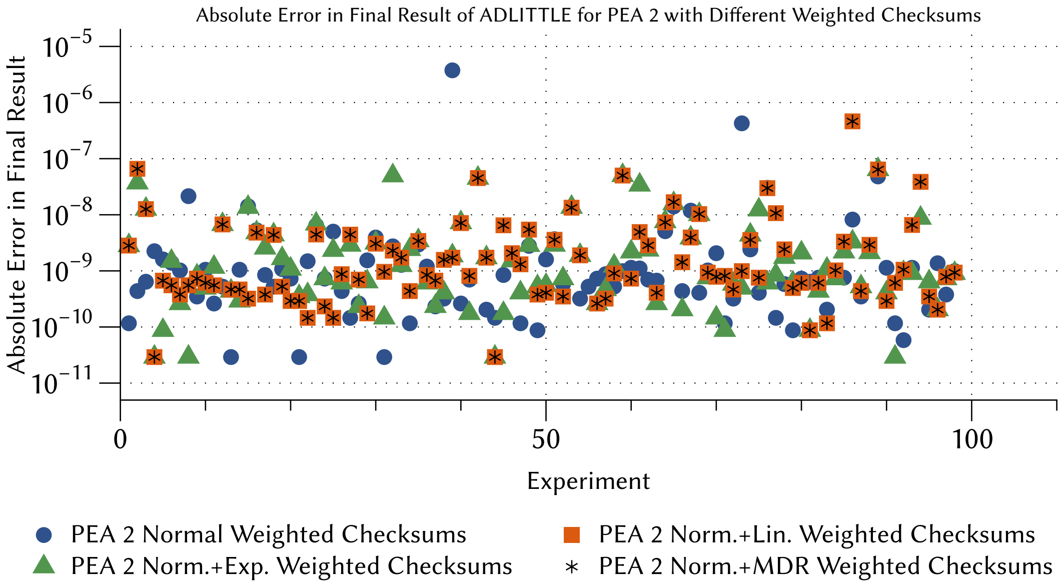
To demonstrate the effectiveness of the A-abft-protection, the fault injection experiments have been evaluated with respect to the absolute error in the final results of the LP problems. In some cases the injected faults lead to failing executions for the unprotected version of the LP solver where no optimum has been found. *All* these cases have been detected and corrected by A-abft, which yielded the correct solution. Figure 7.4 presents the absolute errors in the final results for the problem ADLITTLE. The graph shows that A-abft with PEA 2, as well as SEA, enables a significant reduction of the absolute error in the final result. Most remaining errors reside within a magnitude from 10^{-11} to 10^{-8}. The application of the probabilistic rounding error determination PEA lead to slightly smaller errors in most cases.

Figure 7.5 and Figure 7.6 present the absolute errors in the final results for PEA 2 and SEA with different combinations of weighted checksums. For this case study, the normal weighted checksum encoding has been evaluated together with the combined weighted checksums normal + exponential, normal + linear, and normal + MDR. For PEA 2 and SEA, the differences between the different checksums were rather small.

Figures 7.7, 7.8 and 7.9 exemplary report the absolute errors in the final results for the problems BANDM, BRANDY and STOCFOR2. In all cases, the absolute error in the final results has been significantly reduced by A-abft. For BANDM, the absolute error in the results ranges from 10^{-10} to 10^{-8}, for BRANDY from 10^{-12} to 10^{-9} (PEA 2) and



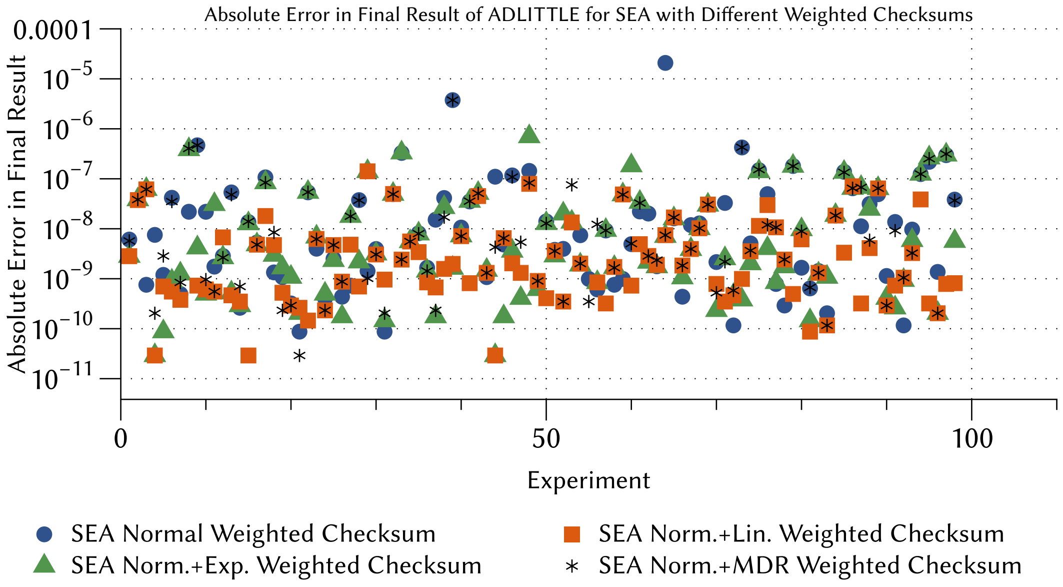
▲ **Figure 7.4** — Absolute error in final result for problem ADLITTLE, comparison between unprotected and A-ABFT protected variants with normal weighted checksums.



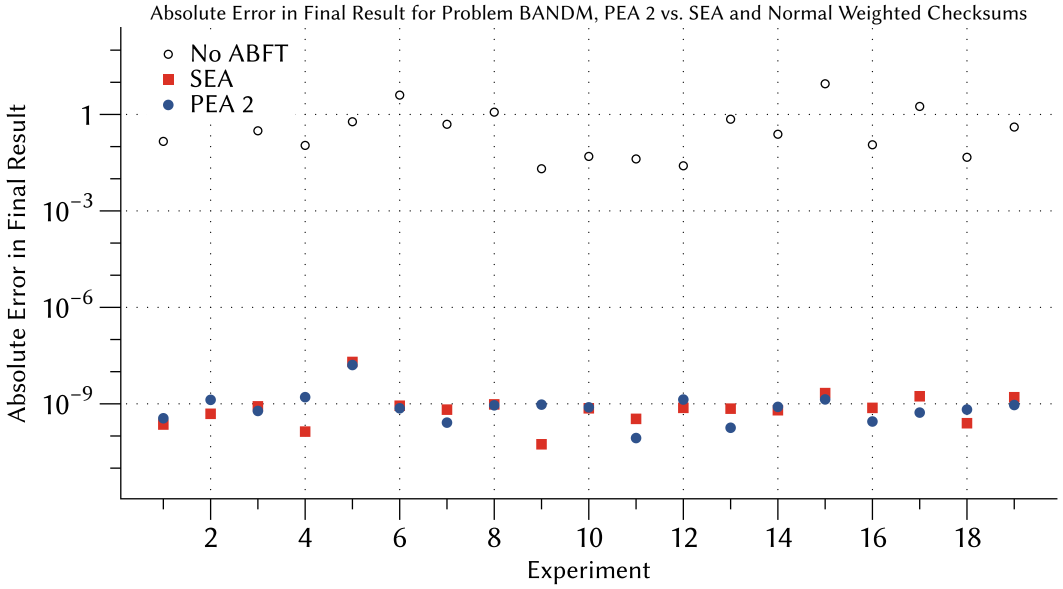
▲ **Figure 7.5** — Absolute error in final result for problem ADLITTLE, comparison between different weighted checksums for PEA 2.

for STOCFOR2 from 10^{-11} to 10^{-8}.

This second case study also emphasizes the benefits of A-ABFT-protected matrix operations on GPU for applications of practical relevance. Although the linear programming

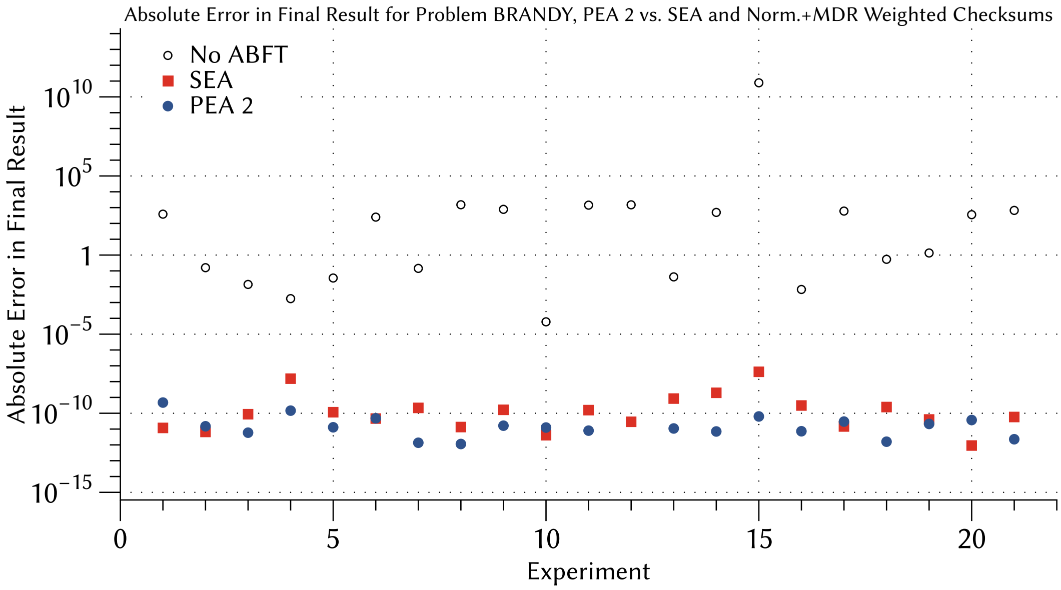


▲ **Figure 7.6** — Absolute error in final result for problem ADLITTLE, comparison between different weighted checksums for SEA.

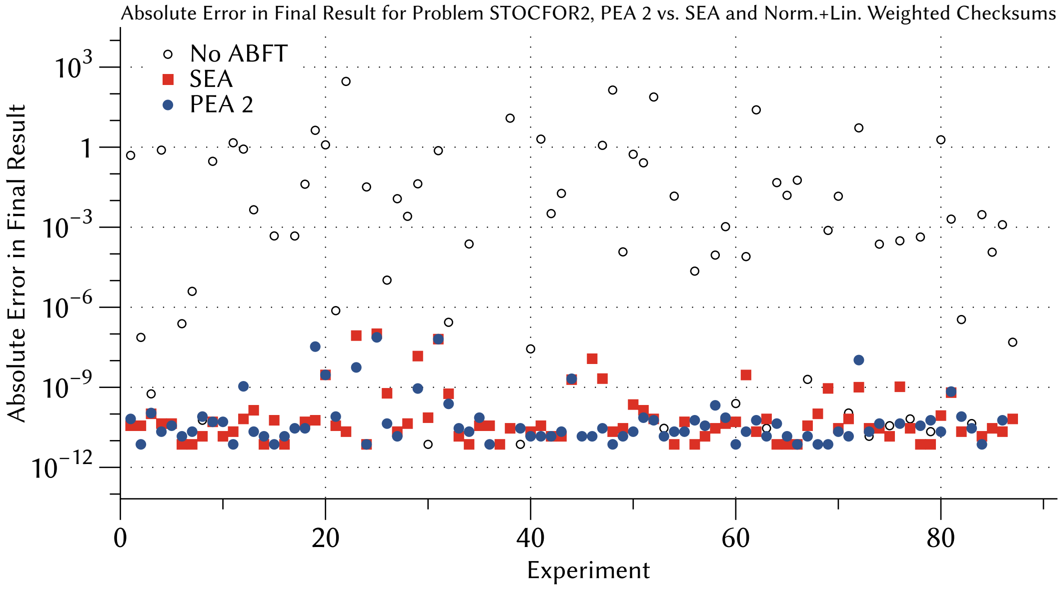


▲ **Figure 7.7** — Absolute error in final result for problem BANDM, comparison between unprotected and A-ABFT protected variants with normal weighted checksums.

solver utilized the A-ABFT matrix multiplication for only one central computation, significant speedups of up to 68x could be achieved in comparison to the non-protected



▲ **Figure 7.8** — Absolute error in final result for problem BANDM, comparison between unprotected and A-ABFT protected variants with normal + MDR weighted checksums.



▲ **Figure 7.9** — Absolute error in final result for problem BANDM, comparison between unprotected and A-ABFT protected variants with normal + linear weighted checksums.

variant of the solver, which used highly optimized BLAS routines on the CPU. Even more important are the results of the fault injection experiments, which show that almost all of the injected faults caused severe errors in the final solution or led to a non-terminating execution of the solver. As in the first case study, the introduced A-Abft method determined rounding error bounds that were almost always closer to the actual rounding error than those determined by the state of the art SEA technique.

CHAPTER 8

EXPERIMENTAL EVALUATION

This chapter presents and discusses the main results of the detailed experimental evaluation of the proposed A-ABFT method for matrix operations on GPUs [Braun14]. Using the example of the matrix multiplication, the novel probabilistic approach for the determination of rounding error bounds (PEA) is compared against the state of the art *simplified error analysis* (SEA) [Roy C93]. In addition, an in-depth analysis of central ABFT design parameters such as different kinds of weighted checksum codes and their combinations, as well as different block sizes for partitioned encodings, is performed. To demonstrate the advantages of A-ABFT over the state of the art, the experimental evaluations consider three main aspects including the *achievable computational throughput and performance*, the *quality of the determined rounding error bounds* and the *error detection capabilities*. In favor of a concise presentation of the experimental results, the most important findings are summarized in this chapter. Section 8.1 presents the evaluation results for the achievable computational performance. Section 8.2 gives the reader details on the quality of the determined rounding error bounds and Section 8.3 summarizes the results of fault injection experiments for the evaluation of the error detection capabilities. Appendix F provides detailed supplemental results on the performed experiments, which have been omitted in this chapter.

The experiments for the evaluation part of this work have been performed on two Intel Xeon-based compute servers, hereinafter referred to as *CPU I* and *CPU II*, which were equipped with Nvidia Tesla GPU accelerators. The detailed technical hardware and software specifications of the two systems are described in Appendix F, Section F.1.

Experimental Parameters

The following experiments have been performed for different weighted checksum codes and combinations of these. Each choice of weighted checksum codes has been evaluated for different ABFT encoding block sizes, matrix dimensions and input value ranges. The used weighted checksum codes with the corresponding formulas for the computation of the checksum elements are listed in Table 8.1

▼ **Table 8.1** — Types of evaluated weighted checksum codes.

Weighted Checksum Code	Formula
Normal weighted checksums	$c = \sum_{i=1}^{bs} 1 \cdot a_i$
Exponential weighted checksums	$c = \sum_{i=1}^{bs} 2^{i+1} \cdot a_i$
Linear weighted checksums	$c = \sum_{i=1}^{bs} (i+1) \cdot a_i$
Periodic weighted checksums	$c = \sum_{i=1}^{bs} (-1)^i \cdot a_i$
Minimum Dynamic Range (MDR) weighted checksums	$c = \sum_{i=1}^{bs} \frac{(i+1)}{bs} \cdot a_i$

The ABFT encoding block sizes include 32 × 32, 64 × 64, 128 × 128 and 256 × 256. For experiments with combinations of two weighted checksum codes, ABFT encoding block sizes of up to 128 × 128 are considered. Since ABFT requires the distribution of computational steps among the available compute resources, the current implementation of the A-Abft method performs redundant computations of checksums. For the standard ABFT setup with a single weighted checksum code $2 \cdot 2 \cdot 256 = 1024$ threads are utilized for the encoding of an ABFT block. This corresponds to the maximum number of active threads per thread block allowed by the current generation of Nvidia GK110 GPUs. If more than one weighted checksum type is used, a maximum ABFT encoding block size of 128 × 128 can be used.

For the proposed A-Abft approach with probabilistic determination of rounding error bounds (PEA x), the added number x refers to the number of elements with largest absolute values that have to be considered per vector. PEA 2 identifies two such elements per vector, PEA 4 four, and so on.

To cover a rather wide range of input data set sizes, the experiments are performed for matrices of the dimensions 512×512, 1024×1024, 2048×2048, 3072×3072, 4096×4096, 5120×5120, 6144×6144, 7168×7168 and 8192×8192.

The test data sets used for the experiments comprise three different types, including test matrices of type T_{pos} with uniformly distributed numbers in the range of $[0, 10^i]$ with $i \in \{0, 1, 2, 3\}$, type T_{full} with uniformly distributed positive and negative numbers in the range $[-10^i, 10^i]$ with $i \in \{0, 1, 2, 3\}$ and test matrices of type T_{ortho}. The latter type comprises randomized orthogonal matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ with a higher dynamic range in the input values, which are generated according to Turmonet al. [Turmo00a] by the formula

$$\boldsymbol{A} = 10^{\alpha} \cdot \boldsymbol{U} \cdot \boldsymbol{D_\kappa} \cdot \boldsymbol{V}^T. \tag{8.1}$$

"The random matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ are the orthogonal factors in the QR factorization of two square matrices with normally distributed entries. The diagonal matrix $\boldsymbol{D}_\kappa$ is filled in by choosing random singular values, such that the largest singular value is unity and the smallest is $\frac{1}{\kappa}$. These matrices all have 2-norm equal to unity; the overall scale is set by α which is chosen uniformly at random between -8 and $+8$" [Turmo00a]. Experiments based on the test data set T_{ortho} have been performed with $\kappa \in \{2, 16, 128, 1024, 8192, 65536\}$.

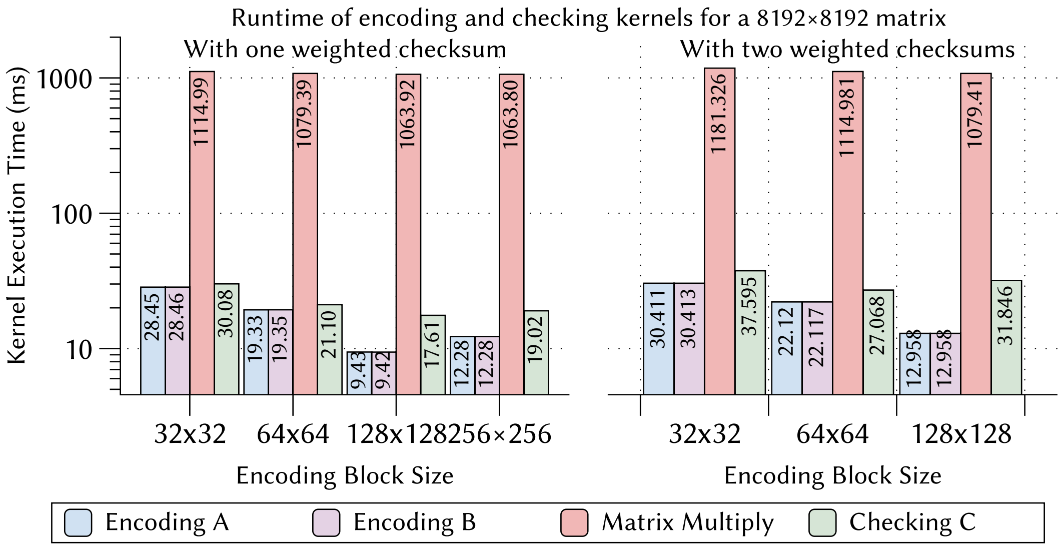
8.1 Computational Performance

For the evaluation of the computational performance the execution times of the GPU kernels for the different ABFT phases have been measured. Section 8.1.1 presents the execution times for the ABFT checksum encoding of the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, as well as the ABFT result check of matrix $\boldsymbol{C}$. These operations are independent of the chosen method for the determination of the required rounding error bounds (SEA and PEA). Section 8.1.2 analyzes the execution times of the GPU kernels required for the preprocessing steps of SEA and PEA, followed by the analysis of the kernel execution times for the final determination of the rounding error bounds (ε-determination kernels) in Section 8.1.3. The overall computational performance and throughput of the A-Abft-protected matrix multiplications is evaluated in Section 8.1.4.

8.1.1 Runtime of ABFT Encoding and Checking Kernels

The kernel execution times for the encoding of the input matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, the actual matrix multiplication, as well as the result check of matrix $\boldsymbol{C}$, have been measured on an Nvidia Tesla K20c GPU. The kernel execution times for these ABFT phases are independent of the chosen method for the determination of rounding error bounds. As will be shown in Section 8.1.4 and Appendix F, the choice of a weighted checksum code does not have any significant impact on these kernel execution times and the overall performance. The following results are therefore averages over multiple experiments that have been performed with different weighted checksum codes.

Figure 8.1 depicts the measured kernel execution times in milliseconds for the ABFT encoding and checking phases in comparison to the actual matrix multiplication for matrices of dimension 8192 × 8192 and ABFT encodings with one and two weighted checksums.



▲ **Figure 8.1** — Kernel execution times for encoding and checking over different encoding block sizes for matrix dimension 8192 × 8192 with one and two weighted checksums.

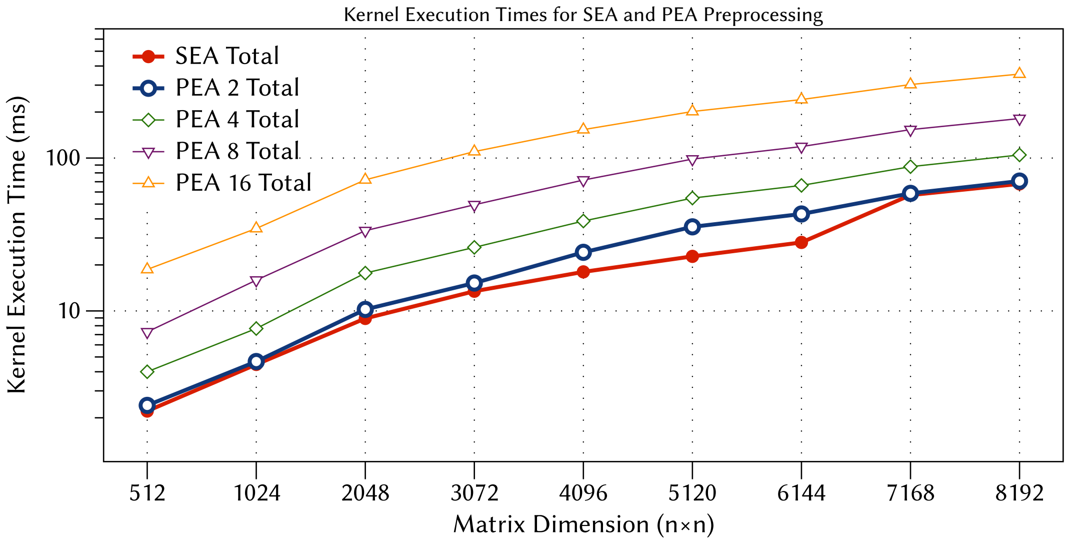
As expected, the figure shows that the matrix multiplication is the computationally most expensive part of this ABFT operation. The variants with one and two weighted checksums applied exhibit a rather similar runtime behavior. For the ABFT encoding block sizes of 32 × 32, 64 × 64 and 128 × 128, the variants with two weighted checksums take about 6.9%, 14.4% and 37.4% more execution time, respectively. For the variants with two weighted checksums, the increase in execution time for the result checking kernel is larger, which can be explained by the fact that this kernel does not increase the number of utilized threads in the way the encoding kernels do. However, the overall kernel execution overhead for the encoding and the checking is negligible compared to the matrix multiplication.

In Appendix F, the Tables F.1, F.2 and F.3 present the measurements of the kernel execution times for matrices of dimensions 512×512, 4096×4096 and 8192×8192 with one weighted checksum. Tables F.4, F.5 and F.6 present the corresponding results for two weighted checksums. The tables show that the overhead for the ABFT encoding and checking becomes smaller with increasing matrix dimensions since the matrix multiplication dominates the overall runtime. For all three matrix dimensions presented, the ABFT encoding block size of 128×128 shows the lowest encoding and checking execution times. The tables also show that smaller problem sizes like 512×512 favor smaller encoding block sizes, whereas larger problem sizes like 4096×4096 and 8192×8192 benefit from larger encoding block sizes.

8.1.2 Runtime of SEA and PEA Preprocessing Kernels

The simplified error analysis (SEA) and the proposed method for the probabilistic determination of rounding error bounds (PEA) consist of a preprocessing phase and a phase for the determination of the rounding error bounds (ε-determination). This section presents the measured kernel execution times for the SEA preprocessing, which mainly includes the computation of vector norms, and the PEA preprocessing, which comprises the identification and sorting of a given number of elements with largest absolute values for each considered vector. The reported kernel execution times in Figure 8.2 are averages over multiple experiments with different ABFT encoding block sizes and show the total kernel execution times (rows and columns) for SEA and the different variants of PEA.

The simplified error analysis (SEA) shows short kernel execution times, which can be expected since the computation of vector norms does not involve any special operations or a complex control flow. In contrast to SEA, the probabilistic determination of rounding error bounds (PEA) includes numerous comparisons and sorting steps, which increase the execution time. Nonetheless, the PEA 2 configuration, which considers the two elements with the largest absolute values per vector, performs comparably well



▲ **Figure 8.2** — Kernel execution times for SEA and PEA preprocessing.

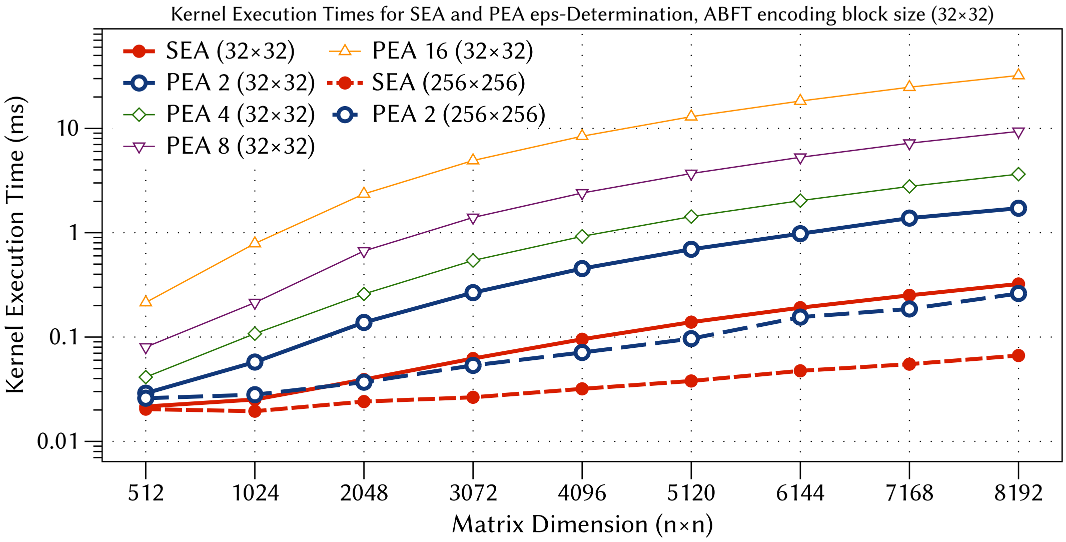
and shows almost identical execution times for mid-sized and large matrix dimensions. The overhead for the identification of elements with largest absolute values increases with the number of such elements. PEA 4 causes up to 1.7x, PEA 8 up to 3.4x and PEA 16 up to 7.7x more kernel execution time, compared to PEA 2.

In Appendix F, Table F.7 presents the kernel execution times for the SEA and PEA preprocessing with separate times reported for the preprocessing of rows and columns. The imbalance between the execution times for rows and columns is mainly related to different memory access patterns, which cause the execution times for the preprocessing of rows to be larger. The difference is more pronounced in the calculation of norms (SEA) than in the identification of elements with largest absolute values (PEA). For PEA, the imbalance decreases with an increasing number of values that has to be considered for the largest absolute values. The internal sorting and comparison of candidate values almost mitigates the differences resulting from memory accesses. For SEA and PEA 2, the single kernel execution times for the preprocessing are in the range of the encoding times for the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$.

8.1.3 Runtime of SEA and PEA ε-Determination Kernels

This section presents the analysis of the kernel execution times for the ε-determination—the final computation of the rounding error bound—for the simplified error analysis

(SEA) and the proposed probabilistic approach (PEA). The implementation of A-Abft takes the ABFT encoding block size as one input parameter for the runtime configuration of the ε-determination GPU kernels. Both SEA and PEA therefore show different kernel execution times for different encoding block sizes. Figure 8.3 depicts the execution times for all methods and an ABFT encoding block size of 32×32. The graph includes the execution times for SEA and PEA 2 with an ABFT encoding block size of 256×256 for comparison.

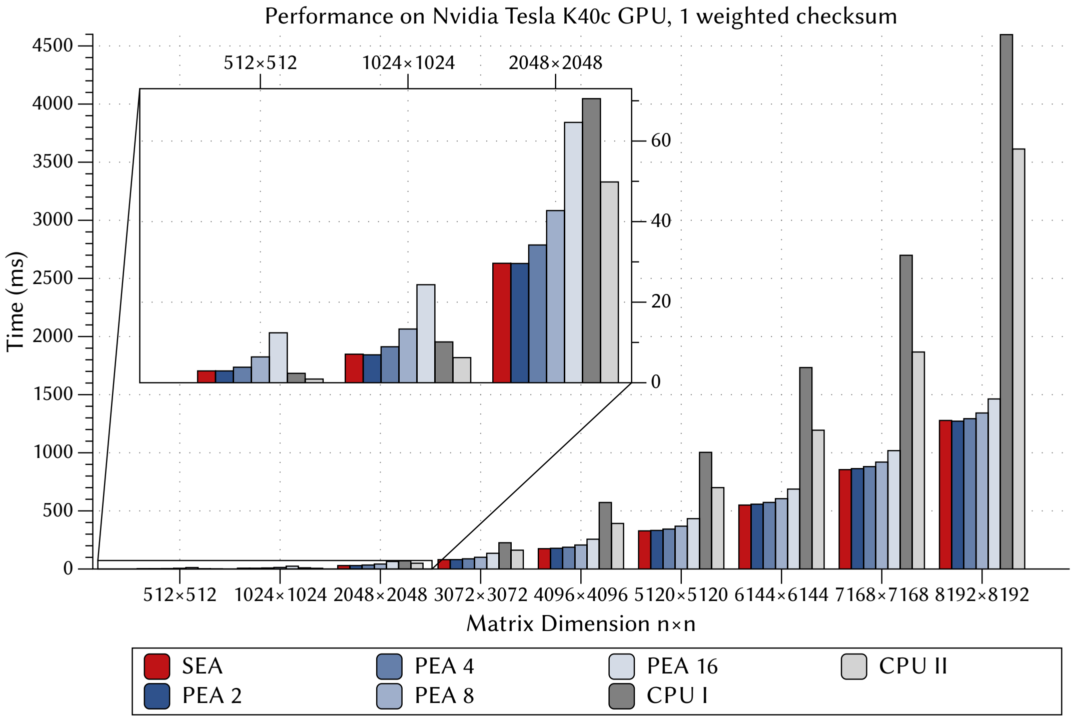


▲ **Figure 8.3** — Kernel execution times for SEA and PEA ε-determination.

In Appendix F, Table F.8 shows the kernel execution times for setups with one weighted checksum and an ABFT encoding block size of 32×32. Table F.9 lists the corresponding execution times for an ABFT encoding block size of 256×256. The simplified error analysis (SEA) exhibits short kernel execution times below 0.5 milliseconds across all test data set sizes and ABFT encoding block sizes. The probabilistic rounding error determination PEA requires between 1.3x and 5.3x more execution time compared to SEA. However, all these kernel execution times are significantly lower than the required times for the encoding of the matrices or the preprocessing steps. Table F.9 also shows that the kernel execution times of SEA and all variants of PEA benefit significantly from larger ABFT encoding block sizes.

8.1.4 Overall Computational Performance

Modern multi-core CPUs are able to provide impressive double precision floating-point performance. The compute servers used for this evaluation each consisted of two Intel Xeon CPUs with a total of 16 processing cores. These machines were able to deliver up to 280 and 396 GFLOPS of theoretical double precision floating-point peak performance. In order to assess the overall achievable performance and to demonstrate the benefits of A-Abft-protected matrix operations on GPU, the following results include performance values for an unprotected and highly optimized BLAS matrix multiplication[1] on these CPUs. The respective values are indicated by *CPU I* and *CPU II*. Figure 8.4 depicts the achieved runtimes for the ABFT-protected matrix multiplication using SEA and the A-Abft-protected matrix multiplication using the different variants of PEA, in comparison to the unprotected multiplication on CPU. For these experiments, one weighted checksum has been used and average values over all weighted checksum



▲ **Figure 8.4** — Performance of SEA and PEA with one weighted checksum on Nvidia Tesla K40c GPU compared to unprotected matrix multiplication on multicore CPUs.

[1] Intel Math Kernel Library (MKL) CBLAS DGEMM routine.

types and encoding block sizes are reported. In Appendix F, Figures F.1 and F.2 compare the overall computational performance for the experiments with one and two weighted checksums. In addition, Tables F.10 to F.38 report the runtimes for the different weighted checksum codes, encoding block sizes and problem sizes in detail.

The experimental results show that the unprotected matrix multiplication on *CPU I* and *CPU II* yields very high performance values. For a problem size of 7168×7168, *CPU I* took 2698.9 ms and *CPU II* 1866.8 ms. This corresponds to 272.9 and 394.6 GFLOPS, which is equivalent to 97.5% and 99.6% of the CPUs' theoretical floating-point peak performance. These results are in good compliance with the observed system CPU utilization of up to 1596% during the experiments, which indicates that all 16 processor cores have been utilized on both machines.

Starting with problems of size 1024×1024, the ABFT-protected matrix multiplications on the graphics processing unit (SEA and PEA2) achieved shorter computation times and outperformed the two multi-core CPUs. With growing problem sizes, the performance of the ABFT-protected matrix multiplications on the GPU increased substantially. With 1001.22 ms (PEA 2) and 1005.57 ms (SEA) for problems of size 8192×8192, a maximum speedup of 4.6× has been achieved over the highly optimized and *unprotected* matrix multiplication on the two multicore CPUs. The implementation of the A-ABFT-protected matrix multiplication achieved up to 1127.2 GFLOPS (PEA 2) for problems of size 8192×8192, which corresponds to 80.5% of the used NVIDIA TESLA K40C GPU's theoretical double precision floating-point peak performance of 1.4 TFLOPS. The implementation of ABFT with SEA achieved up to 1126.1 GFLOPS.

In order to assess the performance impact of the different ABFT design parameters such as the ABFT encoding block size, the types and combinations of applied weighted checksum codes, as well as the different PEA variants, a large number of A-ABFT-protected matrix multiplications has been performed across different problem sizes. Tables F.10 to F.38 in Appendix F report the performance results of these experiments in detail.

In summary, the experimental results show that the different types of weighted checksum codes, as well as their combinations, do not have a significant impact on the overall performance. The achieved performance values are very similar across all codes.

The ABFT encoding block size, however, has more influence on the achievable performance. For experimental setups with one and combinations of two types of weighted checksum codes, the encoding block sizes of 64×64 and 128×128 yielded the highest performance values for SEA and all PEA variants, across all problem sizes.

In comparison to the experimental setups with one weighted checksum type, the combinations of two checksum codes increased the computation times by only 3.3% (SEA) and 3.5% (PEA 2) on average.

For the A-Abft method with its probabilistic determination of rounding error bounds (PEA), the number of elements with largest absolute values that have to be considered has direct influence on the overall performance. As expected, PEA16 shows the lowest performance among the different PEA variants. In comparison of the average computation times, PEA8 is up to 1.9×, PEA4 up to 3.2× and PEA2 up to 4.1× faster for problem size 512×512. However, the performance penalty for PEA16 decreases significantly with increasing problem sizes and becomes almost negligible. As will be shown in Sections 8.2 and 8.3, the quality of the determined rounding error bounds and hence the error detection capabilities of PEA2 are comparable to those of PEA16. For practical applications, PEA2 is therefore the PEA variant of choice, which yields the highest computational performance.

8.2 Quality of Rounding Error Bounds

The quality of a determined rounding error bound, i.e. its distance from the actual rounding error, has direct impact on the error detection capability of an ABFT scheme. The A-Abft method proposed in this work utilizes the probabilistic approach introduced in Chapter 6 to determine such rounding error bounds. This probabilistic PEA method is evaluated and compared in this section against the state of the art simplified error analysis technique SEA from [Roy C93]. All experiments have been performed for the input data sets T_{pos}, T_{full} and T_{ortho}, across all problem sizes and ABFT encoding block sizes.

To measure the quality of the rounding error bounds determined by both methods, a large number of A-Abft-protected matrix multiplications has been performed. For each multiplication, the absolute rounding error that affects each checksum element of the result matrix has been computed together with the SEA and the different variants of the PEA rounding error bound. For reference purposes, all computations of interest have also been performed using a multi-precision floating-point data type with 100-bit precision provided by *The GNU Multiple Precision Arithmetic Library* (GMP) [The

G14]. The absolute rounding error $\lambda_{x,abs}$ that affects a checksum element x has been determined as

$$\lambda_{x,abs} := |x_{fp} - x_{GMP}|, \tag{8.2}$$

where x_{fp} is the value of the checksum element x computed in IEEE-754 double precision and x_{GMP} is the result computed with 100-bit precision.

Throughout *all* experiments and *all* parameter combinations, the rounding error bounds computed by the proposed PEA method showed a significantly higher quality compared to the SEA bounds. The PEA rounding error bounds were up to two orders of magnitude closer to the actual rounding error than the SEA bounds. Table 8.2 shows the average values for the absolute rounding error $\lambda_{x,abs}$, as well as the determined absolute SEA and PEA rounding error bounds for an ABFT encoding block size of 32×32 and a normal weighted checksum code. The reported experiments have been performed over the test data set $T_{ortho(0,2)}$ with $\alpha = 0$ and $\kappa = 2$. Similar experimental results were obtained for other combinations of parameters and test data sets.

▼ **Table 8.2** — Average absolute rounding error and determined absolute rounding error bounds for an ABFT encoding block size of 32×32 and normal weighted checksums over test data set $T_{ortho(0,2)}$.

MATRIX	ROUNDING	SEA	PEA 2	PEA 4	PEA 8	PEA 16
$[n \times n]$	ERROR λ_{abs}	ε_{abs}	ε_{abs}	ε_{abs}	ε_{abs}	ε_{abs}
512	$6.26 \cdot 10^{-11}$	$2.22 \cdot 10^{-6}$	$8.23 \cdot 10^{-8}$	$7.60 \cdot 10^{-8}$	$6.91 \cdot 10^{-8}$	$6.14 \cdot 10^{-8}$
1024	$2.44 \cdot 10^{-10}$	$1.66 \cdot 10^{-5}$	$5.24 \cdot 10^{-7}$	$4.86 \cdot 10^{-7}$	$4.47 \cdot 10^{-7}$	$4.07 \cdot 10^{-7}$
2048	$9.66 \cdot 10^{-10}$	$1.28 \cdot 10^{-4}$	$3.23 \cdot 10^{-6}$	$3.03 \cdot 10^{-6}$	$2.81 \cdot 10^{-6}$	$2.59 \cdot 10^{-6}$
3072	$2.18 \cdot 10^{-9}$	$4.29 \cdot 10^{-4}$	$9.35 \cdot 10^{-6}$	$8.82 \cdot 10^{-6}$	$8.23 \cdot 10^{-6}$	$7.64 \cdot 10^{-6}$
4096	$3.94 \cdot 10^{-9}$	$1.03 \cdot 10^{-3}$	$2.05 \cdot 10^{-5}$	$1.93 \cdot 10^{-5}$	$1.81 \cdot 10^{-5}$	$1.69 \cdot 10^{-5}$
5120	$6.07 \cdot 10^{-9}$	$1.98 \cdot 10^{-3}$	$3.65 \cdot 10^{-5}$	$3.45 \cdot 10^{-5}$	$3.24 \cdot 10^{-5}$	$3.02 \cdot 10^{-5}$
6144	$8.73 \cdot 10^{-9}$	$3.42 \cdot 10^{-3}$	$5.84 \cdot 10^{-5}$	$5.53 \cdot 10^{-5}$	$5.21 \cdot 10^{-5}$	$4.87 \cdot 10^{-5}$
7169	$1.19 \cdot 10^{-8}$	$5.44 \cdot 10^{-3}$	$8.78 \cdot 10^{-5}$	$8.32 \cdot 10^{-5}$	$7.85 \cdot 10^{-5}$	$7.36 \cdot 10^{-5}$
8192	$1.55 \cdot 10^{-8}$	$8.05 \cdot 10^{-3}$	$1.24 \cdot 10^{-4}$	$1.17 \cdot 10^{-4}$	$1.11 \cdot 10^{-4}$	$1.04 \cdot 10^{-4}$

However, since the input data sets exhibit very different characteristics with respect to the random values and the dynamics in the exponents, the absolute values of the rounding error and the corresponding rounding error bounds are only of limited use for a comparison across all test data sets. Relative errors are therefore considered in the following. The relative rounding error $\lambda_{x,rel}$ which affects a checksum element x is defined as

$$\lambda_{x,rel} := \frac{\lambda_{x,abs}}{|x|}. \tag{8.3}$$

The relative rounding error bounds are defined accordingly as

$$\varepsilon_{x,rel} := \frac{\varepsilon_{x,abs}}{|x|}. \tag{8.4}$$

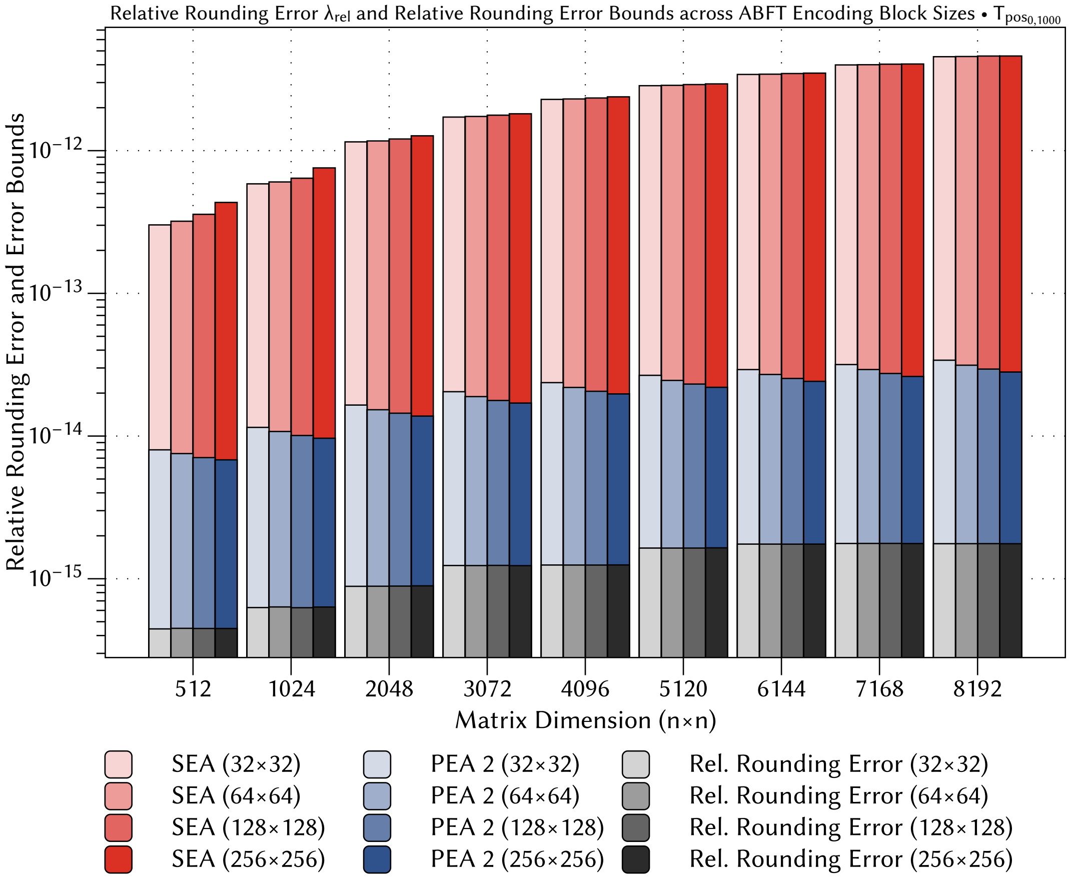
Figure 8.5 summarizes the experimental results of the quality evaluation for the test data set $T_{pos(0,1000)}$ across all problem sizes and all ABFT encoding block sizes. The corresponding results for the test data sets $T_{full(-1000,1000)}$ and $T_{ortho(0,2)}$ are depicted in Figures F.3 and F.4 in Appenix F. The results show that, across all the test data sets, the rounding error bounds determined by the PEA method introduced in this thesis are consistently closer to the actual rounding error, compared to the bounds determined by the simplified error analysis SEA.

For the probabilistic PEA approach, the number of considered elements with largest absolute values per vector has impact on the quality of the rounding error bound. With an increasing number of such elements, the quality of the rounding error bound improves. However, the actual differences between the PEA 2 bounds (two considered elements with largest absolute values) and PEA 16 bounds (sixteen considered elements with largest absolute values) are comparatively small and for the test data sets T_{pos} and T_{full} almost negligible. The differences between the two rounding error bounds are slightly bigger across the test data set T_{ortho}, but still reside within the same magnitude (cfg. Tables F.39, F.40 and F.41 in Appendix F). This makes the PEA 2 rounding error bound a good choice for most application scenarios.

In order to evaluate the differences between SEA and PEA rounding error bounds in more detail, the ratio between the actual rounding error and these rounding error bounds is considered in the following. The relation between rounding error and error bound is expressed by the quotient

$$d := \frac{\varepsilon_{x,rel}}{\lambda_{x,rel}} = \frac{\frac{\varepsilon_{x,abs}}{|x|}}{\frac{\lambda_{x,abs}}{|x|}} = \frac{\varepsilon_{x,abs}}{\lambda_{x,abs}}, \tag{8.5}$$

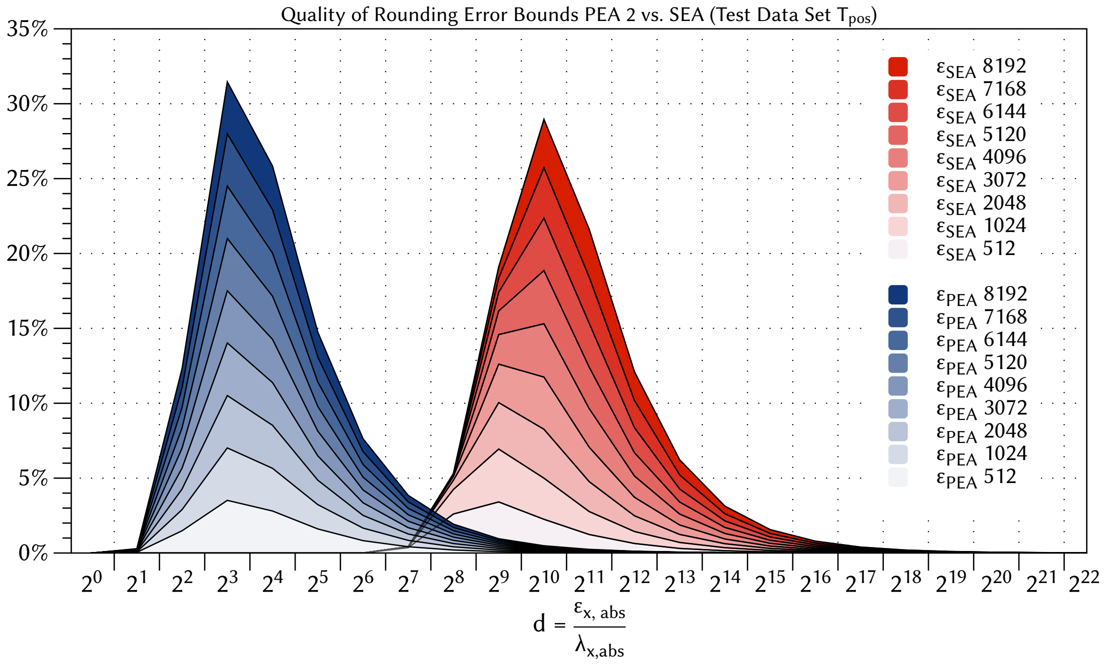
which is independent of the checksum elements x the bounds are determined for. The quotient d is reciprocally linked to the quality of the rounding error bounds. Values $d > 1$ indicate rounding error bounds that are larger than the actual rounding error.



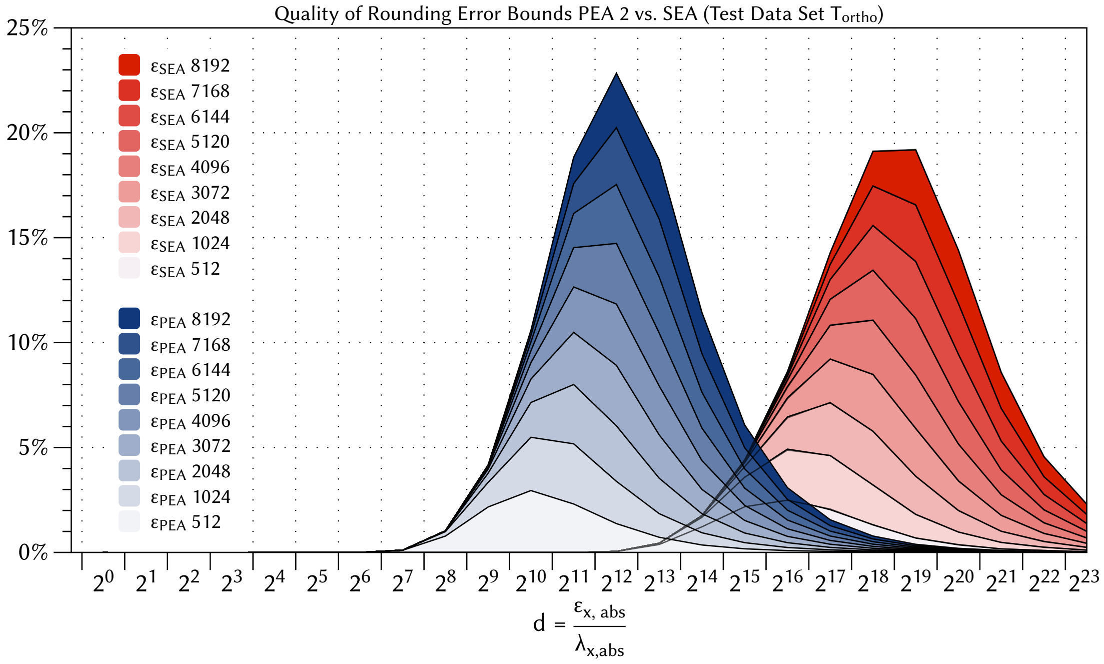
▲ **Figure 8.5** — Average relative rounding error λ_{rel} and determined relative rounding error bounds, test data set $T_{pos(0,1000)}$.

The larger the value of d, the higher the distance, and hence the lower the quality of the rounding error bound. Values $d < 1$ indicate rounding error bounds that reside below the rounding error. Figure 8.6 shows the histogram for d over the test data set T_{pos} and across all problem sizes and ABFT encoding block sizes. Figure 8.7 depicts the histogram for d over the test data set T_{ortho}. The experimental data that has been used to plot these graphs has been generated by Halder [Halde14].

Both histograms clearly reflect the advantage of the proposed PEA method over the SEA approach. For the test data set T_{pos}, the largest fraction of d values for the PEA bounds resides in bin 2^3 with more than 31.5%. For the SEA rounding error bound, 29% of its d values fall into the 2^{10} bin. Across different problem sizes, the distribution of d for the PEA rounding error bounds does not change notably. The largest fraction of d values



▲ **Figure 8.6** — Histogram of rounding error bound quality, test data set T_{pos}.



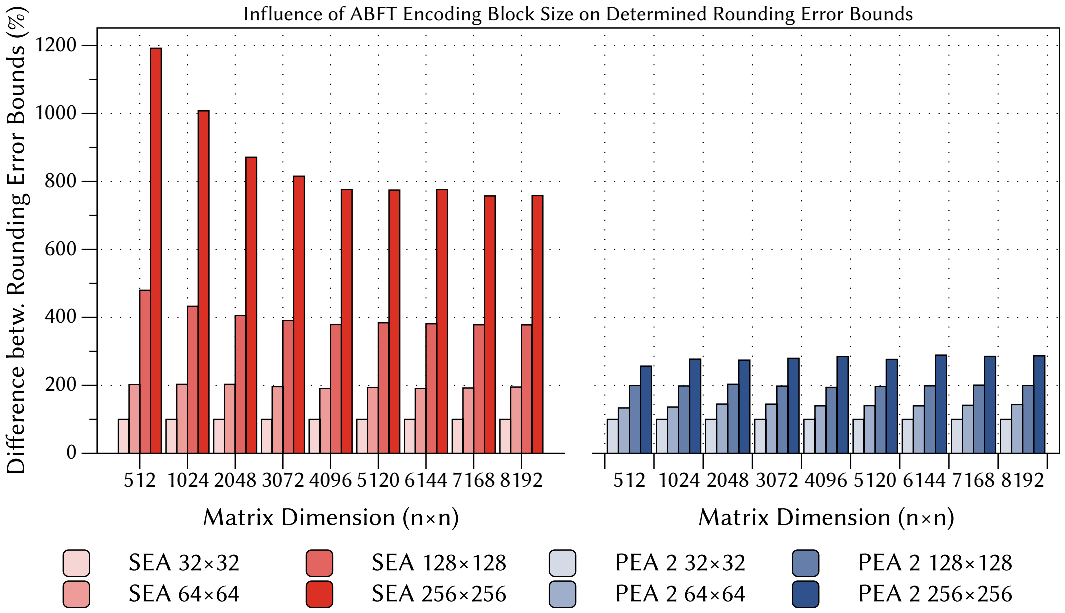
▲ **Figure 8.7** — Histogram of rounding error bound quality, test data set T_{ortho}.

remains within the 2^3 bin. For the SEA rounding error bound a more significant shift in the distribution of d can be observed for increasing problem sizes. For matrices of dimension 512×512, the largest fraction of d values resides within the 2^9 bin, whereas for the problem size 8192×8192, the largest fraction falls into bin 2^{10}. This corresponds to a deterioration of a factor $2x$ for the SEA approach.

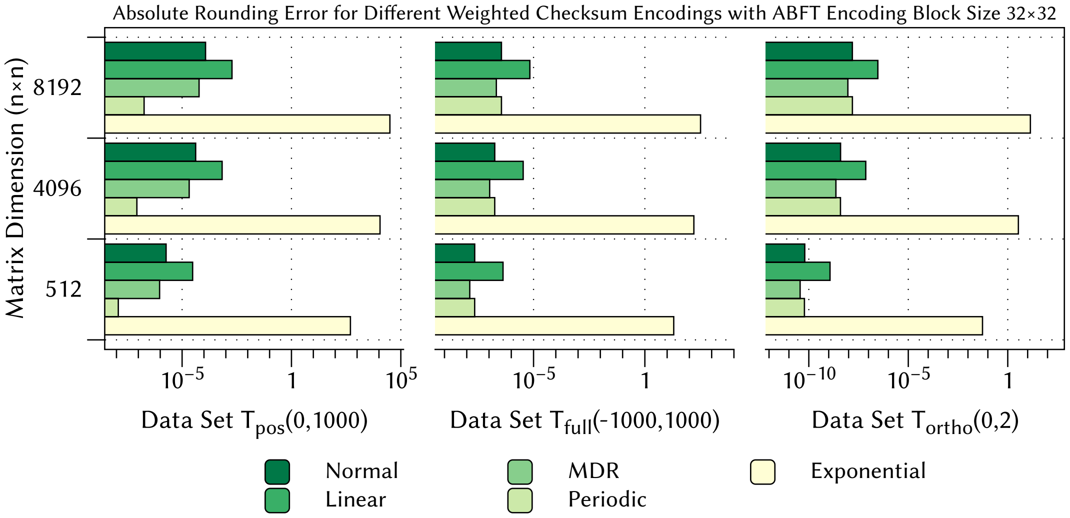
For the test data set T_{ortho}, the largest fraction of d values for the PEA rounding error bounds resides between 2^{10} and 2^{12}. For increasing problem sizes, this indicates a maximum shift of $4x$. Again, the SEA approach exhibits an inferior characteristic. The largest fraction of its d values resides between 2^{16} and 2^{19}, which corresponds to a deterioration of a factor of $8x$ for increasing problem sizes. The evaluations for test data set T_{full} lead to similar results.

Besides the pronounced influence of the problem size on the quality of the SEA rounding error bounds, the ABFT encoding block size also shows significant impact. Figure 8.8 visualizes this impact onto the absolute SEA and PEA rounding error bounds. For both methods, the determined bound for the ABFT encoding block size 32×32 has been chosen as 100% reference. The rounding error bounds for the larger ABFT encoding block sizes are presented in relation to this baseline. As can be seen on the left-hand side graph, the SEA rounding error bounds increase by factors between $2x$ and up to $11.9x$. This behavior can be explained by the fact that the ABFT encoding block size is one of the direct input parameters of the SEA approach and that the involved vector norms have to be calculated over larger vectors for increasing encoding block sizes. The proposed PEA method, in contrast, shows only a weak dependence on the ABFT encoding block size. The determined PEA rounding error bounds increased at most by a factor of $2.9x$.

The experiments that have been conducted for the evaluation of the rounding error bound quality also enabled a closer look at the rounding error associated with the different types of weighted ABFT checksum encodings. Throughout all test data sets, the MDR and the periodic weighted checksum encodings featured the smallest absolute rounding error. For the test data set T_{pos}, the periodic encoding outperformed all other encodings. The exponential weighted checksum encoding introduced the largest rounding error. Figure 8.9 summarizes the absolute rounding error for the different weighted checksum encodings across the three test data sets T_{pos}, T_{full} and T_{ortho}.



▲ **Figure 8.8** — Influence of ABFT encoding block size on determined absolute rounding error bounds. Comparison of SEA and PEA 2 for normal weighted checksums across the test data set $T_{ortho}(0,2)$.



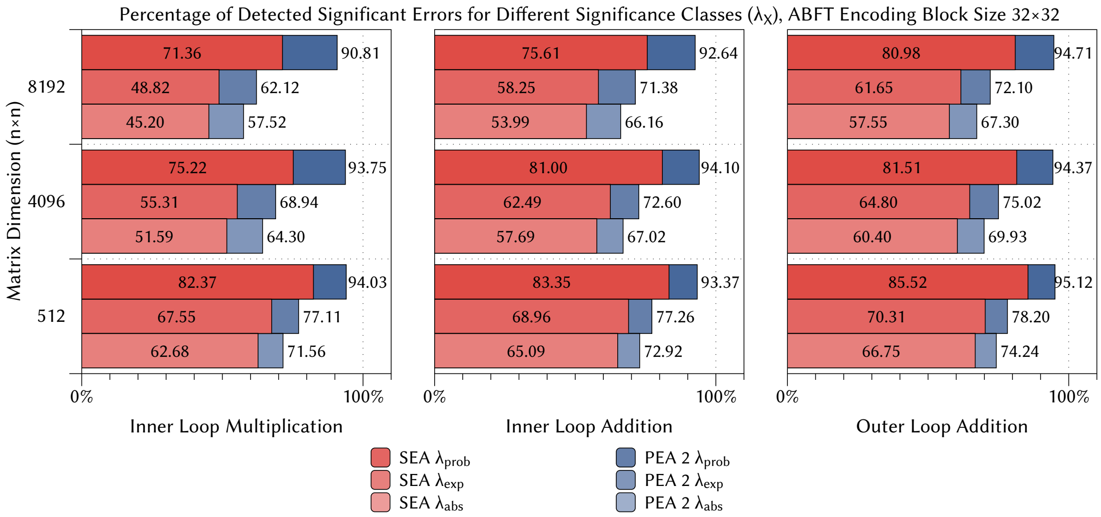
▲ **Figure 8.9** — Absolute rounding error for different weighted checksum encodings with ABFT encoding block size 32×32 over the three input data sets $T_{pos}(0,1000)$, $T_{full}(-1000,1000)$ and $T_{ortho}(0,2)$.

8.3 Error Detection Capabilities

The last part of the experimental evaluation focuses on the error detection capabilities of the A-ABFT-protected matrix multiplication on GPU. Three main parameters have direct influence on the error detection. First, the weighted checksum code that is used to encode the input data before the matrix multiplication is performed. Second, in case of floating-point computations, the applied method for the determination of the required rounding error bounds. Third, the ABFT encoding block size that is used for the partitioned encoding of the input data. To evaluate the interplay and impact of these parameters, a large number of fault injection experiments has been performed across all parameters of interest, as well as all problem sizes and test data sets.

Each fault injection experiment consisted of a full matrix multiplication. First, a streaming multiprocessor (SMX) and a processing core within this SMX has been chosen at random. The fault injection routine then selected randomly one out of three potential targets for the injection. The first potential target is the multiplication operation within the inner loop of the matrix multiplication algorithm. The second potential target is the addition operation that accumulates the intermediate products within the inner loop. The third potential target for the injection of faults is the addition operation in the outer loop of the matrix multiplication. The injection itself has been performed on the floating-point results of the above target operations. The fault injection routine is able to inject single or multiple bit-flips into the sign, the exponent and the significand of the target floating-point number. Since injected faults in the sign and exponent part of these numbers have been successfully detected throughout all experiments, the results of injected single bit-flips in the significand are considered in the following. The outcomes of the fault injection experiments have been analyzed and classified into detected and undetected errors (false negatives).

A meaningful classification of errors requires a proper baseline which allows the distinction into significant and insignificant errors. Significant errors are critical and have to be tackled, whereas insignificant errors in the magnitude of the common rounding error can be tolerated. One option for such a baseline is the absolute rounding error λ_{abs}, which, at first sight, represents a sharp criterion. However, the absolute rounding error is susceptible to factors like the execution order of floating-point operations or the use of special instructions like fused multiply-add operations. Due to this fact, two additional baselines have been derived for this evaluation. The first baseline is the expected rounding error λ_{exp}, determined by the analysis of the squared sums during the formation of the inner product (see Equation 6.51). The second baseline is the expected rounding error λ_{prob}, which is determined using the probabilistic method introduced in Chapter 6 (see Equation 6.52). The three baselines fulfill the following magnitude relation: $\lambda_{abs} \leq \lambda_{exp} \leq \lambda_{prob}$. Figure 8.10 depicts the three different baselines for the determination of significant errors in comparison. In the remainder of this chapter, the results of the error detection evaluation are presented based on λ_{prob}.

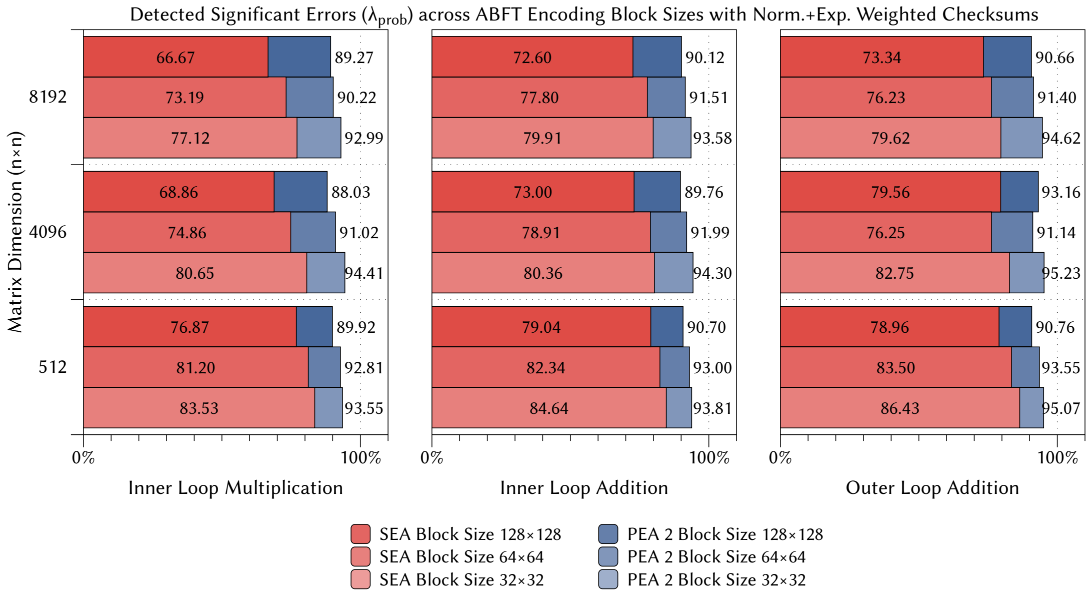


▲ **Figure 8.10** — Comparison of the three different baselines for the classification of significant errors. Test data set $T_{ortho(0,8192)}$ with $\alpha = 0$ and $\kappa = 8192$, for normal weighted checksums.

The following results of the error detection evaluation are presented for the test data set $T_{ortho(0,65536)}$ with $\alpha = 0$ and $\kappa = 65536$. This is the most challenging test data set with the highest diversity in the input values and the highest dynamic in the exponents. In order to allow the comparison of all weighted checksum encodings, results for the ABFT encoding block sizes 32×32, 64×64 and 128×128 are reported. The results for ABFT encoding blocks of size 256×256, which can only be used when a single weighted checksum is applied, closely followed the trend which becomes visible for the presented ABFT block sizes. The detection rates for the block size 256×256 were the lowest throughout all experiments. The results are reported in the following for the problem sizes 512×512, 4096×4096 and 8192×8192. The results for the problem sizes in between these sizes also closely followed the presented trend.

The proposed probabilistic method for the determination of rounding error bounds outperformed the state of the art simplified error analysis consistently throughout *all* experiments[2]. The error detection rates of the PEA method (PEA 2) reached up to 95.23%. The difference in the error detection rate between PEA 2 and SEA reached up to 26.91% (see Figure F.12).

For all types of weighed checksum encodings and across all problem sizes, the ABFT encoding block size of 32×32 yielded the highest error detection rates. The combinations of two weighted checksums yielded consistently high error detection rates across all ABFT encoding block sizes and problem sizes. However, the differences between the peak detection rates for the combined weighted checksums and single weighted checksum such as the linear, the MDR and the periodic encodings were comparatively small. Figure 8.11 depicts the error detection rates for a combination of normal and exponential weighted checksums.



▲ **Figure 8.11** — Detected significant errors (λ_{prob}) across different ABFT encoding block sizes for normal + exponential weighted checksums and test data set $T_{ortho(0,65536)}$.

Although the combination of normal and exponential weighted checksums showed very high detection rates, the stand-alone application of the exponential checksum (see

[2] For all three baselines λ_{abs}, λ_{exp} and λ_{prob}.

Figure F.8) is a conspicuous exception. It showed significantly worse error detection rates and a particularly strong dependence on the ABFT encoding block sizes. For the encoding block size 32 × 32, the exponential weighted checksum reached between 69.51% and 74.32% error detection rates with PEA 2. For the simplified error analysis SEA, the error detection rates resided between 63.5% and 68.92%. For increasing ABFT encoding block sizes, the error detection rates dropped to 42.82% – 47.22% (ABFT encoding block size 64 × 64, problem size 512 × 512) and 22.95% – 27.07% (ABFT encoding block size 128 × 128, problem size 512 × 512). The lowest error detection rates with 17.58% – 19.91% became evident for the simplified error analysis SEA and problem size 8192 × 8192. The sole use of exponential weighted checksums can therefore not be recommended on the basis of these results.

Across all test data sets, the combination of normal and MDR weighted checksums performed very well. For the single weighted checksums the linear (up to 94.19%), the MDR (up to 94.04%) and the periodic (up to 94.57%) weighted checksum encodings performed very similar with high error detection rates. For the test data set T_{pos}, which shows very homogeneous input values, the single weighted checksums linear, MDR and periodic outperformed the combined checksums. For test data set T_{full}, the single weighted checksums linear and MDR yielded the highest error detection rates, followed by the periodic encoding. The normal checksum encoding and the combined weighted checksums achieved in some cases significantly worse error detection rates.

In Appendix F, the Figures F.5 to F.20 present the detailed experimental results for the evaluation of the error detection capabilities for significance class λ_{abs}, as well as for λ_{prob}.

Based on these results, the linear, the MDR and in many cases the periodic weighted checksum encodings can be universally recommended, especially for applications where the characteristics of the input data is unknown.

8.4 Discussion of Experimental Results

In summary, the results of the experimental evaluation with respect to the achievable computational performance, the quality of the rounding error bounds, as well as the error detection capabilities, show that the introduced A-Abft method is superior to the state of the art simplified error analysis method (SEA).

Although A-Abft involves more complex algorithmic steps in comparison to SEA, the high efficiency of the method enables a high computational performance. This performance gain, compared to modern multi-core CPUs, enables a substantial acceleration of sophisticated applications in science and engineering combined with an improved fault tolerance.

The rounding error bounds determined by the A-Abft method (PEA) consistently show a significant improvement over the state of the art, and are up to two orders of magnitude closer to the actual rounding error. The method thereby exhibits a much lower dependence on the ABFT encoding block size and the problem size. High quality rounding error bounds are an essential prerequisite for the reduction of false negative error detections in ABFT-protected floating-point applications.

The results of the fault injection experiments show that, independent of the applied weighted checksum encoding, the problem size and the characteristics of the input data, the A-Abft method consistently achieves better error detection rates, enabling an efficient, effective and autonomously operating ABFT scheme on GPUs.

Chapter 9

Conclusions

The rapid spread and growing importance of scientific computing and computer-based simulation technology fundamentally change the way how research is carried out and how complex products are developed. Modern graphics processing units with their tremendous floating-point compute performance and constantly increasing parallelism act as propellant charge and pave the way for completely new classes of applications. Sophisticated applications in scientific computing and simulation technology, which were only possible in large compute centers with high-performance computing equipment a few years ago, are no brought to the desktop using the power of GPUs.

However, with the spread of these applications and their growing influence on social, economical and political processes and decisions, the trustworthiness of the computed results becomes crucial. Fault tolerance therefore becomes mandatory to improve the reliability and ensure the trustworthiness of such computations. Isolated fault tolerance measures are no longer sufficient. Cross-layer reliability requires fault tolerance at each system layer, from the hardware up to the software and the algorithms.

Algorithm-Based Fault Tolerance is a mature fault tolerance technique that allows to improve the reliability of essential mathematical tasks such as linear algebra matrix operations. Algorithm-Based Fault Tolerance perfectly fits into the concept of cross-layer reliability and can be combined with other hardware- and software-based fault tolerance techniques.

Linear algebra matrix operations like the matrix multiplication are essential components of many applications in scientific computing. The development and provision of fault tolerant GPU kernels for such operations will therefore help to significantly improve the reliability. However, for computations performed in floating-point arithmetic, Algorithm-Based Fault Tolerance schemes often suffer from poor error detection rates due to rounding errors. Solutions like manually set rounding error thresholds or calibration runs are impractical and hamper the application of Algorithm-Based Fault Tolerance. State of the art methods for the analytical derivation of rounding error bounds, such as the simplified error analysis SEA [Roy C93] often lead to weak bounds, which cause insufficient error detection.

In this work, a novel approach called *Autonomous Algorithm-Based Fault Tolerance* (A-ABFT) has been introduced for linear algebra matrix operations on GPUs. The approach utilizes a probabilistic model for the determination of rounding error bounds, which are significantly closer to the actual rounding error than state of the art approaches. The required algorithmic steps are designed in a way that the parallel compute capabilities of modern GPUs are utilized to keep the overhead as small as possible. Since the proposed A-ABFT method operates independently of the input data and does not require any user interaction, an autonomous operation can be achieved. This eases the integration of such fault tolerant GPU kernels into existing scientific applications.

In-depth experimental evaluations were performed to compare the introduced A-ABFT method against the state of the art simplified error analysis. For the computational performance, the experimental results showed that the A-ABFT method does not create any significant overhead compared to state of the art techniques. With peak performance values of 1127.2 GFLOPS, A-ABFT achieved up to 80% of the theoretical peak performance of the used GPU accelerators. The evaluation of the error detection capabilities revealed error detection rates of up to 95.2% for A-ABFT, which corresponds to an improvement of the state of the art of up to 23%. Besides the evaluation with synthetic test data sets, two case studies were performed which demonstrated the effectiveness and efficiency of A-ABFT in real world applications.

Ongoing Research and Future Work

The ongoing research focuses on the integration of the probabilistic method for the determination of rounding error bounds into more linear algebra matrix operations on GPU to provide a library of fault tolerant, GPU-accelerated building blocks for scientific applications. The research also focuses on further improvements of the rounding error bounds. For the matrix multiplication, the consideration of block-based local and global average values has already been investigated for the probabilistic method by Halder [Halde14]. First results indicated improved rounding error bounds, although the performance loss was significantly higher, compared to the proposed A-ABFT method.

Bibliography

[Abdel09] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu. Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster. In *Proceedings of the International Conference on High Performance Computing Simulation (HPCS '09)*, pages 36–43. June 2009. [page 8]

[Agull11] J. I. Agulleiro and J. J. Fernandez. Fast tomographic reconstruction on multicore computers. *Bioinformatics*, 27(4):582–583, 2011. URL http://bioinformatics.oxfordjournals.org/content/27/4/582.abstract. [page 8]

[Alam05] M. Alam and S. Mahapatra. A comprehensive model of PMOS NBTI degradation. *Microelectronics Reliability*, 45(1):71 – 81, 2005. URL http://www.sciencedirect.com/science/article/pii/S0026271404001751. [page 11]

[Ander08] J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008. URL http://www.sciencedirect.com/science/article/pii/S0021999108000818. [page 8]

[Anfin88] C. J. Anfinson and F. T. Luk. A Linear Algebraic Model of Algorithm-Based Fault Tolerance. *IEEE Transactions on Computers*, 37(12):1599–1604, 1988. [pages 62, 64, and 66]

[Arnau11] L. Arnaud, P. Lamontagne, R. Galand, E. Petitprez, D. Ney, and P. Waltz. Electromigration Induced Void Kinetics In Cu Interconnects For Advanced CMOS Nodes. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS'11)*, pages 3E.1.1–3E.1.10. April 2011. [page 11]

[Aseno98] A. Asenov. Random Dopant Induced Threshold Voltage Lowering and Fluctuations in Sub-0.1 μm MOSFET's: A 3-D "Atomistic" Simulation Study. *IEEE Transactions on Electron Devices*, 45(12):2505–2513, Dec 1998. [page 11]

[Aseno03] A. Asenov, S. Kaya, and A. R. Brown. Intrinsic Parameter Fluctuations in Decananometer MOSFETs Introduced by Gate Line Edge Roughness. *IEEE Transactions on Electron Devices*, 50(5):1254–1260, May 2003. [page 11]

[Assad92] F. T. Assad and S. Dutt. More Robust Tests in Algorithm-Based Fault-Tolerant Matrix Multiplication. In *Digest of Papers of the Twenty-Second International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 430–439. 1992. [page 72]

[Avizi04] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan 2004. [pages 17 and 20]

[Aykan87] C. Aykanat and F. Ozguner. A Concurrent Error Detecting Conjugate Gradient Algorithm on a Hypercube Multiprocessor. In *IEEE 17th International Symposium on Fault Tolerant Computing*, pages 204–209. 1987. [page 58]

[Backe12] J. Backer and R. Karri. Balancing Performance and Fault Detection for GPGPU Workloads. In *30th IEEE International Conference on Computer Design (ICCD'12)*, pages 518–519. Sept 2012. [page 50]

[Bakho09] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, pages 163–174. April 2009. [pages 41 and 52]

[Baner88] P. Banerjee and C. B. Stunkel. A Novel Approach to System-level Fault Tolerance in Hypercube Multiprocessors. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues - Volume 1*, pages 307–311. ACM, New York, NY, USA, 1988. ISBN 0-89791-278-0. URL http://doi.acm.org/10.1145/62297.62330. [pages 58 and 71]

[Baner90] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham. Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor. *IEEE Transactions on Computers*, 39(9):1132–1145, 1990. [pages 58 and 71]

[Barlo81] J. L. Barlow. *Probabilistic Error Analysis of Floating Point and CRD Arithmetics.* Ph.D. thesis, Evanston, IL, USA, 1981. AAI8124850. [page 87]

[Barlo83] J. L. Barlow. Fast a posteriori bounds for roundoff error in Gaussian elimination. Technical report, Technical Report No. CS-83-20, Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania (September 1983), 1983. [page 87]

[Barlo85a] J. L. Barlow and E. H. Bareiss. On Roundoff Error Distributions in Floating Point and Logarithmic Arithmetic. *Computing*, 34(4):325–347, 1985. [pages 84, 86, 89, 90, 92, and 94]

[Barlo85b] J. L. Barlow and E. H. Bareiss. Probabilistic Error Analysis of Gaussian Elimination in Floating Point and Logarithmic Arithmetic. *Computing*, 34(4):349–364, 1985. URL http://dx.doi.org/10.1007/BF02251834. [pages 87 and 92]

[Bayra06] I. Bayraktaroglu, J. Hunt, and D. Watkins. Cache Resident Functional Microprocessor Testing: Avoiding High Speed IO Issues. In *Proceedings of the IEEE International Test Conference (ITC'06)*, pages 1–7. Oct 2006. [page 48]

[Belle08] R. G. Belleman, J. Bédorf, and S. F. P. Zwart. High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA. *New Astronomy*, 13(2):103 – 112, 2008. URL http://www.sciencedirect.com/science/article/pii/S1384107607000760. [page 8]

[Benfo38] F. Benford. The Law of Anomalous Numbers. *Proceedings of the American Philosophical Society*, 78(4):pp. 551–572, 1938. URL http://www.jstor.org/stable/984802. [pages 86 and 87]

[Benne97] G. R. Bennett. The application of virtual prototyping in the development of complex aerospace products. *Aircraft Engineering and Aerospace Technology*, 69(1):19–25, 1997. [page 4]

[Bensc08] J. Benschop, V. Banine, S. Lok, and E. Loopstra. Extreme ultraviolet lithography: Status and prospects. *Journal of Vacuum Science & Technology B*, 26(6):2204–2207, 2008. [page 11]

[Berge05] A. Berger, L. Bunimovich, and T. Hill. One-Dimensional Dynamical Systems and Benford's Law. *Transactions of the American Mathematical Society*, 357(1):197–219, 2005. [page 87]

[Berge11] A. Berger and T. P. Hill. A Basic Theory of Benford's Law. *Probability Surveys*, 8:1–126, 2011. [pages 87 and 88]

[Bliss88] W. G. Bliss, M. R. Lightner, and B. Friedlander. Numerical Properties Of Algorithm-Based Fault-Tolerance For High Reliability Array Processors. In *Twenty-Second Asilomar Conference on Signals, Systems and Computers*, volume 2, pages 631–635. 1988. [page 67]

[Bol09] D. Bol, R. Ambroise, D. Flandre, and J.-D. Legat. Interests and Limitations of Technology Scaling for Subthreshold Logic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(10):1508–1519, Oct 2009. [page 11]

[Bonga95] I. Bongartz, A. R. Conn, N. Gould, and P. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. *ACM Transactions on Mathematical Software*, 21(1):123–160, March 1995. URL http://doi.acm.org/10.1145/200979.201043. [page 113]

[Borka05] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, Nov 2005. [page 11]

[Bosil15] G. Bosilca, A. Bouteiller, T. Herault, Y. Robert, and J. Dongarra. Composing Resilience Techniques: ABFT, Periodic and Incremental Checkpointing. *International Journal of Networking and Computing*, 5(1):2–25, 2015. [page 47]

[Bourg04] A. Bourgeat, M. Kern, S. Schumacher, and J. Talandier. The COUPLEX Test Cases: Nuclear Waste Disposal Simulation. *Computational Geosciences*, 8(2):83–98, 2004. [page 9]

[Bowma09] K. Bowman, J. Tschanz, C. Wilkerson, S.-L. Lu, T. Karnik, V. De, and S. Borkar. Circuit Techniques for Dynamic Variation Tolerance. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 4–7. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-497-3. URL `http://doi.acm.org/10.1145/1629911.1629915`. [page 11]

[Braun12a] C. Braun, M. Daub, A. Schöll, G. Schneider, and H.-J. Wunderlich. Parallel Simulation of Apoptotic Receptor-Clustering on GPGPU Many-Core Architectures. In *IEEE International Conference on Bioinformatics and Biomedicine (BIBM'12)*, pages 1–6. 2012. [page 8]

[Braun12b] C. Braun, S. Holst, H.-J. Wunderlich, J. M. Castillo, and J. Gross. Acceleration of Monte-Carlo Molecular Simulations on Hybrid Computing Architectures. In *IEEE 30th International Conference on Computer Design (ICCD'12)*, pages 207–212. 2012. [page 8]

[Braun14] C. Braun, S. Halder, and H.-J. Wunderlich. A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*. 2014. [pages 83, 100, 102, and 121]

[Burke91] J. Burke and E. Kincanon. Benford's Law and Physical Constants: The Distribution of Initial Digits. *American Journal of Physics*, 59(10):952–952, 1991. [page 87]

[Bushn00] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, volume 17 of *Frontiers in Electronic Testing*. Springer, 2000. ISBN 978-0-7923-7991-1. [pages 20 and 47]

[Busto79] J. Bustoz, A. Feldstein, R. Goodman, and S. Linnainmaa. Improved Trailing Digits Estimates Applied to Optimal Computer Arithmetic. *Journal of the ACM*, 26(4):716–730, October 1979. URL `http://doi.acm.org/10.1145/322154.322162`. [page 89]

[Cappe09] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009. URL `http://hpc.sagepub.com/content/23/4/374.abstract`. [page 10]

[Carte10] N. P. Carter, H. Naeimi, and D. S. Gardner. Design Techniques for Cross-layer Resilience. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1023–1028. European Design and Automation Association, Leuven, Belgium, 2010. ISBN 978-3-9810801-6-2. URL `http://dl.acm.org/citation.cfm?id=1870926.1871178`. [page 14]

[Catan08a] B. Catanzaro, K. Keutzer, and B.-Y. Su. Parallelizing CAD: A Timely Research Agenda for EDA. In *Proceedings of the 45th Annual Design Automation Conference (DAC'08)*, DAC '08, pages 12–17. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-115-6. URL `http://doi.acm.org/10.1145/1391469.1391475`. [page 6]

[Catan08b] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 104–111. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-205-4. URL `http://doi.acm.org/10.1145/1390156.1390170`. [page 8]

[Chang10] D. Chang, C. Jenkins, P. Garcia, S. Gilani, P. Aguilera, A. Nagarajan, M. Anderson, M. Kenny, S. Bauer, M. Schulte, and K. Compton. ERCBench: An Open-Source Benchmark Suite for Embedded and Reconfigurable Computing. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'10)*, pages 408–413. Aug 2010. [page 51]

[Chatt09a] D. Chatterjee, A. DeOrio, and V. Bertacco. Event-driven Gate-level Simulation with GP-GPUs. In *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*, pages 557–562. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-497-3. URL `http://doi.acm.org/10.1145/1629911.1630056`. [page 8]

[Chatt09b] D. Chatterjee, A. DeOrio, and V. Bertacco. GCS: High-performance Gate-level Simulation with GP-GPUs. In *Proceedings of the Conference on Design,* *Automation and Test in Europe (DATE'09)*, DATE '09, pages 1332–1337. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2009. ISBN 978-3-9810801-5-5. URL `http://dl.acm.org/citation.cfm?id=1874620.1874941`. [page 8]

[Che09] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*, pages 44–54. IEEE, 2009. [pages 41, 42, and 52]

[Chen86] C.-Y. Chen and J. A. Abraham. Fault-Tolerant Systems for the Computation of Eigenvalues and Singular Values. In *SPIE Proc. 30th Annual Technical Symposium*, pages 228–237. International Society for Optics and Photonics, 1986. [page 58]

[Chen99] P. Chen, L. Wu, G. Zhang, and Z. Liu. A Unified Compact Scalable δi_d Model for Hot Carrier Reliability Simulation. In *Proceedings of the 37th Annual IEEE International Reliability Physics Symposium*, pages 243–248. 1999. [page 11]

[Chen01] L. Chen and S. Dey. Software-Based Self-Testing Methodology for Processor Cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(3):369–380, Mar 2001. [page 48]

[Chen03] L. Chen, S. Ravi, A. Raghunathan, and S. Dey. A Scalable Software-based Self-test Methodology for Programmable Processors. In *Proceedings of the 40th Annual Design Automation Conference (DAC'03)*, DAC '03, pages 548–553. ACM, New York, NY, USA, 2003. ISBN 1-58113-688-9. URL `http://doi.acm.org/10.1145/775832.775973`. [page 48]

[Cohen73] A. M. Cohen. *Numerical Analysis*. European mathematics series. McGraw-Hill, 1973. ISBN 9780070840126. URL `http://books.google.de/books?id=DivvAAAAMAAJ`. [page 68]

[Corno03] F. Corno, G. Cumani, M. Reorda, and G. Squillero. Fully Automatic Test Program Generation for Microprocessor Cores. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 1006–1011. 2003. [page 48]

[Croix13] J. F. Croix, S. P. Khatri, and K. Gulati. Using GPUs to Accelerate CAD Algorithms. *IEEE Design Test*, 30(1):8–16, Feb 2013. [page 8]

[Dantz53] G. B. Dantzig and W. Orchard-Hays. Notes on Linear Programming: Part V—Alternate Algorithm for the Revised Simplex Method using Product Form for the Inverse. *The RAND Corporation Research Memorandum*, 1268, 1953. [page 111]

[Dantz63] G. B. Dantzig. Linear Programming and Extensions. *A RAND Corporation Research Study*, 1963. [page 110]

[Davie11] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High Performance Linpack Benchmark: A Fault Tolerant Implementation Without Checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 162–171. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0102-2. URL http://doi.acm.org/10.1145/1995896.1995923. [page 58]

[DeHon10] A. DeHon, H. M. Quinn, and N. P. Carter. Vision for Cross-Layer Optimization to Address the Dual Challenges of Energy and Reliability. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'10)*, pages 1017–1022. March 2010. [page 13]

[Demat10] L. Dematté and D. Prandi. GPU computing for systems biology. *Briefings in Bioinformatics*, 11(3):323–333, 2010. URL http://bib.oxfordjournals.org/content/11/3/323.abstract. [page 8]

[Di Ca13a] S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta. A Software-based Self Test of CUDA Fermi GPUs. In *18th IEEE European Test Symposium (ETS'13)*, pages 1–6. May 2013. [pages 49 and 52]

[Di Ca13b] S. Di Carlo, G. Gambardella, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta. Fault Mitigation Strategies for CUDA GPUs. In *IEEE International Test Conference (ITC'13)*, pages 1–8. Sept 2013. [page 52]

[Dimit09] M. Dimitrov, M. Mantor, and H. Zhou. Understanding Software Approaches for GPGPU Reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages

94–104. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-517-8. URL http://doi.acm.org/10.1145/1513895.1513907. [pages 50 and 54]

[Du11] P. Du, P. Luszczek, and J. J. Dongarra. High Performance Dense Linear System Solver with Soft Error Resilience. In *IEEE International Conference on Cluster Computing (CLUSTER'11)*, pages 272–280. 2011. [page 58]

[Dutt96] S. Dutt and F. T. Assaad. Mantissa-Preserving Operations and Robust Algorithm Based Fault Tolerance for Matrix Computations. *IEEE Transactions on Computers*, 45(4):408–424, 1996. [page 72]

[Dweik14] W. Dweik, M. Abdel-Majeed, and M. Annavaram. Warped-Shield: Tolerating Hard Faults in GPGPUs. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, pages 431–442. June 2014. [page 51]

[Ehlig10] C. Ehlig-Economides and M. J. Economides. Sequestering carbon dioxide in a closed underground volume. *Journal of Petroleum Science and Engineering*, 70:123 – 130, 2010. URL http://www.sciencedirect.com/science/article/pii/S0920410509002368. [pages 8 and 9]

[Engel03] H.-A. Engel and C. Leuenberger. Benford's Law for Exponential Random Variables. *Statistics & Probability Letters*, 63(4):361 – 365, 2003. URL http://www.sciencedirect.com/science/article/pii/S0167715203001019. [page 87]

[Fang14] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*, pages 221–230. March 2014. [page 41]

[Felds76] A. Feldstein and R. Goodman. Convergence Estimates for the Distribution of Trailing Digits. *Journal of the ACM*, 23(2):287–297, April 1976. URL http://doi.acm.org/10.1145/321941.321948. [page 89]

[Felds82] A. Feldstein and R. H. Goodman. Loss of Significance in Floating Point Subtraction and Addition. *IEEE Transactions on Computers*, C-31(4):328–335, 1982. [page 87]

[Felds86] A. Feldstein and P. Turner. Overflow, Underflow, and Severe Loss of Significance in Floating-Point Addition and Subtraction. *IMA Journal of Numerical Analysis*, 6(2):241–251, 1986. URL `http://imajna.oxfordjournals.org/content/6/2/241.abstract`. [page 87]

[Flehi66] B. J. Flehinger. On the Probability that a Random Integer has Initial Digit A. *The American Mathematical Monthly*, 73(10):pp. 1056–1061, 1966. URL `http://www.jstor.org/stable/2314636`. [page 87]

[Fluhr14] E. J. Fluhr, J. Friedrich, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, A. Hall, D. Hogenmiller, F. Malgioglio, R. Nett, J. Paredes, J. Pille, D. Plass, R. Puri, P. Restle, D. Shan, K. Stawiasz, Z. T. Deniz, D. Wendel, and M. Ziegler. 5.1 POWER8TM: A 12-Core Server-Class Processor in 22nm SOI with 7.6Tb/s Off-Chip Bandwidth. In *Digest of Technical Papers of the 2014 IEEE International Solid-State Circuits Conference (ISSCC'14)*, pages 96–97. Feb 2014. [page 31]

[Flynn72] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972. [pages 6 and 27]

[Forsy67] G. E. Forsythe and C. B. Moler. *Computer Solution of Linear Algebraic Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1967. [page 24]

[Gaikw09] A. Gaikwad and I. M. Toke. GPU Based Sparse Grid Technique for Solving Multidimensional Options Pricing PDEs. In *Proceedings of the 2Nd Workshop on High Performance Computational Finance*, WHPCF '09, pages 6:1–6:9. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-716-5. URL `http://doi.acm.org/10.1145/1645413.1645419`. [page 9]

[Gaikw10] A. Gaikwad and I. M. Toke. Parallel Iterative Linear Solvers on GPU: A Financial Engineering Case. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'10)*, pages 607–614. Feb 2010. [page 9]

[Gent11] P. R. Gent, G. Danabasoglu, L. J. Donner, M. M. Holland, E. C. Hunke, S. R. Jayne, D. M. Lawrence, R. B. Neale, P. J. Rasch, M. Vertenstein, P. H. Worley, Z.-L. Yang, and M. Zhang. The Community Climate System Model

Version 4. *Journal of Climate*, 24(19):4973–4991, 2014/05/18 2011. URL http://dx.doi.org/10.1175/2011JCLI4083.1. [page 4]

[Gizop04] D. Gizopoulos, A. Paschalis, and Y. Zorian. *Embedded Processor-Based Self-Test*, volume 28 of *Frontiers in Electronic Testing*. Springer Science & Business Media, 2004. ISBN 978-1-4419-5252-3. [page 48]

[Goess08] M. Goessel, V. Ocheretny, E. Sogomonyan, and D. Marienfeld. *New Methods of Concurrent Checking*, volume 42. Springer Science+Business Media B.V., 2008. ISBN 978-1-4020-8419-5. [page 39]

[Goldb91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991. URL http://doi.acm.org/10.1145/103162.103163. [page 25]

[Golub12] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 4th edition, 2012. ISBN 978-1-4214-0794-4. [pages 73, 104, and 105]

[Goodn05] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05. ACM, New York, NY, USA, 2005. URL http://doi.acm.org/10.1145/1198555.1198784. [page 8]

[Govin04] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast Computation of Database Operations Using Graphics Processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 215–226. ACM, New York, NY, USA, 2004. ISBN 1-58113-859-8. URL http://doi.acm.org/10.1145/1007568.1007594. [page 8]

[Govin06] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336. ACM, New York, NY, USA, 2006. ISBN 1-59593-434-0. URL http://doi.acm.org/10.1145/1142473.1142511. [page 8]

[Gulat08] K. Gulati and S. P. Khatri. Towards Acceleration of Fault Simulation Using Graphics Processing Units. In *Proceedings of the 45th Annual Design Automation Conference (DAC'08)*, pages 822–827. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-115-6. URL http://doi.acm.org/10.1145/1391469.1391679. [page 8]

[Gunne01] J. A. Gunnels, D. S. Katz, E. S. Quintana-Ortí, and R. A. Van de Gejin. Fault-Tolerant High-Performance Matrix Multiplication: Theory and Practice. In *International Conference on Dependable Systems and Networks (DSN'01)*, pages 47–56. 2001. [page 74]

[Gurum06] S. Gurumurthy, S. Vasudevan, and J. Abraham. Automatic Generation of Instruction Sequences Targeting Hard-to-Detect Structural Faults in a Processor. In *Proceedings of the IEEE International Test Conference (ITC'06)*, pages 1–9. Oct 2006. [page 48]

[Halde14] S. Halder. *Integration von algorithmenbasierter Fehlertoleranz in grundlegende Operationen der linearen Algebra auf GPGPUs.* Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, April 2014. [pages 102, 107, 133, and 145]

[Hamad09] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 TFlops Hierarchical N-body Simulations on GPUs with Applications in Both Astrophysics and Turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 62:1–62:12. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-744-8. URL http://doi.acm.org/10.1145/1654059.1654123. [page 8]

[Hammi50] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160, 1950. [page 184]

[Hammi70] R. W. Hamming. On the Distribution of Numbers. *Bell System Technical Journal*, 49(8):1609–1625, 1970. [pages 86, 87, and 89]

[Hargr06] P. H. Hargrove and J. C. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006. [page 46]

[He08] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-282-5. URL http://doi.acm.org/10.1145/1454115.1454152. [page 9]

[Heath97] M. T. Heath. *Scientific Computing: An Introductory Survey.* The McGraw-Hill Companies, New York, 1997. [page 4]

[Henne12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, Elsevier Inc., 5th edition, 2012. ISBN 9780123838728. [page 6]

[Higha02] N. J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics (SIAM), 2nd edition, 2002. ISBN 0-89871-521-0. [page 73]

[Hill95a] T. P. Hill. A Statistical Derivation of the Significant-Digit Law. *Statistical Science*, 10(4):pp. 354–363, 1995. URL http://www.jstor.org/stable/2246134. [pages 86 and 88]

[Hill95b] T. P. Hill. Base-invariance Implies Benford's Law. *Proceedings of the American Mathematical Society*, 123(3):887–895, 1995. [page 88]

[Hill98] T. P. Hill. The First Digit Phenomenon. *American Scientist*, 86(4):358–363, 1998. [page 87]

[Hollo05] S. Holloway. Underground sequestration of carbon dioxideâ€"a viable greenhouse gas mitigation option. *Energy*, 30:2318 – 2333, 2005. URL http://www.sciencedirect.com/science/article/pii/S0360544204003883, international Symposium on {CO2} Fixation and Efficient Utilization of Energy (CandE 2002) and the International World Energy System Conference (WESC-2002). [pages 8 and 9]

[Holst12] S. Holst, E. Schneider, and H.-J. Wunderlich. Scan Test Power Simulation on GPGPUs. In *Proceedings of the IEEE 21st Asian Test Symposium (ATS'12)*, pages 155–160. Nov 2012. [page 8]

[House58] A. S. Householder. Unitary Triangularization of a Nonsymmetric Matrix. *Journal of the ACM*, 5(4):339–342, October 1958. URL http://doi.acm.org/10.1145/320941.320947. [page 104]

[Huang82] K.-H. Huang and J. A. Abraham. Low Cost Schemes for Fault Tolerance in Matrix Operations with Processor Arrays. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS'82)*, pages 98–105. June 1982. [pages 12, 55, and 56]

[Huang84] K.-H. Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984. [pages 12, 21, 55, 56, 58, 59, 60, 62, 65, and 68]

[Huard06] V. Huard, M. Denais, and C. Parthasarathy. NBTI degradation: From physical mechanisms to modelling. *Microelectronics Reliability*, 46(1):1 – 23, 2006. URL http://www.sciencedirect.com/science/article/pii/S0026271405000351. [page 11]

[Hughe13] J. F. Hughes, A. Van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley. *Computer graphics: principles and practice*. Pearson Education, 3 edition, 2013. [page 5]

[Hwu11] W.-m. W. Hwu. *GPU Computing Gems (Emerald Edition)*. Elsevier, 2011. [page 7]

[IEEE 08] IEEE Computer Society. IEEE 754-2008 Standard for Floating-Point Arithmetic, Aug 2008. [pages 23, 25, 30, 189, 191, 192, and 194]

[Intel13] Intel Corporation. *Desktop 4th Generation Intel® CoreTM Processor Family, Desktop Intel® Pentium® Processor Family, and Desktop Intel® Celeron® Processor Family - Datasheet Volume 2 of 2*, December 2013. [page 6]

[ITRS13] ITRS. International Technology Roadmap for Semiconductors 2013 Edition, 2013. URL http://www.itrs.net/. [pages 10 and 11]

[Jeon12] H. Jeon and M. Annavaram. Warped-DMR: Light-weight Error Detection for GPGPU. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 37–47. IEEE Computer Society, Washington, DC, USA, 2012. ISBN 978-0-7695-4924-8. URL http://dx.doi.org/10.1109/MICRO.2012.13. [page 51]

[Jones95] R. G. Jones, J. M. Murphy, and M. Noguer. Simulation of climate change over europe using a nested regional-climate model. I: Assessment of control climate, including sensitivity to location of lateral boundaries. *Quarterly Journal of the Royal Meteorological Society*, 121(526):1413–1449, 1995. URL http://dx.doi.org/10.1002/qj.49712152610. [page 9]

[Jou85] J.-Y. Jou and J. A. Abraham. Fault-Tolerant FFT Networks. In *Proc. International Symposium on Fault-Tolerant Computing*, pages 338–343. 1985. [page 58]

[Jou86] J.-Y. Jou and J. A. Abraham. Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures. *Proceedings of the IEEE*, 74(5):732–741, 1986. [pages 58, 62, and 66]

[Kalio14] M. Kaliorakis, M. Psarakis, N. Foutris, and D. Gizopoulos. Accelerated Online Error Detection in Many-core Microprocessor Architectures. In *IEEE 32nd VLSI Test Symposium (VTS'14)*, pages 1–6. April 2014. [page 48]

[Karma84] N. Karmarkar. A New Polynomial-time Algorithm for Linear Programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 302–311. ACM, New York, NY, USA, 1984. ISBN 0-89791-133-4. URL http://doi.acm.org/10.1145/800057.808695. [page 110]

[Khach80] L. G. Khachiyan. Polynomial Algorithms in Linear Programming. *USSR Computational Mathematics and Mathematical Physics*, 20(1):53 – 72, 1980. URL http://www.sciencedirect.com/science/article/pii/0041555380900610. [page 110]

[Khron14] Khronos OpenCL Working Group. The OpenCL Specification Version 2.0, Rev. 22. API Reference Manual, March 2014. [page 34]

[Kiss12] I. Kiss, S. Gyimóthy, Z. Badics, and J. Pávó. Parallel Realization of the Element-by-Element FEM Technique by CUDA. *IEEE Transactions on Magnetics*, 48(2):507–510, Feb 2012. [page 8]

[Klee70] V. Klee and G. J. Minty. How good is the simplex algorithm. Technical report, DTIC Document, 1970. [page 110]

[Knuth81] D. E. Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 1981. [page 87]

[Kocht10] M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin. Efficient Fault Simulation on Many-Core Processors. In *Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC'10)*, pages 380–385. June 2010. [page 8]

[Kolb05] A. Kolb and N. Cuntz. Dynamic Particle Coupling for GPU-based Fluid Simulation. In *Proceedings of the Symposium on Simulation Technique*, pages 722–727. 2005. [page 8]

[Koren07] I. Koren and C. M. Krishna. *Fault Tolerant Systems*. Morgan Kaufmann, 2007. ISBN 978-0-12-088568-8. [pages 39, 44, and 45]

[Krani06] N. Kranitis, A. Merentitis, N. Laoutaris, G. Theodorou, A. Paschalis, D. Gizopoulos, and C. Halatsis. Optimal Periodic Testing of Intermittent Faults in Embedded Pipelined Processor Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*, pages 65–70. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2006. ISBN 3-9810801-0-6. URL `http://dl.acm.org/citation.cfm?id=1131481.1131500`. [page 48]

[Krull94] F. N. Krull. The Origin of Computer Graphics within General Motors. *IEEE Annals of the History of Computing*, 16(3):40–56, 1994. [page 5]

[Kuhn11] K. J. Kuhn, M. D. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, S. T. Ma, A. Maheshwari, and S. Mudanai. Process Technology Variation. *IEEE Transactions on Electron Devices*, 58(8):2197–2208, Aug 2011. [page 11]

[Lala01] P. K. Lala. *Self-Checking and Fault-Tolerant Digital Design*. Morgan Kaufmann, 2001. ISBN 0-12-434370-8. [pages 39 and 44]

[Langd09] W. B. Langdon. A Fast High Quality Pseudo Random Number Generator for nVidia CUDA. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, GECCO '09, pages 2511–2514. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-505-5. URL `http://doi.acm.org/10.1145/1570256.1570353`. [page 9]

[Laoso11] S. Laosooksathit, N. Naksinehaboon, and C. Leangsuksan. Two-Level Checkpoint/Restart Modeling for GPGPU. In *9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA'11)*, pages 276–283. IEEE, 2011. [page 46]

[Leemi00] L. M. Leemis, B. W. Schmeiser, and D. L. Evans. Survival Distributions Satisfying Benford's Law. *The American Statistician*, 54(4):236–241, 2000. URL http://www.tandfonline.com/doi/abs/10.1080/00031305.2000.10474554. [page 87]

[Li13] D. Li, Z. Chen, P. Wu, and J. S. Vetter. Rethinking Algorithm-based Fault Tolerance with a Cooperative Software-hardware Approach. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 44:1–44:12. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2378-9. URL http://doi.acm.org/10.1145/2503210.2503226. [page 45]

[Lindh08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE micro*, 28(2):39–55, 2008. [page 5]

[Liu04] Y. Liu, X. Liu, and E. Wu. Real-time 3D Fluid Simulation on GPU with Complex Obstacles. In *Proceedings of the 12th Pacific Conference on Computer Graphics and Applications (PG'04)*, pages 247–256. Oct 2004. [page 8]

[LPSOL14] LPSOLVE Project. The MPS File Format, September 2014. URL http://lpsolve.sourceforge.net/5.5/mps-format.htm. [page 113]

[Luenb08] D. G. Luenberger and Y. Ye. *Linear and Nonlinear Programming*, volume 116. Springer, 3rd edition, 2008. [page 111]

[Luk86] F. T. Luk. Algorithm-Based Fault Tolerance for Parallel Matrix Equation Solvers, 1986. URL http://dx.doi.org/10.1117/12.949703. [pages 58 and 67]

[Luk88] F. T. Luk and H. Park. An Analysis Of Algorithm-Based Fault Tolerance Techniques, 1988. URL http://dx.doi.org/10.1117/12.936896. [page 58]

[Ma09] W. Ma and G. Agrawal. A Translation System for Enabling Data Mining Applications on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 400–409. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-498-0. URL `http://doi.acm.org/10.1145/1542275.1542331`. [page 8]

[Malek85] M. Malek and Y.-H. Choi. A Fault-Tolerant FFT Processor. In *Proc. 15th IEEE International Symposium on Fault-Tolerant Computing Systems (FTCS)*, volume 15, pages 266–271. 1985. [page 58]

[Manav08] S. A. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008. URL `http://www.biomedcentral.com/1471-2105/9/S2/S10`. [page 8]

[Maruy10] N. Maruyama, A. Nukada, and S. Matsuoka. A High-performance Fault-tolerant Software Framework for Memory on Commodity GPUs. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS'10)*, pages 1–12. April 2010. [pages 43 and 53]

[Mieli12] J. Mielikainen, B. Huang, H.-L. A. Huang, and M. D. Goldberg. GPU Acceleration of the Updated Goddard Shortwave Radiation Scheme in the Weather Research and Forecasting (WRF) Model. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(2):555–562, April 2012. [page 8]

[Mille08] S. J. Miller and M. J. Nigrini. Order Statistics and Benford's Law. *International Journal of Mathematics and Mathematical Sciences*, 2008:1–19, 2008. [page 87]

[Moore65] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114ff, April 1965. [pages 6 and 10]

[Morio11] M. Moriondo, C. Giannakopoulos, and M. Bindi. Climate change impact assessment: the role of climate extremes in crop yield simulation. *Climatic Change*, 104(3-4):679–701, 2011. [page 9]

[Mukhe08] S. Mukherjee. *Architecture Design for Soft Errors.* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2008. ISBN 9780123695291. [page 40]

[Mulle10] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, et al. *Handbook of Floating-Point Arithmetic.* Birkhäuser, 1 edition, 2010. [pages 23 and 25]

[Nair88] V. S. S. Nair and J. A. Abraham. General Linear Codes for Fault-tolerant Matrix Operations on Processor Arrays. In *Digest of Papers of the Eighteenth International Symposium on Fault-Tolerant Computing (FTCS-18)*, pages 180–185. June 1988. [pages 67 and 68]

[Nanju10] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC'10)*, pages 149–154. Jan 2010. [page 8]

[Nassi10] S. R. Nassif, N. Mehta, and Y. Cao. A Resilience Roadmap. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1011–1016. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2010. ISBN 978-3-9810801-6-2. URL `http://dl.acm.org/citation.cfm?id=1870926.1871176`. [page 11]

[Neste94] Y. Nesterov, A. Nemirovskii, and Y. Ye. *Interior-Point Polynomial Algorithms in Convex Programming*, volume 13. SIAM, 1994. [page 110]

[Neuba07] A. Neubauer, J. Freudenberger, and V. Kühn. *Coding Theory - Algorithms, Architectures and Applications.* John Wiley & Sons Inc., 1st edition, 2007. ISBN 978-0-470-02861-2. [page 183]

[Newco81] S. Newcomb. Note on the Frequency of Use of the Different Digits in Natural Numbers. *American Journal of Mathematics*, 4(1):pp. 39–40, 1881. URL `http://www.jstor.org/stable/2369148`. [page 86]

[Nicko10] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE micro*, 30(2):56–69, 2010. [pages 5 and 6]

[Nicol11] M. Nicolaidis. *Soft Errors in Modern Electronic Systems.* Frontiers in Electronic Testing, Volume 41. Springer, 2011. [page 12]

[Nigri97] M. J. Nigrini and L. J. Mittermaier. The Use of Benford's Law as an Aid in Analytical Procedures. *Auditing*, 16(2):52–67, 1997. [page 87]

[Nigri12] M. Nigrini. *Benford's Law: Applications for Forensic Accounting, Auditing, and Fraud Detection*, volume 586. John Wiley & Sons, Inc., 2012. [page 87]

[Nithi10] S. K. Nithin, G. Shanmugam, and S. Chandrasekar. Dynamic Voltage (IR) Drop Analysis and Design Closure: Issues and Challenges. In *Proceedings of the 11th International Symposium on Quality Electronic Design (ISQED'10)*, pages 611–617. March 2010. [page 11]

[NVIDI12] NVIDIA Corporation. NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110. Whitepaper, 2012. [pages 27, 28, 29, 31, 33, 35, 37, and 44]

[NVIDI13] NVIDIA Corporation. *TESLA K40 GPU ACTIVE ACCELERATOR - Board Specification*, November 2013. [pages 6 and 44]

[NVIDI14a] NVIDIA Corporation. CUDA C Programming Guide (Version 6.0). Design Guide, February 2014. [pages 34, 37, and 46]

[NVIDI14b] NVIDIA Corporation. CUDA Runtime API (Version 6.0). API Reference Manual, February 2014. [page 37]

[NVIDI14c] NVIDIA Corporation. Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. Whitepaper, February 2014. [page 30]

[NVIDI15] NVIDIA Corporation. CUDA Software Development Kit, https://developer.nvidia.com/cuda-downloads, February 2015. [pages 41, 51, and 53]

[Oberk02] W. L. Oberkampf, S. M. DeLand, B. M. Rutherford, K. V. Diegert, and K. F. Alvin. Error and uncertainty in modeling and simulation. *Reliability Engineering & System Safety*, 75(3):333 – 357, 2002. URL `http://www.sciencedirect.com/science/article/pii/S095183200100120X`. [page 9]

[Oberk10] W. L. Oberkampf and C. J. Roy. *Verification and Validation in Scientific Computing*, volume 5. Cambridge University Press Cambridge, 2010. [pages 4 and 9]

[Obori11] F. Oboril, M. B. Tahoori, V. Heuveline, D. Lukarski, and J.-P. Weiss. Numerical Defect Correction as an Algorithm-Based Fault Tolerance Technique for Iterative Solvers. In *17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'11)*, pages 144–153. 2011. [page 58]

[Ogawa03] E. T. Ogawa, J. Kim, G. S. Haase, H. C. Mogul, and J. W. McPherson. Leakage, Breakdown, and TDDB Characteristics of Porous Low-k Silica-Based Interconnect Dielectrics. In *Proceedings of the 41st Annual IEEE International Reliability Physics Symposium*, pages 166–172. March 2003. [page 11]

[Okamo13] T. Okamoto, H. Takenaka, T. Nakamura, and T. Aoki. Accelerating Large-Scale Simulation of Seismic Wave Propagation by Multi-GPUs and Three-Dimensional Domain Decomposition. In D. A. Yuen, L. Wang, X. Chi, L. Johnsson, W. Ge, and Y. Shi, editors, *GPU Solutions to Multi-scale Problems in Science and Engineering*, Lecture Notes in Earth System Sciences, pages 375–389. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-16404-0. [page 8]

[Olive14] D. Oliveira, P. Rech, L. Pilla, P. Navaux, and L. Carro. GPGPUs ECC Efficiency and Efficacy. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'14)*, pages 209–215. Oct 2014. [pages 42 and 45]

[Owa04] S. Owa and H. Nagasaka. Advantage and feasibility of immersion lithography. *Journal of Micro/Nanolithography, MEMS, and MOEMS*, 3(1):97–103, 2004. URL http://dx.doi.org/10.1117/1.1637593. [page 11]

[Pages12] G. Pagès and B. Wilbertz. GPGPUs in computational finance: massive parallel computing for American style options. *Concurrency and Computation: Practice and Experience*, 24(8):837–848, 2012. URL http://dx.doi.org/10.1002/cpe.1774. [page 9]

[Papad11] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis. A new era in scientific computing: Domain decomposition methods in hybrid CPU-GPU architectures. *Computer Methods in Applied Mechanics and Engineering*, 200(13â€“16):1490 – 1508, 2011. URL http://www.sciencedirect.com/science/article/pii/S0045782511000235. [page 8]

[Pasch01] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian. Deterministic Software-based Self-testing of Embedded Processor Cores. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'01)*, pages 92–96. IEEE Press, Piscataway, NJ, USA, 2001. ISBN 0-7695-0993-2. URL `http://dl.acm.org/citation.cfm?id=367072.367101`. [page 49]

[Pasch05] A. Paschalis and D. Gizopoulos. Effective Software-based Self-test Strategies for On-line Periodic Testing of Embedded Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):88–99, Jan 2005. [page 48]

[Phill11] C. L. Phillips, J. A. Anderson, and S. C. Glotzer. Pseudo-random number generation for Brownian Dynamics and Dissipative Particle Dynamics simulations on GPU devices. *Journal of Computational Physics*, 230(19):7191 – 7201, 2011. URL `http://www.sciencedirect.com/science/article/pii/S0021999111003329`. [page 9]

[Pinkh61] R. S. Pinkham. On the Distribution of First Significant Digits. *The Annals of Mathematical Statistics*, 32(4):1223–1230, 1961. URL `http://www.jstor.org/stable/2237922`. [page 88]

[Press07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007. ISBN 978-0-521-88068-8. [page 4]

[Psara10] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Reorda. Microprocessor Software-Based Self-Testing. *IEEE Design Test of Computers*, 27(3):4–19, May 2010. [page 48]

[Rao89] T. R. N. Rao and E. Fujiwara. Error Control Coding for Computer Systems. *Prentice-Hall Inc.*, 1989. [page 183]

[Rech13] P. Rech, L. Pilla, F. Silvestri, P. Navaux, and L. Carro. Neutron Sensitivity and Software Hardening Strategies for Matrix Multiplication and FFT on Graphics Processing Units. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, pages 13–20. ACM, 2013. [page 42]

[Reddy90] A. L. N. Reddy and P. Banerjee. Algorithm-Based Fault Detection for Signal Processing Applications. *IEEE Transactions on Computers*, 39(10):1304–1308, 1990. [page 58]

[Rexfo92] J. Rexford and N. K. Jha. Algorithm-Based Fault Tolerance for Floating-Point Operations in Massively Parallel Systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '92)*, volume 2, pages 649–652 vol.2. 1992. [pages 68 and 69]

[Roy C93] A. Roy-Chowdhury and P. Banerjee. Tolerance Determination for Algorithm-Based Checks Using Simplified Error Analysis Techniques. In *Digest of Papers of the Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 290–298. 1993. [pages 58, 75, 76, 121, 130, and 144]

[Roy11] C. J. Roy and W. L. Oberkampf. A comprehensive framework for verification, validation, and uncertainty quantification in scientific computing. *Computer Methods in Applied Mechanics and Engineering*, 200(25-28):2131 – 2144, 2011. URL http://www.sciencedirect.com/science/article/pii/S0045782511001290. [page 9]

[Saben14a] D. Sabena, M. S. Reorda, L. Sterpone, P. Rech, and L. Carro. Evaluating the Radiation Sensitivity of {GPGPU} Caches: New Algorithms and Experimental Results. *Microelectronics Reliability*, 54(11):2621 – 2628, 2014. URL http://www.sciencedirect.com/science/article/pii/S0026271414001516. [page 42]

[Saben14b] D. Sabena, L. Sterpone, L. Carro, and P. Rech. Reliability Evaluation of Embedded GPGPUs for Safety Critical Applications. *IEEE Transactions on Nuclear Science*, 61(6):3123–3129, Dec 2014. [page 42]

[Sanch05] E. Sánchez, M. Reorda, and G. Squillero. On The Transformation of Manufacturing Test Sets into On-line Test Sets for Microprocessors. In *Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*, pages 494–502. Oct 2005. [page 48]

[Satis09] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proceedings of the IEEE International*

Symposium on Parallel Distributed Processing (IPDPS'09), pages 1–10. May 2009. [page 8]

[Schat81] P. Schatte. On Random Variables with Logarithmic Mantissa Distribution Relative to Several Bases. *Elektronische Informationsverarbeitung und Kybernetik*, 17(4-6):293–295, 1981. [page 87]

[Schur08] K. Schürger. Extensions of Black-Scholes Processes and Benford's Law. *Stochastic Processes and their Applications*, 118(7):1219 – 1243, 2008. [page 87]

[Segur04] J. Segura and C. F. Hawkins. *CMOS Electronics: How it works, how it fails.* Wiley-IEEE Press, 2004. ISBN 978-0-471-47669-6. [page 11]

[Sha94] C.-C. Sha and R. W. Leavene. An Algorithm-Base Fault Tolerance (More Than One Error) Using Concurrent Error Detection for FFT Processors. In *Proceedings of the Fourth Great Lakes Symposium on Design Automation of High Performance VLSI Systems (GLSV '94)*, pages 56–61. 1994. [page 58]

[Shant12] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 69–78. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1316-2. URL `http://doi.acm.org/10.1145/2304576.2304588`. [page 58]

[Sheaf07] J. W. Sheaffer, D. P. Luebke, and K. Skadron. A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors. In *Graphics Hardware*, volume 2007, pages 55–64. 2007. [page 50]

[Shen98] J. Shen and J. Abraham. Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation. In *Proceedings of the InternationalTest Conference (ITC'98)*, pages 990–999. Oct 1998. [page 48]

[Shimo10] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA

Production Code. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pages 1–11. Nov 2010. [page 8]

[Shu04] L. Shu and D. J. Costello Jr. *Error Control Coding.* (Pearson Prentice Hall) Pearson Education Inc., 2nd edition, 2004. ISBN 0-13-017973-6. [pages 65 and 183]

[Singh06] V. Singh, I. Inoue, K. Saluja, and H. Fujiwara. Instruction-Based Self-Testing of Delay Faults in Pipelined Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(11):1203–1215, Nov 2006. [page 48]

[Sinko13] W. Sinko, S. Lindert, and J. A. McCammon. Accounting for Receptor Flexibility and Enhanced Sampling Methods in Computer-Aided Drug Design. *Chemical Biology & Drug Design*, 81(1):41–49, 2013. URL `http://dx.doi.org/10.1111/cbdd.12051`. [page 8]

[Sinto08] E. Sintorn and U. Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381 – 1388, 2008. URL `http://www.sciencedirect.com/science/article/pii/S0743731508001196`, general-Purpose Processing using Graphics Processing Units. [page 8]

[Soman10] J. Soman, M. K. Kumar, K. Kothapalli, and P. J. Narayanan. Efficient Discrete Range Searching primitives on the GPU with applications. In *Proceedings of the International Conference on High Performance Computing (HiPC'10)*, pages 1–10. Dec 2010. [page 8]

[Sony 10] Sony Corporation. *SONY Semiconductor Quality and Reliability Handbook*, 2010. [page 12]

[Spamp09a] D. G. Spampinato. Linear optimization with cuda. Technical Report TDT4590, Norwegian University of Science and Technology, January 2009. [page 113]

[Spamp09b] D. G. Spampinato and A. C. Elster. Linear Optimization on Modern GPUs. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS'09)*, pages 1–8. May 2009. [pages 113 and 199]

[Stein05] D. Steinkrau, P. Y. Simard, and I. Buck. Using GPUs for Machine Learning Algorithms. *Proceedings of the 12th International Conference on Document Analysis and Recognition*, 0:1115–1119, 2005. [page 8]

[Stine98] B. E. Stine, D. O. Ouma, R. R. Divecha, D. S. Boning, J. E. Chung, D. L. Hetherington, C. R. Harwood, O. S. Nakagawa, and S.-Y. Oh. Rapid Characterization and Modeling of Pattern-Dependent Variation in Chemical-Mechanical Polishing. *IEEE Transactions on Semiconductor Manufacturing*, 11(1):129–140, Feb 1998. [page 11]

[Stone09] J. E. Stone, J. Saam, D. J. Hardy, K. L. Vandivort, W.-m. W. Hwu, and K. Schulten. High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 9–18. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-517-8. URL `http://doi.acm.org/10.1145/1513895.1513897`. [page 8]

[Stone10] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling*, 29(2):116 – 125, 2010. URL `http://www.sciencedirect.com/science/article/pii/S1093326310000914`. [page 8]

[Strat12] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing*, 2012. [pages 41, 51, and 52]

[Stuar11] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU Clusters. In *Proceedings of the IEEE International Parallel Distributed Processing Symposium (IPDPS'11)*, pages 1068–1079. May 2011. [page 9]

[Surko10] V. Surkov. Parallel option pricing with Fourier space time-stepping method on graphics processing units. *Parallel Computing*, 36(7):372 – 380, 2010. URL `http://www.sciencedirect.com/science/article/pii/S0167819110000293`, parallel and Distributed Computing in Finance. [page 9]

[Sussm06] M. Sussman, W. Crutchfield, and M. Papakipos. Pseudorandom Number Generation on the GPU. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '06, pages 87–94. ACM, New York, NY, USA, 2006. ISBN 3-905673-37-1. URL http://doi.acm.org/10.1145/1283900.1283914. [page 9]

[Suthe64] I. E. Sutherland. Sketch Pad a Man-machine Graphical Communication System. In *Proceedings of the SHARE Design Automation Workshop*, DAC '64, pages 6.329–6.346. ACM, New York, NY, USA, 1964. URL http://doi.acm.org/10.1145/800265.810742. [page 5]

[Takiz09] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *Proceedings of the 2009 IEEE International Conference on Parallel and Distributed Computing, Applications and Technologies,*, pages 408–413. IEEE, 2009. [page 46]

[Tan11] J. Tan, N. Goswami, T. Li, and X. Fu. Analyzing Soft-Error Vulnerability on GPGPU Microarchitecture. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'11)*, pages 226–235. Nov 2011. [page 40]

[Tang97] X. Tang, V. K. De, and J. D. Meindl. Intrinsic MOSFET Parameter Fluctuations Due to Random Dopant Placement. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(4):369–376, Dec 1997. [page 11]

[The G14] The GNU Project. The GNU Multiple Precision Arithmetic Library, December 2014. URL https://gmplib.org. [page 130]

[The N14a] The Netlib. Netlib BLAS Basic Linear Algebra Subprogram, October 2014. URL http://www.netlib.org/blas. [page 5]

[The N14b] The Netlib. Netlib LAPACK Linear Algebra PACKage, October 2014. URL http://www.netlib.org/linpack. [page 5]

[The N14c] The Netlib. Netlib Linear Programming Test Problems, September 2014. URL http://www.netlib.org/lp/data/. [page 113]

[The N14d] The Netlib. Netlib LINPACK, October 2014. URL http://www.netlib.org/linpack. [page 5]

[Tolke08] J. Tölke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *International Journal of Computational Fluid Dynamics*, 22(7):443–456, 2008. URL http://dx.doi.org/10.1080/10618560802238275. [page 8]

[Tselo13] S. Tselonis, V. Dimitsas, and D. Gizopoulos. The Functional and Performance Tolerance of GPUs to Permanent Faults in Registers. In *IEEE 19th International On-Line Testing Symposium (IOLTS'13)*, pages 236–239. July 2013. [page 41]

[Turmo00a] M. Turmon and R. Granat. Algorithm-Based Fault Tolerance for Spaceborne Computing: Basis and Implementations. In *Proceedings of the IEEE Aerospace Conference*, volume 4, pages 411–420 vol.4. 2000. [pages 73 and 123]

[Turmo00b] M. Turmon, R. Granat, and D. S. Katz. Software-Implemented Fault Detection for High-Performance Space Applications. In *Proceedings International Conference on Dependable Systems and Networks (DSN'00)*, pages 107–116. 2000. [pages 58 and 73]

[Turmo03] M. Turmon, R. Granat, D. S. Katz, and J. Z. Lou. Tests and Tolerances for High-Performance Software-Implemehted Fault Detection. *IEEE Transactions on Computers*, 52(5):579–591, 2003. [pages 73 and 75]

[Turne82] P. R. Turner. The Distribution of Leading Significant Digits. *IMA Journal of Numerical Analysis*, 2:407–412, 1982. [page 89]

[Turne84] P. R. Turner. Further Revelations on L.S.D. *IMA Journal of Numerical Analysis*, 4(2):225–231, 1984. URL http://imajna.oxfordjournals.org/content/4/2/225.abstract. [page 89]

[Ufimt08] I. S. Ufimtsev and T. J. Martínez. Graphical Processing Units for Quantum Chemistry. *Computing in Science & Engineering*, 10(6):26–34, 2008. [page 8]

[Vouzi11] P. D. Vouzis and N. V. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011. URL http://bioinformatics.oxfordjournals.org/content/27/2/182.abstract. [page 8]

[Walte09] J. P. Walters, V. Balu, S. Kompalli, and V. Chaudhary. Evaluating the use of GPUs in liver image segmentation and HMMER database searches. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS'09)*, pages 1–12. May 2009. [page 8]

[Wang94] S.-J. Wang and N. K. Jha. Algorithm-Based Fault Tolerance for FFT Networks. *IEEE Transactions on Computers*, 43(7):849–854, 1994. [page 58]

[Wen06] C.-P. Wen, L.-C. Wang, and K.-T. Cheng. Simulation-Based Functional Test Generation for Embedded Processors. *IEEE Transactions on Computers*, 55(11):1335–1343, Nov 2006. [page 48]

[Winsb10] E. Winsberg. *Science in the Age of Computer Simulation.* University of Chicago Press, 2010. ISBN 9780226902043. [page 4]

[Wunde10] H.-J. Wunderlich and S. Holst. *Generalized Fault Modeling for Logic Diagnosis*, volume 43, pages 133–155. Springer-Verlag Heidelberg, 2010. [page 20]

[Wunde13] H.-J. Wunderlich, C. Braun, and S. Halder. Efficacy and Efficiency of Algorithm-Based Fault-Tolerance on GPUs. In *19th IEEE International On-Line Testing Symposium (IOLTS'13)*, pages 240–243. 2013. [page 40]

[Xenou09] G. Xenoulis, D. Gizopoulos, M. Psarakis, and A. Paschalis. Instruction-Based Online Periodic Self-Testing of Microprocessors with Floating-Point Units. *IEEE Transactions on Dependable and Secure Computing*, 6(2):124–134, April 2009. [pages 48 and 49]

[Xu10] X. Xu, Y. Lin, T. Tang, and Y. Lin. HiAL-Ckpt: A Hierarchical Application-level Checkpointing for CPU-GPU Hybrid Systems. In *5th International Conference on Computer Science and Education (ICCSE'10)*, pages 1895–1899. Aug 2010. [pages 46 and 47]

[Yang07] J. Yang, Y. Wang, and Y. Chen. GPU accelerated molecular dynamics simulation of thermal conductivities. *Journal of Computational Physics*, 221(2):799 – 804, 2007. URL http://www.sciencedirect.com/science/article/pii/S0021999106003172. [page 8]

[Yim11] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer. Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU. In *IEEE International Parallel Distributed Processing Symposium (IPDPS'11)*, pages 287–300. 2011. [pages 43 and 53]

[Zhang94] Q. Zhang and J. Kim. An Efficient Method to Reduce Roundoff Error in Matrix Multiplication with Algorithm-based Fault Tolerance. In *Proceedings of the Sixth Annual IEEE International Conference on Wafer Scale Integration, 1994*, pages 32–39. Jan 1994. [pages 72 and 73]

[Zhou06] J. Zhou and H.-J. Wunderlich. Software-Based Self-Test of Processors Under Power Constraints. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*, pages 430–435. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 2006. ISBN 3-9810801-0-6. URL `http://dl.acm.org/citation.cfm?id=1131481.1131597`. [page 48]

[Zhou11] Y. Zhou, J. Liepe, X. Sheng, M. P. H. Stumpf, and C. Barnes. GPU accelerated biochemical network simulation. *Bioinformatics*, 27(6):874–876, 2011. URL `http://bioinformatics.oxfordjournals.org/content/27/6/874.abstract`. [page 8]

[Zorri03] F. Zorriassatine, C. Wykes, R. Parkin, and N. Gindy. A survey of virtual prototyping techniques for mechanical product development. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 217(4):513–530, 2003. URL `http://pib.sagepub.com/content/217/4/513.abstract`. [page 4]

Appendix A

Abbreviations and Notation

Linear Algebra Notation

Variables, Parameters Variables and parameters are set in italic lowercase letters: a, b, x.

Vectors Vectors are denoted by bold latin lowercase letters: $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{x}$.

Square brackets are used for the full representation of vectors:

$$\text{row vector } \boldsymbol{r} = \begin{bmatrix} r_1, & \cdots, & r_n \end{bmatrix},$$

$$\text{column vector } \boldsymbol{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}.$$

Vector Elements Elements of vectors are set in italic lowercase letters with subscript index i, representing the position of the element within the vector: a_i, b_i, x_i.

Matrices Matrices are denoted by bold latin capital letters: $\boldsymbol{A}$, $\boldsymbol{M}$.
Square brackets are used for the full representation of matrices:

$$\boldsymbol{A}^{n \times m} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix}.$$

Matrix Elements Elements of matrices are set in italic lowercase letters with subscript indices. The first index represents the row i, the second the column j within the matrix: $a_{i,j}$, $m_{i,j}$.

Matrix Dimensions Dimensions of matrices are either stated explicitly or indicated by superscript: $(n \times m)$-matrix $\boldsymbol{A}$ or $\boldsymbol{A}^{n \times m}$.

Matrix Transpose Transposes of vectors and matrices are indicated by a superscript "T": $\boldsymbol{v}^T$, $\boldsymbol{M}^T$.

Conjugate Transpose Conjugate transposes of matrices are denoted by a superscript "$*$": $\boldsymbol{A}^*$, $\boldsymbol{M}^*$.

Inverse Matrix Inverses of matrices are denoted by a superscript "-1": $\boldsymbol{A}^{-1}$, $\boldsymbol{M}^{-1}$.

Identity Matrix The identity matrix is denoted as $\boldsymbol{I}$.

Scalars If not state otherwise, scalars are denoted by lowercase Greek letters: α, β, χ.

Numbers The different types of numbers are indicated by uppercase blackboard letters: The natural numbers $\mathbb{N}$, the integer numbers $\mathbb{Z}$, the rational numbers $\mathbb{Q}$, the real numbers $\mathbb{R}$, and the complex numbers $\mathbb{C}$.

Vector Space The type and dimension of vector spaces are indicated by superscript, for instance the three-dimensional vector space over the real numbers is $\mathbb{R}^3$.

Coding Theory Notation

Alphabets Alphabets of symbols are denoted by Σ with subscript qualifier, for instance the source alphabet Σ_S and the destination alphabet Σ_D.

Sets of Strings Sets of finite strings over an alphabet Σ are denoted by a superscript "$*$": Σ^*. Sets of finite strings over an alphabet Σ with length k are denoted by a superscript "k": Σ^k.

Codes Codes are denoted by uppercase calligraphic letters: $\mathcal{C}$.

Messages Messages or input information blocks are denoted by lowercase bold letters: $\mathbf{m}$. Symbols of message blocks are denoted by lowercase letters with subscript index: m_0.

Codewords Codewords or output code blocks are denoted by lowercase bold letters: $\mathbf{c}$. Symbols of code blocks or codewords are denoted by lowercase letters with subscript index: c_0.

Floating-Point Notation

Floating-Point Operations Operations that are carried out in floating-point arithmetic are denoted by $fl(\cdot)$. $\odot$ is a placeholder and denotes the four basic arithmetic operations: $\odot \in \{+,-,*,/\}$.

Radix, Base The radix or base of a floating-point number system is denoted by $\boldsymbol{\beta}$.

Significand, Mantissa The significand or mantissa of a floating-point number is denoted by $\boldsymbol{m}$.

Exponent The exponent of a floating-point number is denoted by $\boldsymbol{e}$.

Sign The sign of a floating-point number is denoted by $\boldsymbol{s}$ or $(-1)^s$.

Precision The precision of a floating-point number is represented by $\boldsymbol{p}$.

General Rounding The rounding operation is indicated by $\circ$, i.e. the rounded addition of two floating-point numbers a and b is $\circ(a+b)$.

Round Towards Zero The rounding towards zero is denoted by $\boldsymbol{RZ}(\cdot)$.

Round Towards Nearest The rounding towards the nearest floating-point number is represented by $\boldsymbol{RN}(\cdot)$.

Round Towards $-\infty$ The rounding towards $-\infty$ is denoted by $\boldsymbol{RD}(\cdot)$.

Round Towards $+\infty$ The rounding towards $+\infty$ is denoted by $\boldsymbol{RU}(\cdot)$.

Appendix B

Dependability-related Definitions

This appendix summarizes formal definitions related to the field of dependability and complements the formal foundations in Chapter 2.

Definition B.1 (Lifetime of a system) *The **lifetime** T of a system S is a random variable which is defined as the time until the system fails.*

Definition B.2 (Failure probability) *The **failure probability** $F(t)$ is the probability that a system fails before or at a time t.*

$$F(t) = Pr\{T \leq t\}. \tag{B.1}$$

Definition B.3 (Failure probability density function) *The **failure probability density function** $f(t)$ is defined as*

$$f(t) = \frac{dF(t)}{dt}, \tag{B.2}$$

with $f(t) \geq 0 \forall t \geq 0$ and $\int_0^\infty f(t)dt = 1$.

The failure probability and the density function are related by

$$f(t) = \frac{dF(t)}{dt} \text{ and } F(t) = \int_0^\infty f(s)ds. \tag{B.3}$$

Definition B.4 (Reliability) *The **reliability** of a system is defined as the probability that the system provides its correct service (functionality) during a specified period of time.*

$$R(t) = Pr\{T > t\} = 1 - F(t). \tag{B.4}$$

Definition B.5 (Mean time to failure) *The **meant time to failure** (**MTTF**) is defined as the average time a system provides its correct service until a failure occurs. The meant time to failure can be expressed as the expectation value of the lifetime.*

$$MTTF = \mathtt{EV}(T) = \int_0^\infty t \cdot f(t)dt = \int_0^\infty R(t)dt. \tag{B.5}$$

Definition B.6 (Failure rate) *The **failure rate** λ is the number of failing systems per given time unit compared to the number of surviving systems.*

$$\lambda(t) = \frac{f(t) \cdot N}{(1 - F(t)) \cdot N} \tag{B.6}$$

where N is the number of systems.

Definition B.7 (Availability) *The **availability** $A(t)$ of a system is the fraction of time the system provides its correct services (uptime) during the time interval $[0, t]$*

$$A(t) = \frac{\mathtt{EV}(uptime)}{\mathtt{EV}(uptime) + \mathtt{EV}(downtime)}. \tag{B.7}$$

Appendix C

Definitions from Coding Theory

This appendix summarizes fundamental definitions from the field of coding theory which complement the introduction to linear block codes given in Chapter 2. The reader can find comprehensive introductions to coding theory in [Shu04] and [Rao89], as well as [Neuba07], where most of the following definitions were taken from.

Definition C.1 (Alphabet of symbols) *An **alphabet** Σ is a non-empty set of symbols. A finite string is a finite sequence of symbols over the alphabet Σ. The set of all finite strings over the alphabet Σ is denoted with Σ^*.*

*If the cardinality of Σ is q, $|\Sigma| = q$, the alphabet is called a **q-nary alphabet**.*

*A simple example for an alphabet is the non-empty set $\{0, 1\}$, which is called the **binary alphabet**.*

Definition C.2 (Code) *A **code** is a mathematical rule that describes the transformation and conversion of information from one representation into another.*

Let Σ_S be an alphabet called source alphabet *and Σ_D be an alphabet called* destination alphabet. *A code $\mathcal{C} : \Sigma_S \to \Sigma_D^*$ is a total function that maps each symbol of Σ_S to a sequence of symbols over Σ_D.*

Code Parameters

A small set of parameters is sufficient to characterize block codes with respect to their efficiency and their error detection and error correction capabilities. Theses parameters are the *code rate*, the *Hamming weight*, and the *Hamming distance* [Hammi50].

Definition C.3 (Total number of codewords) *Given an (n,k)-block code $\mathcal{C}$ over the alphabet Σ with $|\Sigma| = q$. The total number M of message blocks and codewords is given as*

$$M = q^k. \tag{C.1}$$

The efficiency of an (n,k)-block code can be expressed in terms of its impact on the transmission of encoded versus non-encoded information over a channel. The corresponding measure is called *code rate*:

Definition C.4 (Code rate) *Given an (n,k)-block code $\mathcal{C}$ over the alphabet Σ with $|\Sigma| = q$ and $M = q^k$. The* ***code rate*** *is defined as*

$$R = \frac{\log_q(M)}{n} = \frac{k}{n}. \tag{C.2}$$

Definition C.5 (Hamming weight) *For a given code $\mathcal{C}$ with codewords $\mathbf{c}_i$, the* ***Hamming weight*** *of a codeword $\mathbf{c}$ is defined as the number of non-zero components in the codeword.*

$$weight_{Ham}(\mathbf{c}) = |\{i : c_i \neq 0, \quad 0 \leq i < n\}|. \tag{C.3}$$

Definition C.6 (Hamming distance) *Given a code $\mathcal{C}$ and two codewords $\mathbf{c}, \mathbf{d} \in \mathcal{C}$. The* ***Hamming distance*** *between two codewords is defined as the number of indices* *at which the codewords differ.*

$$dist_{Ham}(\mathbf{c}, \mathbf{d}) = |\{i : c_i \neq d_i, \quad 0 \leq i < n\}|. \tag{C.4}$$

The ***minimum Hamming distance*** *d of the code $\mathcal{C}$ is defined as the minimum distance between any two codewords in the $\mathcal{C}$*

$$d(\mathcal{C}) = min\{dist_{Ham}(c_i, c_j) | c_i \neq c_j, c_i, c_j \in \mathcal{C}\}. \tag{C.5}$$

Linear Block Codes

Definition C.7 (Linear block code) *A q-nary linear (n, k) block code $\mathcal{C}(n, k, d)$ is defined as a vector subspace of the vector space $\mathbb{F}_q^n$.*

The linearity can be utilized to express the encoding and decoding of information using linear block codes in terms of linear algebra matrix operations. Therefore, the terms *generator matrix* and *parity-check matrix* are introduced.

Definition C.8 (Generator matrix) *Given a q-nary linear block code $\mathcal{C}(n, k, d)$ over the finite field $\mathbb{F}_q$. The $(k \times n)$* ***generator matrix*** *G of $\mathcal{C}$ is formed by a set of k linearly independent basis vectors g_i which span the subspace of the code:*

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \ddots & \vdots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} \tag{C.6}$$

An input information or message block m of dimension k is encoded into an n-dimensional code block c through multiplication with the generator matrix:

$$c = m \cdot G. \tag{C.7}$$

The generator matrix G is able to generate all $M = q^k$ codewords of the code $\mathcal{C}$.

Definition C.9 (Systematic encoding) *For each linear block code $\mathcal{C}(n,k,d)$, an* ***equivalent*** *linear block code can be defined by the generator matrix*

$$G = (I_k | A_{k,n-k}). \tag{C.8}$$

Due to the $(k \times k)$ identity matrix and according to C.8, the first k code symbols c_i are identical to the k message symbols m_i. The remaining $n - k$ code symbols within the codeword $c = m \cdot A_{k,n-k}$ represent parity-check symbols that are attached to the information vector m to enable error detection and correction.

An encoding of this type is called an ***systematic encoding****.*

Definition C.10 (Parity-check matrix) *The $(n - k \times k)$* ***parity-check matrix*** *of a linear block code $\mathcal{C}(n,k,d)$ can be defined using its generator matrix $G = (I_k | A_{k,n-k})$:*

$$H = (B_{n-k,k} | I_{n-k}), \tag{C.9}$$

where I_{n-k} is the $(n - k \times k)$ identity matrix and $B_{n-k,k}$ an $(n - k \times k)$ matrix defined by $B_{n-k,k} = -A^T_{k,n-k}$.

For the generator matrix G and the parity-check matrix H of a linear block code $\mathcal{C}(n,k,d)$, the following relationship holds:

$$H \cdot G^T = B_{n-k,k} + A^T_{k,n-k} = 0_{n-k,k}, \tag{C.10}$$

where $0_{n-k,k}$ is a null matrix. The matrices G and H are orthogonal. This relationship allows the derivation of a direct check for the correctness of codewords, the so-called *parity-check condition.*

Given a linear block code $\mathcal{C}(n,k,d)$ with a generator matrix G and a parity-check matrix H. It is

$$H \cdot G^T = H \cdot (g_0^T, g_1^T, \cdots, g_{k-1}^T) = (Hg_0^T, Hg_1^T, \cdots, Hg_{k-1}^T) = (0, 0, \cdots, 0), \tag{C.11}$$

which is equivalent to

$$\forall 0 \leq i \leq k - 1: \quad H \cdot g_i^T = 0. \tag{C.12}$$

Since each codeword $c_i \in \mathcal{C}(n,k,d)$ can be expressed as

$$c = m \cdot G = m_0 \cdot g_0 + m_1 \cdot g_1 + \cdots + m_{k-1} \cdot g_{k-1}, \tag{C.13}$$

where m is the message block, it follows that

$$H \cdot c^T = m_0 \cdot H \cdot g_0 + m_1 \cdot H \cdot g_1 + \cdots + m_{k-1} \cdot H \cdot g_{k-1} = 0. \tag{C.14}$$

Definition C.11 (Parity-check condition) *Given a linear block code $\mathcal{C}(n,k,d)$ with a generator matrix G and a parity-check matrix H. Every code block $c \in \mathcal{C}$ fulfills the condition*

$$H \cdot c^T = 0. \tag{C.15}$$

*This implies that every received block r with $H \cdot r^T \neq 0$ is not a valid code block and yields the **parity-check condition**:*

$$H \cdot r^T = 0 \Leftrightarrow r \in \mathcal{C}(n,k,d). \tag{C.16}$$

Definition C.12 (Syndrome) *Given a linear block code $\mathcal{C}(n,k,d)$ and a parity-check matrix H. Let r be a received data block obtained from*

$$r = c + e, \tag{C.17}$$

*where c is the transmitted code block and e is an error block. The j-th component of the error block is $e_j = 0$ if no error has occurred at this position. Otherwise the j-th component is $e_j \neq 0$. Based on the parity-check condition, the **syndrome** is defined as*

$$s^T = H \cdot r^T. \tag{C.18}$$

The syndrome is used to check whether the received block r is a valid code block or not. The equation

$$s^T = H \cdot r^T = H \cdot (c + e)^T = H \cdot c^T + H \cdot e^T = H \cdot e^T \tag{C.19}$$

shows that the syndrome only depends on the error block.

It has to be kept in mind that there may be error blocks e which fulfill the equation $H \cdot e^T = 0$ although they are non-zero. These error blocks lead to valid code blocks and hence cannot be detected correctly as error.

An important aspect regarding the computation of the syndrome is the fact that the solution of the equation $s^T = H \cdot e^T$ does not yield a unique error block e, since there exist multiple such error blocks which satisfy the equation. All these error blocks form the coset of the code, which comprises of q^k blocks. With a minimum distance decoding approach, the error block with the smallest number of non-zero elements, the so called *coset leader*, is chosen.

Appendix D

Overview of the IEEE Standard 754™-2008 for Floating-Point Arithmetic

This appendix gives an overview of the basic formats and concepts that are defined in the IEEE Standard 754™-2008 for floating-point arithmetic. The presented information is based on the latest revision of the standard [IEEE 08], which has been published on August 29, 2008.

The IEEE Standard 754™-2008 for floating-point arithmetic defines number formats (binary and decimal) and binary data interchange formats, which are used to represent subsets of real numbers. It also provides methods and algorithms for the processing of floating-point data. The goal of this standard is to ensure identical results of floating-point computations regardless whether these computations are performed in hardware, software or combinations of both. It defines number formats and binary data interchange formats for *single*, *double*, *extended* and *extendable* precision. The arithmetic operations covered by the standard comprise *addition*, *subtraction*, *multiplication*, *division*, *fused multiply add*, *square root*, *compare* and other operations.

In addition to the specification of formats, the standard covers exceptions for floating-point arithmetic, for instance computed results which cannot be represented (Not-a-

Number, NaN), and their handling. It also specifies algorithms for number conversions like integer to floating-point and vice versa.

The number formats within the IEEE Standard 754™-2008 are described by three parameters: a *radix*, a *precision* and a *range for the exponent.* The accompanying data formats are described by a triplet comprising of a *sign*, an *exponent field* and a *significand field* in radix *b*:

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

The set or real numbers which are actually representable by a number format are defined by the following integer parameters:

- b, the radix which can be 2 or 10,
- p, the precision which describes the number of digits in the significand,
- e_{max}, the maximum exponent,
- e_{min}, the minimum exponent.

The standard states that e_{min} shall be $1 - e_{max}$ for all specified number formats. It also specifies that signed zero and non-zero floating-point numbers of the form

$$(-1)^s \times b^e \times m$$

have to be representable. Here, the sign s is 0 or 1, and the exponent e is an integer $e_{min} \leq e \leq e_{max}$. The significand m is a digital string of the form $d_0 \bullet d_1 d_2 ... d_{p-1}$, with integer digits d_i, $0 \leq d_i < b$. The significand is therefore $0 \leq m < b$. The standard further distinguishes two infinite values $+\infty$ an $-\infty$, as well as a quiet (qNaN) and a signaling (sNaN) version of not-a-number.

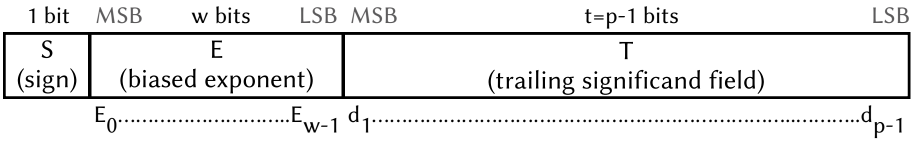
Encoding of Number Formats

The guarantee of unique number representations is an essential design goal of the IEEE Standard 754™-2008. The default representation are *normalized* floating-point numbers between $b^{e_{min}}$ and $b^{e_{max}} \times (b - b^{1-p})$. Non-zero floating-point numbers with magnitudes smaller than $b^{e_{min}}$ are subnormal numbers, which have less than p significant digits.

Each IEEE 754-2008 floating-point number has a unique encoding based on k bits. For the normalization of numbers, the significand m is maximized while the corresponding exponent is decreased until either $e = e_{min}$ or $m \geq 1$. The k-bit encoding uses the following three fields:

- the sign s with 1 bit,
- the biased exponent $E = e + bias$ with w bits,
- the trailing significand string $T = d_1 d_2 ... d_{p-1}$ with $t = p - 1$ bits, where the leading digit d_0 is implicitly encoded in the exponent E.

Figure D.1 depicts this basic structure.



▲ **Figure D.1** — Structure of IEEE-754 floating-point formats. Adopted from [IEEE 08].

The corresponding parameterization of k, p, t, w, and $bias$ for the different types of precision defined in the standard are given in Table D.1. The encoding of the biased exponent E allows the distinction between normal and subnormal numbers, as well as the unique encoding of ±0, ±∞ and NaNs. Every integer value between 1 and $2^w - 2$, inclusive, encodes a normal floating-point number, whereas the reserved value 0 encodes ±0 and subnormal numbers. The value $2^w - 1$ is reserved for the encoding of ±∞ and NaNs.

The bit representation r of an IEEE 754-2008 floating-point number and its value v are derived from the fields introduced above in the following manner:

- If $E = 2^w - 1$ and $T \neq 0$, then r is qNaN or sNaN and v is NaN independent of S.
- If $E = 2^w - 1$ and $T = 0$, then r and $v = (-1)^S \times (+\infty)$.
- If $1 \leq E \leq 2^w - 2$, then r is $(S, (E - bias), (1 + 2^{1-p} \times T))$. The value of the corresponding floating-point number is $v = (-1)^S \times 2^{E-bias} \times (1 + 2^{1-p} \times T)$. Normal numbers have an implicit leading significand bit of 1.
- If $E = 0$ and $T \neq 0$, then r is $(S, e_{min}, (0 + 2^{1-p} \times T))$. The value of the corresponding floating-point number is $v = (-1)^S \times 2^{e_{min}} \times (0 + 2^{1-p} \times T)$. Subnormal numbers have an implicit leading significand bit of 0.

▼ **Table D.1** — Parameters of the IEEE 754-2008 number and data formats. Adopted from [IEEE 08]. The function $RN()$ in this table rounds to the next integer.

Parameter	binary16	binary32	binary64	binary128	binary{k} ($k \geq 128$)
k, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$k - RN(4 \cdot log_2(k)) + 13$
e_{max}, maximum exponent e	15	127	1023	16383	$2^{k-p-1} - 1$
Encoding Parameters					
$bias$, $E - e$	15	127	1023	16383	e_{max}
sign bit	1	1	1	1	1
w, exponent field width in bits	5	8	11	15	$RN(4 \cdot log_2(k)) - 13$
t, trail. significand field width in bits	10	23	52	112	$k - w - 1$
k, storage width in bits	16	32	64	128	$1 + w + t$

- If $E = 0$ and $T = 0$, then r is $(S, e_{min}, 0)$ and $v = (-1)^S \times (+0)$.

Rounding

Floating-point numbers are an approximation of the real numbers which often leads to situations where a real number cannot be represented exactly due to the limited precision. In such cases, an appropriate floating-point number has to be chosen to represent the real number. The process that determines this floating-point number is called *rounding* and introduces the so called *rounding error*. The IEEE Standard 754™-2008 for floating-point arithmetic defines a set of rules and directions for the rounding process which have to be implemented by compliant software and hardware.

The standard also demands that exceptions such as underflow or overflow have to be signaled appropriately.

The main rounding mode defined by the standard is *round to nearest*, which offers three different *rounding directions*. Given an infinitely precise result with a magnitude of at least $b^{e_{max}}(b - \frac{1}{2}b^{1-p})$. This result is rounded to ∞ without changing the sign. The maximum exponent e_{max} and the precision p are determined by the target number format (see Table D.1). The rounding directions are:

- **RoundTiesToEven**: The floating-point number nearest to the infinitely precise result is delivered. If there exist two candidates, the floating-point number with an even last significand digit is chosen.
- **RoundTiesToAway**: The floating-point number nearest to the infinitely precise result is delivered. If there exist two candidates, the floating-point number with the larger magnitude is chosen.

In addition to this main rounding mode, the standard defines three optional rounding directions:

- **RoundTowardPositive**: The floating-point number closest to the infinitely precise result is delivered, which is no less than this result. $+\infty$ is possible as rounded value.
- **RoundTowardNegative**: The floating-point number closest to the infinitely precise result is delivered, which is no greater than this result. $-\infty$ is possible as rounded value.
- **RoundTowardZero**: The floating-point number closest to the infinitely precise result is delivered, which is no greater in magnitude than this result.

Infinity and Not-a-Number

Floating-point arithmetic, as it is defined in the IEEE standard, uses $-\infty$ and $+\infty$ to express operands of arbitrarily large magnitude. Most operations yield exact results when infinity is involved and therefore no exceptions have to be signaled. The standard lists the operations:

- addition(∞, x), addition(x, ∞), subtraction(∞, x), subtraction$(x.\infty)$ for finite x,

- multiplication(∞, x) or multiplication(x, ∞) for fininte x or infinite $x \neq 0$,
- division(∞, x) or division(x, ∞) for finite x,
- square root($+\infty$),
- remainder(x, ∞) for finite normal x,
- conversion of infinity to the same infinity in another format.

Cases where exceptions are signaled are:

- ∞ as an invalid operand,
- ∞ created from finite operands by overflow,
- remainder($subnormal, \infty$) signals underflow.

Values or results that cannot be represented by valid floating-point numbers are set to *Not-a-Number*. The standard distinguishes two different NaN representations, the *signaling NaN* (sNaN) and the *quiet NaN* (qNaN).

Exceptions

The IEEE Standard 754™-2008 for floating-point arithmetic defines several exceptions and accompanying methods for the handling of these exceptions. The most important exceptions are:

- **Invalid operation**: This exception is signaled if the operands are invalid for the target operation or if the result is not defined. The default result for an operation that throws an invalid operation exception is the quiet NaN.
- **Division by zero**: This exception is signaled if an exact infinite result is defined for an operation on finite operands. The default result in this case is ∞.
- **Overflow**: This exception is signaled if the target floating-point number format does not provide a representation which is large enough to map the result.
- **Underflow**: This exception is signaled if a very small non-zero result is detected.
- **Inexact**: This exception is signaled when a result is rounded.

Further details on the floating-point number formats, operations, exceptions and implementation requirements defined in the IEEE Standard 754™-2008 can be found in [IEEE 08].

Appendix E

Implementation Details

This appendix chapter provides additional details on the different algorithmic steps that are performed by the proposed A-Abft method, and which have been omitted in Chapter 6 for the sake of conciseness. The procedures for the checksum encoding and the generation of the candidate sets are described in Algorithm 1. The procedures for the determination of the rounding error bounds and the check of the computed result sub-matrices are given in Algorithm 2.

In addition to the central algorithmic steps of A-Abft, pseudocode descriptions for the QR decomposition based on Householder transformations and the Revised Simplex algorithm, as they have been used for the case studies in Chapter 7, are given in Algorithm 3 and Algorithm 4.

```
Input: padded matrix $A$, $numMax$
Output: encoded matrix $A$, $maxValues$, $maxValueIDs$
Launch Dimensions: $BS \times 1$ threads, one thread block for each $BS \times BS$ sub-matrix of $A$
Data: local registers $sum$, $maxVal$, $maxSum$
Data: shared $Asub[BS][BS]$, $localSums[BS]$
/* loop through rows of block */
for $i \leftarrow 1$ to $BS$ do
    /* each thread calculates column checksum for its own
       column. tid is the thread's ID. */
    load one element of sub-matrix of $A$ into $Asub[i][tid]$;
    $sum \leftarrow sum + Asub[i][tid]$;
    $Asub[i][tid] = abs(Asub[i][tid])$;
end
write back column-checksum $sum$ into $A$;
sync
$sum \leftarrow abs(sum)$
for $m \leftarrow 1$ to $numMax$ do
    /* init local sums and current maximum */
    $maxVal \leftarrow 0$;
    $localSums[tid] \leftarrow abs(sum)$;
    /* search maximum value in row */
    for $i \leftarrow 1$ to $BS$ do
        $maxVal \leftarrow max(maxVal, Asub[tid][i])$;
        update $maxValueID$;
    end
    /* search maximum value of previously calculated
       checksum entries */
    $maxSum \leftarrow$ maxReduce($localSums$);
    update $maxSumID$;
    /* write back maximum values, their location and exclude
       those values from the next round of calculation */
    write back $maxValues$ and $maxValueIDs$;
    $Asub[tid][maxID] \leftarrow 0$;
    if $maxSumID == tid$ then
        write back $maxSum$ and $maxSumID$;
        $sum \leftarrow 0$;
    end
end
```

Algorithm 1: GPU Kernel for Checksum Encoding and Candidate Set Generation.

```
Input: matrix $C$, $maxA$, $maxB$, $maxAIDs$, $maxBIDs$, $numMax$
Output: corrected matrix $C$ or error information

Launch Dimensions: $BS \times 1$ threads, one threadblock for each submatrix of $C$

Data: local registers $A_{idx}[numMax]$, $B_{idx}[numMax]$, $max$, $sum$, $eps$
Data: shared $Csub[BS][BS]$

/* load max indices                                                */
for $i \leftarrow 1$ to $numMax$ do
    $A_{idx}[i] \leftarrow maxAIDs$; $B_{idx}[i] \leftarrow maxBIDs$;
end

/* check for combinations                                          */
$max \leftarrow 0$;
$found \leftarrow$ false;
for $m \leftarrow 1$ to $numMax$ do
    for $n \leftarrow 1$ to $numMax$ do
        if $A_{idx}[m] == B_{idx}[n]$ then
            $newmax \leftarrow maxA[m] \cdot maxB[n]$;
            $max \leftarrow fmax(max, newmax)$;
            $found \leftarrow$ true;
        end
    end
end

$max \leftarrow fmax(max, maxA[numMax-1] \cdot maxB[0])$;
$max \leftarrow fmax(max, maxA[0] \cdot maxB[numMax-1])$;

/* recalculate column checksums                                    */
for $i \leftarrow 1$ to $BS$ do
    load one element of submatrix of $C$ into $Csub[i][tid]$;
    $sum \leftarrow sum + Csub[i][tid]$;
end

/* load checksum element of C, calculate rounding error
   bound eps and compare checksums                                 */
$ref \leftarrow$ columnChecksum($C$, $tid$);
$eps \leftarrow$ calculateEpsilon($max$);
if $abs(ref - sum) > eps$ then
    write back error location or start correction;
end

/* analogously recalculate and check the row checksums, all
   necessary data is already in the shared memory...               */
```

Algorithm 2: GPU Kernel for Computation of Rounding Error Bounds and Checking of Result Sub-Matrices.

```
Input: Matrix $\boldsymbol{A}$, dimensions $rowsA$ and $colsA$ of $\boldsymbol{A}$.
Output: Matrix $\boldsymbol{Q}$ and matrix $\boldsymbol{R}$
/* Storage for intermediate Qs */
Matrix $\boldsymbol{Q_{temp}}[rowsA]$;
Matrix $\boldsymbol{z} \leftarrow \boldsymbol{A}$;
for $k \leftarrow 0; k < colsA$ && $k < rowsA - 1; k++$ do
    Matrix $\boldsymbol{z_1} \leftarrow$ init_matrix($rowsA$, $colsA$);
    /* Copy sub-matrix from $z$ to $z_1$ */
    matrix_minor($\boldsymbol{z}$,$k$,$\boldsymbol{z_1}$);
    /* Copy $k$-th column of $z$ into vector $\boldsymbol{x}$ */
    $\boldsymbol{x} \leftarrow \boldsymbol{z}[\cdot][k]$;
    $a \leftarrow ||\boldsymbol{x}||_2$;
    /* Change sign of norm if $k$-th diag. elem. of $A$ is $>0$ */
    if $\boldsymbol{A}[k][k] > 0$ then
        $a = -a$;
    end
    /* Initialize unit vector $\boldsymbol{e}$ */
    for $i \leftarrow 0; i < rowsA; i++$ do
        $\boldsymbol{e}[i] \leftarrow (i == k)?1 : 0$;
    end
    for $i \leftarrow 0; i < rowsA; i++$ do
        $\boldsymbol{e}[i] \leftarrow \frac{\boldsymbol{e}[i]}{||\boldsymbol{e}||_2}$;
    end
    for $i \leftarrow 0; i < rowsA; i++$ do
        for $j \leftarrow 0; j < rowsA; j++$ do
            $\boldsymbol{Q_{temp}}[i][j] \leftarrow -2.0 \cdot \boldsymbol{e}[i] \cdot \boldsymbol{e}[j]$;
        end
    end
    $\boldsymbol{z_1} \leftarrow \boldsymbol{Q}[k] \cdot \boldsymbol{z}$
end
$\boldsymbol{Q} \leftarrow \boldsymbol{Q_{temp}}[0]$;
$\boldsymbol{R} \leftarrow \boldsymbol{Q_{temp}}[0] \cdot \boldsymbol{A}$;
for $i \leftarrow 1; i < colsA$ && $i < rowsA - 1; i++$ do
    $\boldsymbol{z_1} \leftarrow \boldsymbol{Q_{temp}}[i] \cdot \boldsymbol{Q}$;
    $\boldsymbol{Q} \leftarrow \boldsymbol{z_1}$;
end
$\boldsymbol{z} \leftarrow \boldsymbol{Q} \cdot \boldsymbol{A}$;
$\boldsymbol{R} \leftarrow \boldsymbol{z}$;
$\boldsymbol{Q} \leftarrow \boldsymbol{Q}^T$;
```

Algorithm 3: QR Decomposition using Householder Transformations.

```
Input: Matrix $A$, vectors $b$ and $c$. Linear program in canonical augmented form.
Ensure: Optimal solution or unbounded signal.
/* Initialize data                                          */
$\boldsymbol{B}[m][m] \leftarrow \boldsymbol{I}_m$;
$\boldsymbol{c_B}[m] \leftarrow \boldsymbol{c}[n-m : n-1]$;
$\boldsymbol{x}_B \leftarrow 0$; $Optimum \leftarrow \bot$;
while !$Optimum$ do
    /* Determine entering variable                          */
    $\boldsymbol{y}[m] \leftarrow \boldsymbol{c_B B}^{-1}$;
    $\boldsymbol{e}[n] \leftarrow [1\ \boldsymbol{y}] \cdot [-\boldsymbol{c}; \boldsymbol{A}]$;
    Index $p \leftarrow \{j \mid \boldsymbol{e}_j == min_t(\boldsymbol{e}_t)\}$;
    $\boldsymbol{x_B} \leftarrow \boldsymbol{B}^{-1}\boldsymbol{b}$;
    if $\boldsymbol{e}_p \geq 0$ then
        $Optimum \leftarrow \top$;
        break;
    end
    /* Determine leaving variable                           */
    $\alpha[m] \leftarrow \boldsymbol{B}^{-1}\boldsymbol{A}_p$;
    for $t \leftarrow 0, m-1$ do
        $\Theta_t \leftarrow \alpha_t > 0\ ?\ \frac{\boldsymbol{x_B}}{\alpha_t}\ :\ \infty$;
    end
    Index $q \leftarrow \{j \mid \Theta_j == min_t(\Theta_t)\}$;
    if $\alpha \leq 0$ then
        printerr("Problem is unbounded!");
        break;
    end
    /* Basis update                                         */
    $\boldsymbol{E}[m][m] \leftarrow ComputeE(\alpha, q)$;
    $\boldsymbol{B}^{-1} \leftarrow \boldsymbol{EB}^{-1}$;
    /* Update basis cost                                    */
    $\boldsymbol{c_B} \leftarrow \boldsymbol{c}_p$;
    /* Update basic solution                                */
    $\boldsymbol{x_B} \leftarrow \boldsymbol{B}^{-1}\boldsymbol{b}$;
end
if $Optimum$ then
    Return($\boldsymbol{x_B}, z \leftarrow \boldsymbol{x_B c_B}$);
end
```

Algorithm 4: Revised Simplex algorithm as introduced in [Spamp09b].

Appendix F

Experimental Data

This appendix provides the reader with additional experimental results for the three main evaluation areas: *performance*, *quality of error bounds* and *error detection*.

F.1 Hardware and Software Configuration

The experiments for the evaluation part of this work have been performed on two Intel Xeon-based compute servers, hereinafter referred to as *CPU I* and *CPU II*, which were equipped with Nvidia Tesla GPU accelerators. The detailed technical hardware and software specifications of the two systems are given in the following:

- **Compute server 1 (*CPU I*)**
  - 2x Intel Xeon E5-2650 CPU, 2.2GHz clock frequency
  - 280 (2x140) GFLOPS theo. double precision peak performance at base clock
  - 256 GB memory
  - Cent OS 6 Linux operating system
  - 2x Nvidia Tesla K20c GPU accelerator
  - Nvidia CUDA version 5.5
- **Compute server 2 (*CPU II*)**

  - 2x Intel Xeon E5-2687W3 CPU, 3.4GHz clock frequency
  - 396 (2x198) GFLOPS theo. double precision peak performance at base clock
  - 256 GB memory
  - Cent OS 6 Linux operating system
  - 1x Nvidia Tesla K40c GPU accelerator
  - Nvidia CUDA version 6.5

- **Nvidia Tesla K20c GPU accelerator**
  - CUDA compute capability 3.5
  - Nvidia GK110 "Kepler" architecture
  - 2496 CUDA processing cores
  - 706 MHz GPU clock frequency
  - 1.17 TFLOPS theo. double precision peak performance
  - 3.52 TFLOPS theo. single precision peak performance
  - 5 GB GDDR5 device memory
  - External and internal memories with ECC
  - With ECC on, 6.25% of the GPU memory is used for ECC bits.
  - 2600 MHz memory clock frequency
  - 208 GB/sec memory bandwidth
  - 320 bit memory bus width
  - PCI Express Gen2 ×16 system interface
- **Nvidia Tesla K40c GPU accelerator**
  - CUDA compute capability 3.5
  - Nvidia GK110B "Kepler" architecture
  - 2880 CUDA processing cores
  - 745 MHz GPU clock frequency
  - 1.43 TFLOPS theo. double precision peak performance
  - 4.29 TFLOPS theo. single precision peak performance
  - 12 GB GDDR5 device memory
  - External and internal memories with ECC

- With ECC on, 6.25% of the GPU memory is used for ECC bits.
- 3004 MHz memory clock frequency
- 288 GB/sec memory bandwidth
- 384 bit memory bus width
- PCI Express Gen2 ×16 system interface

F.2 Evaluation of Computational Performance

F.2.1 Runtime of ABFT Encoding and Checking Kernels

▼ **Table F.1** — Runtime of encoding and checking kernels for a 512×512 matrix using 1 weighted checksum and different ABFT encoding block sizes $bs \times bs$.

ABFT Block Size [$bs \times bs$]	Encoding A (ms)	Encoding B (ms)	Matrix Multiply (ms)	Checking C (ms)
32	0.129	0.128	0.484	0.143
64	0.130	0.129	0.479	0.143
128	0.124	0.123	0.483	0.167
256	0.448	0.450	0.481	0.413

▼ **Table F.2** — Runtime of encoding and checking kernels for a 4096×4096 matrix using 1 weighted checksum and different ABFT encoding block sizes $bs \times bs$.

ABFT Block Size [$bs \times bs$]	Encoding A (ms)	Encoding B (ms)	Matrix Multiply (ms)	Checking C (ms)
32	7.133	7.113	139.979	7.559
64	4.879	4.884	136.425	5.244
128	2.406	2.406	136.131	4.479
256	3.205	3.203	136.327	4.838

▼ **Table F.3** — Runtime of encoding and checking kernels for a 8192 × 8192 matrix using 1 weighted checksum and different ABFT encoding block sizes $bs \times bs$.

ABFT Block Size [$bs \times bs$]	Encoding A (ms)	Encoding B (ms)	Matrix Multiply (ms)	Checking C (ms)
32	28.452	28.464	1114.995	30.080
64	19.334	19.354	1079.389	21.103
128	9.425	9.423	1063.915	17.614
256	12.277	12.278	1063.803	19.019

▼ **Table F.4** — Runtime of encoding and checking kernels for a 512 × 512 matrix using 2 weighted checksums and different ABFT encoding block sizes $bs \times bs$.

ABFT Block Size [$bs \times bs$]	Encoding A (ms)	Encoding B (ms)	Matrix Multiply (ms)	Checking C (ms)
32	0.141	0.140	0.479	0.175
64	0.142	0.143	0.480	0.163
128	0.136	0.135	0.483	0.222

▼ **Table F.5** — Runtime of encoding and checking kernels for a 4096 × 4096 matrix using 2 weighted checksums and different ABFT encoding block sizes $bs \times bs$.

ABFT Block Size [$bs \times bs$]	Encoding A (ms)	Encoding B (ms)	Matrix Multiply (ms)	Checking C (ms)
32	7.601	7.600	148.253	9.418
64	5.569	5.568	139.990	6.785
128	3.286	3.288	136.185	8.018

▼ **Table F.6** — Runtime of encoding and checking kernels for a 8192 × 8192 matrix using 2 weighted checksums and different ABFT encoding block sizes $bs \times bs$.

ABFT Block Size [$bs \times bs$]	Encoding A (ms)	Encoding B (ms)	Matrix Multiply (ms)	Checking C (ms)
32	30.411	30.413	1181.326	37.595
64	22.120	22.117	1114.981	27.068
128	12.958	12.958	1079.409	31.846

F.2.2 Runtime of SEA and PEA Preprocessing Kernels

Table F.7 presents the kernel execution times for the SEA and PEA preprocessing with separate times reported for the preprocessing of rows and columns.

▼ **Table F.7** — Kernel Execution Times for SEA and PEA Preprocessing separated for rows/columns.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**
[$n \times n$]	(ms)	(ms)	(ms)	(ms)	(ms)
512	0.81/0.29	0.72/0.49	1.14/0.86	1.91/1.73	4.77/4.57
1024	1.65/0.59	1.40/0.94	2.18/1.65	4.12/3.83	8.86/8.45
2048	3.30/1.17	3.27/1.84	5.07/3.78	8.91/7.84	18.81/17.21
3072	4.97/1.76	4.88/2.73	7.48/5.54	13.29/11.44	28.64/26.48
4096	6.64/2.37	7.94/4.15	11.63/7.73	19.80/16.09	40.21/36.52
5120	8.37/3.00	12.14/5.60	17.03/10.33	27.82/21.40	53.47/47.32
6144	10.35/3.68	14.75/6.79	20.61/12.50	33.50/25.93	64.07/56.60
7168	24.15/4.49	20.74/8.64	28.04/15.88	44.17/32.55	81.51/69.83
8192	28.31/5.58	25.06/10.29	33.58/18.81	52.40/38.26	95.66/81.49

F.2.3 Runtime of SEA and PEA ε-Determination Kernels

▼ **Table F.8** — Kernel Execution Times for SEA and PEA ε-determination, ABFT encoding block size 32×32.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**
[$n \times n$]	(ms)	(ms)	(ms)	(ms)	(ms)
512	0.022	0.029	0.041	0.080	0.214
1024	0.025	0.058	0.108	0.214	0.789
2048	0.039	0.138	0.259	0.668	2.356
3072	0.062	0.266	0.542	1.399	4.920
4096	0.095	0.453	0.923	2.403	8.417
5120	0.139	0.694	1.428	3.693	12.954
6144	0.191	0.978	2.030	5.284	18.282
7168	0.251	1.380	2.779	7.210	24.812
8192	0.323	1.717	3.643	9.358	32.163

▼ **Table F.9** — Kernel Execution Times for SEA and PEA ε-determination, ABFT encoding block size 256×256.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)
512	0.020	0.026	0.031	0.046	0.101
1024	0.019	0.028	0.032	0.052	0.157
2048	0.024	0.037	0.054	0.108	0.259
3072	0.026	0.054	0.101	0.179	0.544
4096	0.032	0.071	0.137	0.295	0.916
5120	0.038	0.096	0.200	0.478	1.563
6144	0.047	0.155	0.291	0.724	2.351
7168	0.055	0.186	0.360	0.945	3.138
8192	0.066	0.262	0.523	1.208	4.086

F.2.4 Computational Performance for different weighted checksum encodings across different ABFT encoding block sizes

▼ **Table F.10** — Performance on Nvidia Tesla K40c GPU with one *Normal* weighted checksum and ABFT encoding block size 32×32.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.27	2.26	2.99	5.06	9.78	2.33	0.91
1024	5.75	5.70	7.33	10.99	20.32	10.13	6.24
2048	24.34	24.28	27.96	35.32	54.32	70.52	49.84
3072	67.66	66.42	71.99	83.28	113.25	225.95	162.00
4096	145.00	147.79	155.64	171.96	216.83	572.15	391.78
5120	276.25	271.48	281.46	303.46	367.58	1003.48	700.21
6144	455.58	461.69	474.47	503.14	580.17	1733.07	1194.06
7168	715.05	722.40	737.88	772.97	866.71	2698.94	1866.83
8192	1061.58	1058.61	1077.56	1122.06	1237.98	4597.47	3613.33

▼ **Table F.11** — Performance on Nvidia Tesla K40c GPU with one *Normal* weighted checksum and ABFT encoding block size 64×64.

Matrix $[n \times n]$	**SEA** (ms)	**PEA 2** (ms)	**PEA 4** (ms)	**PEA 8** (ms)	**PEA 16** (ms)	**CPU I** (ms)	**CPU II** (ms)
512	2.26	2.26	3.00	5.12	9.96	2.33	0.91
1024	5.42	5.35	6.96	10.48	19.20	10.13	6.24
2048	23.08	23.28	26.90	33.85	51.53	70.52	49.84
3072	62.49	62.96	68.34	78.84	106.39	225.95	162.00
4096	135.75	138.36	145.59	160.52	200.34	572.15	391.78
5120	263.15	266.60	276.18	296.13	348.04	1003.48	700.21
6144	440.60	446.67	459.05	485.08	551.33	1733.07	1194.06
7168	685.18	691.30	696.03	727.56	806.85	2698.94	1866.83
8192	1005.57	1005.08	1022.54	1061.70	1158.01	4597.47	3613.33

▼ **Table F.12** — Performance on Nvidia Tesla K40c GPU with one *Normal* weighted checksum and ABFT encoding block size 128×128.

Matrix $[n \times n]$	**SEA** (ms)	**PEA 2** (ms)	**PEA 4** (ms)	**PEA 8** (ms)	**PEA 16** (ms)	**CPU I** (ms)	**CPU II** (ms)
512	2.34	2.31	3.05	5.13	9.92	2.33	0.91
1024	5.55	5.46	7.04	10.49	19.05	10.13	6.24
2048	23.55	23.40	29.44	33.63	50.34	70.52	49.84
3072	64.22	64.62	69.84	80.00	105.63	225.95	162.00
4096	135.57	138.06	145.20	162.61	200.05	572.15	391.78
5120	257.23	260.49	269.69	288.64	337.42	1003.48	700.21
6144	431.63	437.61	449.21	473.94	535.05	1733.07	1194.06
7168	671.58	678.08	691.98	721.88	794.26	2698.94	1866.83
8192	1006.84	1001.51	1017.59	1054.14	1140.65	4597.47	3613.33

▼ **Table F.13** — Performance on Nvidia Tesla K40c GPU with one *Normal* weighted checksum and ABFT encoding block size 256×256.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.59	2.55	3.28	5.14	9.84	2.33	0.91
1024	5.69	5.57	7.16	10.55	19.12	10.13	6.24
2048	23.93	23.76	27.18	33.83	50.01	70.52	49.84
3072	64.31	64.66	69.86	79.80	104.79	225.95	162.00
4096	138.91	141.29	148.22	162.23	197.45	572.15	391.78
5120	257.97	261.05	270.00	288.36	333.77	1003.48	700.21
6144	432.63	438.39	449.74	473.52	530.45	1733.07	1194.06
7168	672.52	679.01	692.72	721.03	791.46	2698.94	1866.83
8192	1008.83	1002.88	1018.60	1054.09	1136.32	4597.47	3613.33

▼ **Table F.14** — Performance on Nvidia Tesla K40c GPU with one *Exponential* weighted checksum and ABFT encoding block size 32×32.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.27	2.27	3.01	5.09	9.90	2.33	0.91
1024	5.77	5.71	7.34	10.99	20.24	10.13	6.24
2048	24.34	24.26	27.95	35.32	54.32	70.52	49.84
3072	65.88	66.46	72.05	83.43	115.44	225.95	162.00
4096	145.04	147.76	155.61	171.81	216.49	572.15	391.78
5120	272.90	276.43	286.36	308.26	367.20	1003.48	700.21
6144	455.49	461.63	474.57	503.34	580.89	1733.07	1194.06
7168	715.29	722.50	738.03	773.46	867.37	2698.94	1866.83
8192	1061.60	1058.69	1077.18	1121.48	1237.24	4597.47	3613.33

▼ **Table F.15** — Performance on Nvidia Tesla K40c GPU with one *Exponential* weighted checksum and ABFT encoding block size 64 × 64.

Matrix [$n \times n$]	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.26	2.26	3.01	5.11	9.96	2.33	0.91
1024	5.44	5.34	6.96	10.46	19.14	10.13	6.24
2048	23.07	23.31	26.88	33.87	51.56	70.52	49.84
3072	62.50	62.97	68.41	79.00	106.83	225.95	162.00
4096	135.72	141.33	148.70	163.70	203.52	572.15	391.78
5120	263.16	266.56	276.11	296.07	347.85	1003.48	700.21
6144	440.57	446.66	459.13	485.09	551.15	1733.07	1194.06
7168	674.26	681.78	696.07	727.59	807.17	2698.94	1866.83
8192	1005.29	1004.91	1022.42	1061.47	1157.32	4597.47	3613.33

▼ **Table F.16** — Performance on Nvidia Tesla K40c GPU with one *Exponential* weighted checksum and ABFT encoding block size 128 × 128.

Matrix [$n \times n$]	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.29	2.30	3.03	5.13	9.92	2.33	0.91
1024	5.64	5.48	7.06	10.59	19.31	10.13	6.24
2048	23.56	23.42	26.93	33.75	50.50	70.52	49.84
3072	64.21	64.65	69.96	80.28	106.47	225.95	162.00
4096	138.63	141.11	148.21	162.62	199.91	572.15	391.78
5120	252.48	255.69	264.87	283.96	333.05	1003.48	700.21
6144	431.65	437.70	449.38	474.28	535.78	1733.07	1194.06
7168	671.47	677.97	691.89	721.51	792.92	2698.94	1866.83
8192	1006.77	1001.61	1017.55	1054.31	1141.38	4597.47	3613.33

▼ **Table F.17** — Performance on Nvidia Tesla K40c GPU with one *Exponential* weighted checksum and ABFT encoding block size 256 × 256.

Matrix $[n \times n]$	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.54	2.55	3.31	5.24	10.08	2.33	0.91
1024	5.80	5.59	7.15	10.58	19.08	10.13	6.24
2048	23.93	23.75	27.19	33.90	50.09	70.52	49.84
3072	64.31	64.65	69.83	80.99	104.77	225.95	162.00
4096	138.92	141.38	148.44	162.70	198.93	572.15	391.78
5120	257.90	261.11	269.94	287.99	332.46	1003.48	700.21
6144	432.64	438.45	449.89	473.64	531.08	1733.07	1194.06
7168	672.66	679.17	692.74	721.44	789.40	2698.94	1866.83
8192	1008.79	1002.53	1018.61	1054.40	1136.40	4597.47	3613.33

▼ **Table F.18** — Performance on Nvidia Tesla K40c GPU with one *Linear* weighted checksum and ABFT encoding block size 32 × 32.

Matrix $[n \times n]$	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.26	2.25	2.99	5.07	9.90	2.33	0.91
1024	5.77	5.71	7.35	10.96	20.36	10.13	6.24
2048	24.33	24.27	27.95	35.34	54.26	70.52	49.84
3072	67.66	68.22	73.84	85.18	115.56	225.95	162.00
4096	144.97	147.76	155.66	171.93	216.73	572.15	391.78
5120	272.87	276.42	286.47	308.38	367.30	1003.48	700.21
6144	455.91	461.87	474.88	503.65	580.14	1733.07	1194.06
7168	715.28	722.42	737.90	772.81	866.02	2698.94	1866.83
8192	1061.53	1058.63	1077.22	1121.38	1236.28	4597.47	3613.33

▼ **Table F.19** — Performance on Nvidia Tesla K40c GPU with one *Linear* weighted checksum and ABFT encoding block size 64×64.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
[$n \times n$]	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.25	2.26	2.99	5.05	9.80	2.33	0.91
1024	5.44	5.34	6.95	10.49	19.24	10.13	6.24
2048	23.07	23.28	26.79	33.73	51.34	70.52	49.84
3072	62.50	62.97	68.43	79.07	107.09	225.95	162.00
4096	138.79	141.35	148.67	163.43	202.79	572.15	391.78
5120	263.15	266.53	276.08	295.98	347.49	1003.48	700.21
6144	440.50	446.71	458.87	484.84	550.99	1733.07	1194.06
7168	674.40	681.81	696.08	727.64	806.89	2698.94	1866.83
8192	1005.38	1004.87	1022.72	1061.72	1158.01	4597.47	3613.33

▼ **Table F.20** — Performance on Nvidia Tesla K40c GPU with one *Linear* weighted checksum and ABFT encoding block size 128×128.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
[$n \times n$]	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.29	2.31	3.06	5.19	10.13	2.33	0.91
1024	5.64	5.48	7.07	10.60	19.31	10.13	6.24
2048	23.57	23.42	26.89	33.67	50.40	70.52	49.84
3072	62.52	62.96	68.27	78.59	104.95	225.95	162.00
4096	138.61	141.08	148.25	162.78	200.48	572.15	391.78
5120	252.47	255.63	264.92	288.73	337.42	1003.48	700.21
6144	431.65	437.52	449.32	473.96	535.00	1733.07	1194.06
7168	671.49	677.97	691.92	721.41	793.22	2698.94	1866.83
8192	1007.07	1001.35	1017.49	1054.58	1142.25	4597.47	3613.33

▼ **Table F.21** — Performance on Nvidia Tesla K40c GPU with one *Linear* weighted checksum and ABFT encoding block size 256 × 256.

Matrix	SEA	PEA 2	PEA 4	PEA 8	PEA 16	CPU I	CPU II
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.54	2.53	3.25	5.04	9.55	2.33	0.91
1024	5.78	5.61	7.20	10.71	19.41	10.13	6.24
2048	23.90	23.75	27.22	33.92	50.25	70.52	49.84
3072	64.31	64.66	69.85	79.77	104.80	225.95	162.00
4096	138.92	141.31	148.33	162.36	200.60	572.15	391.78
5120	257.88	261.01	269.85	287.93	332.37	1003.48	700.21
6144	432.55	438.38	449.69	473.49	530.62	1733.07	1194.06
7168	672.62	678.91	692.63	721.21	788.72	2698.94	1866.83
8192	1008.27	1004.00	1018.22	1053.33	1135.35	4597.47	3613.33

▼ **Table F.22** — Performance on Nvidia Tesla K40c GPU with one *MDR* weighted checksum and ABFT encoding block size 32 × 32.

Matrix	SEA	PEA 2	PEA 4	PEA 8	PEA 16	CPU I	CPU II
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.28	2.33	3.07	5.15	10.00	2.33	0.91
1024	5.94	5.82	7.44	11.09	20.32	10.13	6.24
2048	24.31	24.24	27.87	35.15	53.91	70.52	49.84
3072	67.65	68.23	73.82	85.22	115.59	225.95	162.00
4096	144.99	147.76	155.67	171.80	216.43	572.15	391.78
5120	272.99	276.46	286.34	308.05	366.65	1003.48	700.21
6144	455.75	461.83	474.65	503.34	580.25	1733.07	1194.06
7168	715.27	722.47	737.96	773.01	866.48	2698.94	1866.83
8192	1061.72	1058.76	1077.37	1121.75	1237.04	4597.47	3613.33

▼ **Table F.23** — Performance on Nvidia Tesla K40c GPU with one *MDR* weighted checksum and ABFT encoding block size 64 × 64.

Matrix [$n \times n$]	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.26	2.26	3.00	5.13	9.97	2.33	0.91
1024	5.45	5.35	6.96	10.49	19.16	10.13	6.24
2048	23.07	23.32	26.94	33.92	51.72	70.52	49.84
3072	62.50	62.99	68.41	79.00	106.94	225.95	162.00
4096	138.81	141.39	148.74	163.72	203.72	572.15	391.78
5120	263.23	266.51	276.11	295.98	347.64	1003.48	700.21
6144	440.53	446.57	458.97	484.93	550.94	1733.07	1194.06
7168	674.42	681.85	696.13	727.70	807.11	2698.94	1866.83
8192	1005.08	1004.77	1022.51	1061.92	1171.08	4597.47	3613.33

▼ **Table F.24** — Performance on Nvidia Tesla K40c GPU with one *MDR* weighted checksum and ABFT encoding block size 128 × 128.

Matrix [$n \times n$]	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.29	2.30	3.04	5.15	10.03	2.33	0.91
1024	5.64	5.47	7.09	10.64	19.29	10.13	6.24
2048	23.55	23.42	26.92	33.67	50.42	70.52	49.84
3072	62.52	62.94	68.14	78.32	104.09	225.95	162.00
4096	138.63	141.08	148.10	162.28	199.24	572.15	391.78
5120	257.28	260.50	269.72	288.71	337.48	1003.48	700.21
6144	431.65	437.66	449.33	474.09	535.22	1733.07	1194.06
7168	671.55	677.94	691.86	721.46	793.33	2698.94	1866.83
8192	1006.93	1001.29	1017.38	1053.79	1140.52	4597.47	3613.33

▼ **Table F.25** — Performance on Nvidia Tesla K40c GPU with one *MDR* weighted checksum and ABFT encoding block size 256×256.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.54	2.54	3.29	5.16	9.85	2.33	0.91
1024	5.79	5.59	7.18	10.64	19.22	10.13	6.24
2048	23.93	23.77	27.19	33.84	49.98	70.52	49.84
3072	64.31	64.66	69.90	79.87	104.85	225.95	162.00
4096	138.91	141.32	148.32	162.35	197.75	572.15	391.78
5120	258.04	261.00	269.97	288.07	333.23	1003.48	700.21
6144	432.54	438.43	449.81	473.70	530.99	1733.07	1194.06
7168	672.60	678.98	692.61	720.95	788.30	2698.94	1866.83
8192	1008.50	1002.54	1018.64	1054.04	1135.81	4597.47	3613.33

▼ **Table F.26** — Performance on Nvidia Tesla K40c GPU with one *Periodic* weighted checksum and ABFT encoding block size 32×32.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.27	2.27	3.01	5.10	9.89	2.33	0.91
1024	5.96	5.85	7.52	11.07	20.36	10.13	6.24
2048	24.37	24.28	27.98	35.73	56.19	70.52	49.84
3072	67.65	68.20	73.83	85.72	118.28	225.95	162.00
4096	144.99	147.78	155.91	173.10	221.56	572.15	391.78
5120	272.95	276.47	286.37	308.64	370.96	1003.48	700.21
6144	455.64	461.63	474.69	504.21	585.05	1733.07	1194.06
7168	715.25	722.58	738.16	773.98	871.02	2698.94	1866.83
8192	1082.20	1058.43	1077.47	1123.36	1243.32	4597.47	3613.33

▼ **Table F.27** — Performance on Nvidia Tesla K40c GPU with one *Periodic* weighted checksum and ABFT encoding block size 64×64.

Matrix	SEA	PEA 2	PEA 4	PEA 8	PEA 16	CPU I	CPU II
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.25	2.26	3.00	5.14	10.03	2.33	0.91
1024	5.44	5.34	6.95	10.49	19.23	10.13	6.24
2048	23.07	23.32	26.87	33.92	52.43	70.52	49.84
3072	62.52	62.99	68.41	79.13	107.98	225.95	162.00
4096	138.84	141.34	148.87	164.40	207.37	572.15	391.78
5120	263.16	266.55	276.04	295.99	350.74	1003.48	700.21
6144	440.52	446.76	459.11	485.56	555.33	1733.07	1194.06
7168	674.38	681.88	696.05	727.86	810.00	2698.94	1866.83
8192	1019.50	1017.33	1035.23	1075.73	1176.19	4597.47	3613.33

▼ **Table F.28** — Performance on Nvidia Tesla K40c GPU with one *Periodic* weighted checksum and ABFT encoding block size 128×128.

Matrix	SEA	PEA 2	PEA 4	PEA 8	PEA 16	CPU I	CPU II
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.29	2.29	3.02	5.12	9.94	2.33	0.91
1024	5.64	5.46	7.01	10.46	18.96	10.13	6.24
2048	23.56	23.40	26.91	33.57	50.12	70.52	49.84
3072	62.55	62.91	68.13	78.27	103.97	225.95	162.00
4096	138.60	141.07	148.25	163.09	202.70	572.15	391.78
5120	257.20	260.49	269.68	288.54	338.54	1003.48	700.21
6144	431.55	437.56	449.21	473.91	537.26	1733.07	1194.06
7168	671.56	678.12	691.91	721.64	797.27	2698.94	1866.83
8192	1006.46	1001.22	1017.72	1055.06	1146.28	4597.47	3613.33

▼ **Table F.29** — Performance on Nvidia Tesla K40c GPU with one *Periodic* weighted checksum and ABFT encoding block size 256 × 256.

Matrix $[n \times n]$	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.54	2.54	3.28	5.14	9.84	2.33	0.91
1024	5.78	5.58	7.16	10.54	19.01	10.13	6.24
2048	23.93	23.76	27.18	33.91	50.30	70.52	49.84
3072	64.31	64.62	69.75	79.62	104.45	225.95	162.00
4096	138.93	141.36	148.35	162.32	197.87	572.15	391.78
5120	257.90	261.12	270.08	288.36	333.82	1003.48	700.21
6144	432.60	438.32	449.91	473.70	533.75	1733.07	1194.06
7168	672.64	679.00	692.53	721.08	788.75	2698.94	1866.83
8192	1008.30	1002.48	1018.51	1054.58	1139.82	4597.47	3613.33

▼ **Table F.30** — Performance on Nvidia Tesla K40c GPU with one *Normal* and one *Linear* weighted checksum and ABFT encoding block size 32 × 32.

Matrix $[n \times n]$	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.31	2.31	3.09	5.25	10.29	2.33	0.91
1024	5.91	5.85	7.53	11.26	21.27	10.13	6.24
2048	24.84	25.25	29.10	36.99	58.51	70.52	49.84
3072	68.90	69.77	75.80	88.67	125.01	225.95	162.00
4096	153.78	157.14	165.76	184.45	239.11	572.15	391.78
5120	287.29	291.59	302.73	328.54	426.58	1003.48	700.21
6144	483.86	491.07	505.71	539.73	638.28	1733.07	1194.06
7168	752.90	761.72	779.87	822.80	946.42	2698.94	1866.83
8192	1123.35	1123.87	1146.46	1200.70	1355.53	4597.47	3613.33

▼ **Table F.31** — Performance on Nvidia Tesla K40c GPU with one *Normal* and one *Linear* weighted checksum and ABFT encoding block size 64 × 64.

Matrix $[n \times n]$	**SEA** (ms)	**PEA 2** (ms)	**PEA 4** (ms)	**PEA 8** (ms)	**PEA 16** (ms)	**CPU I** (ms)	**CPU II** (ms)
512	2.33	2.33	3.12	5.26	10.27	2.33	0.91
1024	5.80	5.62	7.26	10.90	20.06	10.13	6.24
2048	24.50	24.42	28.10	35.45	54.38	70.52	49.84
3072	67.36	67.92	73.50	84.68	115.01	225.95	162.00
4096	144.39	147.20	154.91	171.08	215.77	572.15	391.78
5120	272.19	275.98	285.94	307.49	365.72	1003.48	700.21
6144	454.15	460.72	473.86	502.40	572.05	1733.07	1194.06
7168	701.62	710.42	726.17	760.92	854.05	2698.94	1866.83
8192	1057.90	1055.15	1073.62	1117.49	1234.89	4597.47	3613.33

▼ **Table F.32** — Performance on Nvidia Tesla K40c GPU with one *Normal* and one *Linear* weighted checksum and ABFT encoding block size 128 × 128.

Matrix $[n \times n]$	**SEA** (ms)	**PEA 2** (ms)	**PEA 4** (ms)	**PEA 8** (ms)	**PEA 16** (ms)	**CPU I** (ms)	**CPU II** (ms)
512	2.46	2.46	3.20	5.29	10.08	2.33	0.91
1024	6.08	5.90	7.49	11.00	19.68	10.13	6.24
2048	25.35	25.27	28.79	35.85	53.58	70.52	49.84
3072	67.36	67.85	73.19	83.64	111.05	225.95	162.00
4096	144.17	146.75	153.90	168.63	208.09	572.15	391.78
5120	271.88	275.39	284.66	304.62	356.13	1003.48	700.21
6144	453.15	459.38	471.41	497.05	562.72	1733.07	1194.06
7168	701.98	708.19	722.19	753.34	832.48	2698.94	1866.83
8192	1040.92	1038.82	1056.53	1095.64	1192.21	4597.47	3613.33

▼ **Table F.33** — Performance on Nvidia Tesla K40c GPU with one *Normal* and one *Exponential* weighted checksum and ABFT encoding block size 32×32.

Matrix [$n \times n$]	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.32	2.28	3.06	5.23	10.29	2.33	0.91
1024	5.75	5.74	7.45	11.29	21.57	10.13	6.24
2048	24.29	24.70	28.61	36.63	58.39	70.52	49.84
3072	67.76	68.60	74.59	87.37	123.59	225.95	162.00
4096	151.76	155.13	163.79	182.64	237.63	572.15	391.78
5120	284.18	288.49	299.57	325.40	399.44	1003.48	700.21
6144	479.48	486.74	501.27	535.92	635.03	1733.07	1194.06
7168	747.10	755.89	773.74	816.29	939.69	2698.94	1866.83
8192	1115.66	1115.86	1138.40	1192.57	1347.66	4597.47	3613.33

▼ **Table F.34** — Performance on Nvidia Tesla K40c GPU with one *Normal* and one *Exponential* weighted checksum and ABFT encoding block size 64×64.

Matrix [$n \times n$]	SEA (ms)	PEA 2 (ms)	PEA 4 (ms)	PEA 8 (ms)	PEA 16 (ms)	CPU I (ms)	CPU II (ms)
512	2.35	2.31	3.07	5.21	10.12	2.33	0.91
1024	5.62	5.59	7.23	10.90	20.18	10.13	6.24
2048	24.29	24.24	27.92	35.26	54.29	70.52	49.84
3072	66.96	67.51	73.08	84.18	114.22	225.95	162.00
4096	143.58	146.41	154.10	170.23	214.74	572.15	391.78
5120	271.05	274.67	284.85	306.44	365.60	1003.48	700.21
6144	452.68	459.05	472.18	500.67	577.24	1733.07	1194.06
7168	699.40	708.14	723.84	758.61	851.61	2698.94	1866.83
8192	1041.06	1039.77	1058.28	1102.61	1230.58	4597.47	3613.33

▼ **Table F.35** — Performance on Nvidia Tesla K40c GPU with one *Normal* and one *Exponential* weighted checksum and ABFT encoding block size 128 × 128.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.49	2.44	3.17	5.24	10.06	2.33	0.91
1024	5.89	5.80	7.40	10.91	19.57	10.13	6.24
2048	24.95	24.89	28.41	35.41	53.05	70.52	49.84
3072	66.50	66.99	72.38	82.95	110.72	225.95	162.00
4096	142.59	145.22	152.50	167.49	207.41	572.15	391.78
5120	269.58	273.01	282.29	302.23	353.83	1003.48	700.21
6144	449.81	456.03	468.28	494.37	560.95	1733.07	1194.06
7168	697.40	703.75	717.71	749.15	828.48	2698.94	1866.83
8192	1034.87	1032.79	1050.45	1089.66	1186.26	4597.47	3613.33

▼ **Table F.36** — Performance on Nvidia Tesla K40c GPU with one *Normal* and one *MDR* weighted checksum and ABFT encoding block size 32 × 32.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.31	2.32	3.11	5.32	10.49	2.33	0.91
1024	5.90	5.86	7.57	11.40	21.63	10.13	6.24
2048	24.84	25.61	29.16	37.20	59.09	70.52	49.84
3072	68.91	69.74	75.72	88.51	124.66	225.95	162.00
4096	153.74	157.13	165.83	184.70	239.69	572.15	391.78
5120	282.40	286.79	302.71	328.53	402.44	1003.48	700.21
6144	483.87	491.05	505.75	540.19	639.28	1733.07	1194.06
7168	753.07	761.80	779.93	822.80	946.51	2698.94	1866.83
8192	1123.45	1123.61	1146.37	1200.67	1355.58	4597.47	3613.33

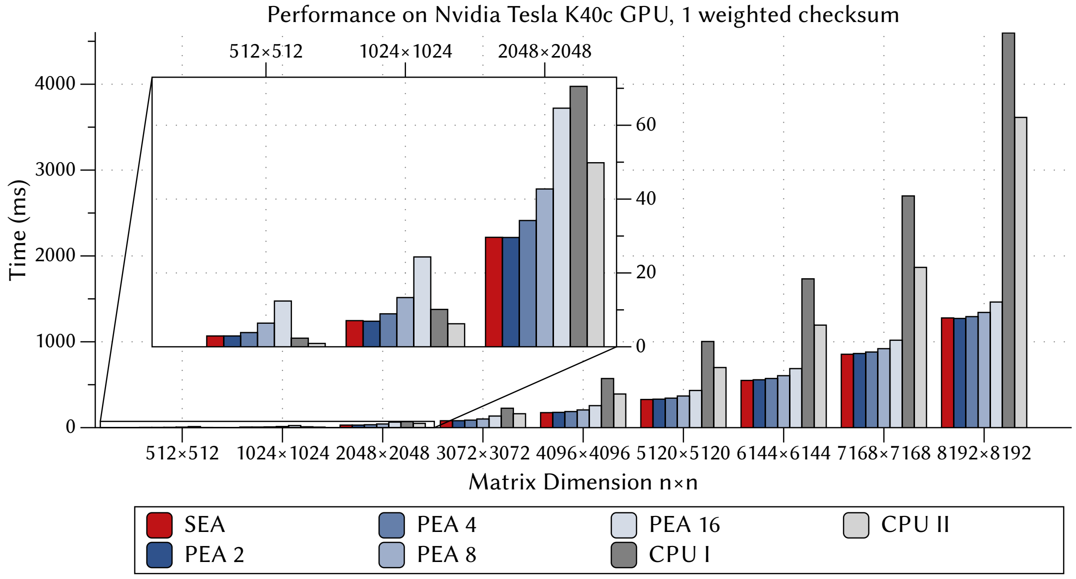
▼ **Table F.37** — Performance on Nvidia Tesla K40c GPU with one *Normal* and one *MDR* weighted checksum and ABFT encoding block size 64 × 64.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.32	2.31	3.07	5.18	10.04	2.33	0.91
1024	5.77	5.63	7.30	10.93	20.35	10.13	6.24
2048	24.47	24.40	28.04	35.24	53.92	70.52	49.84
3072	67.42	67.92	73.48	84.63	114.81	225.95	162.00
4096	144.31	147.11	154.79	170.92	215.53	572.15	391.78
5120	271.98	275.86	285.87	307.25	365.62	1003.48	700.21
6144	446.35	453.58	466.81	495.34	572.65	1733.07	1194.06
7168	701.41	710.29	726.13	761.19	854.93	2698.94	1866.83
8192	1057.79	1054.77	1073.52	1117.50	1238.21	4597.47	3613.33

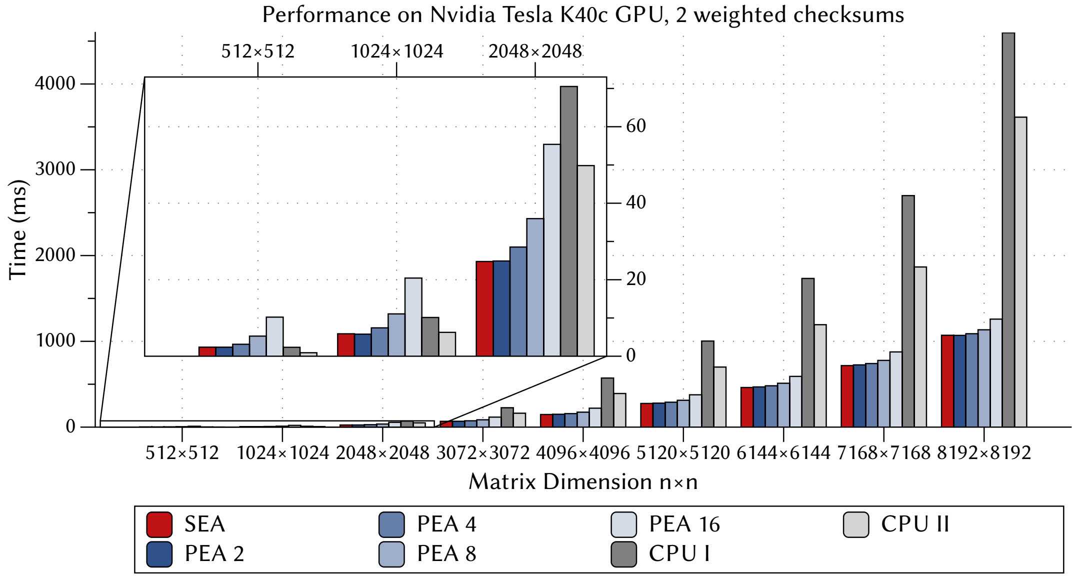
▼ **Table F.38** — Performance on Nvidia Tesla K40c GPU with one *Normal* and one *MDR* weighted checksum and ABFT encoding block size 128 × 128.

Matrix	**SEA**	**PEA 2**	**PEA 4**	**PEA 8**	**PEA 16**	**CPU I**	**CPU II**
$[n \times n]$	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)	(ms)
512	2.46	2.48	3.24	5.38	10.33	2.33	0.91
1024	6.08	5.90	7.49	10.98	19.59	10.13	6.24
2048	25.35	25.27	28.80	35.76	53.29	70.52	49.84
3072	67.34	67.88	73.26	83.81	111.54	225.95	162.00
4096	144.14	146.76	154.04	168.94	208.65	572.15	391.78
5120	271.82	275.27	284.55	304.50	355.90	1003.48	700.21
6144	452.99	459.04	471.18	497.07	562.69	1733.07	1194.06
7168	701.91	708.30	722.46	753.86	833.47	2698.94	1866.83
8192	1040.78	1038.67	1056.40	1095.38	1191.54	4597.47	3613.33

F.3 Overall Computational Performance of A-ABFT

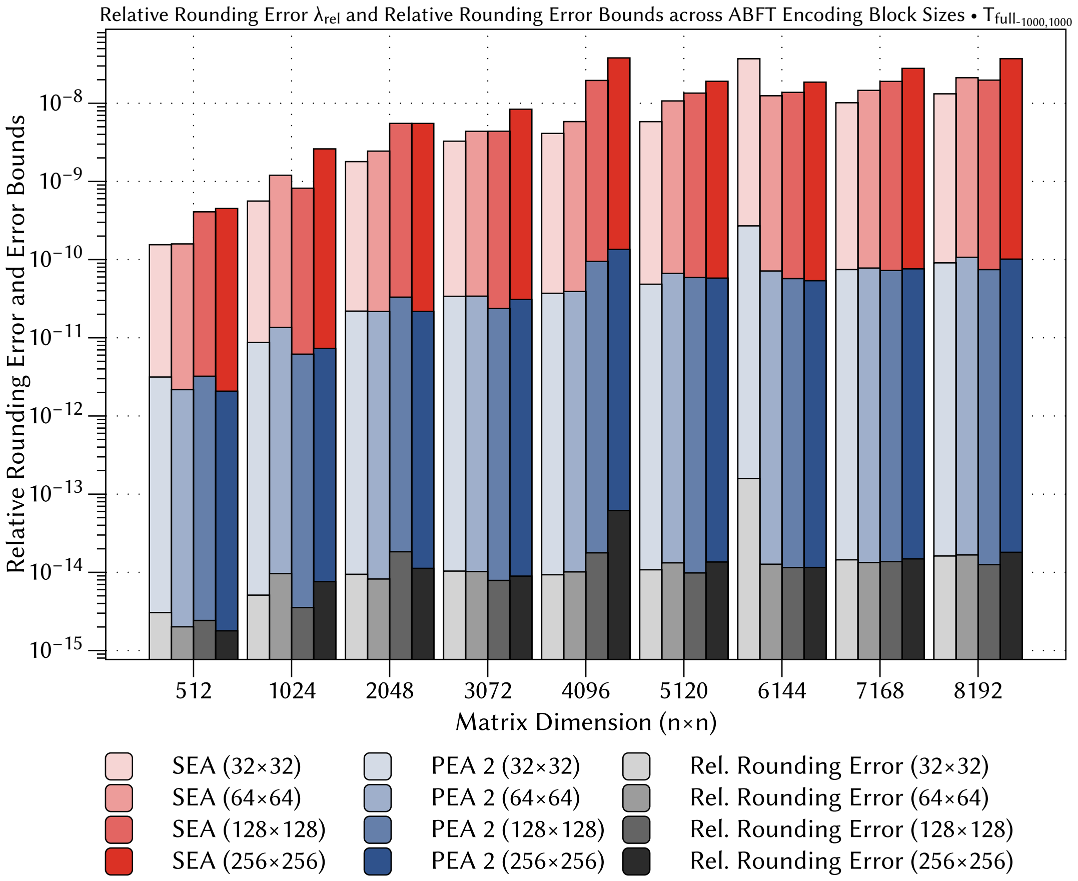


▲ **Figure F.1** — Performance of SEA and PEA with one weighted checksum on NVIDIA TESLA K40c GPU compared to unprotected matrix multiplication on multicore CPUs.

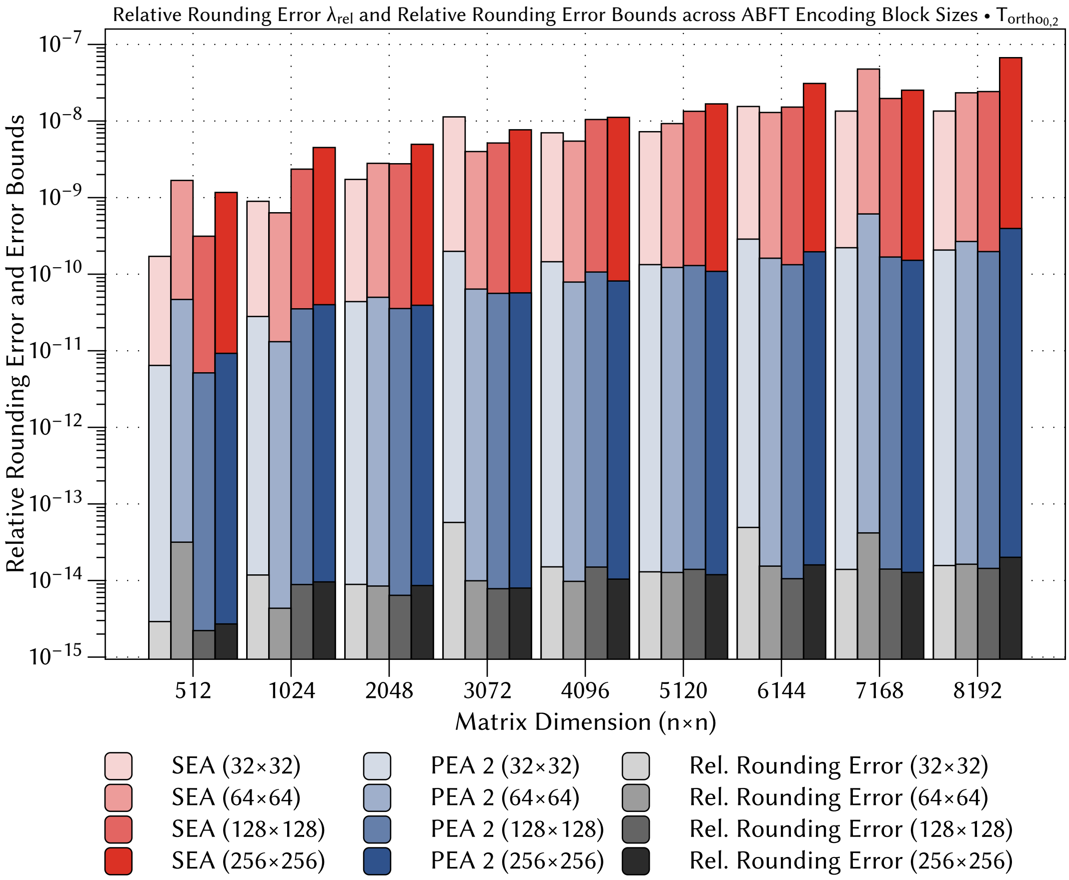


▲ **Figure F.2** — Performance of SEA and PEA with two weighted checksums on NVIDIA TESLA K40c GPU compared to unprotected matrix multiplication on multicore CPUs.

F.4 Quality of Rounding Error Bounds



▲ **Figure F.3** — Average relative rounding error λ_{rel} and determined relative rounding error bounds, test data set $T_{full(-1000,1000)}$.



▲ **Figure F.4** — Average relative rounding error λ_{rel} and determined relative rounding error bounds, test data set $T_{ortho(0,2)}$.

▼ **Table F.39** — Average relative rounding error and determined rounding error bounds for an ABFT encoding block size of 32 × 32 and normal weighted checksums over test data set $T_{pos(0,1000)}$.

Matrix	Rounding	SEA	PEA 2	PEA 4	PEA 8	PEA 16
$[n \times n]$	Error λ_{rel}	ε_{rel}	ε_{rel}	ε_{rel}	ε_{rel}	ε_{rel}
512	$4.53 \cdot 10^{-16}$	$3.02 \cdot 10^{-13}$	$7.99 \cdot 10^{-15}$	$7.96 \cdot 10^{-15}$	$7.90 \cdot 10^{-15}$	$7.78 \cdot 10^{-15}$
1024	$6.39 \cdot 10^{-16}$	$5.85 \cdot 10^{-13}$	$1.16 \cdot 10^{-14}$	$1.16 \cdot 10^{-14}$	$1.15 \cdot 10^{-14}$	$1.14 \cdot 10^{-14}$
2048	$8.99 \cdot 10^{-16}$	$1.15 \cdot 10^{-12}$	$1.66 \cdot 10^{-14}$	$1.66 \cdot 10^{-14}$	$1.65 \cdot 10^{-14}$	$1.65 \cdot 10^{-14}$
3072	$1.26 \cdot 10^{-15}$	$1.72 \cdot 10^{-12}$	$2.04 \cdot 10^{-14}$	$2.04 \cdot 10^{-14}$	$2.04 \cdot 10^{-14}$	$2.03 \cdot 10^{-14}$
4096	$1.27 \cdot 10^{-15}$	$2.29 \cdot 10^{-12}$	$2.37 \cdot 10^{-14}$	$2.37 \cdot 10^{-14}$	$2.37 \cdot 10^{-14}$	$2.36 \cdot 10^{-14}$
5120	$1.63 \cdot 10^{-15}$	$2.85 \cdot 10^{-12}$	$2.66 \cdot 10^{-14}$	$2.66 \cdot 10^{-14}$	$2.66 \cdot 10^{-14}$	$2.65 \cdot 10^{-14}$
6144	$1.78 \cdot 10^{-15}$	$3.42 \cdot 10^{-12}$	$2.93 \cdot 10^{-14}$	$2.93 \cdot 10^{-14}$	$2.93 \cdot 10^{-14}$	$2.92 \cdot 10^{-14}$
7169	$1.81 \cdot 10^{-15}$	$3.99 \cdot 10^{-12}$	$3.17 \cdot 10^{-14}$	$3.17 \cdot 10^{-14}$	$3.17 \cdot 10^{-14}$	$3.17 \cdot 10^{-14}$
8192	$1.80 \cdot 10^{-15}$	$4.55 \cdot 10^{-12}$	$3.40 \cdot 10^{-14}$	$3.40 \cdot 10^{-14}$	$3.39 \cdot 10^{-14}$	$3.39 \cdot 10^{-14}$

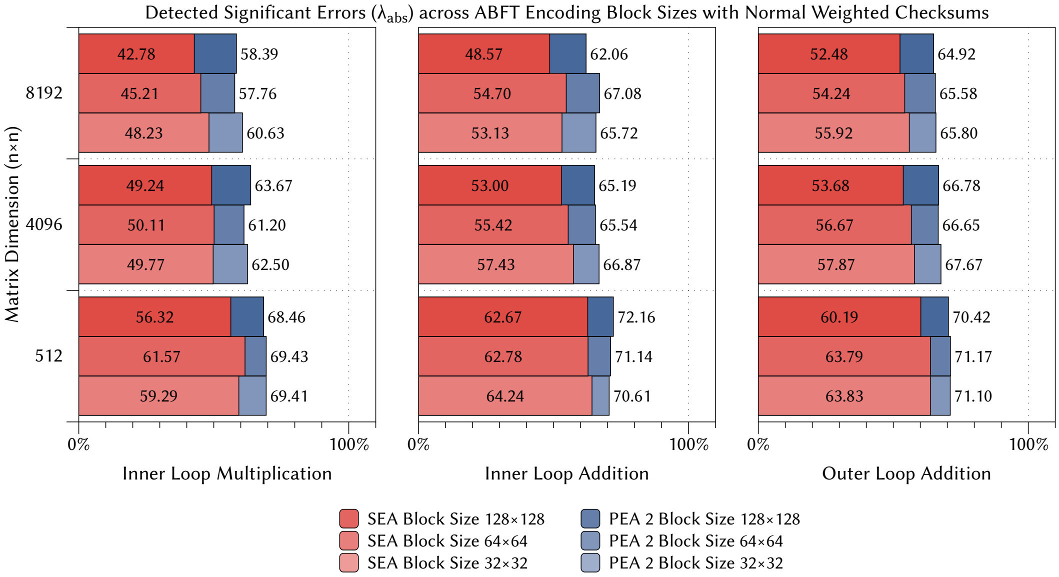
▼ **Table F.40** — Average relative rounding error and determined rounding error bounds for an ABFT encoding block size of 32 × 32 and normal weighted checksums over test data set $T_{full(-1000,1000)}$.

Matrix	Rounding	SEA	PEA 2	PEA 4	PEA 8	PEA 16
$[n \times n]$	Error λ_{rel}	ε_{rel}	ε_{rel}	ε_{rel}	ε_{rel}	ε_{rel}
512	$3.06 \cdot 10^{-15}$	$1.55 \cdot 10^{-10}$	$3.15 \cdot 10^{-12}$	$3.14 \cdot 10^{-12}$	$3.12 \cdot 10^{-12}$	$3.07 \cdot 10^{-12}$
1024	$5.11 \cdot 10^{-15}$	$5.61 \cdot 10^{-10}$	$8.73 \cdot 10^{-12}$	$8.70 \cdot 10^{-12}$	$8.67 \cdot 10^{-12}$	$8.60 \cdot 10^{-12}$
2048	$9.44 \cdot 10^{-15}$	$1.79 \cdot 10^{-9}$	$2.19 \cdot 10^{-11}$	$2.19 \cdot 10^{-11}$	$2.19 \cdot 10^{-11}$	$2.18 \cdot 10^{-11}$
3072	$1.04 \cdot 10^{-14}$	$3.27 \cdot 10^{-9}$	$3.40 \cdot 10^{-11}$	$3.39 \cdot 10^{-11}$	$3.39 \cdot 10^{-11}$	$3.38 \cdot 10^{-11}$
4096	$9.31 \cdot 10^{-15}$	$4.11 \cdot 10^{-9}$	$3.71 \cdot 10^{-11}$	$3.71 \cdot 10^{-11}$	$3.71 \cdot 10^{-11}$	$3.70 \cdot 10^{-11}$
5120	$1.08 \cdot 10^{-14}$	$5.82 \cdot 10^{-9}$	$4.84 \cdot 10^{-11}$	$4.84 \cdot 10^{-11}$	$4.84 \cdot 10^{-11}$	$4.83 \cdot 10^{-11}$
6144	$1.58 \cdot 10^{-13}$	$3.71 \cdot 10^{-8}$	$2.70 \cdot 10^{-10}$	$2.70 \cdot 10^{-10}$	$2.70 \cdot 10^{-10}$	$2.69 \cdot 10^{-10}$
7169	$1.45 \cdot 10^{-14}$	$1.02 \cdot 10^{-8}$	$7.46 \cdot 10^{-11}$	$7.45 \cdot 10^{-11}$	$7.45 \cdot 10^{-11}$	$7.44 \cdot 10^{-11}$
8192	$1.62 \cdot 10^{-14}$	$1.32 \cdot 10^{-8}$	$9.09 \cdot 10^{-11}$	$9.09 \cdot 10^{-11}$	$9.08 \cdot 10^{-11}$	$9.07 \cdot 10^{-11}$

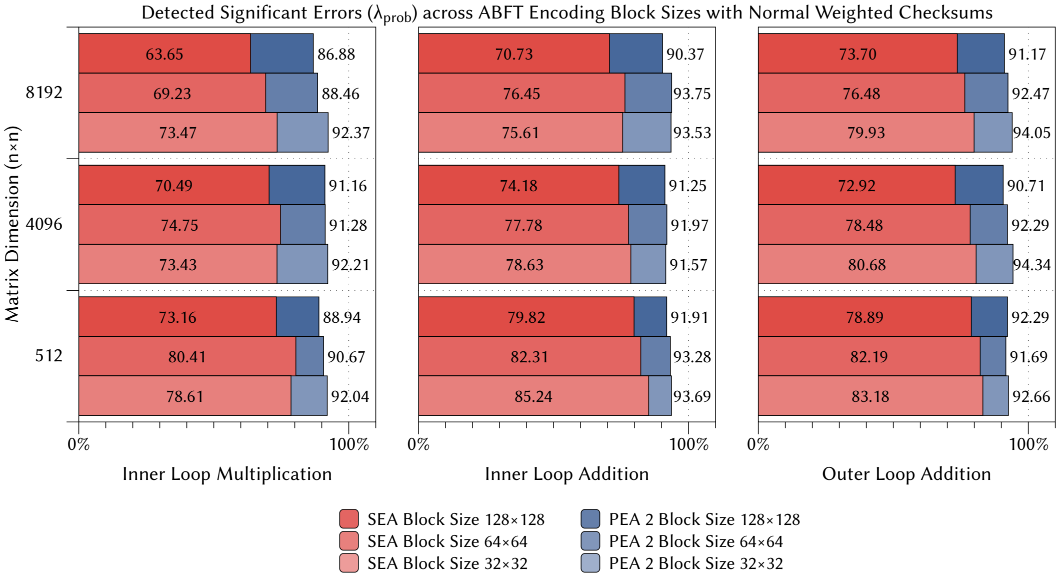
▼ **Table F.41** — Average relative rounding error and determined rounding error bounds for an ABFT encoding block size of 32 × 32 and normal weighted checksums over test data set $T_{ortho(0,2)}$.

Matrix $[n \times n]$	**Rounding Error** λ_{rel}	**SEA** ε_{rel}	**PEA 2** ε_{rel}	**PEA 4** ε_{rel}	**PEA 8** ε_{rel}	**PEA 16** ε_{rel}
512	$2.91 \cdot 10^{-15}$	$1.71 \cdot 10^{-10}$	$6.44 \cdot 10^{-12}$	$5.94 \cdot 10^{-12}$	$5.40 \cdot 10^{-12}$	$4.79 \cdot 10^{-12}$
1024	$1.18 \cdot 10^{-14}$	$8.95 \cdot 10^{-10}$	$2.80 \cdot 10^{-11}$	$2.51 \cdot 10^{-11}$	$2.32 \cdot 10^{-11}$	$2.10 \cdot 10^{-11}$
2048	$8.90 \cdot 10^{-15}$	$1.73 \cdot 10^{-9}$	$4.38 \cdot 10^{-11}$	$4.09 \cdot 10^{-11}$	$3.80 \cdot 10^{-11}$	$3.49 \cdot 10^{-11}$
3072	$5.73 \cdot 10^{-14}$	$1.13 \cdot 10^{-8}$	$1.99 \cdot 10^{-10}$	$1.89 \cdot 10^{-10}$	$1.79 \cdot 10^{-10}$	$1.68 \cdot 10^{-10}$
4096	$1.51 \cdot 10^{-14}$	$7.02 \cdot 10^{-9}$	$1.45 \cdot 10^{-10}$	$1.37 \cdot 10^{-10}$	$1.27 \cdot 10^{-10}$	$1.19 \cdot 10^{-10}$
5120	$1.30 \cdot 10^{-14}$	$7.25 \cdot 10^{-9}$	$1.33 \cdot 10^{-10}$	$1.26 \cdot 10^{-10}$	$1.19 \cdot 10^{-10}$	$1.11 \cdot 10^{-10}$
6144	$4.93 \cdot 10^{-14}$	$1.55 \cdot 10^{-8}$	$2.87 \cdot 10^{-10}$	$2.62 \cdot 10^{-10}$	$2.48 \cdot 10^{-10}$	$2.33 \cdot 10^{-10}$
7169	$1.40 \cdot 10^{-14}$	$1.35 \cdot 10^{-8}$	$2.22 \cdot 10^{-10}$	$2.08 \cdot 10^{-10}$	$1.96 \cdot 10^{-10}$	$1.82 \cdot 10^{-10}$
8192	$1.57 \cdot 10^{-14}$	$1.35 \cdot 10^{-8}$	$2.07 \cdot 10^{-10}$	$1.95 \cdot 10^{-10}$	$1.85 \cdot 10^{-10}$	$1.74 \cdot 10^{-10}$

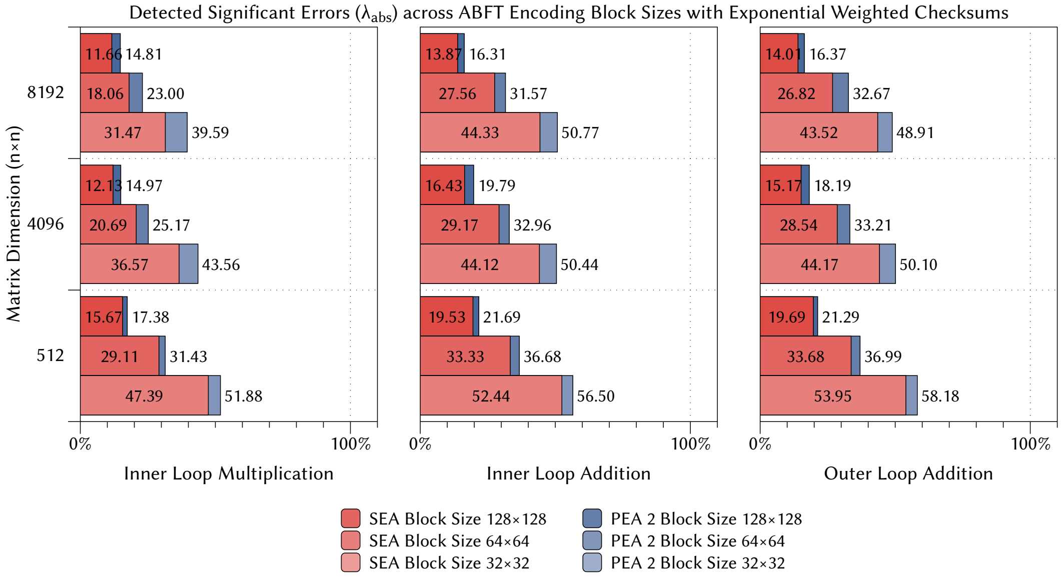
F.5 Error Detection



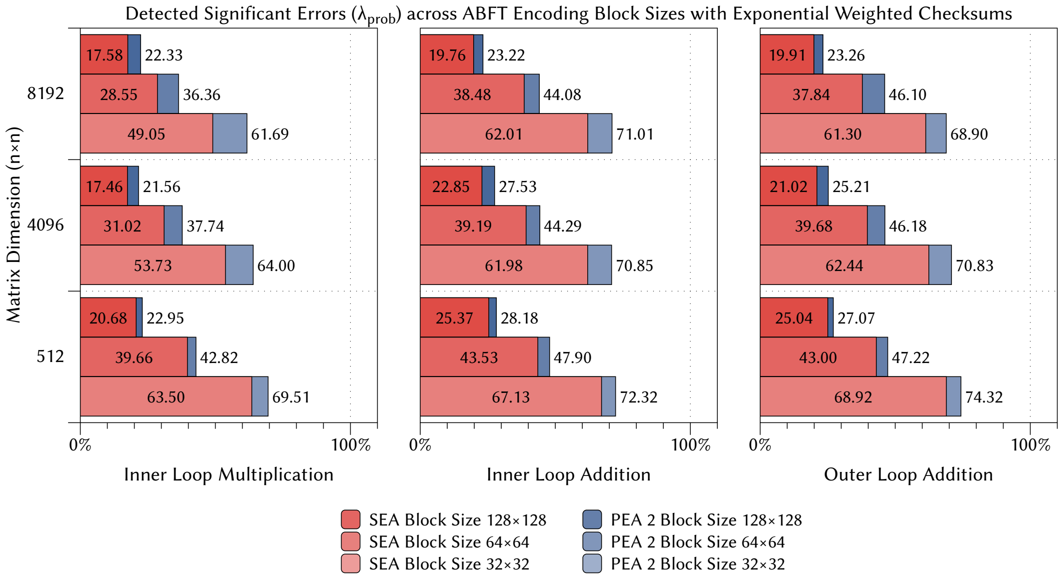
▲ **Figure F.5** — Detected significant errors (λ_{abs}) across different ABFT encoding block sizes for normal weighted checksums and test data set $T_{ortho(0,65536)}$.



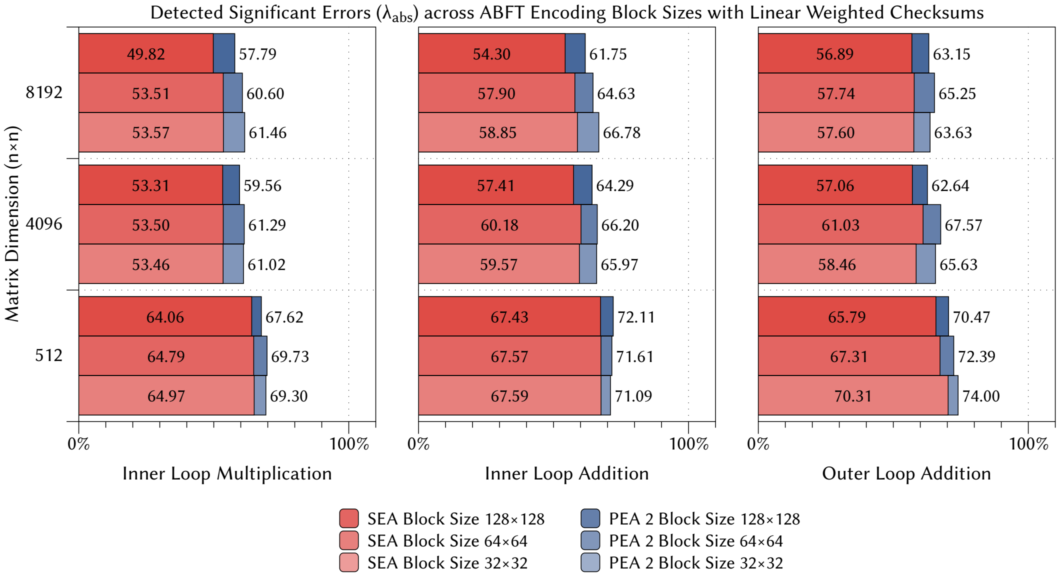
▲ **Figure F.6** — Detected significant errors (λ_{prob}) across different ABFT encoding block sizes for normal weighted checksums and test data set $T_{ortho(0,65536)}$.



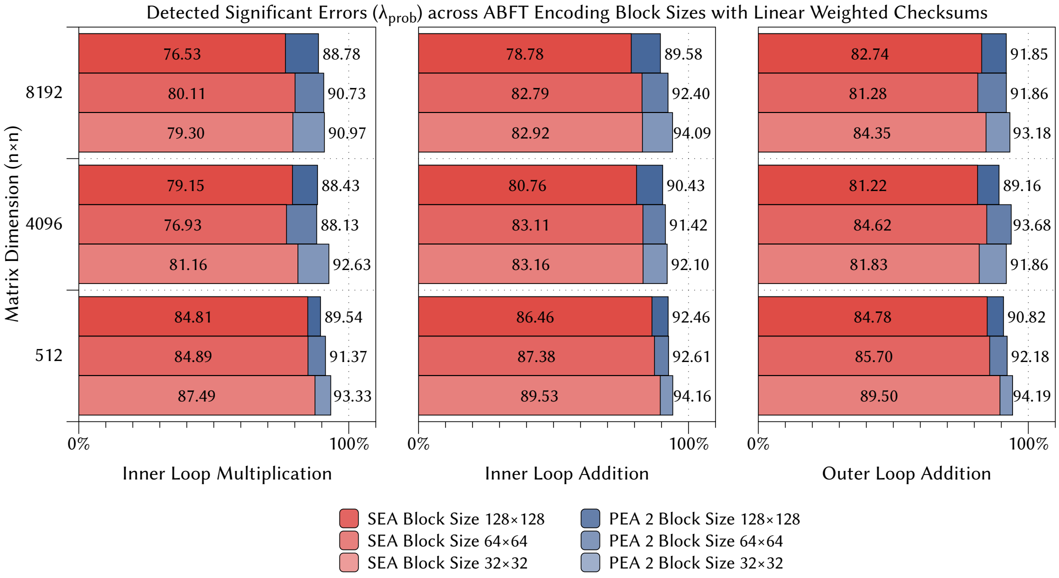
▲ **Figure F.7** — Detected significant errors (λ_{abs}) across different ABFT encoding block sizes for exponential weighted checksums and test data set $T_{ortho(0,65536)}$.



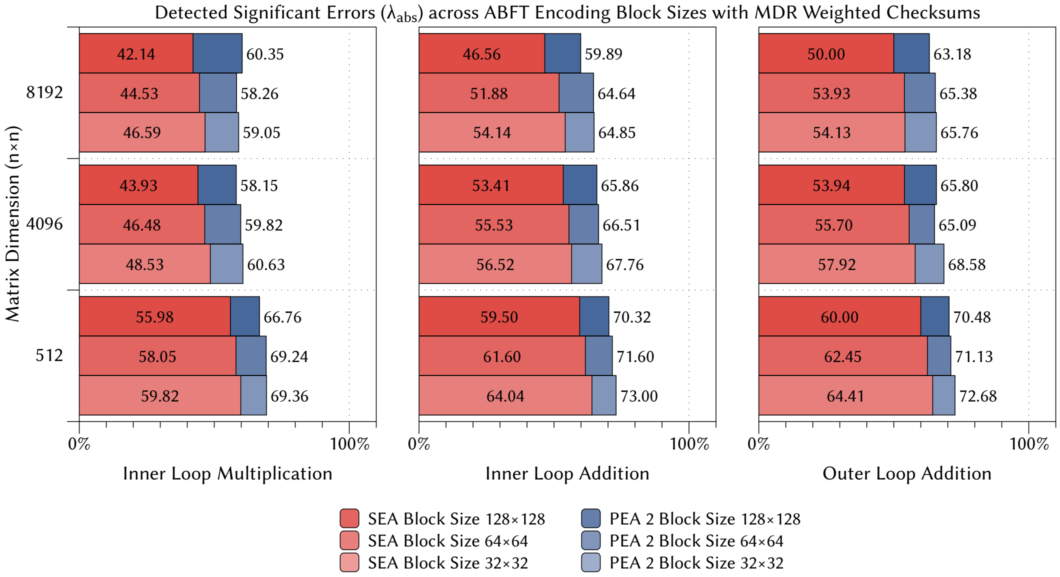
▲ **Figure F.8** — Detected significant errors (λ_{prob}) across different ABFT encoding block sizes for exponential weighted checksums and test data set $T_{ortho(0,65536)}$.



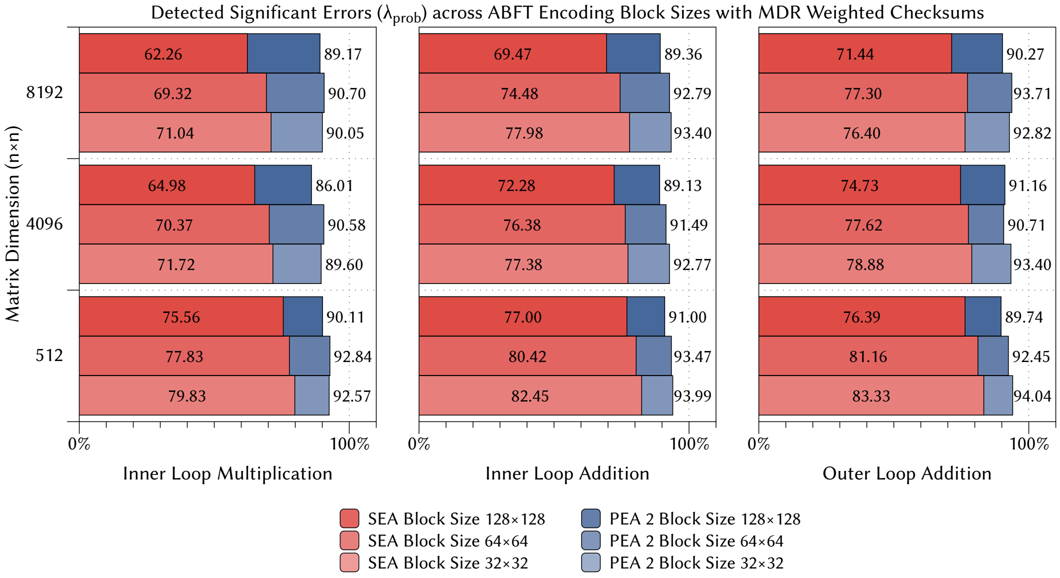
▲ **Figure F.9** — Detected significant errors (λ_{abs}) across different ABFT encoding block sizes for linear weighted checksums and test data set $T_{ortho(0,65536)}$.



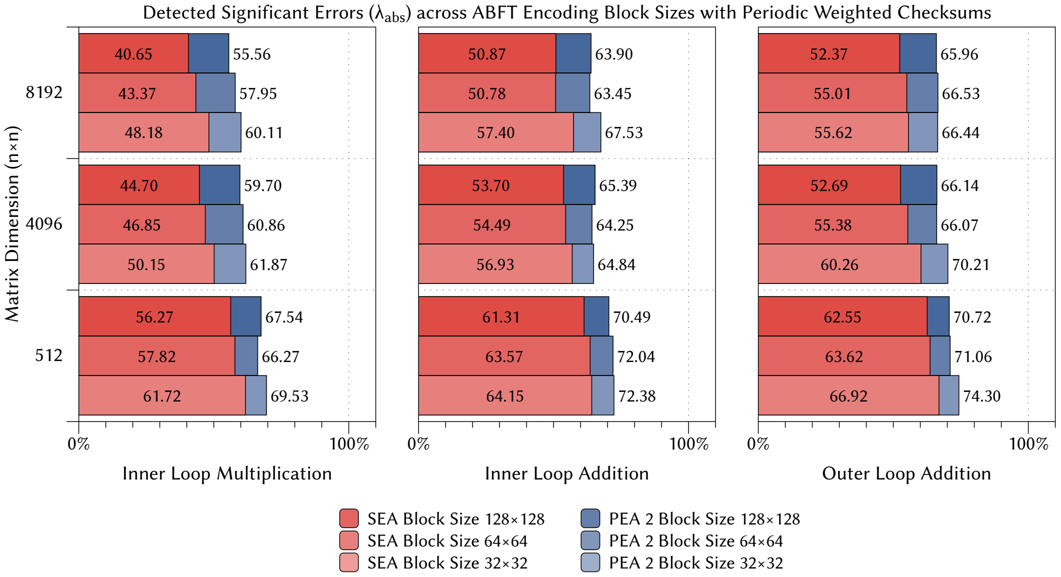
▲ **Figure F.10** — Detected significant errors (λ_{prob}) across different ABFT encoding block sizes for linear weighted checksums and test data set $T_{ortho(0,65536)}$.



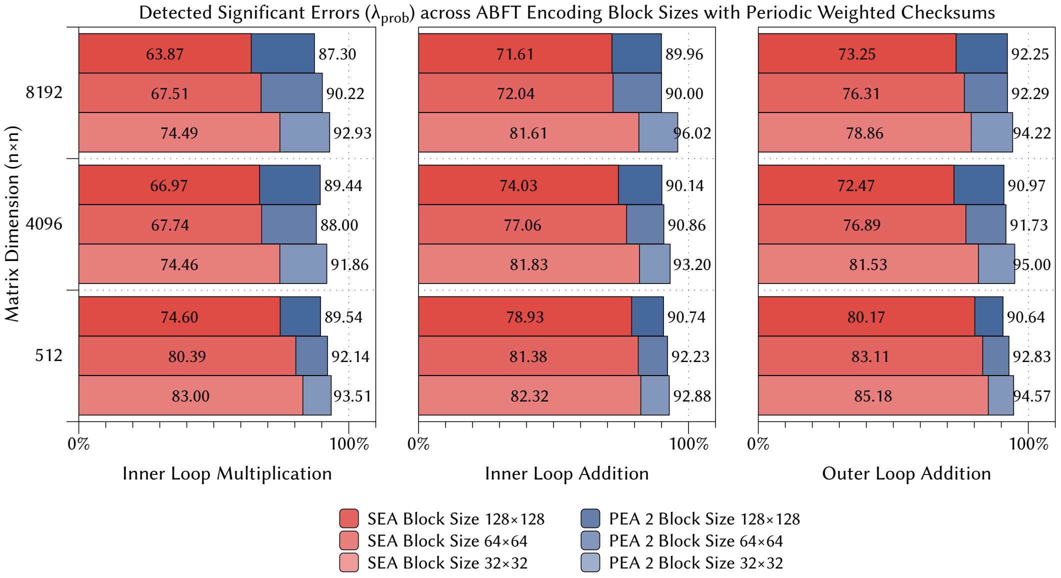
▲ **Figure F.11** — Detected significant errors (λ_{abs}) across different ABFT encoding block sizes for MDR weighted checksums and test data set $T_{ortho(0,65536)}$.



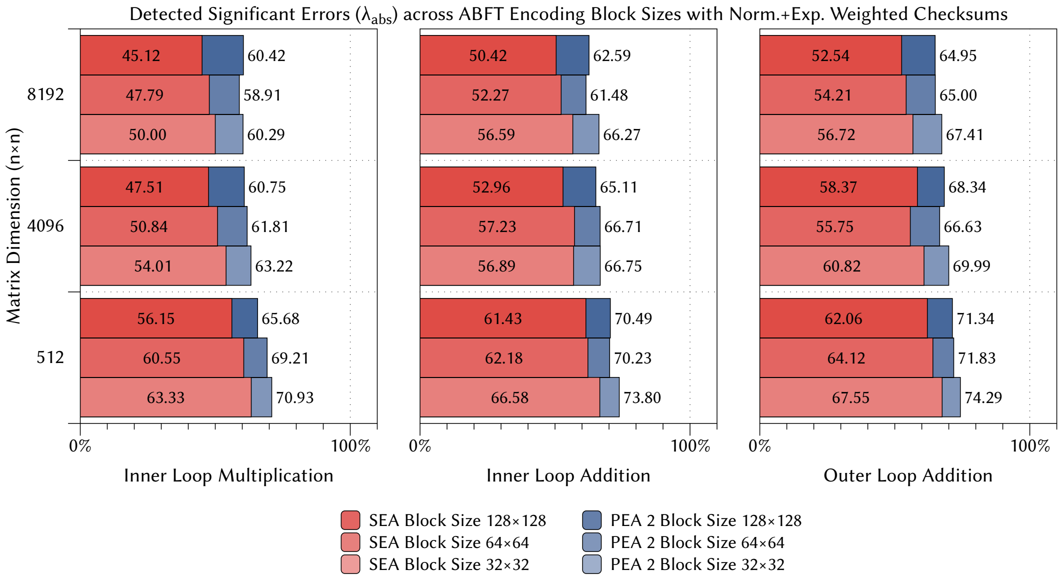
▲ **Figure F.12** — Detected significant errors (λ_{prob}) across different ABFT encoding block sizes for MDR weighted checksums and test data set $T_{ortho(0,65536)}$.



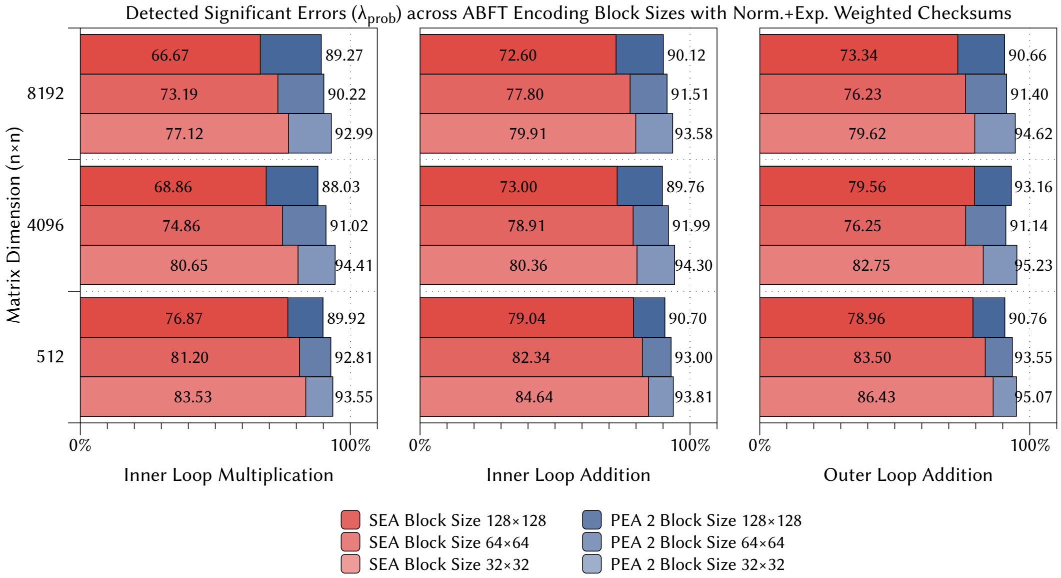
▲ **Figure F.13** — Detected significant errors (λ_{abs}) across different ABFT encoding block sizes for periodic weighted checksums and test data set $T_{ortho(0,65536)}$.



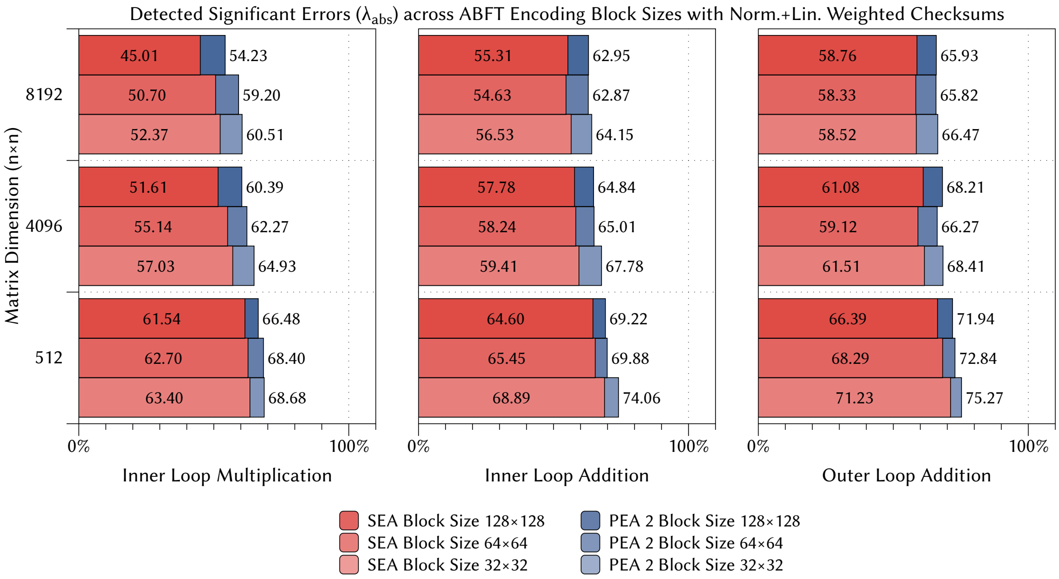
▲ **Figure F.14** — Detected significant errors (λ_{prob}) across different ABFT encoding block sizes for periodic weighted checksums and test data set $T_{ortho(0,65536)}$.



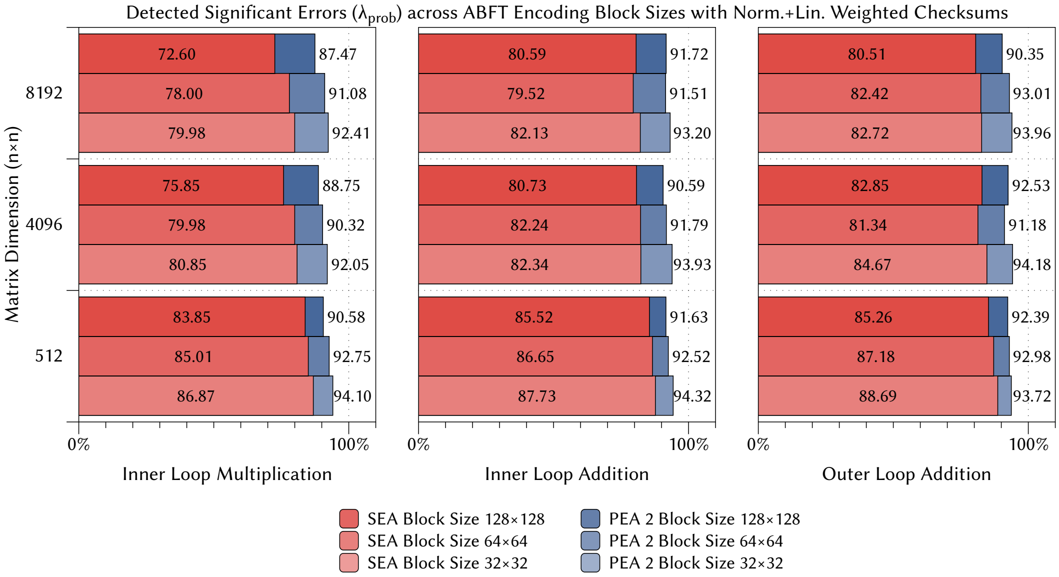
▲ **Figure F.15** — Detected significant errors (λ_{abs}) across different ABFT encoding block sizes for normal + exponential weighted checksums and test data set $T_{ortho(0,65536)}$.



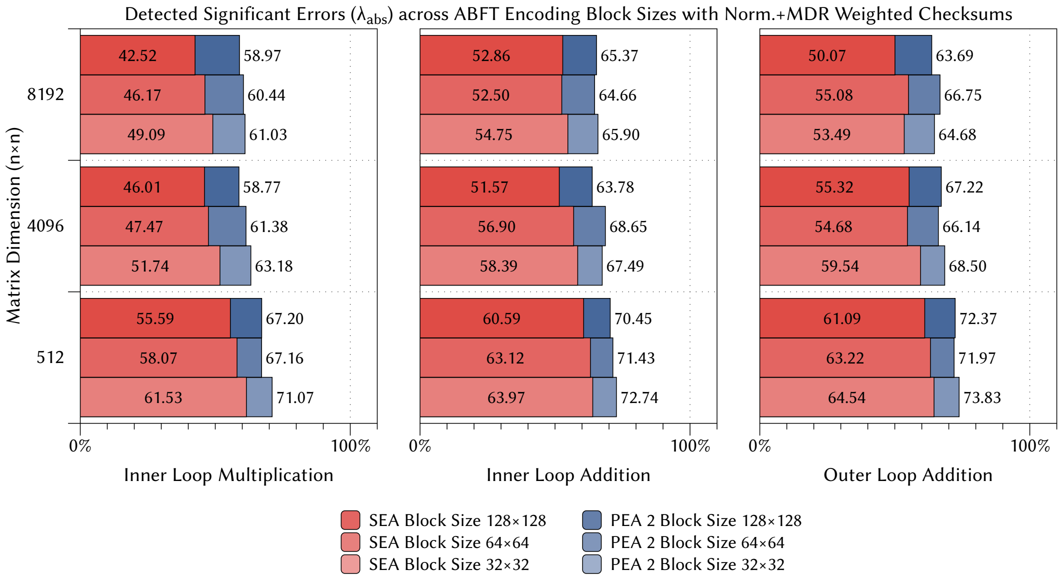
▲ **Figure F.16** — Detected significant errors (λ_{prob}) across different ABFT encoding block sizes for normal + exponential weighted checksums and test data set $T_{ortho(0,65536)}$.



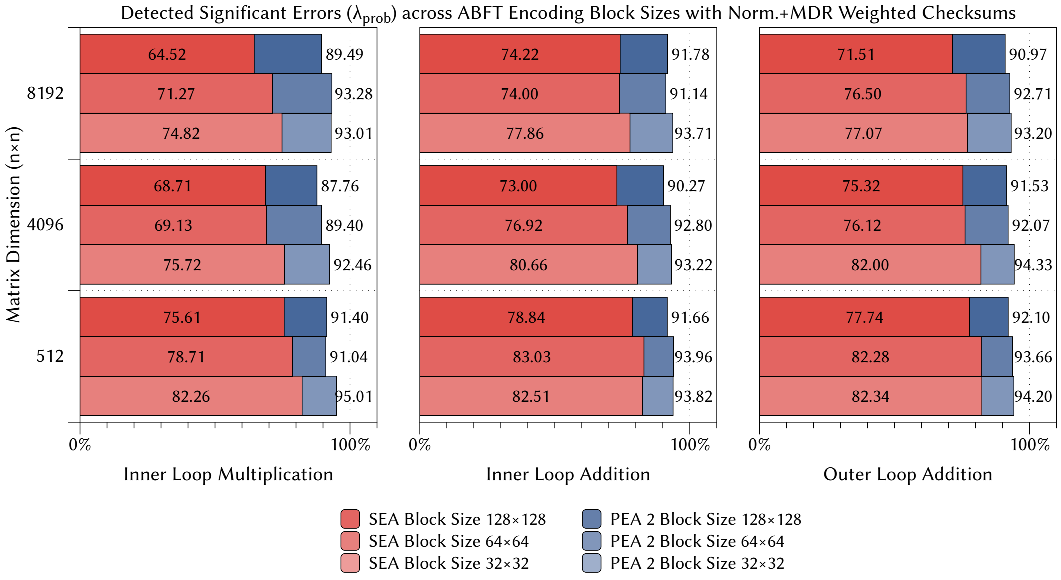
▲ **Figure F.17** — Detected significant errors (λ_{abs}) across different ABFT encoding block sizes for normal + linear weighted checksums and test data set $T_{ortho(0,65536)}$.



▲ **Figure F.18** — Detected significant errors (λ_{prob}) across different ABFT encoding block sizes for normal + linear weighted checksums and test data set $T_{ortho(0,65536)}$.



▲ **Figure F.19** — Detected significant errors (λ_{abs}) across different ABFT encoding block sizes for normal + MDR weighted checksums and test data set $T_{ortho(0,65536)}$.



▲ **Figure F.20** — Detected significant errors (λ_{prob}) across different ABFT encoding block sizes for normal + MDR weighted checksums and test data set $T_{ortho(0,65536)}$.

Index

Curriculum Vitae of the Author

Claus Braun received a Diploma degree in Computer Science (Diplom-Informatik) from the Eberhard Karls Universität Tübingen, Germany in 2008. In the same year, he joined the Institute of Computer Architecture and Computer Engineering (Institut für Technische Informatik, ITI) at the University of Stuttgart, Germany.

The author has been working as a research and teaching assistant under the supervision of Prof. Dr. rer. nat. habil. Hans-Joachim Wunderlich and supported the graduate course ”Advanced Processor Architecture”. He also supervised students in several seminars, Master and Diploma theses, and undergraduate projects.

With the *Stuttgart Research Centre for Simulation Technology* (SRC SimTech), the author was part of the *DFG Cluster of Excellence in Simulation Technology* (ExC SimTech) and the Graduate School SimTech. He has been involved in the research projects "*SimTech I: Mapping Simulation Algorithms to NoC-MPSoC Computers*", "*SimTech II: Simulation on Reconfigurable Heterogeneous Computer Architectures*", "*OTERA: Online Test Strategies for Reliable Reconfigurable Architectures (DFG Priority Project SPP-1500)*", and "*PARSIVAL: Parallel High-Throughput Simulations for Efficient Nanoelectronic Design and Test Validation*", which were all supported by the German Research Foundation (DFG).

His research interests include algorithm- and software-based fault tolerance, hardware reliability, parallel many-core and runtime-reconfigurable heterogeneous computer architectures, as well as computer-based simulation technology.

Publications of the Author

- A. Schöll, C. Braun, M.A. Kochte and H.-J. Wunderlich. Efficient On-Line Fault-Tolerance for the Preconditioned Conjugate Gradient Method. *to appear in Proceedings of the IEEE International On-Line Testing Symposium (IOLTS'15)*, Halkidiki, Greece, 6-8 July 2015.

- A. Schöll, C. Braun, M. Daub, G. Schneider and H.-J. Wunderlich. Adaptive Parallel Simulation of a Two-Timescale-Model for Apoptotic Receptor-Clustering on GPUs. *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine (BIBM'14)*, Belfast, United Kingdom, 2-5 November 2014, pp. 424-431.

- C. Braun, S. Halder and H.-J. Wunderlich. A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units. *Proceedings of The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, Atlanta, Georgia, USA, 23-26 June 2014, pp. 443-454.

- C. Braun, S. Halder and H.-J. Wunderlich. A-ABFT: Autonomous Algorithm-Based Fault Tolerance on GPUs. *Proceedings of the International Workshop on Dependable GPU Computing, in conjunction with the ACM/IEEE DATE'14 Conference*, Dresden, Germany, 28 March 2014.

- H.-J. Wunderlich, C. Braun and S. Halder. Efficacy and Efficiency of Algorithm-Based Fault Tolerance on GPUs. *Proceedings of the IEEE International On-Line Testing Symposium (IOLTS'13)*, Crete, Greece, 8-10 July 2013, pp. 240-243.

- H. Zhang, L. Bauer, M.A. Kochte, E. Schneider, C. Braun, M.E. Imhof, H.-J. Wunderlich and J. Henkel. Module Diversification: Fault Tolerance and Aging Mitigation for Runtime Reconfigurable Architectures. *Proceedings of the IEEE International Test Conference (ITC'13)*, Anaheim, California, USA, 10-12 September 2013, pp. 1-10.

- L. Bauer, C. Braun, M.E. Imhof, M.A. Kochte, E. Schneider, H. Zhang, J. Henkel and H.-J. Wunderlich. Test Strategies for Reliable Runtime Reconfigurable Architectures. *IEEE Transactions on Computers*, Vol. 62(8), Los Alamitos, California, USA August 2013, pp. 1494-1507.

- C. Braun, M. Daub, A. Schöll, G. Schneider and H.-J. Wunderlich. Parallel Simulation of Apoptotic Receptor-Clustering on GPGPU Many-Core Architectures. *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine (BIBM'12)*, Philadelphia, Pennsylvania, USA, 4-7 October 2012, pp. 1-6.

- C. Braun, S. Holst, J. M. Castillo, J. Gross and H.-J. Wunderlich. Acceleration of Monte-Carlo Molecular Simulations on Hybrid Computing Architectures. *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD'12)*, Montreal, Canada, 30 September-3 October 2012, pp. 207-212.

- M.S. Abdelfattah, L. Bauer, C. Braun, M.E. Imhof, M.A. Kochte, H. Zhang, J. Henkel and H.-J. Wunderlich. Transparent Structural Online Test for Reconfigurable Systems. *Proceedings of the 18th IEEE International On-Line Testing Symposium (IOLTS'12)*, Sitges, Spain, 27-29 June 2012, pp. 37-42.

- L. Bauer, C. Braun, M.E. Imhof, M.A. Kochte, H. Zhang, H.-J. Wunderlich and J. Henkel. OTERA: Online Test Strategies for Reliable Reconfigurable Architectures. *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS'12)*, Erlangen, Germany, 25-28 June 2012, pp. 38-45.

- C. Braun and H.-J. Wunderlich. Algorithmen-basierte Fehlertoleranz für Many-Core-Architekturen (Algorithm-based Fault-Tolerance on Many-Core Architectures). *it - Information Technology*, Vol. 52(4), August 2010, pp. 209-215.

- C. Braun and H.-J. Wunderlich. Algorithm-based fault tolerance for many-core architectures. *Proceedings of the 15th IEEE European Test Symposium (ETS'10)*, Praha, Czech Republic, 24-28 May 2010, pp. 253.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

Claus Braun