Institut für Technische Informatik
Universität Stuttgart
Pfaffenwaldring 47
D–70569 Stuttgart

Diplomarbeit Nr. 3576

Integration von algorithmenbasierter Fehlertoleranz in grundlegende Operationen der linearen Algebra auf GPGPUs

Sebastian Halder

Studiengang:	Informatik
Prüfer:	Prof. Dr. Hans-Joachim Wunderlich
Betreuer:	Dipl.-Inform. Claus Braun, Dipl.-Inf. Alexander Schöll
begonnen am:	16. Oktober 2013
beendet am:	17. April 2014
CR-Klassifikation:	B.8.1, C.1.2, G.1.3

Kurzfassung

Der Einsatz algorithmenbasierter Fehlertoleranz bietet eine Möglichkeit, auftretende Fehler bei Operationen der linearen Algebra zu erkennen, zu lokalisieren und zu korrigieren. Diese Operationen der linearen Algebra können durch den Einsatz hochoptimierter Bibliotheken mit einem großen Geschwindigkeitszuwachs gegenüber Mehrkernprozessoren auf GPGPUs ausgeführt werden. Die Integration der algorithmenbasierten Fehlertoleranz unter Verwendung dieser Bibliotheken für einige ausgewählte Operationen der linearen Algebra ist Kern dieser Arbeit.

Bei der Überprüfung der Ergebnisse bezüglich aufgetretener Fehler müssen dabei Werte verglichen werden, die durch einen Rundungsfehler behaftet sind und somit nicht mit einem Test auf Gleichheit abgeprüft werden können. Deshalb werden Fehlerschwellwerte benötigt, bei deren Überschreitung ein Fehler erkannt und anschließend korrigiert werden kann.

In dieser Arbeit wurden deterministische Methoden zur Fehlerschwellwertbestimmung untersucht und eine auf einer probabilistische Methode zur Abschätzung des Rundungsfehlers basierende Methode zur Fehlerschwellwertbestimmung angepasst und weiterentwickelt. Diese Methoden zur Fehlerschwellwertbestimmung wurden anhand experimenteller Untersuchungen bezüglich der Qualität im Sinne der Differenz zum gemessenen Rundungsfehler, der Fehlererkennungsraten bei Fehlerinjektion und der Performanz der Methoden bei Implementierung auf GPGPUs miteinander verglichen. Die probabilistische Methode zeichnet sich dabei durch einen näher am auftretenden Rundungsfehler liegenden Fehlerschwellwert aus, ist dadurch in der Lage einen größeren Anteil auftretender Fehler zu erkennen und zeigt eine hohe Performanz bei der Verwendung auf GPGPUs.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1. Einleitung

Moderne Anwendungen des wissenschaftlichen Rechnens und der numerischen Simulation von Systemen nehmen heutzutage einen großen Stellenwert in Wissenschaft und Industrie ein. Immer mehr praktische Experimente werden durch Simulationen ersetzt, da diese schneller günstigere und meist besser verständlichere Ergebnisse produzieren [1].

Wird beispielsweise ein Kollisionsversuch von Fahrzeugen in der Realität durchgeführt, dann muss enormer Aufwand betrieben werden um den zeitlichen Ablauf der Kollision festzuhalten: Hochgeschwindigkeitskameras erfassen äußere Auswirkungen der Kollision, Sensoren ermöglichen eine punktuelle Messung von auf das Material wirkenden Kräften. Aus diesen Informationen muss im Nachhinein der Ablauf der Kollision für die nicht sichtbaren Teile rekonstruiert werden. Wird eine modellbasierte Computersimulation für das gleiche Experiment verwendet, dann können für jedes Teil des Modells zu jedem Zeitpunkt die exakt wirkenden Kräfte betrachtet werden. Sollen nun weitere Experimente mit leicht veränderten Parametern durchgeführt werden, so lässt sich erkennen, dass der finanzielle Aufwand der Simulation asymptotisch kleiner ist als der des realen Experiments: Für die Simulation fallen lediglich Fixkosten, wie zum Beispiel Personal- und Stromkosten an, für das reale Experiment muss zusätzlich ein zweites Testfahrzeug zur Verfügung gestellt werden. Manche Experimente sind darüber hinaus aufgrund ihrer räumlichen oder zeitlichen Auflösung ohne die Hilfe von Simulationen gar nicht untersuchbar. In diese Kategorie fallen beispielsweise weitreichende Simulationen von Klimamodellen, sowie Simulationen im Bereich der Astrophysik oder der Quantenmechanik [2].

Diese Anwendungen des wissenschaftlichen Rechnens stellen hohe Anforderungen an die eingesetzten Rechnersysteme und Algorithmen. Immer größere und komplexere Systeme müssen simuliert werden, entsprechend steigt der Speicherbedarf und die Nachfrage nach Rechenkapazität. Diese Nachfrage kann meist nur durch große Rechnerverbünde beantwortet werden, bei denen viele Rechenknoten über eine entsprechende Netzinfrastruktur miteinander verbunden werden. Trotz der großen Zahl der eingesetzten Knoten benötigen viele Applikationen dennoch Stunden, Tage oder sogar Wochen, bis das endgültige Ergebnis verfügbar ist. Eine weitere Anforderung stellt die Verlässlichkeit bezüglich der Korrektheit der berechneten Ergebnisse dar. Abweichungen vom korrekten Ergebnis können im schlimmsten Fall zu falschen politischen oder ökonomischen Entscheidungen führen, zumindest aber zu einem finanziellen Schaden durch die Neuberechnung des Ergebnisses.

Durch den Einsatz programmierbarer Grafikprozessoren (GPUs) ist es möglich, rechenintensive Anwendungen des wissenschaftlichen Rechnens in annehmbarer Zeit auch auf kleineren, dezentralen Systemen durchzuführen. Dies erlaubt es Anwendern aus der Wissenschaft und Industrie, Simulationen und Berechnungen mit einem niedrigeren Kostenaufwand auf

lokalen Rechnersystemen durchzuführen und zu evaluieren. Darüber hinaus werden Grafikprozessoren zur Beschleunigung bestimmter Teilaufgaben in großen Rechnerverbünden eingesetzt und werden in Zukunft auch in eingebetteten Systemen vermehrt Anwendung finden. Die hierbei eingesetzten Grafikprozessoren werden jedoch primär für den Spiele- und Multimediabereich entwickelt, bei dem andere Anforderungen an die Rechnerarchitektur gestellt werden als im Bereich des *High-Performance Computings* (HPC). Hierbei steht nicht die Absicherung gegen mögliche Hard- und Softwarefehler und eine hohe Verfügbarkeit im Vordergrund, sondern das Erreichen eines möglichst hohen Datendurchsatzes. Die für das wissenschaftliche Rechnen verwendeten Prozessoren entstammen dabei meist der gleichen Produktionsserie wie die Prozessoren für den Spielebereich, werden jedoch aufwendiger getestet und mit anderen Komponenten und einer meist geringeren Taktfrequenz für den Einsatz im wissenschaftlichen Rechnen ausgelegt. In Folge dessen kann die verwendete Hardware nur begrenzt mit hardwarebasierten Fehlertoleranzmethoden angepasst werden, um die im Bereich des HPC geforderte Zuverlässigkeit zu erreichen.

Es bietet sich daher an, softwarebasierte Methoden zur Fehlertoleranz einzusetzen um dennoch einen höheren Grad der Zuverlässigkeit zu erreichen. Die Herausforderung besteht dabei darin, diese Ideen aus der softwarebasierten Fehlertoleranz auf die durch den Einsatz in GPUs entstehenden Anforderungen anzupassen. Besonders interessant ist dieses Vorgehen für Algorithmen und Funktionen, die einen signifikanten Geschwindigkeitszuwachs durch den Einsatz auf GPUs erfahren und in einem weiten Bereich von Anwendungen eingesetzt werden. Ein Beispiel hierfür sind Funktionen aus der linearen Algebra, wie die Multiplikation, die Addition und die Transposition von Matrizen. Hierfür existieren hochoptimierte Bibliotheken, die einen performanten Einsatz durch auf die Architektur angepasste Algorithmen ermöglichen. Die algorithmenbasierte Fehlertoleranz ist eine Methode, wie diese Operationen der linearen Algebra gegen auftretende Fehler abgesichert werden können. Bei der Überprüfung der Ergebnisse bezüglich aufgetretener Fehler müssen dabei Werte miteinander verglichen werden, die durch einen Rundungsfehler behaftet sind und somit nicht mit einem Test auf Gleichheit abgeprüft werden können. Deshalb werden Fehlerschwellwerte benötigt, bei deren Überschreitung ein Fehler erkannt und anschließend korrigiert werden kann.

In dieser Arbeit wurde der Einsatz von algorithmenbasierter Fehlertoleranz unter Verwendung hochoptimierter Bibliotheken auf GPUs für ausgewählte Operationen der linearen Algebra untersucht und angewandt. Der Fokus dieser Arbeit liegt neben der Anpassung der Methoden an den Einsatz auf GPUs darin, geeignete Fehlerschwellwerte für die algorithmenbasierte Fehlertoleranz zu untersuchen. Dabei wurden deterministische Methoden zur Fehlerschwellwertbestimmung betrachtet und eine probabilistische Methode zur Rundungsfehlerabschätzung für die Bestimmung von Fehlerschwellwerten angepasst und weiterentwickelt. Diese wurden anhand experimenteller Untersuchungen bezüglich der Qualität im Sinne der Differenz zum real aufgetretenen Rundungsfehler, der Fehlererkennung bei Fehlerinjektion und der Performanz der Methoden bei Implementierung auf GPUs miteinander verglichen. Die probabilistische Methode zeichnet sich dabei durch einen näher am Rundungsfehler liegenden Fehlerschwellwert aus, ist dadurch in der Lage einen größeren Anteil auftretender Fehler zu erkennen und kann ohne großen Mehraufwand auf GPUs implementiert werden.

2. Stand der Technik: Fehlertolerante Systeme

Die Verlässlichkeit eines Systems kann durch Attribute wie Zuverlässigkeit (engl. *Reliability*), Verfügbarkeit (engl. *Availablity*), Sicherheit (engl. *Safety*), Integrität (engl. *Integrity*) und Wartbarkeit (engl. *Maintainability*) beschrieben werden [3]. Diese Attribute sind oftmals eng miteinander verknüpft und erhalten je nach System und Standpunkt des Betrachters eine andere Gewichtung, wenn von der Verlässlichkeit eines Systems gesprochen wird. Im Fokus dieser Arbeit liegen Untersuchungen von Maßnahmen zur Integration von Fehlertoleranz in ausgewählten algebraischen Operationen. Bei der Diskussion wird daher das Hauptaugenmerk auf die Zuverlässigkeit und die damit verbundene Verfügbarkeit des Systems gelegt.

In großen Rechnerverbünden auftretende Hard- und Softwarefehler wertet eine Studie anhand von gesammelten Fehlerdaten aus [4]. Diese Daten umfassen einen Zeitraum von neun Jahren für den Rechnerverbund des Los Alamos National Laboratory (LANL), sowie einen Zeitraum von einem Jahr an einem ungenannten Supercomputer mit über 10^4 Prozessoren. Über 50% des Systemversagens kann dabei für beide Systeme auf Hardwarefehler zurückgeführt werden. Bei Betrachtung der Ursachen dieser Hardwarefehler für den Rechnerverbund des LANL können rund 30,1% der Fehler den Speichermodulen zugeordnet werden. Weitere Fehlerquellen sind die Hauptplatine (16,4%), die Stromversorgung (9,7%), das Netzwerk (9,7%) und der Hauptprozessor (2,4%). Für den Supercomputer konnten ebenfalls mehr als 20% der Hardwarefehler den Speichermodulen als Fehlerursache zugeordnet werden. Die Fehlerraten variieren dabei zwischen 20 und 1000 Fehlern pro Jahr, der Supercomputer liegt mit 600 Fehlern pro Jahr dabei im oberen Mittelfeld. Für den schlechtesten Fall bedeutet dies, dass eine Anwendung mit einer Laufzeit von über acht Stunden nur mit einer geringen Wahrscheinlichkeit von keinem Fehler betroffen ist. Dies macht deutlich, dass gerade im Bereich von großen Rechensystemen verschiedene Maßnahmen zur Integration von Fehlertoleranz mit Variation in Aufwand und Kosten benötigt werden, um einen produktiven Einsatz dieser Systeme gewährleisten zu können.

Hardwarebasierte Methoden zur Fehlertoleranz zielen darauf ab, einen Teil des Systems durch Einbringen zusätzlicher Strukturen in die Hardware resistenter gegen auftretende Fehler zu machen. Zwei generelle Ansätze können dabei unterschieden werden. Auf der einen Seite können auftretende Fehler durch Nutzung zusätzlicher, redundanter Hardware maskiert werden. Klassische Beispiele hierfür sind die *Triple Modular Redundancy* [5], die Verwendung von fehlerkorrigierenden Codes (engl. *error correcting codes*, ECC) zur Absicherung von Bitmustern bei der Speicherung und Kommunikation von Daten [6] und der Einsatz selbsttestender Schaltungen [7]. Auf der anderen Seite können auftretende Fehler zunächst detektiert und das System im Anschluss rekonfiguriert werden. Die Detektion von Fehlern

kann beispielsweise durch periodisch durchgeführte Selbsttests [8] oder die Verwendung von *Watchdog-Prozessoren* [9] erfolgen.

Diese hardwarebasierten Methoden zur Fehlertoleranz sind jedoch mit hohen Entwicklungskosten verbunden und können nicht nachträglich integriert werden. Der Einsatz beschränkt sich dadurch meist auf sehr sicherheitskritische Systeme, bei denen dieser hohe Aufwand gerechtfertigt werden kann. Eine weitere Möglichkeit zur Integration von Fehlertoleranz bieten die softwarebasierten Methoden zur Fehlertoleranz.

2.1. Softwarebasierte Fehlertoleranz

Klassische Methoden der softwarebasierten Fehlertoleranz sind zwei universell einsetzbare Ansätze: Die Replikation von Daten und Berechnungen, sowie das Sichern des Systemzustands an Fixpunkten und die Wiederaufnahme der Berechnung an diesem Punkt bei Auftreten eines Fehlers.

Die Replikation von Daten und Berechnungen stellt das softwareseitige Äquivalent zur Einführung redundanter Module in Hardware dar. Die Redundanz auf Softwareebene beim Einsatz von *recovery blocks* [10] wird dadurch erreicht, dass für eine gegebene Spezifikation unterschiedliche Versionen oder Implementierungen dieser Funktion verwendet werden. Das Ergebnis der primären Version wird nach der Berechnung einem Akzeptanztest unterzogen. Dieser Test entscheidet, ob das Ergebnis akzeptiert werden soll oder nicht. Schlägt der Akzeptanztest fehl, wird die Berechnung mit einer alternativen Implementierung auf der gleichen Hardware erneut durchgeführt. Eine mögliche Schwierigkeit besteht darin, den Akzeptanztest zu definieren. Die Methode des *n-Version Programming* [11] verzichtet auf den Einsatz eines Akzeptanztests, vielmehr werden simultan mehrere unterschiedliche Versionen eingesetzt, die die gleiche Aufgabe bearbeiten. Eine nachfolgende Entscheidungsfindung generiert aus allen diesen berechneten Ergebnissen ein finales Ergebnis. Signifikanter Nachteil dieser Methode ist der Mehraufwand, der zum Einen bei der Entwicklung der Software, zum Anderen bei Einsatz dieser Methode betrieben werden muss.

Das von Reis *et al.* [12] entwickelte Schema der *Software Implemented Fault Tolerance* (*SWIFT*) dupliziert die Berechnung innerhalb eines definierten Blocks durch Hinzufügen redundanter Instruktionen und Datenregister. Dadurch kann die dynamische und statische Parallelität auf Instruktionsebene eines Prozessors ausgenutzt werden und bietet zusammen mit der Verwendung von ECC-geschütztem Speicher und Methoden zur Überprüfung des Kontrollflusses eine umfassende Möglichkeit zur Integration von Fehlertoleranz auf Softwareebene. Diese Technik benötigt jedoch Zugriff auf den Quellcode des Compilers, da signifikante Änderungen vorgenommen werden müssen um die Überprüfung des Kontrollflusses zu implementieren.

Die Replikation von Daten und Berechnungen zeichnet sich durch einen je nach Verfahren unterschiedlich hohen Entwicklungsaufwand aus. Während bei Verwendung von *recovery blocks* und *n-Version Programming* die redundanten Funktionen für jeden Algorithmus implementiert werden müssen, kann *SWIFT* prinzipiell als Bibliothek ausgelegt werden.

Die Sicherung des Systemzustands ist eine weitere, universell einsetzbare Methode zur Integration softwarebasierter Fehlertoleranz, welche 1975 von Randell [10] vorgestellt wurde. Dabei wird in regelmäßigen Abständen ein komplettes Abbild des Systemzustands auf ein geschütztes Medium geschrieben. Bei Detektion eines Fehlers kann dieser Systemzustand wieder hergestellt werden und die Berechnung von diesem Fixpunkt aus erneut gestartet werden. Die Geschwindigkeitseinbußen dieser Methode liegen vor Allem in der Zeit für das Erstellen eines Fixpunkts, sowie der Zeit für die Wiederherstellung bei Auftreten eines Fehlers und sind somit eng an die Datenmenge gekoppelt, die gespeichert werden muss. Cappello [13] führt ein inhärentes Problem dieser Technik für besonders große Systeme an: Je kürzer die Perioden zwischen zwei aufeinanderfolgenden Fehlern und der dadurch gestarteten Wiederherstellung, desto weniger Zeit bleibt für die Applikation um Fortschritte bei der Berechnung zu machen. Ist die durchschnittliche Zeit zwischen zwei auftretenden Fehlern gleich der Zeit, die das System für die Wiederherstellung benötigt, wird die Anwendung keine Fortschritte mehr erzielen können.

Einige Weiterentwicklungen zielen darauf ab, nur kritische Teile des Speichers zu sichern. Die Klassifizierung, welche Daten als kritisch betrachtet werden, kann manuell durch den Programmierer vorgegeben werden, zu einem gewissen Teil aber auch durch den Compiler bestimmt werden. Ein weiterer Ansatz zur Reduktion der Datenmenge stellen inkrementelle Zustandsabbildungen dar. Hierbei werden bei der Erstellung von sukzessiven Fixpunkten nur von Änderungen betroffene Daten gespeichert. Bei der Methode des *diskless checkpointing* wird der lokale Zustand jedes Knotens in dessen lokalen Speicher abgelegt [14]. Bei der Wiederherstellung muss demnach nicht auf das langsamere, festplattenbasierte Dateisystem zugegriffen werden. Das Aufkommen schneller SSD-Datenträger ermöglicht eine weitere Variante dieser Methode, bei der jeder Knoten ein Abbild seines Zustands auf einem lokalen SSD-Datenspeicher sichert. Dadurch können schnelle Zugriffszeiten und Datentransferraten erreicht werden, ohne einen großen Teil des Hauptspeichers durch das Abbild des Zustands zu belegen [15].

Das Sichern des Systemzustands stellt die in großen Rechensystemen häufig eingesetzte Methode zur Fehlertoleranz dar und wird ständig weiterentwickelt, um bei der steigenden Größe der Rechnerysteme weiterhin mit akzeptablem Mehraufwand eingesetzt werden zu können.

Neben diesen allgemein einsetzbaren Ansätzen wurden auch Methoden der Fehlertoleranz für spezielle Algorithmen und Operationen entwickelt. Diese sind speziell auf die Algorithmen zugeschnitten und nutzen Eigenschaften dieser Operationen aus. Der Vorteil dieser algorithmenbasierten Fehlertoleranz liegt in der Unabhängigkeit von der eingesetzten Hardware und dem Erreichen einer performanten und feingranularen Fehlertoleranz. Der Nachteil ergibt sich ebenfalls durch die Spezialisierung: Die algorithmenbasierte Fehlertoleranz muss für jede Operation separat entwickelt und kann nicht für jeden beliebigen Algorithmus angewendet werden.

2.2. Algorithmenbasierte Fehlertoleranz

Die algorithmenbasierte Fehlertoleranz ist eine Methode, mit der bestimmte grundlegende mathematische Operationen und Funktionen auf Systemebene gegen das Auftreten von Hardwarefehlern resistenter gemacht werden können. Diese Verfahren bauen auf spezifischen Eigenschaften dieser Funktionen auf und sind so in der Lage, einen hinreichenden Grad an Fehlertoleranz mit geringem Aufwand zu gewährleisten. Alle Verfahren der algorithmenbasierten Fehlertoleranz folgen dem gleichen Schema: In einem ersten Schritt werden die Eingangsdaten zunächst mit zusätzlichen Informationen kodiert. Das Ergebnis wird daraufhin durch einen angepassten Algorithmus berechnet. Im Anschluss können durch die Dekodierung des Ergebnisses Rückschlüsse über das Auftreten von Fehlern im Ergebnis gezogen werden und diese lokalisiert werden. Dieses Schema ist in Abbildung 2.1 dargestellt.

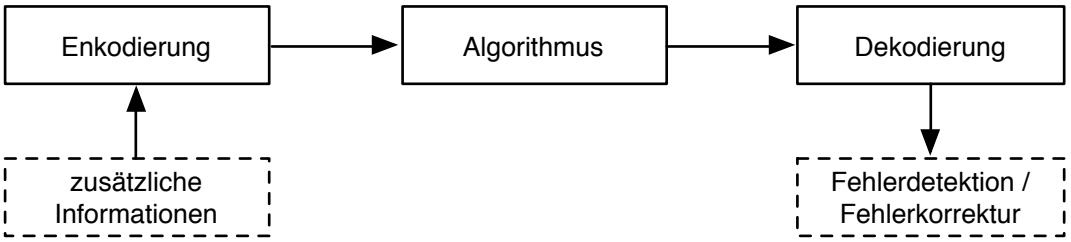


Abbildung 2.1.: Das Schema der algorithmenbasierten Fehlertoleranz.

Algorithmenbasierte Methoden zur Fehlertoleranz wurden zunächst für die Matrixaddition, die Matrixmultiplikation, das Skalarprodukt, die LU-Zerlegung und die Transposition vorgestellt [16]. Weitere Beispiele für den Einsatz algorithmenbasierter Fehlertoleranz sind die QR-Faktorisierung [17] und die schnelle Fourier-Transformation [18, 19, 20]. Im Folgenden wird die algorithmenbasierte Fehlertoleranz für Matrixoperationen näher erläutert.

2.2.1. Algorithmenbasierte Fehlertoleranz für Matrixoperationen

Die durch algorithmenbasierte Fehlertoleranz geschützte Matrixmultiplikation wurde 1984 von Huang und Abraham vorgestellt [16]. Bei diesem Verfahren werden die Matrizen $\mathbf{A}$ und $\mathbf{B}$ durch zusätzliche Reihen-, beziehungsweise Spaltenvektoren erweitert. Die $m \times n$ Matrix $\mathbf{A}$ wird dabei um Spaltenprüfsummen a_{m+1}, j erweitert:

$$\mathbf{A_{cc}} = \left(\begin{array}{ccc} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \\ \hline a_{m+1,1} & \dots & a_{m+1,n} \end{array}\right) \quad \text{mit} \quad a_{m+1,j} = \sum_{i=1}^{m} a_{i,j}. \tag{2.1}$$

Die $n \times q$ Matrix $\mathbf{B}$ wird um Reihenprüfsummen $b_{j,q+1}$ erweitert:

$$\mathbf{B_{rc}} = \left(\begin{array}{ccc|c} b_{1,1} & \dots & b_{1,q} & b_{1,q+1} \\ \vdots & \ddots & \vdots & \vdots \\ b_{n,1} & \dots & b_{n,q} & b_{n,q+1} \end{array}\right) \quad \text{mit} \quad b_{j,q+1} = \sum_{j=1}^{q} b_{i,j}. \tag{2.2}$$

Nach der linearen Operation der Matrixmultiplikation $\mathbf{A_{cc}} \cdot \mathbf{B_{rc}} = \mathbf{C_{fc}}$ sind in der Ergebnismatrix $\mathbf{C_{fc}}$ sowohl Spalten- als auch Reihenprüfsummen vorhanden:

$$\mathbf{C_{fc}} = \left(\begin{array}{ccc|c} c_{1,1} & \dots & c_{1,q} & c_{1,q+1} \\ \vdots & \ddots & \vdots & \vdots \\ c_{m,1} & \dots & c_{m,q} & c_{m,q+1} \\ \hline c_{m+1,1} & \dots & c_{m+1,q} & c_{m+1,q+1} \end{array}\right) \tag{2.3}$$

$\mathbf{C_{fc}}$ wird als vollständige Prüfsummenmatrix (*full checksum matrix*) bezeichnet und hat die Größe $(m+1) \times (q+1)$. Die Abbildung 2.2 zeigt die ABFT-Matrixmultiplikation. Prüfsummenelemente sind hierbei grau, Datenelemente weiß dargestellt.

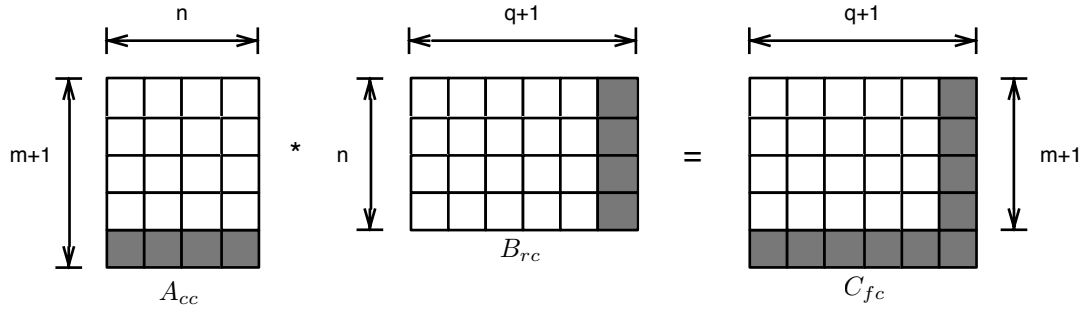


Abbildung 2.2.: Die ABFT-Matrixmultiplikation: Eine um Spaltenprüfsummen erweiterte Matrix $\mathbf{A_{cc}}$ wird mit einer um Reihenprüfsummen erweiterten Matrix $\mathbf{B_{rc}}$ multipliziert. Das Ergebnis ist eine vollständige Prüfsummenmatrix $\mathbf{C_{fc}}$.

Nach Ausführung der Matrixmultiplikation können über die Datenelemente der Matrix $\mathbf{C_{fc}}$ die Prüfsummen erneut berechnet werden:

$$c^*_{m+1,j} = \sum_{i=1}^{m} c_{i,j}, \quad \text{mit} \quad 1 \leq j \leq q \tag{2.4}$$

und

$$c^*_{i,q+1} = \sum_{j=1}^{q} c_{i,j}, \quad \text{mit} \quad 1 \leq i \leq m. \tag{2.5}$$

Über den Vergleich der durch die lineare Operation entstandene Spaltenprüfsummen $c_{m+1,j}$ mit $1 \leq j \leq q$ und den entstandenen Reihenprüfsummen $c_{i,q+1}$ mit $1 \leq i \leq m$, im folgenden

Referenzprüfsummen genannt, mit den aus den Elementen der Ergebnismatrix neu berechneten Prüfsummen $c^*_{i,j}$ lassen sich einzelne fehlerhafte Elemente lokalisieren und korrigieren. Im fehlerfreien Fall gilt für die Spaltenprüfsummen:

$$\forall j : \; c_{m+1,j} = c^*_{m+1,j} \tag{2.6}$$

und für die Reihenprüfsummen:

$$\forall i : \; c_{i,q+1} = c^*_{i,q+1}. \tag{2.7}$$

Manifestiert sich ein Fehler bei der Berechnung in genau einem Element der Ergebnismatrix, so kann dieser über den Vergleich von Prüfsumme und Referenzprüfsumme lokalisiert werden:

- $\exists j : c_{m+1,j} \neq c^*_{m+1,j}$ und $\forall i : c_{i,q+1} = c^*_{i,q+1}$: Fehler im Spaltenprüfsummenelement
- $\forall j : c_{m+1,j} = c^*_{m+1,j}$ und $\exists i : c_{i,q+1} \neq c^*_{i,q+1}$: Fehler im Reihenprüfsummenelement
- $\exists j : c_{m+1,j} \neq c^*_{m+1,j}$ und $\exists i : c_{i,q+1} \neq c^*_{i,q+1}$: Fehlerhaftes Datenelement ist $c_{i,j}$.

Die Differenz zwischen Referenzprüfsumme und neu berechneter Prüfsumme wird als Syndrom bezeichnet und gibt den Fehler an, der in diesem Prüfsummeneintrag aufgetreten ist. Für ein fehlerhaftes Datenelement in $c_{i,j}$ ist das Syndrom der Reihenprüfsumme definiert als

$$S_r = c^*_{j,q+1} - c_{j,q+1}, \tag{2.8}$$

für die Spaltenprüfsumme definiert als

$$S_c = c^*_{m+1,i} - c_{m+1,i}. \tag{2.9}$$

Für die Korrektur eines einzelnen fehlerhaften Datenelements $c_{i,j}$ ergeben sich drei Möglichkeiten: Zum Einen kann das Datenelement $c_{i,j}$ erneut aus den Eingangsmatrizen $\mathbf{A}$ und $\mathbf{B}$ berechnet werden:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}. \tag{2.10}$$

Zum Anderen lässt sich das fehlerhafte Element über das Syndrom der Reihenprüfsumme oder das Syndrom der Spaltenprüfsumme korrigieren:

$$c_{i,j} = c_{i,j} - S_r \quad \text{oder} \quad c_{i,j} = c_{i,j} - S_c. \tag{2.11}$$

Ergeben sich bei der Überprüfung der Reihen-, beziehungsweise Spaltenprüfsummen mehrere Abweichungen, ist die eindeutige Zuordnung und damit die Korrektur über die Syndrome nicht mehr möglich. Die Abbildung 2.3 verdeutlicht diesen Zusammenhang. Während bei einem einzelnen auftretenden Fehler (links) das fehlerhafte Element eindeutig identifiziert werden kann, ergeben sich bei mehreren abweichenden Prüfsummen mehrere Möglichkeiten der Zuordnung (Mitte, rechts).

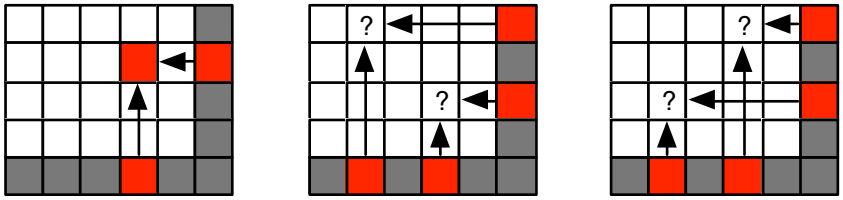


Abbildung 2.3.: Eindeutige Zuordnung eines einzelnen fehlerhaften Elements (links) und Problem der Zuordnung bei mehreren fehlerhaften Prüfsummen (Mitte, rechts) in der Ergebnismatrix $\mathbf{C_{fc}}$.

2.2.2. ABFT für Matrixoperationen mit gewichteten Prüfsummen

Jou und Abraham [21] beschrieben den Einsatz gewichteter Prüfsummen für die algorithmenbasierte Fehlertoleranz in verschiedenen Matrixoperationen. Ein Vektor $\mathbf{a}$ der Länge n wird zu einem kodierten Vektor $\hat{\mathbf{a}}$, indem $\mathbf{a}$ um t gewichtete Prüfsummen erweitert wird:

$$\mathbf{a}^T = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix} \tag{2.12}$$

$$\hat{\mathbf{a}}^T = \begin{pmatrix} a_1 & a_2 & \dots & a_n & cs_1 & cs_2 & \dots & cs_t \end{pmatrix} \tag{2.13}$$

Die gewichteten Prüfsummen $cs_1, \dots, cs_t$ ergeben sich dabei aus dem Skalarprodukt des Vektors $\mathbf{a}$ mit einem Vektor von n Gewichten:

$$cs_i = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix} \times \begin{pmatrix} w_{i,1} & w_{i,2} & \dots & w_{i,n} \end{pmatrix} \tag{2.14}$$

$$= \sum_{j=1}^{n} a_j \cdot w_{i,j} \tag{2.15}$$

Diese Gewichte lassen sich in einer $t \times (t+n)$ Matrix $\mathbf{H}$ zusammenfassen, über die sich Aussagen über die Eigenschaften des Kodierungsschemas (des *Codes*) bezüglich der Fehlerdetektions- und Fehlerkorrekturfähigkeit treffen lassen.

$$\mathbf{H} = \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n} & -1 & 0 & \dots & 0 \\ w_{2,1} & w_{2,2} & \dots & w_{2,n} & 0 & -1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ w_{t,1} & w_{t,2} & \dots & w_{t,n} & 0 & 0 & \dots & -1 \end{pmatrix} \tag{2.16}$$

Die Gewichte $w_{i,j}$ müssen für einen *Code* mit Distanz d so gewählt werden, dass bei jeder Kombination von $d-1$ Spaltenvektoren von $\mathbf{H}$ die Vektoren linear unabhängig sind. Ein *Code* mit Distanz $d+1$ kann mindestens d Fehler detektieren, für die Korrektur von c Fehlern ist eine Distanz von $2c+1$ nötig.

Die Korrektur von fehlerhaften Elementen kann nun durch Lösen eines Gleichungssystems durchgeführt werden. Für einen kodierten Vektor $\hat{\mathbf{a}}$ mit t gewichteten Prüfsummen kann

analog zum Syndrom ein Syndromvektor **s** der Größe t definiert werden. Die Elemente dieses Syndromvektors bestehen aus der Differenz von mitgeführten Referenzprüfsummen cs_i und neu berechneten Prüfsummen cs_i^*:

$$s_i = cs_i - cs_i^* \quad \text{mit} \quad 1 \leq i \leq t. \tag{2.17}$$

Gesucht wird nun ein Korrekturvektor **cv** der Länge $n + t$, mit dem sich c fehlerhafte Elemente in **â** korrigieren lassen. Dieser Korrekturvektor kann per Definition maximal c Einträge ungleich 0 enthalten. Gesucht werden also die maximal c fehlerhaften Positionen $i_1 \dots i_c$ in **â** und die dazugehörigen Werte $cv_{i_1} \dots cv_{i_c}$. Hierfür wird ein Gleichungssystem mit t Gleichungen aufgestellt:

$$\begin{aligned}
w_{1,i_1} \cdot cv_{i_1} + w_{1,i_2} \cdot cv_{i_2} + \dots + w_{1,i_c} \cdot cv_{i_c} &= s_1 \\
w_{2,i_1} \cdot cv_{i_1} + w_{2,i_2} \cdot cv_{i_2} + \dots + w_{2,i_c} \cdot cv_{i_c} &= s_2 \\
&\vdots \\
w_{t,i_1} \cdot cv_{i_1} + w_{t,i_2} \cdot cv_{i_2} + \dots + w_{t,i_c} \cdot cv_{i_c} &= s_t
\end{aligned}$$

Bekannt sind in diesem Gleichungssystem lediglich die Einträge des Syndromvektors $s_1, \dots, s_t$. Für die Lösbarkeit wird aufgrund der großen Anzahl an Unbekannten eine spezielle Beziehung zwischen den Gewichten benötigt. Je nach verwendeten Gewichten kann dieses Gleichungssystem linear oder auch nichtlinear sein. Wird anstelle dieser generischen Darstellung ein konkreter Wert für c und eine passende Gewichtung gewählt, lässt sich die Lösung dieses Gleichungssystem in einem effizienten Algorithmus implementieren [22].

Die Kodierung von Vektoren durch gewichtete Prüfsummen kann für die Matrixmultiplikation verwendet werden, indem jeder Spaltenvektor von **A** und jeder Reihenvektor von **B** nach einem solchen Schema kodiert wird. Der Einsatz gewichteter Prüfsummen in der ABFT-Matrixmultiplikation ermöglicht es dadurch, Fehler innerhalb jedes Spalten- und Reihenvektors der Ergebnismatrix zu lokalisieren und gegebenenfalls zu korrigieren. Neben dieser pro Vektor geltenden Eigenschaft gibt auch die Kombination von abweichenden Reihen- und Spaltenprüfsummen Aufschluss über den Ort eines aufgetretenen Fehlers. Abbildung 2.4 zeigt den Einsatz gewichteter Prüfsummen bei der ABFT-Matrixmultiplikation. Jeder Spaltenvektor von **A** und jeder Reihenvektor von **B** wird nicht um eine, sondern mehrere Prüfsummen erweitert.

Folgende Matrixoperationen erhalten die Eigenschaft der gewichteten Prüfsummen: Die Multiplikation, Addition und Transposition von Matrizen, sowie die LU-Zerlegung und die Multiplikation einer Matrix mit einem skalaren Wert. Aufgrund der einfachen Implementierung durch Schiebe-Operationen wurde von den Autoren [21] die Verwendung exponentieller Gewichte $w_{i,j}$ mit

$$w_{i,j} = (2^{j-1})^{i-1} \tag{2.18}$$

vorgeschlagen. Diese Kodierung hat jedoch den Nachteil, dass selbst für die Korrektur eines einzelnen fehlerhaften Elements ($t = 2$) im Vektor die Gewichte exponentiell mit der Größe des Vektors wachsen. Daraus folgt zum Einen, dass ab einer gewissen Größe des Vektors

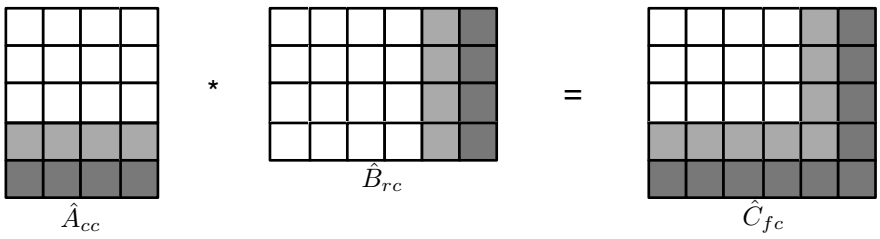


Abbildung 2.4.: Die ABFT-Matrixmultiplikation mit gewichteten Prüfsummen.

die Zahl in Gleitkommaarithmetik nicht mehr darstellbar ist, zum Anderen treten bei der Berechnung der Prüfsummenelemente vermehrt signifikant große Rundungsfehler auf. Diese erschweren die Erkennung und die Lokalisierung von aufgetretenen Fehlern. Aufgrund dieses Zusammenhangs beschäftigen sich einige Arbeiten mit der Suche nach polynomiellen Gewichten.

Nair und Abraham beschreiben eine Menge von linearen *Codes* für den Einsatz in algorithmenbasierter Fehlertoleranz [23]. Da es nicht möglich ist, ein Kodierungsschema zu finden, welches unabhängig von der zu kodierenden Datenmenge den numerischen Fehler minimiert, stellen die Autoren verschiedene Kodierungsschemata vor. Diese umfassen Durchschnitts- und gewichtete Durchschnittsprüfsummen, sowie periodische Kodierungsvektoren und normalisierte Kodierungsvektoren. Auf Basis experimenteller Untersuchungen geben die Autoren eine Empfehlung, welches Kodierungsschema sich für den Einsatz bei Datenmengen mit einer bestimmten Charakteristik eignet. Die Autoren Anfinson und Luk [22] stellen einige Begrifflichkeiten und Distanzeigenschaften von *Codes* im Bezug auf [21] klar und geben einen schnellen Korrekturalgorithmus für einen *Code* mit Distanz 5 bei Verwendung von exponentiell gewichteten Prüfsummen an.

Bliss *et al.* sprechen sich für den Einsatz linear gewichteter Prüfsummen bei Einsatz in Arithmetik mit beschränkter Genauigkeit aus [24]. Des Weiteren wird ein Kodierungsschema zur Korrektur eines einzelnen Fehlers mit minimalem dynamischen Wertebereich vorgestellt, der bessere numerische Eigenschaften als exponentiell gewichtete und andere linear gewichtete Prüfsummen bietet.

2.2.3. ABFT für Matrixoperationen mit partitionierter Kodierung

Jedes Element der $m \times q$ Matrix $\mathbf{C}$, die das Ergebnis der Multiplikation einer Matrix $\mathbf{A}$ der Größe $m \times n$ und Matrix $\mathbf{B}$ der Größe $n \times q$ ist, kann durch das Skalarprodukt eines Reihenvektors von $\mathbf{A}$ und eines Spaltenvektors von $\mathbf{B}$ dargestellt werden:

$$c_{i,j} =< \mathbf{a_i}, \mathbf{b_j}^T > \tag{2.19}$$

Diese Eigenschaft erlaubt eine Unterteilung der Matrizen $\mathbf{A}$, $\mathbf{B}$ und $\mathbf{C}$ in Blöcke der Größe $t \times t$, wobei n, m und q durch t teilbar sein müssen. Zur Erfüllung dieser Voraussetzung

für Matrizen beliebiger Größe müssen die Eingangsmatrizen um maximal $t-1$ mit Null initialisierte Reihenvektoren für **A**, beziehungsweise Spaltenvektoren für **B** ergänzt werden. Dadurch ergibt sich für die Berechnung eines Blocks $\mathbf{C_{r,s}}$ der Ergebnismatrix:

$$\mathbf{C_{r,s}} = \sum_{k=1}^{n/t} \mathbf{A_{r,k}} \cdot \mathbf{B_{k,s}} \tag{2.20}$$

Bei der partitionierten Kodierung wird jeder der $t \times t$ Blöcke um separate Prüfsummenvektoren erweitert. Die Abbildung 2.5 zeigt die Verwendung von partitionierter Kodierung in der ABFT-Matrixmultiplikation. Jeder $t \times t$ Block von **A** wird um Spaltenprüfsummen, jeder $t \times t$ Block von **B** wird um Reihenprüfsummen erweitert.

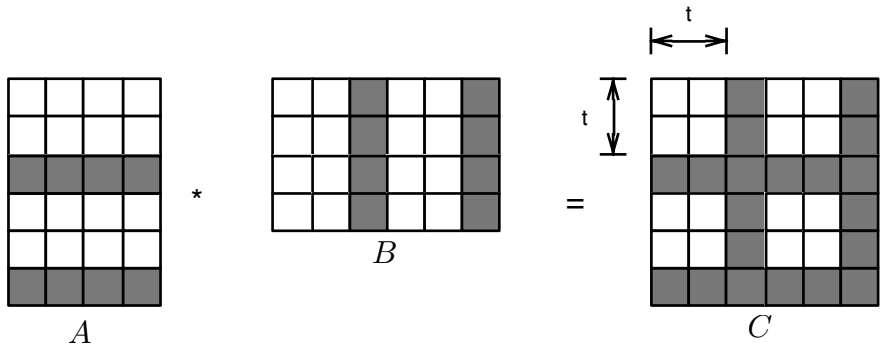


Abbildung 2.5.: Partitionierte Kodierung von Matrizen durch Blöcke der Größe $t \times t$.

Die Vorteile der Verwendung von partitionierter Kodierung werden in [25] ausführlich besprochen. Diese umfassen:

- Die Detektion von mehreren Fehlern: Durch die Verwendung der partitionierten Kodierung können mehrere auftretende Fehler erkannt werden, wenn diese sich in unterschiedlichen Blöcken von **C** manifestieren. Die Prüfsummeneigenschaft wird für jeden Block separat erhalten.
- Bessere numerische Eigenschaften bei gewichteten Prüfsummen: Während bei der unpartitionierten Version alle m Elemente des Spaltenvektors von **A**, beziehungsweise alle q Elemente des Reihenvektors von **B** bei der Berechnung der Prüfsummen einfließen, ist die Anzahl der Elemente bei partitionierter Kodierung konstant. Besonders bei Verwendung von exponentiell gewichteten Prüfsummen wird dadurch der auftretende Rundungsfehler bei der Prüfsummenberechnung begrenzt.
- Begrenzung der zusätzlichen benötigten Hardware für die Kodierung: Werden Baumstrukturen von Addierern eingesetzt, um die Matrix durch Berechnung der Prüfsummen zu kodieren, dann werden für die partitionierte Kodierung weniger Addierer als für die unpartitionierte Kodierung benötigt.

2.2.4. Verteilung von Berechnungsaufgaben

Ein wichtiger Aspekt für die Korrektheit der ABFT-Matrixmultiplikation besteht darin, dass die Prüfsummen über die Datenelemente der Ergebnismatrix durch einen anderen Prozessor berechnet werden, als durch den Prozessor, der diese Datenelemente bei der Matrixmultiplikation berechnet hat. Ist diese Voraussetzung nicht gegeben, kann es zu einer Maskierung von Fehlern kommen.

In weiterführenden Arbeiten wurde die ABFT-geschützte Matrixmultiplikation für Mehrkernprozessoren angepasst [26, 27]. Die Arbeit von Banerjee *et al.* [26] beschäftigt sich dabei mit dem Einsatz von ABFT auf Hypercube-Multiprozessoren. Ein Hypercube besteht aus 2^N Prozessoren, wobei jeder Prozessor mit N anderen Prozessoren direkt kommunizieren kann. Die Matrix $\mathbf{A}$ wird dabei unterteilt und jede Teilmatrix an einen Prozessor und den benachbarten Prozessor des Hypercubes verteilt. Prozessor i berechnet das Ergebnis $\mathbf{C_i} = \mathbf{A_i} \cdot \mathbf{B}$, sowie die Spaltenprüfsumme der dem benachbartem Prozessor zugeteilten Matrix $\mathbf{A_{mate(i)}}$ und daraus die erwartete Prüfsumme für das Ergebnis des Nachbarprozessors. Nach der Berechnung wird $\mathbf{C_i}$ an den benachbarten Prozessor geschickt, dort die Prüfsumme berechnet und mit der erwarteten Prüfsumme verglichen. Zur Lokalisierung eines fehlerhaften Prozessors wird in einem nachfolgenden Schritt jeder Prozessor mit einem anderen Nachbarn gepaart und der Überprüfungsschritt erneut durchgeführt. Darüber hinaus werden in dieser Arbeit Techniken zur Rekonfiguration des Hypercubes vorgestellt, bei der fehlerhafte Prozessoren durch zusätzliche, unbenutzte Prozessoren ausgetauscht werden können.

Für MIMD-Architekturen (*Multiple Instruction, Multiple Data*) mit p Prozessoren liefern Roy-Chowdhury und Banerjee [27] ein skalierbares Verfahren, bei dem jeweils eine bestimmte Anzahl Prozessoren zu einer Prüfgruppe zusammengefasst werden und gegenseitig die Überprüfung der Prüfsummeneigenschaft übernehmen. Die Größe dieser Prüfgruppen ist dabei abhängig von der Anzahl Fehler, die lokalisiert werden sollen. Über gewichtete Prüfsummen kann eine Fehlerlokalisierung und -korrektur auch bei mehreren fehlerhaften Prozessoren sichergestellt werden.

Analytische Untersuchungen durch graphentheoretische Modelle setzen die Anzahl zu erkennender Fehler und die Anzahl verwendeter Prozessoren mit den dafür benötigten Überprüfungen in Relation [28] und ergeben untere und obere Grenzen für den Zeit- und Prozessoroverhead. In [29] wurden diese Grenzen weiter verfeinert.

3. Gleitkommaarithmetik nach IEEE 754

3.1. Darstellung von reellen Zahlen

Aufgrund der beschränkten Anzahl an Bits, die in einem Rechner für die Darstellung von Datentypen zur Verfügung stehen, muss für die Darstellung von reellen Zahlen eine approximative Darstellung verwendet werden. Die Darstellung einer solchen Zahl x in Gleitkommaarithmetik ist allgemein gegeben durch ein Vorzeichen s, eine Basis B, einen Exponenten e und eine Mantisse m:

$$x = s \cdot m \cdot B^e. \tag{3.1}$$

Je nachdem, wie viele Bits insgesamt für die Darstellung einer solchen Zahl zur Verfügung stehen, werden dem Vorzeichen, dem Exponent und der Mantisse eine bestimmte Anzahl von Bits zur Darstellung vergeben. Für den IEEE 754 Standard ist diese Verteilung von Vorzeichen-, Exponenten- und Mantissenbits für Darstellungen mit 32 und 64 zur Verfügung stehenden Bits bei $B = 2$:

Darstellung	Vorzeichen S	Exponent E	Mantisse M
Single Precision (32 Bit)	1 Bit	8 Bit	23 Bit
Double Precision (64 Bit)	1 Bit	11 Bit	52 Bit

Um sowohl negative, als auch positive Exponenten e darstellen zu können, wird für die Exponentendarstellung E ein fester Biaswert R verwendet. Dieser Biaswert ist abhängig von der Anzahl Bits für die Exponentendarstellung definiert als $R = 2^{bits(E)-1} - 1$. Die Umrechnung von Exponent zu Exponentdarstellung ergibt sich durch $e = E - R$. Die Mantisse wird bei IEEE 754 als normalisierte Zahl mit impliziter 1 vor dem Komma gespeichert. Dadurch steht für normalisierte Zahlen ein weiteres Bit Genauigkeit zur Verfügung. Für die Umrechnung zwischen Mantisse und Mantissendarstellung gilt mit der impliziten 1 die Beziehung $m = 1, M$. Für das Vorzeichen gilt $s = (-1)^S$. Spezielle Bitmuster des Exponenten E stehen für die Sonderfälle der denormalisierten Zahlen und der Kodierung von Überlauf und von undefinierten Zahlen zur Verfügung. Bei der denormalisierten Darstellung mit $E = 0$ ist die implizite 1 vor dem Komma der Mantisse aufgehoben, wobei der Exponent e als der kleinste, normalisiert darstellbare Wert des Exponenten definiert ist. Dadurch wird der Bereich zwischen 0 und dem kleinsten normalisiert darstellbaren Wert kodiert. Die Auflösung ist in diesem Bereich durch die Anzahl Mantissenbits beschränkt. Der kleinste Wert, für den in einer Gleitkommadarstellung $1 + \epsilon_{mach} > 1$ gilt, wird als Maschinengenauigkeit bezeichnet und findet oftmals Anwendung bei der Abschätzung von Rundungsfehlern.

3.2. Rundung

Bei den meisten in Gleitkommaarithmetik ausgeführten Operationen kommt es aufgrund der beschränkten Genauigkeit zu Rundungsfehlern. Werden beispielsweise zwei Zahlen $x = 1.010 \cdot 2^4$ und $y = 1.000 \cdot 2^0$ bei einer fiktiven Gleitkommadarstellung mit drei Mantissenbits addiert, muss y zunächst für den Exponent von x angeglichen werden:

$$y = 1.000 \cdot 2^0 = 0.0001 \cdot 2^4.$$

Das Ergebnis der Addition

$$\begin{aligned} & 1.0100 \cdot 2^4 \\ + \; & 0.0001 \cdot 2^4 \\ = \; & 1.0101 \cdot 2^4 \end{aligned}$$

kann nur mit ebenfalls drei Mantissenbits dargestellt werden und muss zu $1.010 \cdot 2^4$ oder $1.011 \cdot 2^4$ gerundet werden. Der IEEE 754 Standard fordert, dass die vier Grundoperationen der Addition, Subtraktion, Multiplikation und Division exakt gerundet werden. Exakt gerundet heisst hierbei, dass das Ergebnis der Operation exakt berechnet und dann zur nächsten darstellbaren Zahl, bei gleichem Abstand zur nächsten geraden Zahl gerundet wird. Diese Forderung kann durch das Einführen von zwei *Guard Bits* und einem *Sticky Bit* effizient implementiert werden. Die *Guard Bits* und das *Sticky Bit* werden dabei für eine erhöhte Genauigkeit beim Anpassen der Zahlen vor der Addition verwendet. Bits, die beim Anpassen der Zahlen aus dem darstellbaren Bereich der Mantisse geschoben werden würden, landen in den zwei *Guard Bits*. Wird mindestens eine 1 über diese *Guard Bits* hinaus verschoben, dann wird das *Sticky Bit* auf 1 gesetzt. Basierend auf den *Guard Bits* und dem *Sticky Bit* kann eine im Sinne des Standards exakte Rundung durchgeführt werden. Neben dem Runden zur nächsten darstellbaren Zahl, bei gleichem Abstand zur nächsten gerade Zahl, werden noch vier weitere Rundungsmodi vom IEEE 754 Standard gefordert: Das Runden zur nächsten darstellbaren Zahl, bei gleichem Abstand zur betragsmäßig größeren Zahl, das Runden in Richtung 0, das Runden in Richtung $+\infty$, sowie das Runden in Richtung $-\infty$ [30].

Es gibt jedoch Abweichungen von der Forderung nach der Rundung nach jeder Operation. Ein Beispiel hierfür ist die *Fused Multiply-Add*-Operation $a \leftarrow a + (b \cdot c)$, welche zur Steigerung der Performanz in aktuellen Grafikprozessoren eingesetzt wird. Hierbei wird das Zwischenergebnis $b \cdot c$ vor der Addition mit a nicht gerundet, sondern in voller Auflösung nach der Multiplikationseinheit dem Addierer zugeführt. Erst nach der Addition wird das Ergebnis gerundet, was neben einer erhöhten Performanz ein genaueres Ergebnis als bei Rundung des Zwischenergebnisses erzeugt. Jedoch wird eine angepasste Hardware mit breiteren Datenpfaden benötigt. Die *Fused Multiply-Add*-Operation mit einer Rundungsoperation wurde 2008 dem Standard IEEE 754 zugefügt.

4. Stand der Technik: Methoden zur Fehlerschwellwertbestimmung

Durch die Akkumulation von Rundungsfehlern bei der Berechnung der Matrixelemente sowie der Prüfsummen kommt es auch im fehlerfreien Fall zu Abweichungen bei der Überprüfung der Gleichheit zwischen neu berechneter und mitgeführter Prüfsumme. Entsprechend muss der Vergleich der neu berechneten Prüfsummen mit den Referenzprüfsummen so angepasst werden, dass diese auftretenden Rundungsfehler toleriert werden und nicht zu einer falsch-positiven Fehlerdetektion (engl. *False Positives*) führen. Für die Spaltenprüfsummen wird dazu Gleichung 2.6 durch einen Schwellwert ϵ erweitert:

$$|c_{i,q+1} - c^*_{i,q+1}| > \epsilon_{i,q+1}. \tag{4.1}$$

Für die Reihenprüfsummen wird Gleichung 2.7 ebenfalls angepasst:

$$|c_{m+1,j} - c^*_{m+1,j}| > \epsilon_{m+1,j}. \tag{4.2}$$

Die Ermittlung von geeigneten Schwellwerten stellt somit für die Performanz und Fähigkeit zur Fehlererkennung der ABFT-geschützten Matrixmultiplikation eine wichtige Aufgabe dar: Sind die Fehlerschwellwerte zu niedrig gewählt, kommt es durch den akkumulierten Rundungsfehler zu *False Positives*. Dadurch werden die auftretenden Rundungsfehler inkorrekt als Hardwarefehler identifiziert. Die anschließende, eigentlich unnötige Korrektur dieses Fehlers verschlechtert die Performanz der ABFT-geschützten Matrixmultiplikation. Sind die Fehlerschwellwerte zu hoch gewählt, werden Fehler nicht erkannt und haben somit Auswirkungen auf die Korrektheit des Ergebnisses.

In der Literatur werden unterschiedliche Methoden zur Bestimmung eines solchen Fehlerschwellwerts diskutiert. Diese Methoden lassen sich in experimentelle und deterministische Methoden zur Fehlerschwellwertbestimmung einteilen.

4.1. Experimentelle Fehlerschwellwertbestimmung

Eine experimentelle Bestimmung der Fehlerschwellwerte kann eingesetzt werden, wenn für einen bestimmten Anwendungsfall mehrere verschiedene Trainingsdatensätze vorliegen, mit denen die Fehlerschwellwerte trainiert werden können. Pro Datensatz werden zunächst die Fehlerschwellwerte für alle Prüfsummenelemente mit 0 initialisiert und bei mehrmaligem Ausführen der Prüffunktion inkrementell erhöht, bis keine *False Positives* mehr für diesen Datensatz auftreten. Dieses Verfahren wird für jeden der Trainingsdatensätze durchgeführt.

Die Fehlerschwellwerte für den Einsatz in der Praxis ergeben sich aus dem Maximum aller für die Trainingsdatensätze ermittelten Fehlerschwellwerte [31]. Banerjee *et al.* [26] beschreiben ein Verfahren für variierende Problemgrößen. Die Norm des Fehlers wird dabei definiert als eine Funktion $K \cdot F(N) \cdot 2^{-2t}$ mit einer anwendungsspezifischen Konstante K, einer von der Problemgröße N abhängenden Funktion $F(N)$ und der Anzahl Mantissenbits t. Für eine gegebene Problemgröße N und eine gegebene Anzahl Mantissenbits t kann analog zum oberen beschriebenen Verfahren experimentell eine Konstante K' gefunden werden, bei der für mehrere Testdatensätze kein *False Positive* mehr auftritt.

Diese experimentellen Methoden haben zum Einen den Nachteil, dass die Kalibrierungsläufe sehr rechenintensiv und zeitaufwändig sind. Zum Anderen sind die ermittelten Fehlerschwellwerte sehr von den in der Trainingsphase verwendeten Datensätzen und des Dynamikbereichs der Werte dieser Datensätze abhängig. Ergeben sich Änderungen in der Charakteristik der Datensätze, so werden die ermittelten Fehlerschwellwerte höchstwahrscheinlich den Anforderungen nicht mehr genügen und vermehrt Fehler nicht erkennen, beziehungsweise korrekte Daten als *False Positives* klassifizieren.

4.2. Deterministische Fehlerschwellwertbestimmung

Bei den deterministisch ermittelten Fehlerschwellwerten werden Matrixnormen eingesetzt, um die Fehlerschwellwerte zu bestimmen. Für die Matrixmultiplikation einer $m \times n$ Matrix $\mathbf{A}$ mit einer $n \times q$ Matrix $\mathbf{B}$ gilt nach Golub und Van Loan [32] für den auftretenden Rundungsfehler:

$$||fl(\mathbf{A} \cdot \mathbf{B}) - \mathbf{A} \cdot \mathbf{B}||_\infty \leq max(m, n, q) \cdot ||\mathbf{A}||_\infty \cdot ||\mathbf{B}||_\infty \cdot \epsilon_{mach} \ , \tag{4.3}$$

wobei $fl(\mathbf{A} \cdot \mathbf{B})$ das durch Rundungsfehler behaftete Ergebnis der Matrixmultiplikation in Maschinengenauigkeit und $||\mathbf{A}||_\infty$ die Zeilensummennorm notiert:

$$||\mathbf{A}||_\infty = \max_i \sum_j |a_{ij}|. \tag{4.4}$$

Basierend auf dieser Betrachtung wurden weitergehende Verfahren entwickelt, die einen Fehlerschwellwert für die durch algorithmenbasierte Fehlertoleranz geschützte Matrixmultiplikation ermitteln. Gunnels und Katz [33] definieren normbasierte Schwellwerte, deren Überschreitung eine Fehlerkorrektur nach sich zieht. Angepasst auf die partitionierte Kodierung mit einer ABFT-Blockgröße BS sind diese gegeben als:

$$||\mathbf{d}||_\infty > \tau \cdot ||\mathbf{A}||_\infty \cdot ||\mathbf{B}||_\infty \quad \text{und} \tag{4.5}$$

$$||\mathbf{e^T}||_\infty > \tau \cdot ||\mathbf{A}||_\infty \cdot ||\mathbf{B}||_\infty \quad \text{mit} \tag{4.6}$$

$$\tau = max(BS, n) \cdot \epsilon_{mach}. \tag{4.7}$$

Der Vektor $\mathbf{d}$ ist definiert als die Differenz aus dem Reihenprüfsummenvektor nach der (möglicherweise fehlerbehafteten) Multiplikation $\mathbf{C}^* \cdot \mathbf{w}$ und dem durch die lineare Operation

entstandenen Reihenprüfsummenvektor $\mathbf{A} \cdot (\mathbf{B} \cdot \mathbf{w})$ mit dem Vektor von uniformen Gewichten $\mathbf{w} = (1, \ldots, 1)$. Der Vektor $\mathbf{e}$ ist analog für den Spaltenprüfsummenvektor definiert. In weiteren Arbeiten der Autoren [34, 35] wurde der Faktor τ als von der Eingabe unabhängiger Faktor definiert, dessen Einfluss auf die Fehlererkennung in Experimenten untersucht wurde und somit eine Mischung aus deterministischem und experimentellem Ansatz vereint. Der Fehlerschwellwert bezieht sich nicht auf einzelne Prüfsummenelemente, sondern auf einen gesamten ABFT-Block einer Matrix. Abbildung 4.1 zeigt die für den Fehlerschwellwert dieses Verfahrens benötigte Daten.

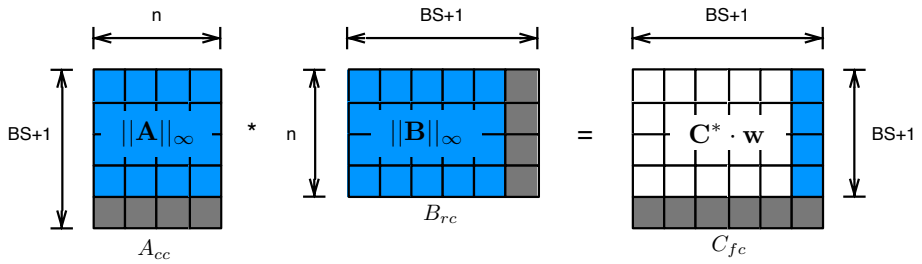


Abbildung 4.1.: Für den Fehlerschwellwert benötigte Daten für das Verfahren von Gunnels und Katz [33].

Ein deterministisches Verfahren zur Fehlerschwellwertbestimmung für einzelne Prüfsummenelemente wurde von Roy-Chowdhurry und Banerjee vorgestellt [36]. Der hierbei ermittelte Fehlerschwellwert für ein Prüfsummenelement ist abhängig von der gemeinsamen Dimension n der Eingangsmatrizen $\mathbf{A}$ und $\mathbf{B}$ der Größe $m \times n$ und $n \times q$, sowie der ABFT-Blockgröße BS. Zusätzlich haben die euklidischen Normen der an der Prüfsummenbildung beteiligten Vektoren $\mathbf{a}$, $\mathbf{b}$ und dem Prüfsummenvektor der Ergebnismatrix $\mathbf{s}$ Einfluss auf die Fehlerschwellwerte. Die euklidische Norm für Vektoren der Größe n ist definiert als

$$||\mathbf{a}|| = \sqrt{\sum_{i=1}^{n} |a_i|^2}. \tag{4.8}$$

Abbildung 4.2 zeigt die für diese Schwellwertbestimmung benötigten Daten am Beispiel der Reihenprüfsummen.

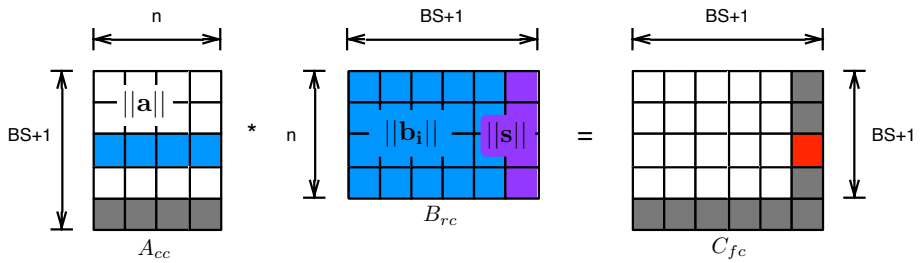


Abbildung 4.2.: Für den Fehlerschwellwert eines Reihenprüfsummenelements benötigte Daten nach Roy-Chowdhury und Banerjee [36].

Angepasst auf die partitionierte Kodierung ergibt sich der Fehlerschwellwert für die Spaltenprüfsummen zu:

$$\epsilon = |fl(c) - fl(c^*)| < ((n + 2 \cdot BS - 2) \cdot ||\mathbf{b}|| \cdot \sum_{i=1}^{BS} ||\mathbf{a_i}|| + n \cdot ||\mathbf{s}|| \cdot ||\mathbf{b}||) \cdot \epsilon_{mach}. \quad (4.9)$$

Für Reihenprüfsummen ergibt sich der Fehlerschwellwert zu:

$$\epsilon = |fl(c) - fl(c^*)| < ((n + 2 \cdot BS - 2) \cdot ||\mathbf{a}|| \cdot \sum_{i=1}^{BS} ||\mathbf{b_i}|| + n \cdot ||\mathbf{s}|| \cdot ||\mathbf{a}||) \cdot \epsilon_{mach}. \quad (4.10)$$

4.3. Weitere Verfahren

In weiterführenden Verfahren wird ein zusätzlicher Schutz der Mantissen verwendet, um die Fähigkeit der Fehlerdetektion bei der ABFT-geschützten Matrixmultiplikation zu erhöhen [37]. Dabei werden für die Eingangsmatrizen $\mathbf{A}$ und $\mathbf{B}$ die Mantissen aller Matrixelemente extrahiert und in Integer-Matrizen $\mathbf{A}_{mant}$ und $\mathbf{B}_{mant}$ gespeichert. Diese Integer-Matrizen werden wie die Eingangsmatrizen mit Reihen- bzw. Spaltenprüfsummen versehen. Bei Durchführung der Matrixmultiplikation $\mathbf{A_{cc}} \cdot \mathbf{B_{rc}} = \mathbf{C_{fc}}$ wird für jedes dabei berechnete Produkt die Mantisse extrahiert und gegebenenfalls denormalisiert. Zur Überprüfung wird eine weitere Integer-Matrix in der Größe der Ergebnismatrix berechnet. Der Eintrag an der Stelle (i, j) dieser Matrix ergibt sich durch Summation der extrahierten Mantissen jedes auftretenden Produkts für dieses Element bei der Durchführung der Matrixmultiplikation $\mathbf{A_{cc}} \cdot \mathbf{B_{rc}} = \mathbf{C_{fc}}$. Im Anschluss können für diese Integer-Matrix wiederum die Prüfsummen gebildet werden und mit den Prüfsummen verglichen werden, die durch Multiplikation der Mantisse des Prüfsummenelements in $\mathbf{A_{cc}}$ und der Matrix $\mathbf{B}_{mant}$ berechnet wird.

Dieses Verfahren hat den Nachteil, dass Integer-Operationen in der Größenordnung $\mathcal{O}(n^3)$ bei der Extraktion und Denormalisierung der Mantissen für jedes bei der Matrixmultiplikation gebildete Produkt auftreten. Zusätzlich müssen für die Ausgangsmatrizen die Mantissen extrahiert werden ($\mathcal{O}(n^2)$) und Referenzprüfsummen für die Mantissen berechnet werden ($\mathcal{O}(n^2)$). Bei dieser Methode wird die Addition in Gleitkommaarithmetik nicht betrachtet und es können nur Fehler detektiert werden, die in den unteren Bits des Mantissenprodukts auftreten.

5. Probabilistische Rundungsfehlerabschätzung

Während analytisch bestimmte obere Schranken für den bei der Überprüfung der Prüfsummen auftretenden Rundungsfehler [38] meist zu pessimistisch sind, wurden bessere Schranken durch vereinfachte Fehleranalysetechniken vorgestellt [36]. Bei experimentellen Untersuchungen stellte sich heraus, dass auch diese Schranken noch einige Magnituden vom echten, akkumulierten Rundungsfehler entfernt sind. Ein Problem hierbei ist, dass das Auftreten des schlimmstmöglichen Falls bezüglich des akkumulierten Rundungsfehlers sehr unwahrscheinlich ist. Um einen praktikablen Fehlerschwellwert zu erhalten ist es sinnvoll, von einer deterministisch bestimmten oberen Schranke abzuweichen und eine Schranke zu finden, die mit genügend hoher Wahrscheinlichkeit nicht überschritten wird und eine realistischere Abschätzung für den Rundungsfehler liefert.

Das Ziel bei der probablistischen Rundungsfehlerabschätzung ist es, für einen zu überprüfenden Wert x ein Konfidenzintervall, bestehend aus einem Erwartungswert $\mathtt{EV}(x)$ und einer Varianz $\mathtt{Var}(x)$ zu finden. Bei Annahme einer Standardnormalverteilung kann mit Hilfe der Standardabweichung $\sigma(x) = \sqrt{\mathtt{Var}(x)}$ und einem Faktor ω dieses Konfidenzintervall so gewählt werden, dass dieser Wert zu einer beliebigen Wahrscheinlichkeit kleiner 100% innerhalb dieses Intervalls liegt. Das Konfidenzintervall ist dabei definiert als:

$$[\mathtt{EV}(x) - \omega \cdot \sigma(x),\ \mathtt{EV}(x) + \omega \cdot \sigma(x)]. \tag{5.1}$$

Oft verwendete Werte für ω sind in der nachfolgenden Tabelle dargestellt:

$\omega \cdot \sigma$	% der Proben innerhalb des Konfidenzintervalls
$1 \cdot \sigma$	68,3 %
$2 \cdot \sigma$	95,4 %
$3 \cdot \sigma$	99,7 %

Die Arbeit von Barlow und Bareiss [39] untersucht die Rundungsfehlerverteilung bei verschiedenen Operationen der Fließkommaarithmetik. Bei Anwendung der grundlegenden Gleitkommaoperationen Addition, Subtraktion, Multiplikation und Division kommt es zu einem absoluten Rundungsfehler ϵ:

$$s = a_1 \text{ op } a_2 = s^* + \epsilon. \tag{5.2}$$

Hierbei bezeichnet s das exakte Ergebnis der Operation und s^* das mit einem Rundungsfehler behaftete Ergebnis. Für eine Darstellung in einem Gleitkommasystem mit Basis 2, Exponent

E und Darstellung der Mantissen $x, x^* \in [1/2, 1]$ ist $s = x \cdot 2^E$ und $s^* = x^* \cdot 2^E$. Der absolute Rundungsfehler kann damit durch den Mantissenfehler β dargestellt werden:

$$\epsilon = (x - x^*) \cdot 2^E = \beta \cdot 2^E. \tag{5.3}$$

Wird für die vier Gleitkommaoperationen angenommen, dass x reziprokal verteilt ist, kann eine Verteilung für β abgeleitet werden. Sind Erwartungswert und Varianz von β bekannt, lassen sich diese auf das Ergebnis einer oder mehrerer Gleitkommaoperationen s^* beziehen und sich damit der absolute Rundungsfehler abschätzen:

$$\mathtt{EV}(\epsilon) = sgn(s^*) \cdot 2^E \cdot \mathtt{EV}(\beta) \tag{5.4}$$

$$\mathtt{Var}(\epsilon) = 2^{2E} \cdot \mathtt{Var}(\beta) \tag{5.5}$$

$$E = \lceil log_2 |s^*| \rceil \tag{5.6}$$

5.1. Die reziproke Verteilung von Mantissenbits

Die reziproke Verteilung lässt sich in vielen empirisch ermittelten Datensätzen bei Betrachtung der Verteilung der ersten Ziffern finden und wird allgemein als *Benfordsches Gesetz* [40] bezeichnet. Für eine Basis B ist die reziproke Verteilung definiert als

$$r(x) = \frac{1}{x \cdot \ln(B)} \quad \text{mit} \quad x \in [1/B, 1] \tag{5.7}$$

und in Abbildung 5.1 für die Basis $B = 2$ dargestellt.

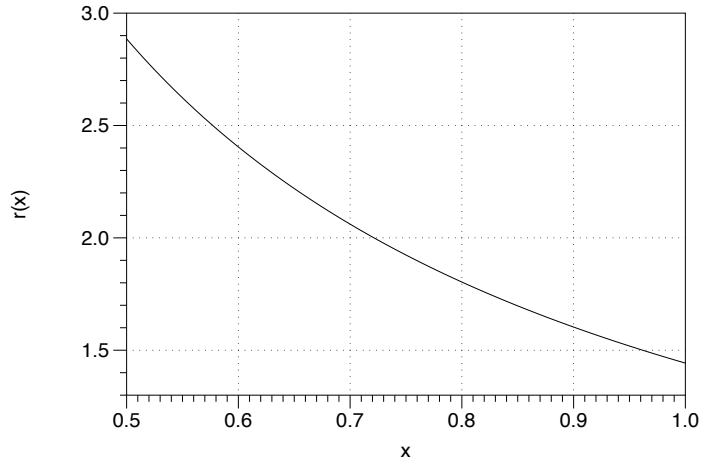


Abbildung 5.1.: Die reziproke Verteilung für die Basis $B = 2$.

Diese statistische Eigenheit wird beispielsweise zur Aufdeckung von Datenfälschungen in Wirtschaft und Forschung benutzt oder kann zur Aufdeckung von Wahlbetrug eingesetzt werden [41, 42].

Für den Bereich der Gleitkommaarithmetik beschäftigt sich die Arbeit von Hamming [43] mit der Verteilung von Mantissenbits und der Auswirkung verschiedener Operationen auf diese Verteilung. Sei $a = x \cdot 2^{E_a}$ und $b = y \cdot 2^{E_b}$ die Darstellung zweier Zahlen in Gleitkommaarithmetik mit Mantissen x und y, sowie Exponenten E_a und E_b. Dann kann für eine angenommene Verteilung $f(x)$ der Mantisse von a und eine Verteilung $g(y)$ der Mantisse von b eine Aussage über die Verteilung der Mantisse $h(z)$ des Ergebnisses $c = a \odot b = z \cdot 2^{E_c}$ mit $\odot \in \{+, -, \cdot, /\}$ getroffen werden. Die Arbeit von Pinkham [44] zeigt, dass für eine beliebige Verteilung der Mantissenbits von a und b bei Anwendung von Multiplikation und Division die Verteilung der Mantissenbits des Ergebnisses sich der reziproken Verteilung annähert. Hamming erläutert daraufhin, dass die Verteilung der Mantissenbits des Ergebnisses einer Multiplikation oder Division der reziproken Verteilung folgt, sobald die Verteilung der Mantissenbits eines Operanden dieser Verteilung folgt [43]. Bei einer Reihe von Multiplikationen und Divisionen nähert sich die Verteilung der Mantissenbits also der reziproken Verteilung an, sobald nur ein Faktor der reziproken Verteilung folgt. Für die Addition und Subtraktion findet sich eine ausführliche Betrachtung in [39]. Zusammengefasst lässt sich sagen, dass bei Anwendung genügend vieler Operationen die Verteilung der Mantissenbits einer reziproken Verteilung folgt.

Basierend auf der probabilistischen Rundungsfehlerabschätzung lassen sich Fehlerschwellwerte für die Prüfsummenelemente der ABFT-Matrixmultiplikation ermitteln [45].

5.2. Rundungsfehlerabschätzung für die Summation

Bei der Summation von n Werten $x_1, \ldots, x_n$ gilt die folgende Rekursion für Zwischenergebnisse in Maschinengenauigkeit s_k^* und Rundungsfehler ϵ_k:

$$s_{k+1}^* + \epsilon_{k+1} = s_k^* + x_{k+1} \quad \text{für } k = 1, \ldots, n-1. \tag{5.8}$$

In jedem Summationsschritt wird durch die Addition des Werts x_{k+1} auf das Zwischenergebnis s_k^* ein neues, durch einen zusätzlichen Rundungsfehler ϵ_{k+1} behaftetes Ergebnis in Maschinengenauigkeit s_{k+1}^* berechnet. Die Rekursion für die Differenz Δs_k zwischen exaktem Ergebnis und Ergebnis in Maschinengenauigkeit $\Delta s_k = s_k - s_k^*$ ist $\Delta s_{k+1} = \Delta s_k + \epsilon_{k+1}$ und kann damit aufgelöst werden zu:

$$\Delta s_n = \sum_{k=2}^{n} \epsilon_k. \tag{5.9}$$

Nun soll für den Rundungsfehler Δs_n dieser Summe s_n der Rundungsfehler probabilistisch bestimmt werden, also ein Konfidenzintervall bestehend aus $\mathtt{EV}_{\mathtt{Sum}}(\Delta s_n)$ und $\mathtt{Var}_{\mathtt{Sum}}(\Delta s_n)$ gefunden werden:

$$\mathtt{EV}_{\mathtt{Sum}}(\Delta s_n) = \sum_{k=2}^{n} \mathtt{EV}(\epsilon_k) \tag{5.10}$$

und

$$\mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) = \sum_{k=2}^{n} \mathtt{Var}(\epsilon_k). \tag{5.11}$$

Für die Addition und Subtraktion zweier Gleitkommazahlen bei symmetrischem Runden gilt für den Mantissenfehler β unter der Annahme, dass jeder Summand groß genug ist um eine Änderung im Ergebnis der Operation hervorzurufen [39]:

$$\mathtt{EV}(\beta) = 0 \tag{5.12}$$

und

$$\mathtt{Var}(\beta) \leq \frac{1}{8} \cdot 2^{-2t}. \tag{5.13}$$

Mit Gleichung 5.4 und 5.5 ergibt sich für den Erwartungswert der gesamten Summe:

$$\mathtt{EV}_{\mathtt{Sum}}(\Delta s_n) = 0 \tag{5.14}$$

und für die Varianz

$$\begin{aligned} \mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) &= \sum_{k=2}^{n} \mathtt{Var}(\epsilon_k) && (5.15)\\ &\leq \sum_{k=2}^{n} 2^{2E_k} \cdot \mathtt{Var}(\beta) && (5.16)\\ &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} 2^{2E_k}, && (5.17) \end{aligned}$$

wobei E_k den Exponent des Zwischenergebnisses s_k^* nach Addition des k-ten Summanden denotiert. Durch den Einsatz einer normalisierten Darstellung gilt $E_k \leq s_k^*$ und der Term kann direkt auf die Zwischenergebnisse der Summation s_k^* bezogen werden:

$$\mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) \leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} (s_k^*)^2. \tag{5.18}$$

Eine Möglichkeit zur weiteren Vereinfachung dieses Terms besteht darin, eine obere Schranke y für die dabei addierten Summanden zu finden. Das Zwischenergebnis im k-ten Schritt ist dadurch maximal, wenn k mal dieser Wert aufaddiert wurde und $s_k^* \leq k \cdot y$ gilt. Mit dieser Abschätzung ergibt sich die Gleichung 5.18 zu

$$\begin{aligned} \mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} (k \cdot y)^2 && (5.19)\\ &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \left(\frac{n \cdot (n+1) \cdot (2n+1)}{6} \right) \cdot y^2. && (5.20) \end{aligned}$$

Mit $\sigma(x) = \sqrt{\mathtt{Var}(x)}$ ergibt sich das Konfidenzintervall zu:

$$\mathtt{EV}(\Delta s_n) = 0 \tag{5.21}$$

und

$$\sigma(\Delta s_n) \leq \sqrt{\frac{n \cdot (n+1) \cdot (2n+1)}{48}} \cdot y \cdot 2^{-t}. \tag{5.22}$$

5.3. Rundungsfehlerabschätzung für das Skalarprodukt

Das Skalarprodukt zweier Vektoren **a** und **b** mit n Elementen ist definiert als

$$< \mathbf{a}, \mathbf{b} >= \sum_{k=1}^{n} c_k = \sum_{k=1}^{n} a_k \cdot b_k. \tag{5.23}$$

Zusätzlich zu dem bei der Summation auftretenden Rundungsfehler kommt es bei jedem Multiplikationsschritt $c_k = a_k \cdot b_k$ zu einem weiteren Rundungsfehler α_k:

$$c_k^* + \alpha_k = c_k. \tag{5.24}$$

Auch hier bezeichnet c_k^* das Ergebnis nach der Rundung in Maschinengenauigkeit und c_k das exakte Ergebnis. Für die Rekursion der beim Skalarprodukt gebildeten Summe gilt $\Delta s_{k+1} = \Delta s_k + \epsilon_{k+1} + \alpha_{k+1}$ mit dem bei der Addition eingeführten Rundungsfehler ϵ_{k+1} und dem bei der Multiplikation eingeführten Rundungsfehler α_{k+1} und kann aufgelöst werden zu einem Rundungsfehler für das gesamte Skalarprodukt:

$$\Delta s_n = \sum_{k=2}^{n} \epsilon_k + \sum_{k=1}^{n} \alpha_k. \tag{5.25}$$

Analog zu Gleichung 5.10 und 5.11 kann der Erwartungswert definiert werden als

$$\mathtt{EV}_{\mathtt{Skal}}(\Delta s_n) = \mathtt{EV}_{\mathtt{Sum}}(\Delta s_n) + \mathtt{EV}_{\mathtt{Mul}}(\Delta s_n) \tag{5.26}$$

$$= \sum_{k=2}^{n} \mathtt{EV}(\epsilon_k) + \sum_{k=1}^{n} \mathtt{EV}(\alpha_k) \tag{5.27}$$

und die Varianz mit

$$\mathtt{Var}_{\mathtt{Skal}}(\Delta s_n) = \mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) + \mathtt{Var}_{\mathtt{Mul}}(\Delta s_n) \tag{5.28}$$

$$= \sum_{k=2}^{n} \mathtt{Var}(\epsilon_k) + \sum_{k=1}^{n} \mathtt{Var}(\alpha_k) \tag{5.29}$$

beschrieben werden.

Der bei der Summation auftretende Erwartungswert und die Varianz des Rundungsfehlers wurde bereits erläutert, es gilt für die Bildung eines Konfidenzintervalls den Erwartungswert und die Varianz für die Multiplikation zu bestimmen. Für die Multiplikation und Division zweier Gleitkommazahlen gilt für die Varianz [39]:

$$\mathtt{Var}(\beta) = \frac{1}{12} \cdot 2^{-2t}. \tag{5.30}$$

Für den Erwartungswert gilt bei Multiplikation und Division mit symmetrischem Runden:

$$\mathtt{EV}(\beta) = \frac{1}{3} \cdot 2^{-2t}. \tag{5.31}$$

Der Anteil für die Varianz des Rundungsfehler, der durch die k Multiplikationen bei der Bildung des Skalarprodukts eingeführt wird, ist gegeben durch die Summe aller Varianzen:

$$\mathtt{Var}_{\mathtt{Mul}}(\Delta s_n) = \sum_{k=1}^{n} \mathtt{Var}(\alpha_k). \tag{5.32}$$

Mit der Annahme eines Werts $y = a_q \cdot b_q$ mit $q \in \{1, \ldots, n\}$, bei dem die Varianz des Rundungsfehler $\mathtt{Var}(\alpha_q)$ maximal ist kann durch

$$\mathtt{Var}_{\mathtt{Mul}}(\Delta s_n) \leq n \cdot \mathtt{Var}(\alpha_q) \tag{5.33}$$

der gesamte, beim Skalarprodukt durch Multiplikation auftretende Rundungsfehler abgeschätzt werden. Bei Betrachtung der Beziehung zwischen absolutem Rundungsfehler und der Verteilung des Mantissenfehlers mit $\mathtt{Var}(\epsilon) = 2^{2E} \cdot \mathtt{Var}(\beta)$ (Gleichung 5.5) und $E = \lceil log_B |s^*| \rceil$ (Gleichung 5.6) kann festgestellt werden, dass die Varianz maximal ist, wenn der Exponent des Ergebnisses der Multiplikation maximal ist. Für alle $k \in \{1, \ldots, n\}$ gilt deshalb

$$\mathtt{Var}(\alpha_q) = 2^{2E_q} \cdot \mathtt{Var}(\beta) \geq 2^{2E_k} \cdot \mathtt{Var}(\beta) = \mathtt{Var}(\alpha_k) \tag{5.34}$$

und y kann als obere Schranke verwendet werden. Damit lässt sich die Varianz des Rundungsfehlers, der bei der Multiplikation eingeführt wird abschätzen:

$$\begin{aligned} \mathtt{Var}_{\mathtt{Mul}}(\Delta s_n) &\leq n \cdot \mathtt{Var}(\alpha_y) && (5.35)\\ &\leq n \cdot y^2 \cdot \mathtt{Var}(\beta) && (5.36)\\ &= \frac{n}{12} \cdot 2^{-2t} \cdot y^2. && (5.37) \end{aligned}$$

Für den Erwartungswert des Rundungsfehlers bei der Multiplikation lässt sich eine analoge Betrachtung durchführen, der Erwartungswert bei symmetrischem Runden ergibt sich dadurch zu:

$$\begin{aligned} \mathtt{EV}_{\mathtt{Mul}}(\Delta s_n) &= n \cdot y \cdot \mathtt{EV}(\beta) && (5.38)\\ &\leq \frac{n}{3} \cdot 2^{-2t} \cdot y. && (5.39) \end{aligned}$$

Zusammengefasst mit dem Erwartungswert und der Varianz für die Summation ergibt sich eine Standardabweichung

$$\begin{aligned} \sigma(\Delta s_n) &= \sqrt{\mathtt{Var}_{\mathtt{Skal}}(\Delta s_n)} && (5.40)\\ &= \sqrt{\mathtt{Var}_{\mathtt{Mul}}(\Delta s_n) + \mathtt{Var}_{\mathtt{Sum}}(\Delta s_n)} \quad \leq \sqrt{\frac{n \cdot (n+1) \cdot (n+\frac{1}{2}) + 2 \cdot n}{24}} \cdot 2^{-t} \cdot y && (5.41) \end{aligned}$$

und ein Erwartungswert in $\mathcal{O}(2^{-2t})$, der angesichts des $\mathcal{O}(2^{-t})$ Terms für $\sigma(\Delta s_n)$ vernachlässigt werden kann.

5.4. Fehlerschwellwerte für Prüfsummenelemente

Bei der Überprüfung der Prüfsummen bei der ABFT-Matrixmultiplikation werden zwei Prüfsummen miteinander verglichen:

1. Die durch die lineare Operation der Matrixmultiplikation entstandene Referenzprüfsumme $c_{i,j}$.
2. Die durch Summation der Elemente in der Ergebnismatrix entstandene Prüfsumme $\hat{c}_{i,j}$.

Das Element $c_{i,j}$ entstehe dabei durch das Skalarprodukt eines Daten(reihen)vektors $\mathbf{a_i}$ der Matrix $\mathbf{A_{cc}}$ und einem Prüfsummen(spalten)vektor $\mathbf{b_j}$ der Matrix $\mathbf{B_{rc}}$. Die Betrachtung für Prüfsummenreihenvektoren $\mathbf{a_i}$ und Datenspaltenvektoren $\mathbf{b_j}$ kann analog durchgeführt werden. Wird bei der Bildung des Skalarprodukts symmetrisches Runden bei der Multiplikation verwendet, kann für jedes Element $c_{i,j}$ das Konfidenzintervall direkt nach Gleichung 5.41 bestimmt werden, da der Erwartungswert hier vernachlässigbar klein ist. Zusätzlich dazu wird ein Rundungsfehler bei der Bildung des Referenzprüfsummenvektors eingeführt: Für jedes Element $b_{k,j}$ dieses Prüfsummenvektors werden BS Datenelemente summiert. Der gesamte Rundungsfehler, der bei dieser Operation auftritt, kann damit beschrieben werden durch:

$$\begin{aligned}
\mathtt{Var}(\Delta c_{i,j}) = & \underbrace{\mathtt{Var}_{\mathtt{Skal}}(\Delta c_{i,j})}_{n \text{ Prüfsummenelemente}} + \sum_{k=1}^{n} \underbrace{\mathtt{Var}_{\mathtt{Sum}}(\Delta b_{k,j})}_{BS \text{ Datenelemente}} && (5.42)\\
= & \underbrace{\mathtt{Var}_{\mathtt{Mul}}(\Delta c_{i,j})}_{n \text{ Prüfsummenelemente}} + \underbrace{\mathtt{Var}_{\mathtt{Sum}}(\Delta c_{i,j})}_{n \text{ Prüfsummenelemente}} + \sum_{k=1}^{n} \underbrace{\mathtt{Var}_{\mathtt{Sum}}(\Delta b_{k,j})}_{BS \text{ Datenelemente}} && (5.43)\\
= & \quad \mathcal{O}(BS \cdot n) \quad + \quad \mathcal{O}(BS \cdot n^3) \quad + \quad \mathcal{O}(n \cdot BS^3) && (5.44)
\end{aligned}$$

Für die neu berechnete Prüfsumme $\hat{c}_{i,j}$ werden bei einer ABFT-Blockgröße BS Elemente der Ergebnismatrix summiert, die durch Bildung des Skalarprodukts von Datenvektoren entstanden sind:

$$\hat{c}_{i,j} = \sum_{l=1}^{BS} c_{i-l,j}. \tag{5.45}$$

Zusätzlich zu den durch die Bildung des Skalarprodukts entstandenen BS Rundungsfehlern in jedem Element der Ergebnismatrix wird ein Rundungsfehler bei der Summation dieser BS Werte eingeführt. Auch hier kann der gesamte Rundungsfehler beschrieben werden:

$$\begin{aligned}
\mathtt{Var}(\Delta \hat{c}_{i,j}) = & \sum_{i=1}^{BS} \underbrace{\mathtt{Var}_{\mathtt{Skal}}(\Delta c_{i-l,j})}_{n \text{ Datenelemente}} + \underbrace{\mathtt{Var}_{\mathtt{Sum}}(\Delta \hat{c}_{i,j})}_{BS \text{ Datenelemente}} && (5.46)\\
= & \sum_{i=1}^{BS} \underbrace{\mathtt{Var}_{\mathtt{Mul}}(\Delta c_{i-l,j})}_{n \text{ Datenelemente}} + \sum_{i=1}^{BS} \underbrace{\mathtt{Var}_{\mathtt{Sum}}(\Delta c_{i-l,j})}_{n \text{ Datenelemente}} + \underbrace{\mathtt{Var}_{\mathtt{Sum}}(\Delta \hat{c}_{i,j})}_{BS \text{ Datenelemente}} && (5.47)\\
= & \quad \mathcal{O}(BS \cdot n) \quad + \quad \mathcal{O}(BS \cdot n^3) \quad + \quad \mathcal{O}(BS^3) && (5.48)
\end{aligned}$$

Wird das Skalarprodukt mittels *Fused Multiply-Add*-Operationen berechnet, muss lediglich die Varianz für die Summation betrachtet werden. Die Multiplikationsterme entfallen, da bei der *Fused Multiply-Add*-Operation nur ein Rundungsschritt nach der Addition eingesetzt wird und das Zwischenergebnis aus der Multiplikation mit doppelter Mantissengenauigkeit für die Addition zum vorhergehenden Wert zur Verfügung steht.

Diese Betrachtung macht deutlich, dass die beiden Abschätzungen für $c_{i,j}$ und $\hat{c}_{i,j}$ asymptotisch gleich sind. Eine probabilistische Abschätzung erhebt weiterhin nicht den Anspruch, keine *False Positives* bei der Überprüfung zu haben, nur sollte die Wahrscheinlichkeit dafür entsprechend gering sein. Da bei der Abschätzung maximale Elemente verwendet werden, um beispielsweise die Summe der Quadrate aller Zwischensummen abzuschätzen, wird der Fehlerschwellwert weiter erhöht.

Zusammengefasst legt diese Betrachtung nahe, den Rundungsfehler für die durch die lineare Operation der Matrixmultiplikation entstandene Prüfsumme $c_{i,j}$ abzuschätzen und diesen Wert als Fehlerschwellwert zu nutzen.

6. ABFT-BLAS-Bibliothek

6.1. Massiv-parallele Rechnerarchitekturen

Parallelisierung und parallele Architekturen

Bei der Parallelisierung von Algorithmen werden Teilbereiche dieser Algorithmen identifiziert, die untereinander keine Abhängigkeiten zeigen und somit nicht nacheinander ausgeführt werden müssen. Dadurch ist es möglich, diese Teile von mehreren Berechnungseinheiten parallel ausführen zu lassen. Das Ergebnis dieser Operationen kann nach der Ausführung wieder zusammengeführt werden, wofür explizite oder implizite Synchronisations- und Transferanweisungen benutzt werden. Eine genaue Analyse des zu parallelisierenden Problems ist dabei notwendig, um diese unabhängigen Teilbereiche identifizieren zu können. Allgemeine Parallelisierungsansätze basieren auf der örtlichen, der zeitlichen und der örtlich-zeitlichen Unabhängigkeit von Daten. Soll beispielsweise ein Histogramm für die Pixel eines Bildes erstellt werden, kann dieses Bild in Teilbereiche unterteilt werden. Das Histogramm jedes Teilbereichs kann örtlich unabhängig voneinander berechnet werden und in einem nachfolgenden Schritt zu einem Histogramm für das gesamte Bild zusammengeführt werden. Die Frage, welche Bildwerte sich in zwei aufeinander folgenden Bildern einer Sequenz von mehreren Bildern ändern, kann durch Ausnutzung zeitlicher Unabhängigkeit oder zeitlich-örtlicher Unabhängigkeit parallelisiert werden: Jeder Prozessor kann für ein Paar von aufeinander folgenden Bildern diese Information berechnen. Bei zeitlich-örtlicher Unabhängigkeit kann jedes Bildpaar zusätzlich in Teilbereiche aufgeteilt werden, die unabhängig voneinander berechnet werden können.

Es existiert eine große Bandbreite an Systemen und Rechnerarchitekturen, die eine Parallelisierung erlauben. Diese unterschieden sich in der Anzahl parallel ausführbarer Operationen und der Methode, wie die Kommunikation innerhalb des Systems integriert ist. Ein Mehrkernprozessor besteht beispielsweise aus einer Anzahl von unabhängig arbeitenden Prozessorkernen. Da jeder dieser Prozessorkerne auf einen einheitlichen Speicher zugreifen kann muss kein Transfer von Daten durchgeführt werden, es genügen Synchronisationsanweisungen um die Datenintegrität zu wahren. Werden mehrere Rechner in ein Netzwerk integriert, kann jeder Knoten dieses Rechnerverbunds ein oder mehrere Teilprobleme bearbeiten. Dieses verteilte Rechnen wird besonders in großen Rechnerverbünden eingesetzt. Die Systeme unterschieden sich in der Art, wie die Rechner untereinander vernetzt sind, aus welcher Hardware die einzelnen Knoten bestehen und wie die Speicherung von Daten erfolgt. Die Kommunikation und Synchronisation zwischen den einzelnen Knoten muss durch explizite Kommunikationsanweisungen durchgeführt werden, bei denen Daten zwischen den Knoten ausgetauscht werden. Darüber hinaus gibt es speziell auf die parallele

Verarbeitung von Daten ausgelegt Rechnerarchitekturen, wie beispielsweise Vektorrechner oder Grafikprozessoren.

Nicht jedes Problem kann allerdings beliebig parallelisiert werden, der maximale Geschwindigkeitszuwachs ist nach dem *Amdahlschen Gesetz* [46] durch die Anzahl eingesetzter Prozessoren und dem rein sequenziellen Anteil am gesamten Algorithmus begrenzt. Für einen parallel ausführbaren Anteil P eines Programms und der Verwendung von N Prozessoren ergibt sich dadurch der maximal erreichbare Geschwindigkeitszuwachs S zu:

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{6.1}$$

GPGPUs

Der Begriff *General Purpose Computation on Graphics Processing Units* (GPGPUs) beschreibt die Verwendung von Grafikprozessoren für Berechnungen außerhalb des ursprünglich für diese Prozessoren vorgesehenen Einsatzgebiets. Diese massiv-parallelen Rechnerarchitekturen zeichnen sich im Vergleich zu herkömmlichen Prozessoren, den *Central Processing Units* (CPUs), durch eine signifikant größere Anzahl an Prozessorkernen aus. Während CPUs zu den latenzoptimierten Prozessoren gezählt werden, bei denen der Architekturentwurf auf eine möglichst schnelle Abarbeitung eines Instruktionsstroms abgestimmt ist, können Grafikprozessoren den durchsatzoptimierten Architekturen zugeordnet werden. Ziel beim Entwurf dieser Architekturen ist es, einen Instruktionsstrom auf eine große Datenmenge anzuwenden und somit in Summe mehr Daten verarbeiten zu können. Nach der Flynn'schen Einteilung von Prozessoren sind die GPGPUs zur Klasse der *Single Instruction Multiple Data* (SIMD) zu zählen, ein treffenderer Begriff ist jedoch *Single Programm Multiple Data* (SPMD), da dieser das Ausführen von verschiedenen Abzweigungen in bedingten Anweisungen in die Einteilung mit einbezieht.

Die CUDA-Architektur

Die CUDA-Architektur besteht aus mindestens einem *Streaming Multi-Processor* und einem für dieses Gerät globalen Speicher. Jedes auf der GPU ausgeführte Programm kann sowohl in diesen globalen Speicher schreiben, als auch daraus lesen. Ein *Streaming Multi-Processor* besteht aus einer großen Anzahl an Prozessorkernen, die über einen im *Streaming Multi-Processor* lokalen Speicher miteinander verbunden sind, dem *Shared Memory*. Daten von einem *Shared Memory* zum anderen können nur über den globalen Speicher ausgetauscht werden. Die Prozessorkerne verfügen des Weiteren über eine Menge an lokale Registern, die nur für diesen Kern sichtbar sind.

Das CUDA-Programmiermodell

Der grundlegende Aufbau des CUDA-Programmiermodells besteht in der Definition von Kerneln und der hierarchischen Gruppierung von *Threads*. Ein Kernel ist ein Programmstück, das von mehreren *Threads* parallel ausgeführt wird, meist also den gleichen Instruktionsstrom auf unterschiedliche Datensätze anwendet. Diese *Threads* sind je nach Position in der Hierarchieebene in der Lage, sich über in der Latenz unterscheidende Speicher untereinander auszutauschen.

Die kleinste Gruppierung von *Threads* ist der *Warp*. Ein *Warp* besteht aus einer von der Hardware abhängenden Anzahl von *Threads*, deren Instruktionsstrom durch denselben *Instruction Scheduler* vorgegeben wird. In der aktuellen Generation der Hardware umfasst ein *Warp* 32 *Threads*. Die Abarbeitung dieser Instruktionen erfolgt einheitlich für alle *Threads* innerhalb dieses *Warps*, bei bedingten Anweisungen werden alle Abzweigungen ausgeführt, wobei jeweils nur die *Threads* aktiv sind, für die die Bedingung der derzeitigen Abzweigung erfüllt ist. Währenddessen warten alle anderen *Threads* innerhalb dieses *Warps*. Innerhalb eines *Warps* können Daten zwischen *Threads* durch sogenannte *Shuffle Instructions* ausgetauscht werden, bei denen kein expliziter Speicher für den Austausch verwendet werden muss und Daten aus Registern von anderen *Threads* in ein lokales Register gespeichert werden.

Die nächst größere Gruppierung von *Threads* ist der *Thread-Block*. Hierbei werden mehrere *Warps* zusammengefasst, der Austausch von Daten innerhalb eines *Thread-Blocks* erfolgt durch das *Shared Memory*. Diese Gruppierung von *Threads* kann in maximal drei Dimensionen erfolgen, jeder *Thread* hat eine eindeutige, lokal abrufbare Koordinate in x-, y- und z-Richtung. Durch explizite Synchronisationsanweisungen können alle *Threads* innerhalb eines *Thread-Blocks* synchronisiert werden. In der aktuellen Generation der Hardware kann ein *Thread-Block* aus maximal 1.024 *Threads* bestehen. Die Anzahl *Threads* pro *Thread-Block* ist nicht der einzige limitierende Faktor, auch die zur Verfügung stehenden Speicherressourcen sind limitiert. Ausschlaggebend für die maximale Anzahl *Threads* pro *Thread-Block* ist zum Einen die pro *Thread* angeforderte Menge an Registern, zum Anderen die für den *Thread-Block* angeforderte Kapazität des *Shared Memories*. Wird diese Kapazität überschritten, können nur *Thread-Blocks* von geringerer Anzahl *Threads* eingesetzt werden.

Wird mehr als ein *Thread-Block* für die Berechnung benötigt, können mehrere dieser *Thread-Blocks* in einem Gitter, dem *Grid* angeordnet werden. Dieses *Grid* kann ebenfalls in maximal drei Dimensionen definiert werden. Analog zum *Thread-Block* hat auch hier jeder *Thread* eine eindeutige, lokal abrufbare Information darüber, zu welchem *Thread-Block* er in x-, y- und z-Richtung gehört. Die Anzahl *Thread-Blocks* ist im Gegensatz zu den *Threads* pro *Thread-Block* nicht begrenzt, jedoch kann je nach Ressourcenanforderung des Kernels und Dimensionierung des *Thread-Blocks* nur eine maximale Anzahl *Thread-Blocks* pro *Streaming Multi-Processor* parallel bearbeitet werden. Über die Grenzen von *Thread-Blocks* hinaus können Instruktionen von *Threads* nicht explizit synchronisiert werden, der Austausch von Daten ist damit nur möglich, indem der Algorithmus in verschiedene Kernelaufrufe partitioniert wird und die Daten über den globalen Speicher ausgetauscht werden.

Eine grobgranulare Möglichkeit zur Parallelisierung von *Threads* besteht durch das Konzept der *CUDA-Streams*. Jeder aufgerufene Kernel kann einem bestimmten *CUDA-Stream* zugeordnet werden, die Abarbeitung von Kerneln innerhalb jedes *CUDA-Streams* erfolgt sequenziell. *Threads* aus anderen *CUDA-Streams* können durch die CUDA-Architektur nebenläufig zu den *Threads* aus diesem *CUDA-Stream* bearbeitet werden. Während beispielsweise auf das Ergebnis von stark latenzbehafteten Instruktionen, wie zum Beispiel dem Lesen oder Schreiben aus dem globalen Speicher gewartet wird, können auf der Hardware andere Berechnungen aus anderen *CUDA-Streams* durchgeführt werden. Über *CUDA-Events* können diese *CUDA-Streams* synchronisiert werden.

6.2. Analyse der Anforderungen

Verschiedene Anforderungen werden von unterschiedlicher Seite an eine effiziente ABFT-BLAS-Bibliothek gestellt. Dabei spielen nicht nur die Bedingungen und Anforderungen der eingesetzten Hardware und der algorithmenbasierten Fehlertoleranz eine große Rolle. Auch durch den Benutzer vorgegebene Rahmenbedingungen müssen analysiert und eingehalten werden, um eine effiziente und korrekte ABFT-BLAS-Bibliothek definieren zu können. Im Folgenden werden die durch die drei zentrale Faktoren gegebenen Anforderungen dargelegt.

Durch die Verwendung von GPUs werden folgende Anforderungen gestellt:

- Die ABFT-BLAS-Bibliothek muss aufgrund der physikalisch getrennten Speicherbereiche von Host und GPU die Verwaltung der Daten übernehmen. Dabei müssen Änderungen der Matrixelemente sowohl von Host-Seite, als auch von GPU-Seite her erkannt werden. Die von Änderungen betroffenen Teile der Matrix müssen transferiert werden, bevor von der anderen Seite aus auf diese Daten zugegriffen werden kann. Die Einhaltung dieser Konsistenz der Daten auf beiden Seiten ist zwingend erforderlich. Eine Abweichung von dieser Anforderung würde zwangsläufig zu falschen Ergebnissen führen.
- Die ABFT-BLAS-Bibliothek muss für eine performante Konzeption externe BLAS-Bibliotheken für grundlegende Operationen verwenden. Mit jeder neuen Hardwaregeneration werden von den Herstellern der GPUs neue, auf die jeweilige Hardware optimierte BLAS-Bibliotheken zur Verfügung gestellt. Diese Bibliotheken sind darauf ausgelegt, einen möglichst hohen Datendurchsatz zu erreichen. Um die ABFT-BLAS-Bibliothek auch in kommenden Hardwaregenerationen mit hoher Performanz einsetzen zu können, sollen elementare Operationen wie die Matrixmultiplikation, die Matrixaddition und die Matrixtransposition als *Black-Box-Modell* aus diesen hochoptimierten Bibliotheken verwendet werden.

Durch das ABFT-Schema werden folgende Anforderungen gestellt:

- Die ABFT-BLAS-Bibliothek muss die durch das ABFT-Schema geforderte Verteilung von Berechnungsaufgaben auf verschiedene Prozessoren einhalten. Besonders bei Einsatz eines *Black-Box-Modells* für die Berechnung der Daten innerhalb der ABFT-BLAS-Bibliothek muss in den Kodierungs- und Überprüfungsschritten Wert darauf gelegt werden, diese Operationen durch zusätzliche Maßnahmen so abzusichern, dass eine Maskierung von Fehlern durch eine fehlerhafte Berechnungseinheit unterbunden wird.

Durch den Benutzer werden folgende Anforderungen gestellt:

- Die ABFT-BLAS-Bibliothek sollte die standardisierte BLAS-Nomenklatur für Funktionsaufrufe übernehmen, um die Einbindung der ABFT-BLAS-Bibliothek in bestehende Anwendungen zu erleichtern. Durch diese Funktionsaufrufe können beispielsweise Teilmatrizen miteinander multipliziert werden oder eine Skalierung der Eingangsdaten vorgenommen werden. Bei der Fehlerschwellwertbestimmung in der ABFT-BLAS-Bibliothek müssen diese Faktoren berücksichtigt werden.
- Die ABFT-BLAS-Bibliothek sollte dem Benutzer eine detaillierte Rückmeldung darüber ermöglichen, in welchen Matrixelementen Fehler aufgetreten sind und wie diese korrigiert wurden.
- Die ABFT-BLAS-Bibliothek sollte dem Benutzer die Möglichkeit geben, auch nicht-BLAS Algorithmen auf der GPU integrieren zu können. Um das Potenzial der GPUs auch außerhalb der Funktionen einer BLAS-Bibliothek dem Anwender zur Verfügung zu stellen, sollen Datenstrukturen und Funktionsumfang der ABFT-BLAS-Bibliothek so ausgelegt sein, dass eine Integration spezialisierter nicht-BLAS Algorithmen durch den Anwender ermöglicht wird.

Analyse des Einsatzschemas

Um ein Konzept für eine ABFT-BLAS-Bibliothek zu erstellen, das eine effiziente Umsetzung für GPUs erlaubt, muss zunächst eine möglichst umfassende Analyse des Einsatzschemas von BLAS-Operationen durchgeführt werden. Da die GPU eine dedizierte Hardware mit eigenem physikalischem Speicher ist, müssen sämtliche verwendete Daten initial in diesen Speicher geladen werden. Analog dazu muss das Ergebnis nach einer Berechnung wieder aus dem Speicher der GPU in den Speicher des Host transferiert werden. Dieser Datentransfer ist dabei mit einer gewissen Latenz und einer maximalen Datentransferrate behaftet. Diese Werte sind durch die verwendete Busarchitektur bestimmt. In aktuellen Generationen der Hardware sind diese signifikant niedriger als die Anzahl Gleitkommaoperationen, die von einer GPU verarbeitet werden können. Der Datentransfer stellt damit einen der wichtigsten, potentiellen Engpässe dar, welcher bei der Konzeption der ABFT-BLAS-Bibliothek beachtet werden muss. Für eine effiziente ABFT-BLAS-Bibliothek bedeutet dies, dass der Datentransfer zwischen Host- und Devicearchtektur bedarfsorientiert ausgelegt sein muss und nur wirklich angefragte, beziehungsweise benötigte Daten transferiert werden.

Zwischen dem initialen und dem finalen Datentransfer wird eine bestimmte Folge von BLAS-Operationen auf die Daten im Speicher der GPU angewandt. Diese Folge von BLAS-Operationen und die verwendeten Operanden sind nicht immer statisch für die gesamte Anwendung vorgegeben. Vielmehr muss davon ausgegangen werden, dass diese Folge von Operationen erst dynamisch, beispielsweise in Abhängigkeit von Zwischenergebnissen, bestimmt wird. Für die ABFT-BLAS-Bibliothek bedeutet dies, dass erst zur Laufzeit der Anwendung festgestellt werden kann, welche BLAS-Operationen mit welchen Operanden ausgeführt werden. Da die Definition und Ausführungsreihenfolge der BLAS-Operationen durch den Benutzer erfolgt, muss bei der Konzeption der ABFT-BLAS-Bibliothek darauf geachtet werden, alle möglichen Operationen zu jedem Zeitpunkt mit einer möglichst geringen Latenz ausführen zu können. Seien beispielsweise m Matrizen der Größe $n \times n$ gegeben: $\mathbf{A_1}, \ldots, \mathbf{A_m}$. So kann erst beim Aufruf der entsprechenden Multiplikationsanweisung festgestellt werden, welche dieser Matrizen mit welcher Matrix multipliziert wird. Hinzu kommt, dass die Multiplikation von Matrizen nicht kommutativ ist, also bis auf wenige Ausnahmen $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A}$ gilt. Selbst wenn davon ausgegangen wird, dass nur eine einzelne Matrixmultiplikation mit diesen Daten ausgeführt wird, sind damit m^2 unterschiedliche Kombinationen möglich. Je mehr Operationen mit diesen Matrizen und den daraus entstehenden Ergebnismatrizen durchgeführt werden, desto mehr Kombinationsmöglichkeiten ergeben sich. Dies legt nahe, alle in der ABFT-BLAS-Bibliothek auftretenden zusätzlichen Operationen, wie zum Beispiel die Kodierung von Matrizen und die Fehlerschwellwertbestimmung, in kontextunabhängige und kontextabhängige Schritte aufzuteilen. Die kontextunabhängigen Operationen können damit zum frühst möglichen Zeitpunkt ausgeführt werden, sobald die zugehörigen Daten auf der GPU zur Verfügung stehen. Erst beim Aufruf einer expliziten BLAS-Operation ist der Kontext, in dem diese Daten eingesetzt werden, hinreichend definiert und die kontextabhängigen Schritte können ausgeführt werden. Diese können dabei auf die Zwischenergebnissen aus dem kontextunabhängigen Schritt zurückgreifen.

Neben dem Transfer von Zwischenergebnissen und deren Verarbeitung auf dem Host besteht weiterhin die Möglichkeit das Potenzial der Verwendung von GPUs auch in Operationen zu nutzen, die nicht der Klasse der BLAS-Operationen zuzuordnen sind. Diese nicht-BLAS Algorithmen sind vom Benutzer der ABFT-BLAS-Bibliothek selbst zu implementieren. Für die Konzeption der ABFT-BLAS-Bibliothek bedeutet dies, dem Benutzer eine Möglichkeit zur Bearbeitung der Daten auf der GPU zur Verfügung zu stellen. Dies umfasst neben dem Zugriff auf die Daten selbst auch eine Möglichkeit, eventuelle aufgetretene Wertänderungen der Matrixelemente an die ABFT-BLAS-Bibliothek mitzuteilen, sodass die Konsistenz der Daten zwischen Host und GPU aufrecht erhalten werden kann.

Der letzte zu betrachtende Faktor ist die Anwendung des ABFT-Schemas auf die in der ABFT-BLAS-Bibliothek verwendeten Daten. Nach dem Schema der algorithmenbasierten Fehlertoleranz müssen die Daten vor der Berechnung durch den ABFT-Algorithmus zunächst mit zusätzlichen Informationen kodiert werden. Nach der Berechnung durch den ABFT-Algorithmus muss das Ergebnis vor der Weiterverwendung wieder dekodiert werden. Die Abbildung 6.1 fasst diese Analyse anhand eines Ablaufdiagramms zusammen. Die in der ABFT-BLAS-Bibliothek verwendeten Daten durchlaufen verschiedene Schritte und Algorithmen, zulässige aufeinanderfolgende Schritte sind mit einem Pfeil gekennzeichnet.

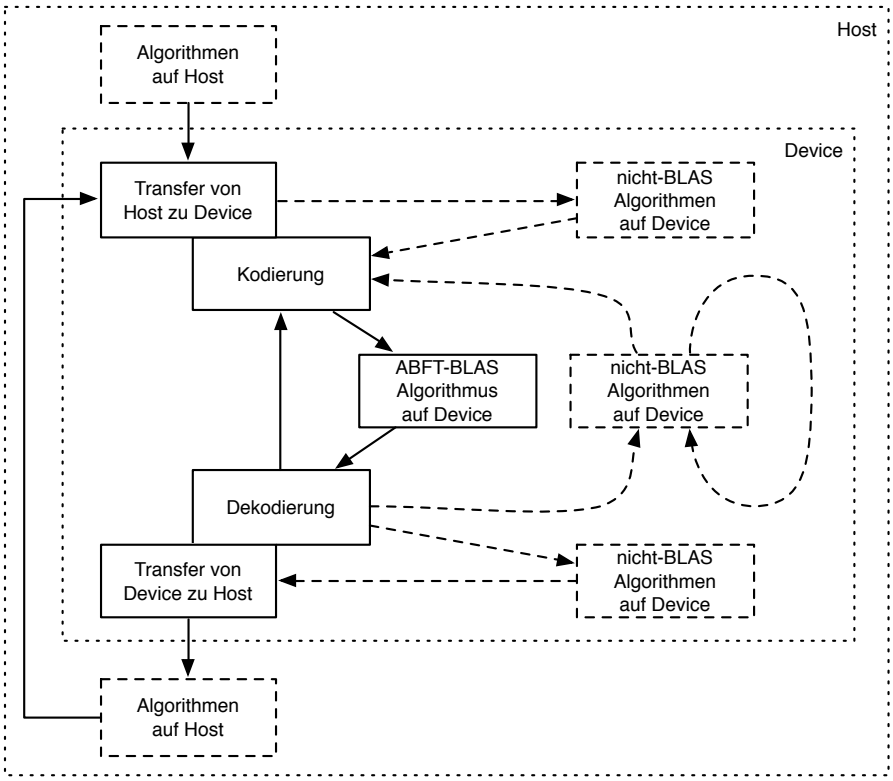


Abbildung 6.1.: Ablaufdiagramm für Daten in der ABFT-BLAS-Bibliothek. Zulässige Transitionen sind durch einen Pfeil markiert.

Analyse der ABFT-Matrixmultiplikation

Nach der allgemeinen Betrachtung der Verwendung von BLAS-Operationen mit algorithmenbasierter Fehlertoleranz auf GPUs wird diese Analyse nun auf spezielle Operationen der ABFT-BLAS-Bibliothek ausgeweitet. Von zentraler Bedeutung für die ABFT-Matrixmultiplikation ist die Abfolge:

$$\text{Kodierung} \rightarrow \text{ABFT-BLAS Algorithmus} \rightarrow \text{Dekodierung}.$$

Werden die daraus entstehenden Matrizen weiterverwendet, müssen sie ebenfalls kodiert werden. Die für diese Kodierung benötigte Zeit kann dabei geringer sein als die für eine initiale Kodierung benötigte Zeit. Dies begründet sich unter anderem darauf, dass im Dekodierungsschritt und dem darauffolgenden Kodierungsschritt redundante Operationen eingespart werden können. Eine Betrachtung am Beispiel von aufeinanderfolgenden Matrixmultiplikationen macht dies deutlich: Bei der initialen Kodierung wird die Matrix $\mathbf{A}$ mit zusätzlichen Spaltenprüfsummen zu einer Matrix $\mathbf{A_{cc}}$, die Matrix $\mathbf{B}$ mit zusätzlichen Reihenprüfsummen zu einer Matrix $\mathbf{B_{rc}}$ kodiert. Nach der Ausführung der Matrixmultiplikation muss die vollständige Prüfsummenmatrix $\mathbf{C_{fc}}$ dekodiert werden. Die Prüfsummenelemente werden für die Ergebnismatrix erneut berechnet und mit den Prüfsummenelemente verglichen, die durch die lineare Operation der Matrixmultiplikation berechnet wurden. Wird diese

Matrix $\mathbf{C_{fc}}$ in einer weiteren ABFT-Matrixmultiplikation verwendet, können diese bereits beim Dekodierungsschritt berechneten Prüfsummenelemente weiterverwendet werden. Je nachdem, ob bei der nachfolgenden Operation die Matrix von der linken oder von der rechten Seite an eine weitere kodierte Matrix $\mathbf{D_{fc}}$ multipliziert wird, müssen lediglich die Spalten- oder Reihenprüfsummenelemente von $\mathbf{C_{fc}}$ und die Reihen- oder Spaltenprüfsummenelemente von $\mathbf{D_{fc}}$ von der Berechnung ausgeschlossen werden. Sie können beispielsweise auf Null gesetzt werden. Dies ist signifikant weniger aufwendig, als die bereits bei der Dekodierung der Matrix durchgeführte Neuberechnung der Prüfsummenelemente erneut durchzuführen. Abbildung 6.2 zeigt das Einsparungspotential durch Substitution redundanter Dekodierungs- und Kodierungsoperationen anhand eines Zeitstrahls. Während in Version 1) Prüfsummen für die Dekodierung und erneute Kodierung von **C** und **E** mehrmals berechnet werden, wird bei der Dekodierung *Dec** in Version 2) diese redundante Operation weggelassen.

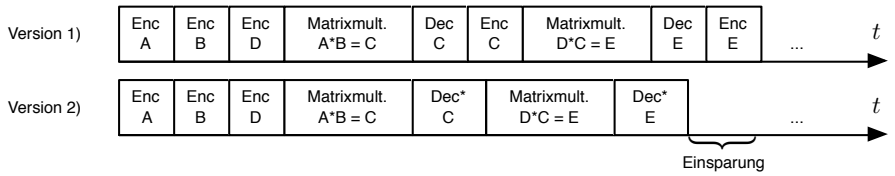


Abbildung 6.2.: Einsparungspotenzial bei erneuter Kodierung.

In Verbindung mit der Analyse des Einsatzschemas legt diese Betrachtung nahe, alle in der ABFT-BLAS-Bibliothek verwendeten Matrizen im Format einer vollständigen Prüfsummenmatrix zu speichern. Die kontextabhängige Kodierung von $\mathbf{A} \rightarrow \mathbf{A_{cc}}$ und $\mathbf{B} \rightarrow \mathbf{B_{rc}}$ wird dadurch zu einer kontextunabhängigen Kodierung $\mathbf{A} \rightarrow \mathbf{A_{fc}}$ und $\mathbf{B} \rightarrow \mathbf{B_{fc}}$. Spätestens nach der ersten Multiplikation mit der zugehörigen Ergebnismatrix $\mathbf{C_{fc}}$ kann zur Laufzeit nicht ohne zusätzliche Informationen durch den Benutzer bestimmt werden, ob die Matrix $\mathbf{C_{fc}}$ im nächsten Multiplikationsschritt von der linken oder von der rechten Seite an eine weitere Matrix multipliziert wird. Ein weiteres Argument für die Verwendung einer allgemeinen Kodierung von Matrizen als vollständige Prüfsummenmatrizen stellt die Möglichkeit dar, diese Kodierung zur Absicherung der Daten außerhalb der ABFT-BLAS-Operationen zu nutzen. Ein mögliches Anwendungsszenario hierfür ist der Einsatz der ABFT-BLAS-Bibliothek in GPUs ohne durch fehlerkorrigierende Codes geschützten Speicher. Ein weiterer Überprüfungsschritt der Prüfsummen vor der eigentlichen ABFT-BLAS-Operation kann dabei zur Fehlerkorrektur eingesetzt werden.

Die Trennung von Berechnungsschritten in kontextunabhängige und kontextabhängige Operationen lässt sich auch auf die Methoden zur Fehlerschwellwertbestimmung anwenden. Die eingesetzten Verfahren lassen sich hier in zwei Schritte unterteilen: Einen von der Verwendung in Links- oder Rechtsmultiplikationen unabhängigen und damit kontextunabhängigen Vorverarbeitungsschritt, sowie einer kontextabhängigen finale Bestimmung der Fehlerschwellwerte, die die Informationen aus der Vorverarbeitung mit einbezieht. Soll eine Matrix in mehreren Multiplikationen als Operand verwendet werden, muss nicht für jede Multiplikation die gesamte Fehlerschwellwertbestimmung erneut durchgeführt werden. Die Abbildung 6.3 zeigt die zeitlichen Abhängigkeiten der unterschiedlichen Schritte bei der ABFT-Matrixmultiplikation mit datenabhängigen Fehlerschwellwertbestimmungsmethoden.

Sobald eine Matrix vollständig kodiert ist, können für die Fehlerschwellwertbestimmung benötigte kontextunabhängige Informationen in einem Vorverarbeitungsschritt bestimmt werden. Parallel zu diesem Schritt kann die eigentliche Matrixmultiplikation durchgeführt werden. Für die Überprüfung der Prüfsummen werden neben der Ergebnismatrix die entsprechenden Fehlerschwellwerte benötigt. Diese werden in einem kontextabhängigen Bestimmungsschritt aus den im Vorverarbeitungsschritt ermittelten Werten berechnet.

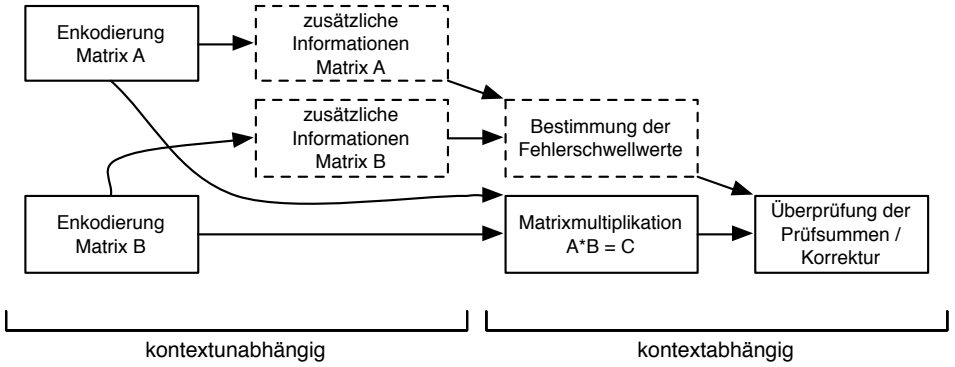


Abbildung 6.3.: Zeitliche Abhängigkeiten bei der ABFT-Matrixmultiplikation mit Fehlerschwellwertbestimmung durch einen kontextunabhängigen Vorverarbeitungsschritt und einen kontextabhängigen Bestimmungsschritt.

BLAS-Konformität

Die BLAS-Aufrufe [47] enthalten neben den Quell- und Zieloperanden oftmals zusätzliche skalare Faktoren. Mit diesen Faktoren kann das Ergebnis der Operation und der Zieloperand skaliert werden kann, bevor das Ergebnis zum Zieloperand addiert wird. Für die Matrixmultiplikation ist die Funktionalität beispielsweise gegeben durch die Faktoren α und β:

$$\mathbf{C} \leftarrow \alpha \cdot \mathbf{A} \cdot \mathbf{B} + \beta \cdot \mathbf{C}. \tag{6.2}$$

Wird kein *Black-Box-Modell* für die Matrixmultiplikation innerhalb der durch algorithmenbasierte Fehlertoleranz geschützten Multiplikation verwendet, kann das Ergebnis von $\mathbf{C_{fc}} = \mathbf{A_{cc}} \cdot \mathbf{B_{rc}}$ zunächst überprüft werden, bevor es auf den Zieloperand addiert wird. Ist dies zum Beispiel durch Verwendung des *Black-Box-Modells* für die Matrixmultiplikation nicht möglich, müssen einige Änderungen bei der Überprüfung der Prüfsummen durchgeführt werden, um diese erweiterten Operation dennoch unterstützen zu können.

Die erweiterte Operation der Matrixmultiplikation mit Addition auf den Zieloperand vereint zwei BLAS-Operationen: Zum Einen die Matrixmultiplikation $\mathbf{D} = \mathbf{A} \cdot \mathbf{B}$, zum Anderen die Addition des Ergebnisses dieser Operation auf den Zieloperand $\mathbf{C_{new}} = \mathbf{C_{old}} + \mathbf{D}$. Bei Verwendung des *Black-Box Modells* in der ABFT-BLAS-Bibliothek werden diese beiden Operationen in einem Schritt ausgeführt. Es steht also nicht das Zwischenergebnis $\mathbf{D}$ explizit zur Verfügung. Entsprechend muss die Methode zur Fehlerschwellwertbestimmung angepasst

werden, um diese Operation unterstützen zu können. Aufgrund der Trennung in kontextunabhängigen Vorverarbeitungsschritt und kontextabhängigen Bestimmungsschritt stehen Informationen über die Matrizen **A** und **B** zur Verfügung. Somit kann der bei der Operation $\mathbf{A} \cdot \mathbf{B}$ in den Prüfsummenelementen auftretende Rundungsfehler abgeschätzt werden. Die Referenzprüfsummen nach Ausführung der erweiterten Operation ergeben sich aus der Summe der bereits in $\mathbf{C_{old}}$ vorhandenen Prüfsummen und der in der linearen Operation $\mathbf{A} \cdot \mathbf{B}$ berechneten Prüfsummen:

$$c = \alpha \cdot d + \beta \cdot c_{old} \tag{6.3}$$

Die Rundungsfehlerabschätzung für die Prüfsummenelemente der Ergebnismatrix muss die am Ende der Operation durchgeführte Addition berücksichtigen. Für die Abschätzung des Rundungsfehlers über das probabilistische Verfahren ergibt sich dadurch für die Varianz:

$$\mathtt{Var}_{\mathtt{Erw}}(c) = \mathtt{Var}_{\mathtt{Skal}}(d) + \mathtt{Var}_{\mathtt{Sum}}(\beta \cdot c_{old} + d) \tag{6.4}$$

$$\leq \mathtt{Var}_{\mathtt{Skal}}(d) + \frac{1}{8} \cdot 2^{-2t} \cdot c_{new}^2. \tag{6.5}$$

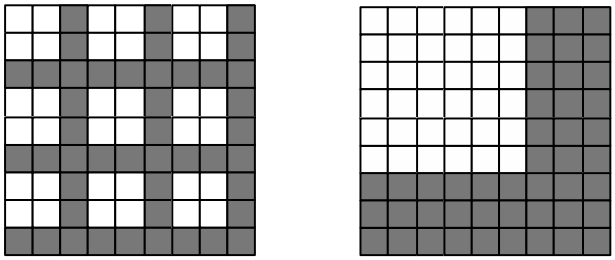
6.3. Konzept einer effizienten ABFT-Bibliothek für BLAS-Operationen

Die Konzeption der ABFT-BLAS-Bibliothek integriert die gewonnenen Erkenntnisse aus den Anforderungen und der Analyse des Einsatzschemas. Zentrales Element dieses Konzepts ist die Verwendung von optimierten BLAS-Bibliotheken, die vom Hersteller der GPUs bereitgestellt werden. Dabei werden elementare Operationen aus dieser Bibliothek für die benötigten Berechnungen als *Black-Box-Modell* verwendet. Im Folgenden werden weitere Teilaspekte dieser Konzeption genauer erläutert.

6.3.1. Kodierung und Speicherung von ABFT-Matrizen

Für die Speicherung von Datenelementen und Prüfsummenelementen einer als vollständige Prüfsummenmatrix kodierten ABFT-Matrix ergeben sich zwei grundlegende Möglichkeiten: Zum Einen das Speichern der Prüfsummenelemente innerhalb der Matrix am Rand jedes ABFT-Blocks und zum Anderen das Speichern der Prüfsummenelemente am Rand der Matrix. Jede dieser Möglichkeiten bietet gewisse Vor- und Nachteile. Beim Speichern der Prüfsummenelemente innerhalb der Matrix am Rand jedes ABFT-Blocks ist die Zugehörigkeit zwischen Prüfsummenelementen und Datenelementen ohne eine spezielle Indizierung ersichtlich. Nach jeweils einer ABFT-Blockgröße werden Spalten- beziehungsweise Reihenprüfsummen eingefügt. Wird eine Matrix in einer ABFT-BLAS-Operation verwendet, müssen jedoch die nicht benötigten Prüfsummenelemente auf Null gesetzt werden. Ein weiterer Nachteil bei der Speicherung von Prüfsummenelementen innerhalb der Matrix ergibt sich dadurch, dass die Datenelemente in der ABFT-Matrix nicht die gleiche Adressierung wie in der Originalmatrix besitzen. Dadurch müssen die Datenelemente vor oder nach dem Datentransfer zwischen den Architekturen an eine anderen Position verschoben werden,

oder der Transferschritt in mehrere kleine Blöcke äquivalent zur ABFT-Blockgröße unterteilt werden, um den für die Prüfsummenelemente benötigten Platz bereitstellen zu können. Abbildung 6.4 zeigt diese Schemata zur Speicherung der Prüfsummenelemente anhand einer 6×6 Matrix mit ABFT-Blockgröße 2.



(a) **(b)**

Abbildung 6.4.: Zwei Möglichkeiten zur Speicherung von Prüfsummenelementen einer ABFT-Matrix: innerhalb der Matrix (a) und am Rand der Matrix (b).

Das Verschieben von Datenelementen innerhalb der Matrix entfällt bei der Speicherung der Prüfsummenelemente am Rand der Matrix, die Adressierung von Datenelementen in der ABFT-Matrix entspricht der Adressierung von Datenelementen in der Originalmatrix. Des Weiteren müssen bei der ABFT-Matrixmultiplikation keine Prüfsummenelemente auf Null gesetzt werden. Es reicht hierbei aus, bei Verwendung des *Black-Box-Modells* für die Matrixmultiplikation innerhalb der ABFT-BLAS-Bibliothek die Matrixmultiplikation mit geänderten Dimensionen durchzuführen. Soll eine vollständige Prüfsummenmatrix $\mathbf{A_{fc}}$ der Größe $(n+d) \times (n+d)$ in einer Linksmultiplikation mit einer vollständigen Prüfsummenmatrix $\mathbf{B_{fc}}$ der Größe $(n+d) \times (n+d)$ multipliziert werden, kann einfach die Matrix $\mathbf{A_{fc}}$ mit der Angabe von n Spalten und die Matrix $\mathbf{B_{fc}}$ mit der Angabe von n Reihen verwendet werden. Die Abbildung 6.5 zeigt die dadurch ausgewählten Matrixbereiche für die Linksmultiplikation, Abbildung 6.6 zeigt die ausgewählten Matrixbereiche für die Rechtsmultiplikation von $\mathbf{A_{fc}}$ an $\mathbf{B_{fc}}$.

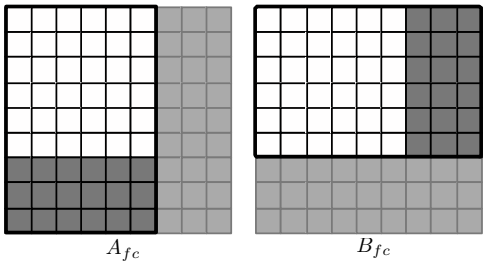


Abbildung 6.5.: Für die Linksmultiplikation $\mathbf{A} \cdot \mathbf{B} = \mathbf{C_{fc}}$ ausgewählte Bereiche bei Speicherung als vollständiger Prüfsummenmatrix.

Diese Betrachtung macht deutlich, dass die Speicherung von Prüfsummenelementen am Rand der Matrix die flexiblere Möglichkeit darstellt.

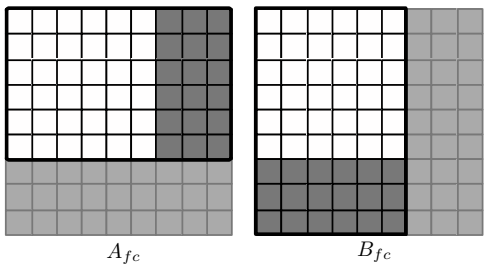


Abbildung 6.6.: Für die Rechtsmultiplikation $\mathbf{B} \cdot \mathbf{A} = \mathbf{C_{fc}}$ ausgewählte Bereiche bei Speicherung als vollständiger Prüfsummenmatrix.

6.3.2. Ausnutzung grobgranularer Parallelität

Die Einteilung der bei der ABFT-Matrixmultiplikation ausgeführten Schritte in kontextunabhängige und kontextabhängige Schritte (siehe Abbildung 6.3) erlaubt eine grobgranulare Parallelisierung der Berechnungsaufgaben, wenn bestimmte Datenabhängigkeiten eingehalten und Synchronisationsanweisungen eingefügt werden. Die im Folgenden vorgestellte grobgranulare Parallelisierung kann prinzipiell durch verschiedene Prozessthreads unabhängig von der Rechnerarchitektur genutzt werden. Um jedoch bei der Nomenklatur der Zielarchitektur zu bleiben und Verwechslungen mit CUDA-Threads zu vermeiden, werden Prozessthreads mithilfe von CUDA-Streams beschrieben. Diese können eingesetzt werden, um auf einer CUDA-GPU mehrere Aufgaben parallel zu bearbeiten. Dadurch können entstehende Latenzen, beispielsweise beim Lesen und Schreiben in den globalen Speicher, durch weitere Berechnungen versteckt werden. Dazu werden pro Matrix drei CUDA-Streams definiert: *fullStream*, *rowStream* und *colStream*.

Der CUDA-Stream *fullStream* wird zunächst für den Transfer von Daten, sowie die initiale Kodierung dieser Matrix verwendet. Die CUDA-Streams *colStream* und *rowStream* werden für den kontextunabhängigen Vorverarbeitungsschritt für die Fehlerschwellwertbestimmung verwendet. Dieser darf erst nach Fertigstellung der Kodierung der jeweiligen Matrix gestartet werden. Ist eine Matrix das Ergebnis einer Matrixoperation, wird *fullStream* für die Operation und den anschließenden Überprüfungs- beziehungsweise Korrekturschritt verwendet. In diesem Fall wird explizit auf *fullStream* der Operanden gewartet, um die vollständige Kodierung der Operanden sicherzustellen. Den zur Ergebnismatrix gehörenden CUDA-Streams *rowStream* und *colStream* wird der kontextabhängige Bestimmungsschritt der Fehlerschwellwertbestimmung zugewiesen. Dieser Schritt darf erst nach Fertigstellung der kontextunabhängigen Schritte für die Operanden erfolgen, in welchen die benötigten Daten ermittelt wurden. Abbildung 6.7 zeigt die Verteilung von unterschiedlichen Aufgaben auf die entsprechenden CUDA-Streams für die Matrixmultiplikation $\mathbf{A_{fc}} \cdot \mathbf{B_{fc}} = \mathbf{C_{fc}}$.

Abhängigkeiten zwischen Aufgaben und CUDA-Streams sind dabei durch Pfeile dargestellt. Zeigt ein Pfeil auf einen CUDA-Stream, so wird auf die Ausführung der letzten, auf diesem CUDA-Stream ausgeführten Anweisung gewartet. Aufgaben, die demselben CUDA-Stream zugeordnet sind, werden immer sequenziell abgearbeitet.

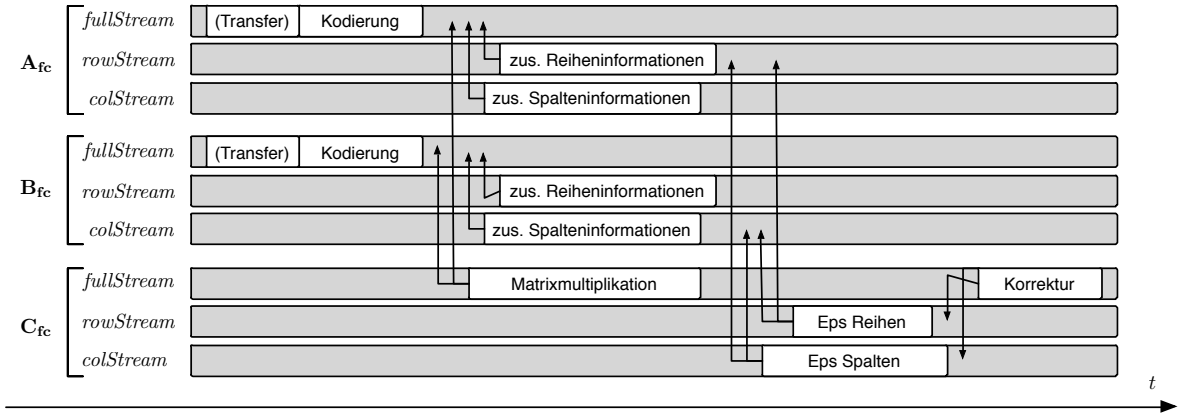


Abbildung 6.7.: Die Verteilung von Aufgaben auf verschiedene CUDA-Streams und die zur korrekten Abarbeitung einzuhaltenden Abhängigkeiten.

6.3.3. Überlappung von Transfer- und Kodierungsschritten

Bei der Verwendung einer partitionierten Kodierung für ABFT-Matrizen kann der Transfer von Host zu GPU zeitlich überlappend mit der Kodierung ausgelegt werden. Da für die Prüfsummenberechnung jedes ABFT-Blocks nur die in diesem Block gespeicherten Daten benötigt werden, kann prinzipiell jeder Block kodiert werden, sobald die dazugehörigen Daten transferiert wurden. Parallel dazu können die weiteren Blöcke kopiert werden. Abbildung 6.8 verdeutlicht diese Überlappung anhand eines Zeitstrahls, Transferschritte sind dabei mit *H to D* gekennzeichnet, Kodierungsschritte mit *Enc*.

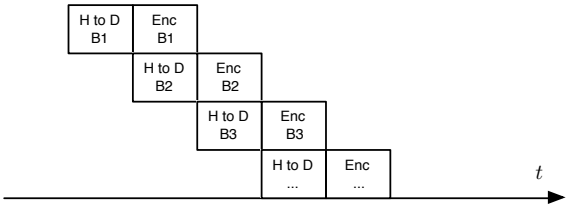


Abbildung 6.8.: Überlappung von Transfer und Kodierung.

6.3.4. Erhaltung der Konsistenz

Bei asynchron ausgeführten Operationen auf Host und GPU muss besonderer Wert darauf gelegt werden, die Konsistenz der Daten auf beiden Seiten zu erhalten. Ist beispielsweise eine Matrix das Ergebnis einer Matrixmultiplikation auf der GPU und soll von Hostseite auf diese Daten zugegriffen werden, muss die Matrixmultiplikation beendet und der Transferschritt von GPU zu Host durchgeführt sein, bevor auf diese Daten zugegriffen werden kann. Ein schwierigeres Szenario stellt sich dar, wenn zusätzlich von der Seite des Hosts Schreiboperationen durchgeführt werden, bevor die Matrixmultiplikation beendet ist. In

diesem Fall müssen die Schreiboperationen des Benutzers Vorrang vor den in der Multiplikation berechneten Matrixelementen erhalten. Für die Konsistenzerhaltung wurden in dieser Arbeit zwei Variablen definiert, welche die Änderungen von Host und GPU für jede Matrix indizieren: Die Variable *hostChanged* und die Variable *deviceChanged*. Für den Benutzer stehen Abfragemethoden zum Lesen und Änderungsmethoden zum Setzen von Matrixelementen zur Verfügung, die den direkten Zugriff auf den Hostspeicher unterbinden. Zusätzlich dazu führt eine Methode die Synchronisation von Daten durch. Dabei gilt:

- Wird eine Änderungsmethode auf Hostseite aufgerufen, wird der Indikator *hostChanged* dieser Matrix gesetzt, das Element mit dem neuen Wert belegt und die Position, sowie der neue Wert in einer Änderungsliste gespeichert.
- Wird eine Abfragemethode auf Hostseite aufgerufen und der Indikator *deviceChanged* ist gesetzt, wird die Synchronisationsmethode gestartet.
- Wird eine ABFT-BLAS-Operation ausgeführt und der Indikator *hostChanged* ist gesetzt, wird die Synchronisationsmethode gestartet. Vor Durchgeführung der Operation wird der Indikator *deviceChanged* für die Ergebnismatrix gesetzt.
- Werden nicht-BLAS Algorithmen auf Deviceseite durch den Benutzer integriert, müssen diese nach Beendigung den Indikator *deviceChanged* setzen.

Die Synchronisationsmethode wartet auf die Abarbeitung des CUDA-Streams *fullStream* der zu synchronisierenden Matrix. Danach wird zwischen vier Fällen unterschieden:

1.) Weder der Indikator *hostChanged*, noch der Indikator *deviceChanged* sind gesetzt:
Die Matrix ist bereits synchronisiert, es ist kein Transfer notwendig.

2.) Nur der Indikator *hostChanged* ist gesetzt:
Die Matrix wurde nur auf Hostseite geändert, es erfolgt ein Transfer von Host zu Device und aufgrund geänderter Matrixelemente die anschließende erneute Kodierung der Matrix und die weiteren kontextunabhängigen Schritte.

3.) Nur der Indikator *deviceChanged* ist gesetzt:
Die Matrix wurde nur auf Deviceseite geändert, es erfolgt ein Transfer von Device zu Host.

4.) Sowohl *hostChanged*, als auch *deviceChanged* sind gesetzt:
In diesem Fall ist diese Matrix das Ergebnis einer ABFT-BLAS-Operation und es wurden (ohne vorhergehende Abfragemethoden) Elementen der Matrix neue Werte zugewiesen. Zur Synchronisation werden die folgenden Schritte durchgeführt:

   a) Transfer von Device zu Host

   b) erneutes Setzen der in der Änderungsliste gespeicherten Elemente auf Hostseite

   c) Transfer der durch Änderungen aus der Änderungsliste betroffenen Submatrix von Host zu Device und die erneute Kodierung der Matrix und die weiteren kontextunabhängigen Schritte.

Im Anschluß werden die Indikatoren *deviceChanged* und *hostChanged* wieder zurückgesetzt.

6.3.5. Effiziente Bestimmung von oberen Schranken für das probabilistische Verfahren

Im vorigen Kapitel wurde dargelegt, warum eine Einteilung der Fehlerschwellwertbestimmung in einen kontextunabhängigen Vorverarbeitungsschritt und einen kontextabhängigen Bestimmungsschritt sinnvoll ist. Für das probabilistische Verfahren zur Rundungsfehlerabschätzung kann für jedes Prüfsummenelement ein Fehlerschwellwert nach der Formel

$$\omega \cdot \sigma = \omega \cdot \sqrt{\frac{n \cdot (n+1) \cdot (n+\frac{1}{2}) + 2 \cdot n}{24}} \cdot y \cdot 2^{-t} \tag{6.6}$$

bestimmt werden. Bei Einsatz der *Fused Multiply-Add*-Operation für die Matrixmultiplikation wird dieser Fehlerschwellwert nach der Formel

$$\omega \cdot \sigma = \omega \cdot \sqrt{\frac{n \cdot (n+1) \cdot (n+\frac{1}{2})}{24}} \cdot y \cdot 2^{-t} \tag{6.7}$$

berechnet.

Der Wert y ist dabei als obere Schranke definiert, sodass für alle Summanden $c_k = a_k \cdot b_k$ des Skalarprodukts der Vektoren **a** und **b** gilt: $y \geq c_k$.

Betragsmäßig größte Elemente

Für jeden Reihen- und Spaltenvektor einer Matrix können im Vorverarbeitungsschritt die p betragsmäßig größten Elemente und ihre zugehörige Position im Vektor bestimmt werden. Eine obere Schranke für ein beliebiges Element aus der Ergebnismatrix **C** an der Stelle (i, j) lässt sich dann durch eine Kombinationsüberprüfung basierend auf den p betragsmäßig größten Elementen des Reihenvektors **a** der Matrix **A** und des Spaltenvektors **b** der Matrix **B** bestimmen: Sei A_{idx} die Menge der Indizes der p betragsmäßig größten Elemente von **a**, B_{idx} die Menge der Indizes der p betragsmäßig größten Elemente in **b**. Die obere Schranke lässt sich bestimmen als das Maximum der drei Werte:

- Ist die Schnittmenge $S = A_{idx} \cap B_{idx}$ nicht leer, werden Elemente aus den Mengen der p betragsmäßig größten Elemente miteinander multipliziert und für die obere Schranke y gilt: $y = max(|a_s \cdot b_s|)$ mit $s \in S$.
- Das betragsmäßig größte Element aus dem Vektor **a** wird auf jeden Fall mit einem Element aus dem Vektor **b** multipliziert. Da nur Informationen über p Elemente aus **b** vorliegen, ist $max(|a_r|) \cdot min(|b_s|)$ mit $r \in A_{idx}, s \in B_{idx}$ ein weiterer Kandidat für eine obere Schranke.
- Analoges gilt für das betragsmäßig größte Element des Vektors **b**, deshalb ist auch $min(|a_r|) \cdot max(|b_s|)$ mit $r \in A_{idx}, s \in B_{idx}$ ein weiterer Kandidat für eine obere Schranke.

Blockbasierte Durchschnittsmetriken

Eine Weiterentwicklung der Methode zur Bestimmung einer oberen Schranke für das probabilistische Verfahren zur Fehlerschwellwertbestimmung basiert darauf, dass bei der Bildung des Skalarprodukts nur Elemente der Vektoren **a** und **b** mit gleichen Indizes multipliziert werden. Die Vektoren lassen sich in Blöcke der Größe *bs* einteilen, wobei diese Blockgröße unabhängig von der ABFT-Blockgröße ist:

$$\mathbf{a} = \begin{pmatrix} \mathbf{a_1^*} & \mathbf{a_2^*} & \dots & \mathbf{a_s^*} \end{pmatrix} \quad \text{mit} \quad \mathbf{a_i^*} = \begin{pmatrix} a_{bs\cdot i+1} & a_{bs\cdot i+2} & \dots & a_{bs\cdot i+bs} \end{pmatrix} \quad \text{und} \tag{6.8}$$

$$\mathbf{b} = \begin{pmatrix} \mathbf{b_1^*} & \mathbf{b_2^*} & \dots & \mathbf{b_s^*} \end{pmatrix} \quad \text{mit} \quad \mathbf{b_i^*} = \begin{pmatrix} b_{bs\cdot i+1} & b_{bs\cdot i+2} & \dots & b_{bs\cdot i+bs} \end{pmatrix}. \tag{6.9}$$

Das Skalarprodukt der Vektoren **a** und **b** ist dann:

$$< \mathbf{a}, \mathbf{b} >= \sum_{i=1}^{s} < \mathbf{a_i^*}, \mathbf{b_i^*} >= \sum_{i=1}^{s} \sum_{t=1}^{bs} a_t^* \cdot b_t^* \tag{6.10}$$

Es lässt sich damit für das Skalarprodukt jedes Blocks $< \mathbf{a_i^*}, \mathbf{b_i^*} >$ eine eigene obere Schranke bestimmen. Um aus diesen Schranken eine obere Schranke für das Skalarprodukt der gesamten Vektoren **a** und **b** zu erhalten, wird dann das Maximum dieser Schranken benutzt.

Die Motivation hinter der blockbasierten Durchschnittsmetrik besteht darin, für jeden Block eines Vektors in einem konextunabhängigen Vorverarbeitungsschritt eine Reihe von Informationen über die in diesem Block auftretenden Werte und deren örtliche Verteilung zu ermitteln. Anhand dieser Informationen kann im anschließenden kontextabhängigen Schritt unmittelbar Rückschlüsse darauf gezogen werden, ob Elemente von betragsmäßig großem Wert miteinander multipliziert werden. Können solche Kombinationen ausgeschlossen werden, lässt sich die zur Fehlerschwellwertbestimmung benötigte obere Schranke näher an den tatsächlich auftretenden betragsmäßig größten Summanden c_k bei der Bildung des Skalarprodukts legen.

Bei der blockbasierten Durchschnittsmetrik wird zunächst der Durchschnittswert als arithmetisches Mittel der Absolutwerte für einen Block bestimmt. Im Anschluß wird die Information gespeichert, an welchen Positionen innerhalb dieses Blocks der entsprechende Absolutwert über diesem ermittelten Durchschnittswert liegt. Für eine effiziente Verarbeitung der Daten wird diese Information in einem Bitvektor gespeichert. Die Abbildung 6.9 zeigt exemplarisch einen solchen Block des Vektors *a* mit den Absolutwerten seiner Daten und den zugehörigen Durchschnittswert `avgA`$[0]$. Der zugehörige Bitvektor `bitVecA`$[0]$ kodiert nun die Positionen im Vektor, bei denen der Absolutwert an dieser Stelle über dem ermittelten Durchschnittswert liegt. Diese Positionen sind in der Abbildung schwarz markiert. Im nächsten Schritt wird von den Elementen, deren Werte über dem gerade ermittelten Durchschnittswert liegen, ein neuer Durchschnittswert `avgA`$[1]$ bestimmt. Dies wird im Folgenden als die nächst höhere Kodierungsebene bezeichnet. Die Positionen im Vektor, bei denen der Absolutwert größer als `avgA`$[1]$ ist, werden dann in einem weiteren Bitvektor `bitVecA`$[1]$ kodiert. Dieses Vorgehen kann nun beliebig oft wiederholt werden, so lange noch Elemente über dem zuletzt ermittelten Durchschnittswert liegen. Für den letzten Bitvektor wird der maximal auftretende

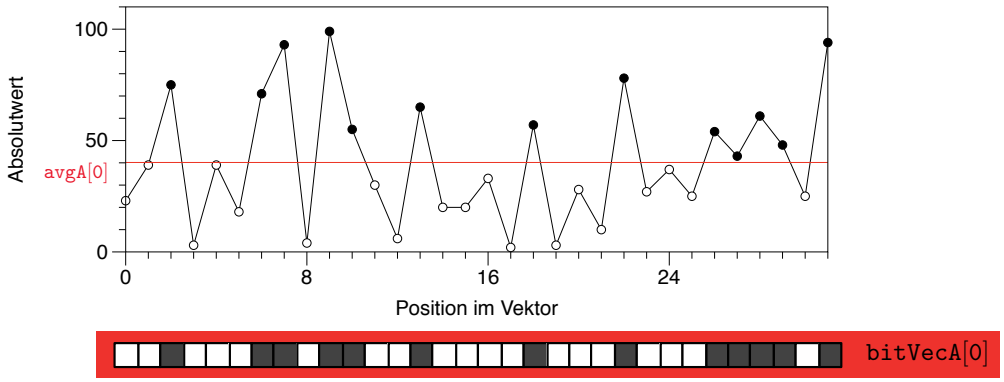


Abbildung 6.9.: Die Bestimmung des Durchschnittswerts `avgA`[0] der Absolutwerte eines Vektors a und der zugehörige Bitvektor `bitVecA`[0]. Der Bitvektor kodiert die Positionen des Vektors, bei denen der Absolutwert des Elements größer als der ermittelte Durchschnittswert ist.

Absolutwert `maxA` unabhängig davon gespeichert, wie viele Elemente in diesem Bitvektor kodiert werden oder wie der Durchschnittswert dieser Elemente ist. Abbildung 6.10 zeigt das Ergebnis dieses Vorgehens für vier Ebenen. Die rechts neben dem Vektor dargestellte Grafik zeigt das Profil der Absolutwerte des Vektors und damit die Einteilung, wie viele Elemente über und unter dem ermittelten Durchschnittswert liegen.

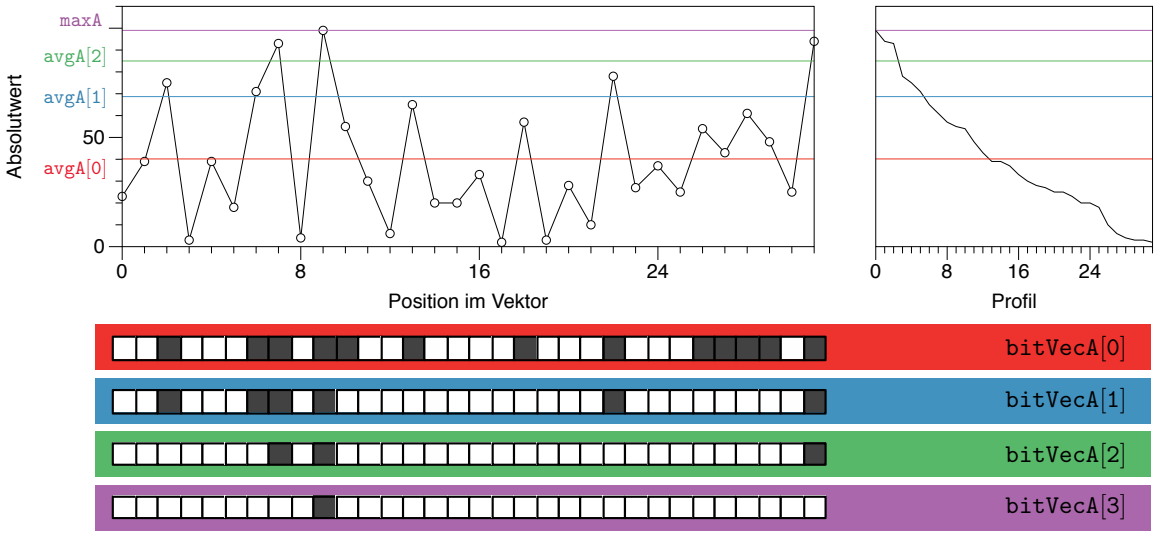


Abbildung 6.10.: Der Vektor a und die dazu bestimmten Durchschnittswerte und dazugehörigen Bitvektoren für die blockbasierte Durchschnittsmetrik (links), die sortierten Werte des Vektors als Profil (rechts) und die zugehörigen Bitvektoren (unten).

Diese Kodierung erfüllt folgende Eigenschaften:

- Die Durchschnittswerte steigen mit steigender Kodierungsebene an. Aufgrund der Tatsache, dass nur Elemente in den nächst höheren Durchschnittswert gezählt werden, die echt größer als der vorhergehende Durchschnittswert sind, ist diese Folge streng monoton steigend.
- Die im Bitvektor kodierten Positionen der Kodierungsebene $i+1$ ist eine (echte) Teilmenge der im Bitvektor der Kodierungsebene i kodierten Positionen.
- Die Anzahl der Elemente zweier aufeinanderfolgenden Kodierungsebenen wird im Durchschnitt halbiert, da nur Absolutwerte betrachtet werden. Dies bedeutet, dass für eine Blockgröße bs nach maximal $\log_2(bs)$ Schritten kein Element mehr über dem letzten Durchschnittswert liegt.
- Alle Elemente, die in Ebene i, jedoch nicht in Ebene $i+1$ im Bitvektor kodiert sind, sind kleiner als der Durchschnittswert $\texttt{avgA}[i+1]$ und größer als der Durchschnittswert $\texttt{avgA}[i]$.

Bei der kontextabhängigen Kombination zweier Blöcke im Skalarprodukt kann durch die Bitvektoren und die zugehörigen Durchschnittswerte eine Aussage bezüglich der oberen Schranke für diesen Block getroffen werden. Die Kodierung durch Bitvektoren erlaubt die Verwendung der bitweisen *UND*-Operation (&) zur Überprüfung, ob und welche Elemente aus den jeweiligen Kodierungsebenen miteinander multipliziert werden. Werden keine Elemente miteinander multipliziert, ist $(\texttt{bitVecA}[i]\ \&\ \texttt{bitVecB}[j]) = 0$. Diese Operation entspricht der Bildung der Schnittmenge der im Bitvektor $\texttt{bitVecA}[i]$ kodierten Positionen und der im Bitvektor $\texttt{bitVecB}[j]$ kodierten Positionen.

Für alle Kodierungsebenen i des Blocks von **a** und alle Kodierungsebenen j des Blocks von **b** gilt:

- $(\texttt{bitVecA}[i]\ \&\ \texttt{bitVecB}[j]) \neq 0 \Rightarrow y > \texttt{avgA}[i] \cdot \texttt{avgB}[j]$
  Elemente, deren Wert größer als der Durchschnittswert dieser Kodierungsebenen ist, werden im Skalarprodukt multipliziert. Der Wert der oberen Schranke muss demnach größer als das Produkt der zu i und j gehörenden Werte $\texttt{avgA}[i]$ und $\texttt{avgB}[j]$ sein.
- $(\texttt{bitVecA}[i]\ \&\ \texttt{bitVecB}[j]) = 0 \Rightarrow \forall i^* \geq i\ \forall j^* \geq j : (\texttt{bitVecA}[i^*]\ \&\ \texttt{bitVecB}[j^*]) = 0$
  Elemente der Kodierungsebenen i und j werden nicht im Skalarprodukt multipliziert. Höhere Kodierungsebenen i^* und j^* müssen nicht betrachtet werden, da die Teilmengeneigenschaft gilt.

Des Weiteren gilt:

- Der Wert $y_{maxA} = \texttt{maxA} \cdot \texttt{avgB}[0]$ muss unabhängig von der Kombinationsüberprüfung als Kandidat für eine obere Schranke betrachtet werden, da der betragsmäßig größte, in **a** auftretende Wert auf jeden Fall mit einem Wert multipliziert wird, der unterhalb des Durchschnittswerts der Kodierungsebene 0 des Vektors **b** liegt.
- Analog dazu muss der Wert $y_{maxB} = \texttt{maxB} \cdot \texttt{avgA}[0]$ als Kandidat für eine obere Schranke betrachtet werden.

Der Wert y_{maxK} bezeichnet den maximalen Wert

$$y_{maxK} = \max_{(i,j)\in K} \{\texttt{avgA}[i+1] \cdot \texttt{avgB}[j+1]\} \tag{6.11}$$

aus der Menge aller Kandidaten

$$K = \{(i,j) \quad | \quad \texttt{bitVecA}[i] \;\&\; \texttt{bitVecA}[j] \neq 0 \wedge \texttt{bitVecA}[i+1] \;\&\; \texttt{bitVecA}[j+1] = 0\}\,. \tag{6.12}$$

Diese Menge K beschreibt alle möglichen, auftretenden Kombinationen von Kodierungsebenen i und j, in denen Elemente von gleicher Position multipliziert werden, jedoch nicht in den nächst höheren Kodierungsebenen $i+1$ und $j+1$.

Die kleinste obere Schranke ist damit definiert als

$$y = \max\{y_{maxA}, y_{maxB}, y_{maxK}\}. \tag{6.13}$$

Wird nun im kontextabhängigen Bestimmungsschritt ein Block des Vektors **a** mit einem Block des Vektors **b** kombiniert, lässt sich hierfür eine obere Schranke bestimmen:

1. Finde die höchste Kodierungsebene i_{max} von **a**, in der der zugehörige Bitvektor $\texttt{bitVecA}[i_{max}]$ mindestens eine Position kodiert.

2. Von diesem Bitvektor ausgehend suche von der untersten Kodierungseben $j = 0$ von **b** aufsteigend nach der größten Kodierungsebene j, in der $\texttt{bitVecA}[i_{max}]$ und $\texttt{bitVecB}[j]$ noch eine Übereinstimmung haben, die in $\texttt{bitVecA}[i_{max}]$ und $\texttt{bitVecB}[j+1]$ kodierte Positionen also nicht kombiniert werden. Der erste Kandidat für eine obere Schranke ist $\texttt{maxA} \cdot \texttt{avgB}[j+1]$. Kann keine solche Kodierungsebene gefunden, wird die niedrigste Kodierungsebene von **b** gewählt, $\texttt{maxA} \cdot \texttt{avgB}[0]$ wird als erster Wert für eine obere Schranke verwendet.

3. In den darauffolgenden Schritten wird nach einer Übereinstimmung der Bitvektoren in höheren Ebenen von **b** und niedrigeren Ebenenen von **a** gesucht, da die Kombination solcher Werte größer als die bisher gefundene obere Schranke sein kann.

   Durch das Suchen in niedrigeren Ebenen von **a** werden mehr Elemente betrachtet, allerdings wird auch der zugehörige Durchschnittswert kleiner. Da die neue obere Schranke größer sein soll als die bereits gefundene, muss die Ebene von **b** erhöht werden, um eine insgesamt größere neue obere Schranke zu finden. Sei $\lambda_{a,k}$ der Faktor, um den der Durchschnittswert von **a** bei Wahl einer um k niedrigeren Ebene verringert wird: $\lambda_a = \texttt{avgA}[i-k]/\texttt{avgA}[i]$. Sei weiterhin $\lambda_{b,1}$ der Faktor, um den der Durchschnittswert von **b** bei Wahl der nächst höheren Ebene erhöht wird: $\lambda_b = \texttt{avgB}[j+1]/\texttt{avgB}[j]$. Um eine insgesamt größere neue obere Schranke zu erhalten, muss $\lambda_{b,1} > \lambda_{a,k}$ sein. Beginnend bei den zuvor bestimmten Kodierungsebenen $i = i_{max}$ und j können unter Ausnutzung der durch die Kodierung gegeben Eigenschaften alle möglichen Kandidaten für eine größere obere Schranke betrachtet werden:

   a) Erhöhe die Ebene von **b**: $j \rightarrow j+1$

b) Erniedrige die Ebene von **a** bis $\lambda_{b,1} > \lambda_{a,k}$ ist und eine Übereinstimmung auftritt. Ein neuer Kandidat ist $\texttt{avgA}[i+1] \cdot \texttt{avgB}[j+1]$, da in den darüberliegenden Ebenen keine Übereinstimmung auftrat.

c) Stoppe, sobald i die niedrigste Kodierungsebene und j die höchste Kodierungsebene erreicht hat.

Diese Bestimmung erfolgt unter Ausnutzung der Eigenschaften der Kodierung in maximal $2 \cdot l$ Schritten, wobei l die Anzahl verwendeter Kodierungsebenen ist.

Abbildung 6.11 zeigt anhand des Profils der Vektoren **a** und **b**, welche Elemente durch die jeweiligen Bitvektoren kodiert werden, Abbildung 6.12 die exemplarisch durchgeführten Schritte zur Bestimmung der oberen Schranke.

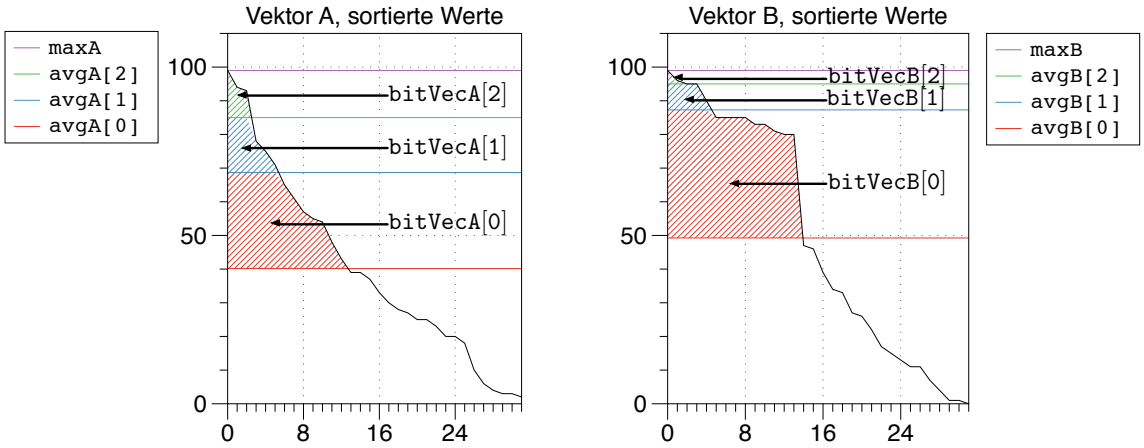


Abbildung 6.11.: Profile der Werte der Vektoren **a** und **b**, die ermittelten Durchschnittswerte sowie die durch die zugehörigen Bitvektoren kodierten Elemente.

1) $(\texttt{bitVecA}[2]\ \&\ \texttt{bitVecB}[0]) = 0 \Rightarrow$ initiale Schranke : $\texttt{maxA} \cdot \texttt{avgB}[0]$, Ebene A $\downarrow$, Ebene B $\uparrow$

2) $(\texttt{bitVecA}[1]\ \&\ \texttt{bitVecB}[1]) = 0 \Rightarrow$ Ebene A $\downarrow$

3) $(\texttt{bitVecA}[0]\ \&\ \texttt{bitVecB}[1]) \neq 0 \Rightarrow$ neue Schranke : $\texttt{avgA}[1] \cdot \texttt{avgB}[2]$, (Ebene A $\downarrow$), Ebene B $\uparrow$

4) $(\texttt{bitVecA}[0]\ \&\ \texttt{bitVecB}[2]) \neq 0 \Rightarrow$ neue Schranke : $\texttt{avgA}[1] \cdot \texttt{maxB}$

Abbildung 6.12.: Exemplarische Schritte zur Bestimmung der oberen Schranke.

Im Schritt 1) werden Elemente aus der höchsten Kodierungsebene i_{max} von **a** mit Elementen aus der Kodierungsebene 0 von **b** nicht kombiniert, entsprechend ist die initiale Schranke gegeben durch $\texttt{maxA} \cdot \texttt{avgB}[0]$. Daraufhin wird die Ebene von **a** erniedrigt, sowie die Ebene von **b** erhöht um nach einer insgesamt größeren Schranke zu suchen. Da in Ebene 1 von **a** und Ebene 1 von **b** keine Kombination von Elementen erfolgt, kann die Ebene von **a** erniedrigt werden, um mehr Elemente in die Betrachtung mit einzuschließen. In Ebene 0 von **a** erfolgt eine Kombination, ein neuer Kandidat für eine obere Schranke ist damit $\texttt{avgA}[1] \cdot \texttt{avgB}[2]$. Da die Ebene von **a** in diesem Punkt nicht weiter erniedrigt werden kann,

wird nur die Ebene von **b** erhöht. Auch hier ergibt sich eine Kombination und der letzte Kandidat für eine obere Schranke ist damit $\texttt{avgA}[1] \cdot \texttt{maxB}$.

Verwendung von globalen Durchschnittswerten

Eine weitere blockbasierte Durchschnittsmetrik kann definiert werden, indem beim kontextunabhängigen Vorverarbeitungsschritt der Durchschnittswert nicht lokal für jeden Block bestimmt wird, sondern Durchschnittswerte über alle Blöcke des Vektor gebildet werden. Sei Vektor **a** beispielsweise unterteilt in drei Blöcke: $\mathbf{a_1^*}$, $\mathbf{a_2^*}$ und $\mathbf{a_3^*}$. Für den ersten Durchschnittswert wird der Durchschnitt aller Absolutwerte des gesamten Vektors **a** betrachtet. Für höhere Kodierungsebenen wird der Durchschnittswert aus den Werten aller Blöcke $\mathbf{a_1^*}$, $\mathbf{a_2^*}$ und $\mathbf{a_3^*}$ gebildet, die über dem vorhergehenden Durchschnittswert liegen. Der Vorteil dieser Variante liegt im niedrigeren Speicherbedarf für die Speicherung der Durchschnittswerte: Während beim Verfahren der blockbasierten lokalen Durchschnittsmetrik pro Vektorblock und pro Kodierungsebene jeweils ein Bitvektor und ein Durchschnittswert gespeichert werden müssen, wird hier neben dem Bitvektor pro Vektorblock und pro Kodierungsebene nur ein Durchschnittswert pro Kodierungsebene für alle Vektorblöcke benötigt. Der Nachteil dieser Variante liegt darin, dass bei der Kodierung aufgrund der Bildung eines globalen Durchschnittswerts pro Kodierungsebene nicht jeder Block des Vektors unabhängig voneinander betrachtet werden kann. Bei der kontextabhängigen Bestimmung der oberen Schranke kann das gleiche Verfahren wie bei der lokalen blockbasierten Durchschnittsmetrik angewandt werden. Dabei erfüllt die Kodierung ähnliche Eigenschaften:

- Die Durchschnittswerte sind ebenfalls streng monoton steigend mit steigender Kodierungsebene.
- Die in Kodierungsebene $i+1$ enthaltenen Elemente sind eine Teilmenge der in Kodierungsebene i kodierten Elemente.
- Die Anzahl der Elemente zweier aufeinanderfolgender Kodierungsebenen wird bei Betrachtung aller Blöcke im Durchschnitt ebenfalls halbiert.

Die Fehlerschwellwertbestimmung erfolgt analog zur Bestimmung basierend auf der lokalen Durchschnittsmetrik.

Tabelle 6.1 fasst die wichtigsten Eigenschaften der vorgestellten Methoden zur Bestimmung einer oberen Schranke für das probabilistische Verfahren zusammen. Die für den Speicherbedarf angegebene Blockgröße bs bezieht sich auf die Breite der bei der Durchschnittsmetrik verwendeten Bitvektoren. In Abhängigkeit von der Größe bs kann die Anzahl Kodierungsebenen l gewählt werden, in dieser Arbeit wurden $bs = 32$ und $l = 5$ verwendet. Bei Verwendung der betragsmäßig größten Elemente kann die Anzahl der im Bestimmungsschritt betrachteten betragsmäßig größten Elemente p variiert werden. Die Zeitkomplexität des Bestimmungsschritts ist dabei quadratisch in der Anzahl betrachteter Elemente, da jede Kombination überprüft werden muss.

Eigenschaft	Betragsmäßig größte Elemente	Blockbasierte lokale Durchschnittsmetrik	Blockbasierte globale Durchschnittsmetrik
Parameter	Anzahl größte Elemente p	Blockgröße bs, Anz. Ebenen l	Blockgröße bs, Anz. Ebenen l
Speicherbedarf bei einer $n \times m$ Matrix	$(n+m) \cdot p$ Werte und $(n+m) \cdot p$ Positionen	$(n+m) \cdot bs \cdot l$ Werte und $(n+m) \cdot bs \cdot l$ Bitvektoren	$(n+m) \cdot l$ Werte und $(n+m) \cdot bs \cdot l$ Bitvektoren
Zeitkomplexität des Bestimmungsschritts in Abhängigkeit der Veränderlichen	$\mathcal{O}(p^2)$	$\mathcal{O}(2 \cdot l)$	$\mathcal{O}(2 \cdot l)$
Maximaler Wert der Veränderlichen, bis kein Informationszugewinn mehr erreicht wird	$p = n$, bzw. $p = m$	$\log_2(bs)$	$\log_2(n)$, bzw. $\log_2(m)$
Bestimmungsschritt unabhängig für blockweise Bestimmung	ja	ja	nein

Tabelle 6.1.: Vergleich unterschiedlicher Verfahren bezüglich des Speicherbedarfs, der Zeitkomplexität und des Informationszugewinns bei Änderung der Parameter.

6.3.6. Gewichtete Prüfsummen und Fehlerkorrekturmaßnahmen

Die Verwendung von unterschiedlichen Gewichten und die Variation in der Anzahl dieser Gewichte hat Auswirkungen auf die zur Kodierung benötigten Operationen, den Platzbedarf der kodierten Matrix und die Fehlererkennungs- und Fehlerkorrekturmöglichkeiten. Die ABFT-BLAS-Bibliothek wurde so ausgelegt, dass eine beliebige Anzahl gewichteter Prüfsummen verwendet werden kann. Die Gewichte werden dem Kodierungskernel als Parameter übergeben, für die Fehlerlokalisierung müssen jedoch spezialisierte Korrekturkernel eingesetzt werden. Folgende Prüfsummenschemata wurden in der ABFT-BLAS-Bibliothek exemplarisch integriert:

- Eine Prüfsumme mit einfacher Fehlerlokalisierung, die die Fehlerkorrektur eines fehlerhaften Elements pro ABFT-Block über die Lokalisierung durch die Reihen- und Spaltenprüfsumme ermöglicht:
  - Normale Prüfsumme (uniform mit 1 gewichtet)
- Zwei Prüfsummen mit einfacher Fehlerlokalisierung:
  - Normale Prüfsumme und Durchschnittsprüfsummen
  - Normale Prüfsumme und periodischen Kodierungsvektoren [23]

- Zwei Prüfsummen mit erweiterte Fehlerlokalisierung, die die Fehlerkorrektur eines fehlerhaften Elements innerhalb jedes Reihen- und Spaltenvektors eines ABFT-Blocks ermöglicht.
  - Normale Prüfsumme und exponentiell gewichtete Prüfsumme
  - Normale Prüfsumme und linear gewichtete Prüfsumme [24]
  - Normale Prüfsumme und *Minimum Dynamic Range SEC Code* [24]

Die Formeln zur Bildung dieser Prüfsummen sind in Tabelle 6.2 aufgeführt.

Prüfsummenschema	Formel
normale Prüfsumme	$c = \sum_{i=1}^{BS} 1 \cdot a_i$
Durchschnittsprüfsumme	$c = \sum_{i=1}^{BS} \frac{1}{BS} \cdot a_i$
periodische Kodierungsvektoren	$c = \sum_{i=1}^{BS} (-1)^i \cdot a_i$
exponentiell gewichtete Prüfsumme	$c = \sum_{i=1}^{BS} 2^{i+1} \cdot a_i$
linear gewichtete Prüfsumme	$c = \sum_{i=1}^{BS} (i+1) \cdot a_i$
MDR-Code	$c = \sum_{i=1}^{BS} \frac{(i+1)}{BS} \cdot a_i$

Tabelle 6.2.: Gewichte zur Bildung verschiedener Prüfsummen bei ABFT-Blockgröße .

Im Folgenden werden die Fehlerkorrekturfähigkeiten und daraus abgeleitete Schemata zur Korrektur von fehlerhaften Elementen dargestellt.

Korrekturmaßnahmen für Prüfsummen mit einfacher Lokalisierung

Bei der Verwendung von Prüfsummen mit einfacher Lokalisierung ergeben sich zwei, beziehungsweise vier Indikatoren für ein fehlerhaftes Element in der Ergebnismatrix:

1. Mindestens ein Syndrom der bei der Überprüfung der Reihenprüfsumme einer Reihe betrachteten Syndrome ist größer als der Fehlerschwellwert: $|S_r| > \epsilon_r$
2. Mindestens ein Syndrom der bei der Überprüfung der Spaltenprüfsumme einer Spalte betrachteten Syndrome ist größer als der Fehlerschwellwert: $|S_c| > \epsilon_c$

Indiziert mehr als eine Reihe oder mehr als eine Spalte einen Fehler durch die Überschreitung des Fehlerschwellwerts, kann das fehlerhafte Element nicht oder nicht eindeutig lokalisiert und damit nicht korrigiert werden. Das Schema zur Korrektur lautet:

1. Betrachte die Syndrome der Reihenprüfsummen:
   a) Überschreitet mindestens ein Syndrom genau einer Reihe den Fehlerschwellwert, speichere die zugehörige Reihe.
   b) Überschreitet mindestens ein Syndrom von mehr als einer Reihe den Fehlerschwellwert, berechne den ABFT-Block neu.
2. Betrachte die Syndrome der Spaltenprüfsummen:
   a) Überschreitet mindestens ein Syndrom genau einer Spalte den Fehlerschwellwert und im ersten Schritt wurde eine fehlerhafte Reihe gespeichert, korrigiere dieses Element. Wurde im ersten Schritt keine Reihe lokalisiert, ist ein nicht lokalisierbarer Fehler aufgetreten und der ABFT-Block muss neu berechnet werden.
   b) Überschreitet mindestens ein Syndrom von mehr als einer Spalte den Fehlerschwellwert, dann muss der ABFT-Block ebenfalls neu berechnet werden.
3. Überschreiten keine Syndrome der Reihen- und Spaltenprüfsummen den Fehlerschwellwert, ist der ABFT-Block fehlerfrei und muss nicht korrigiert werden.

Korrekturmaßnahmen für Prüfsummen mit erweiterter Lokalisierung

Bei der Verwendung von zwei Prüfsummen mit der Möglichkeit der Lokalisierung von fehlerhaften Elementen innerhalb der Reihen- und Spaltenvektoren ergeben sich vier Indikatoren für ein fehlerhaftes Element in der Ergebnismatrix:

1. Das Syndrom der *ungewichteten* Prüfsumme bei der Überprüfung der *Reihenprüfsummen* ist größer als der Fehlerschwellwert: $|S_{r,1}| > \epsilon_{r,1}$
2. Das Syndrom der *gewichteten* Prüfsumme bei der Überprüfung der *Reihenprüfsummen* ist größer als der Fehlerschwellwert: $|S_{r,w}| > \epsilon_{r,w}$
3. Das Syndrom der *ungewichteten* Prüfsumme bei der Überprüfung der *Spaltenprüfsummen* ist größer als der Fehlerschwellwert: $|S_{c,1}| > \epsilon_{c,1}$
4. Das Syndrom der *gewichteten* Prüfsumme bei der Überprüfung der *Spaltenprüfsummen* ist größer als der Fehlerschwellwert: $|S_{c,w}| > \epsilon_{c,w}$

Aus dem Syndrompaar der gewichteten und ungewichteten Prüfsummen lässt sich für 1.) und 2.) die fehlerhafte Spalte berechnen, für 3.) und 4.) die fehlerhafte Reihe. Stimmen diese Angaben miteinander überein, kann das fehlerhafte Element direkt über die Syndrome korrigiert werden. Für die Syndrome der ungewichteten Prüfsummen gilt bei einem fehlerhaften Element pro Reihe und Spalte

$$S_{r,1} = S_{c,1}, \tag{6.14}$$

da der Fehler bei der Bildung der Prüfsummen der entsprechenden Reihen- und Spaltenprüfsummen mit dem gleichen Gewicht addiert wird.

Für jedes Element (i, j) gibt es dadurch drei Möglichkeiten für die Lokalisierung von auftretenden Fehlern:

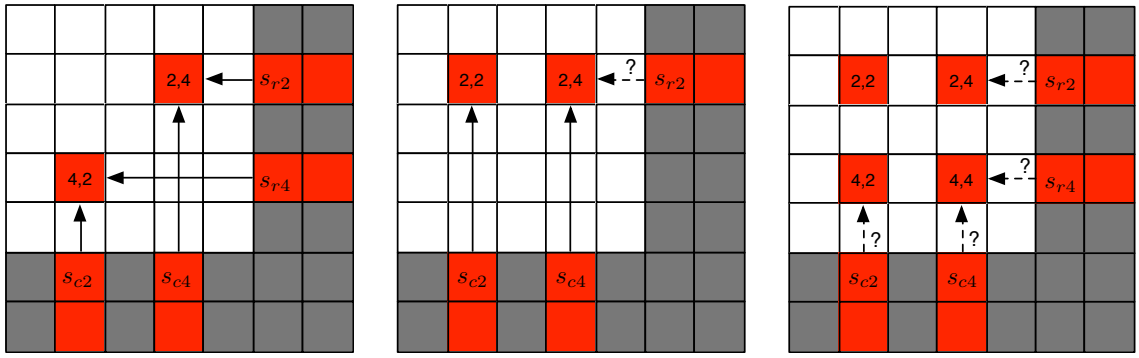
1. Die ungewichteten Syndrome der Reihenprüfsumme i und der Spaltenprüfsumme j überschreiten den Fehlerschwellwert und stimmen überein.
2. Das Syndrom der gewichteten Prüfsumme der Reihe i überschreitet den Fehlerschwellwert und lokalisiert das fehlerhafte Element in Spalte j.
3. Das Syndrom der gewichteten Prüfsumme der Spalte j überschreitet den Fehlerschwellwert und lokalisiert das fehlerhafte Element in Reihe i.

Auftretende Fehler können dadurch in Fehlerklassen eingeteilt werden:

- Fehlerklasse F_2 (*Eindeutiger, doppelt lokalisierter Fehler*):
  Sowohl für die Reihen- als auch für die Spaltenprüfsumme überschreitet jeweils mindestens ein Syndrom den Fehlerschwellwert. Die Lokalisierung des fehlerhaften Elements über die Syndrome von gewichteten und ungewichteten Spaltenprüfsummen in (i, j) stimmt mit der Lokalisierung über die Syndrome von gewichteten und ungewichteten Reihenprüfsummen überein. Darüber hinaus stimmt das Syndrom der ungewichteten Reihen- und Spaltenprüfsummen ebenfalls überein ($S_{r,1} = S_{c,1}$).
- Fehlerklasse F_1 (*Eindeutiger, einfach lokalisierter Fehler*):
  Es überschreitet entweder mindestens ein Syndrom der Reihenprüfsummen oder mindestens ein Syndrom der Spaltenprüfsummen den Fehlerschwellwert. Über das Syndrom der gewichteten und ungewichteten Reihenprüfsumme (Spaltenprüfsumme) kann die zugehörige Spalte (Reihe) ermittelt werden. Das Syndrom der ungewichteten Reihen- und Spaltenprüfsummen stimmt überein.
- Fehlerklasse F_0 (*Nicht eindeutiger, beliebig lokalisierter Fehler*):
  Ein Fehler wurde durch eine beliebige Prüfsummenkombination lokalisiert, die Syndrome der ungewichteten Prüfssummen stimmen nicht überein. Daraufhin muss davon ausgegangen werden, dass sich in einer Reihe oder einer Spalte mehrere fehlerhafte Elemente befinden.

Durch die Verwendung von gewichteten Prüfsummen in einer vollständigen Prüfsummenmatrix können prinzipiell mehrere Fehler in einem Vektor korrigiert werden. Befinden sich beispielsweise mehrere fehlerhafte Elemente innerhalb eines Reihenvektors, können diese durch die Syndrome der Spaltenprüfsummen lokalisiert und damit korrigiert werden. Die durch die Reihenprüfsumme gegebene Information ist dadurch nicht verwendbar. Eine nicht korrigierbare Situation ergibt sich, wenn mehrere Fehler pro Reihenvektor und pro Spaltenvektor auftreten. Abbildung 6.13 zeigt die Auswirkungen von mehreren fehlerhaften Elementen pro Vektor. In Abbildung 6.13a kann die Eindeutigkeit jedes fehlerhaften Elements bestimmt werden, indem neben der Lokalisierung über die Syndrome der gewichteten und ungewichteten Spalten- und Reihenprüfsummen die zugehörigen ungewichteten Syndrome der Spalten- und Reihenprüfsummen übereinstimmen: $s_{c2} = s_{r4}$ und $s_{c4} = s_{r2}$. Bei zwei

fehlerhaften Elementen pro Reihenvektor in Abbildung 6.13b lokalisieren die Syndrome der Spaltenprüfsummen der Spalten 2 und 4 das fehlerhafte Element in Reihe 2. Bei der Überprüfung der Syndrome der ungewichteten Reihen- und Spaltenprüfsummen gilt $s_{r2} = s_{c2} + s_{c4}$. Bei mehreren fehlerhaften Elementen pro Reihenvektor und pro Spaltenvektor, wie in Abbildung 6.13c dargestellt, kann keine solche Beziehung festgestellt werden und diese Situation dadurch detektiert werden. Eine Ausnahme hiervon bildet die Situation eines uniformen Fehlers über alle fehlerhaften Elemente hinweg.



(a) **(b)** **(c)**

Abbildung 6.13.: Auswirkungen mehrerer fehlerhafter Elemente in einem ABFT-Block. (a) Im Fall von unterschiedlichen Reihen und Spalten können Fehler eindeutig lokalisiert werden. (b) Bei Auftreten von mehreren Fehlern in einer Reihe können diese dennoch über die Spaltenprüfsummen lokalisiert werden. (c) Eine nicht korrigierbare Situation mit mehreren Fehlern pro Spalten- und Reihenvektor.

Als Folge dieser Betrachtung wird die Korrekturmaßnahme für zwei Prüfsummen mit erweiterter Lokalisierung auf maximal ein fehlerhaftes Element pro Spaltenvektor und pro Reihenvektor beschränkt. Alle Fehler, die dieser Restriktion nicht entsprechen, sollen bei Detektion durch eine Neuberechnung des ABFT-Blocks korrigiert werden. Aus dieser Betrachtung lässt sich ein Schema zur Korrektur von maximal BS_{ABFT} fehlerhaften Elementen pro ABFT-Block der Größe BS_{ABFT} definieren:

1. Überprüfe die gewichteten und ungewichteten Syndrome für jede Reihenprüfsumme und speichere das Syndrom für jede ungewichtete Reihenprüfsumme. Überschreitet mindestens ein Syndrom den zugehörigen Schwellwert, dann berechne die Lokalisierung der entsprechenden Spalte j durch die Syndrome von gewichteten und ungewichteten Prüfsummen. Speichere zu jeder so lokalisierten Spalte j die zugehörige Reihe i und zu jeder Reihe i die zugehörige lokalisierte Spalte j.
2. Überprüfe die gewichteten und ungewichteten Syndrome für jede Spaltenprüfsumme.
   a) Überschreitet mindestens ein Syndrom den zugehörigen Schwellwert, dann berechne die Lokalisierung der Reihe i über die Syndrome der gewichteten und ungewichteten Prüfsummen. Für jede Spalte, die eine Lokalisierung durchgeführt hat, wird verglichen, ob die zu dieser Spalte im ersten Schritt ermittelte Information

über die Reihe mit dieser Reihe übereinstimmt. Kommt es zu einer Abweichung, muss der ABFT-Block neu berechnet werden, da die Lokalisierung nicht eindeutig ist. Des Weiteren muss überprüft werden, ob das Syndrom der ungewichteten Prüfsummen dieser Spalte mit dem Syndrom der ungewichteten Prüfsumme der lokalisierten Reihe übereinstimmt. Ist dies nicht der Fall, muss der ABFT-Block neu berechnet werden.

b) Überschreitet kein Syndrom den zugehörigen Schwellwert und für diese Spalte wurde im ersten Schritt eine zugehörige Reihe gespeichert, kann hier ebenfalls ein Überprüfungsschritt durchgeführt werden. Dazu werden die Syndrome der ungewichteten Prüfsummen verglichen. Bei einer Abweichung wird der ABFT-Block neu berechnet.

6.4. Implementierung

Der Entwurf von Kodierungs- und Überprüfungskernel muss mehrere Faktoren berücksichtigen. An erster Stelle steht die Notwendigkeit, Kodierung und Überprüfung durch unterschiedliche Prozessorkerne durchzuführen. Da keine Aussage darüber getroffen werden kann, auf welchem Multiprozessor verschiedene Blöcke bearbeitet werden, muss die Verteilung innerhalb eines Multiprozessors auf verschiedene Prozessorkerne erfolgen. Ausreichende Information kann über die Warp-Größe bereitgestellt werden. Diese beschreibt die Anzahl zusammengeschalteter Prozessorkerne, die dem gleichen Kontrollfluss unterliegen und bei denen davon ausgegangen werden kann, dass die Verteilung von *Threads* statisch erfolgt. Der zweite Faktor zu berücksichtigende Faktor ist die Auslastung der GPU. Eine massiv parallele Architektur erreicht einen hohen Datendurchsatz durch eine große Anzahl an parallel ausgeführten Instruktionen. Ein dritter Faktor ist die Berücksichtigung von Thread-Blockgrößen. Pro Threadblock kann nur eine maximale Anzahl von 1024 Threads bei der aktuellen Generation der Hardware verwendet werden. Der innerhalb eines Threadblocks nutzbare Speicher ist das *Shared Memory*.

6.4.1. Kodierungskernel

Der Kodierungskernel wurde so konzipiert, dass beliebige ABFT-Blockgrößen und eine beliebige Anzahl von Gewichten zur Kodierung einer Matrix eingesetzt werden kann. Um eine möglichst hohe Auslastung der GPU zu erreichen und die im Sinne der Performanz kostspielige Abfrage von Sonderfällen zu vermeiden wurde die ABFT-Blockgröße auf den Einsatz von Zweierpotenzen beschränkt. Prinzipiell ist diese Implementierung jedoch in der Lage, mit absehbarem Aufwand auch weitere ABFT-Blockgrößen unterstützen zu können.

Algorithmisches Konzept

Der grundlegende Gedanke für den Kodierungskernel besteht darin, eine Anzahl von BS_{ABFT} *Threads* äquivalent zur ABFT-Blockgröße zur Kodierung zu verwenden. Jeder *Thread-Block* berechnet dabei für einen ABFT-Block der Matrix **A** die Reihen- und Spaltenprüfsummen. Dazu wird dieser ABFT-Block von **A** in das *Shared Memory* geladen, jeder *Thread* kodiert dann zunächst eine Reihe, im Anschluß eine Spalte des ABFT-Blocks. Die Abbildung 6.14 zeigt dieses Vorgehen. Datenelemente sind dabei weiß dargestellt, Prüfsummenelemente in grau. Die einzelnen *Threads* sind durch schwarze Punkte markiert.

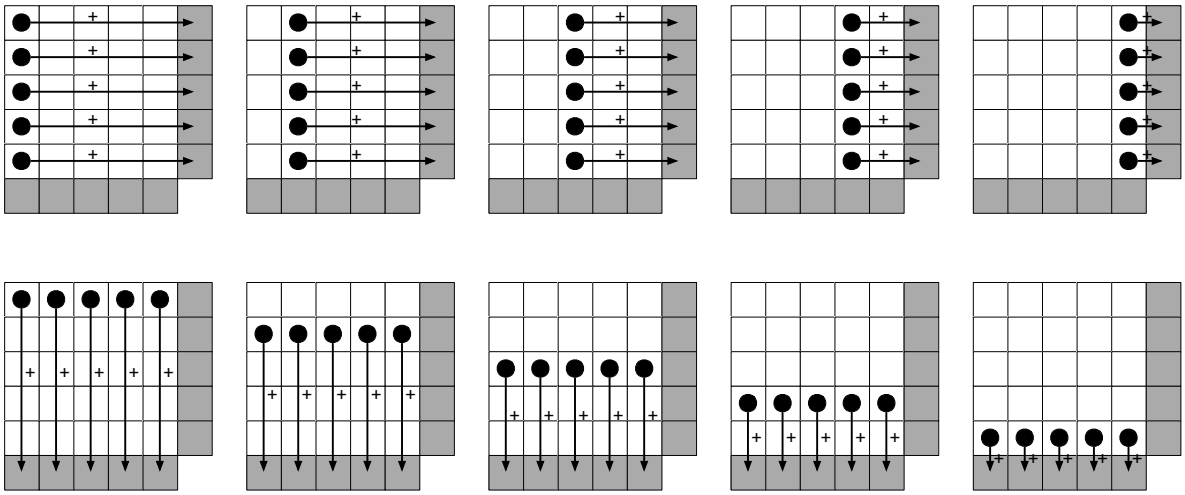


Abbildung 6.14.: Kodierung einer vollständigen Prüfsummenmatrix der Größe $BS_{ABFT} \times BS_{ABFT}$ mit BS_{ABFT} Threads.

Aufgrund der begrenzten Kapazität des *Shared Memory* muss der ABFT-Block partitioniert werden, wenn die ABFT-Blockgröße diese Kapazität überschreitet. Dazu wird der ABFT-Block in Streifen von Reihen unterteilt, die Spaltenanzahl und Anzahl verwendeter *Threads* entspricht weiterhin der ABFT-Blockgröße BS_{ABFT}. Die Breite BS_X dieses Streifens kann bestimmt werden durch

$$y = BS_{ABFT} \cdot BS_X, \tag{6.15}$$

wobei y kleiner als die Kapazität des *Shared Memories* und BS_{ABFT} durch BS_X teilbar sein muss. Der zu kodierende ABFT-Block wird nun streifenweise mit BS_X Reihen und der vollen Anzahl Spalten in das *Shared Memory* geladen. Auch hier summiert jeder *Thread* zunächst die Werte aus der ihm zugewiesenen Spalte, bis alle Elemente aus dem im *Shared Memory* befindlichen Streifen abgearbeitet wurden. Für die Berechnung der Reihenprüfsummen kann weiterhin die gesamte Anzahl an *Threads* verwendet werden, indem der zu kodierende Streifen in BS_{ABFT}/BS_X Blöcke unterteilt wird. In Abbildung 6.15 wurde der ABFT-Block beispielsweise in zwei Streifen unterteilt. Zunächst summiert jeder *Thread* für seine Spalte die Hälfte der Elemente auf (links), dann wird der Streifen wiederum in zwei Spaltenblöcke unterteilt, je zwei *Threads* summieren damit eine Reihe (rechts). Die Reihenprüfsummen für die

obere Hälfte des ABFT-Blocks sind damit vollständig berechnet, für die Spaltenprüfsummen steht das Zwischenergebnis für die weitere Bearbeitung bereit.

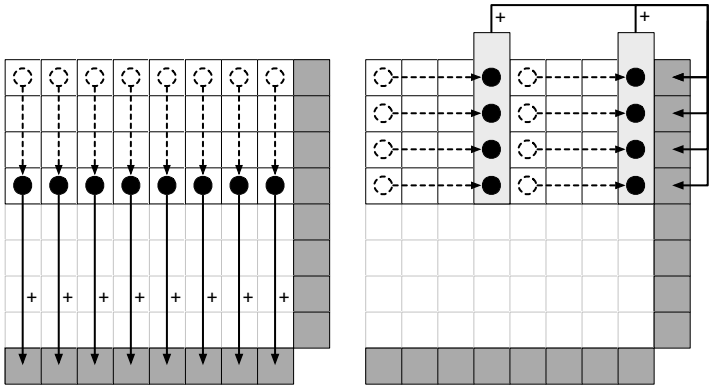


Abbildung 6.15.: Erster Schritt des Kodierungskernels für ABFT-Blockgrößen, die die Kapazität des *Shared Memory* übersteigen. Es wird ein Streifen von Reihen in das *Shared Memory* geladen und mit der vollen Anzahl *Threads* der obere Teil der Spaltenprüfsummen addiert (links). Zur Kodierung der Reihenprüfsummen kann ebenfalls die volle Anzahl *Threads* durch weitere Partitionierung und anschließende Akkumulation genutzt werden (rechts).

Im Anschluß wird, dargestellt in Abbildung 6.16, der zweite Streifen in das *Shared Memory* geladen und damit die untere Hälfte der Elemente von jedem *Thread* für die Spaltenprüfsummen summiert (links). Diese Werte werden dabei auf die im vorhergehenden Schritt berechneten Teilergebnisse der Spaltenprüfsummen addiert. Als letzter Schritt werden die Reihenprüfsummen dieses Streifens wieder mit zwei *Threads* pro Reihe berechnet (rechts). Damit ist der gesamte ABFT-Block kodiert.

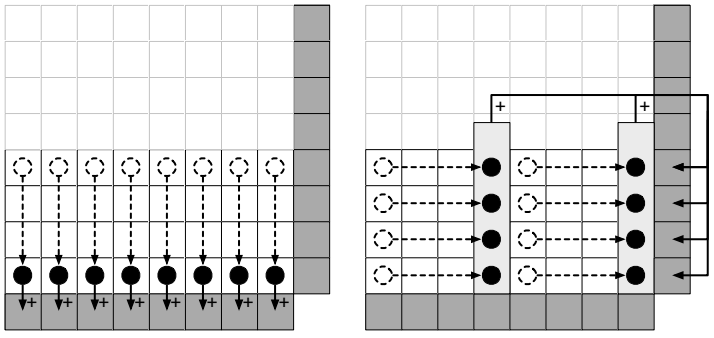


Abbildung 6.16.: Zweiter Schritt des Kodierungskernels für ABFT-Blockgrößen, die die Kapazität des *Shared Memory* überschreiten. Hier wird der zweite Streifen von Reihen nach dem gleichen Schema mit der vollen Anzahl an *Threads* kodiert.

Um nun die durch das ABFT-Schema gegebene Anforderung der redundanten Prüfsummenberechnung durch zwei unabhängige Berechnungseinheiten zu integrieren, müssen zusätzliche *Threads* verwendet werden. Die Identifikation von *Threads* innerhalb eines *Thread-Blocks* erfolgt durch ein maximal dreidimensionale *Thread-ID tid*, auf die in jedem *Thread* mittels $tid.x$, $tid.y$ und $tid.z$ zugegriffen werden kann. Die Einteilung von *Threads* in *Warps* erfolgt nach dem CUDA-Programmiermodell zuerst in x-Richtung, dann in y-Richtung und zum Schluß in z-Richtung. Für den Kodierungskernel wird daher die x-Dimension für verschieden gewichtete Prüfsummen und die redunante Berechnung verwendet. Die y-Dimension wird für die BS_{ABFT} benötigten *Threads* zur Kodierung benutzt. Im Fall von zwei Prüfsummen berechnen beispielsweise die *Threads* mit $tid.x = 0$ die normale Prüfsumme,*Threads* mit $tid.x = 1$ die redundante Version der normalen Prüfsumme, *Threads* mit $tid.x = 2$ die zweite Prüfsumme und *Threads* mit $tid.x = 3$ die redundante Version der zweiten Prüfsumme. Insgesamt werden also $2 \cdot numWCS$ *Threads* in x-Dimension verwendet, die alle in den selben *Warp* eingeteilt werden können. Für die Überprüfung der redundant berechneten Prüfsummen können nun *Shuffle Instructions* benutzt werden, die den direkten Austausch von Daten innerhalb eines *Warps* von verschiedenen *Threads* ohne den Weg über das *Shared Memory* ermöglichen. Die Funktion `shuffleAndCheck()` in Algorithmus A.1 verbindet diesen Austausch mit einem Test auf Gleichheit und gibt dem Benutzer und der ABFT-BLAS-Bibliothek im Fehlerfall eine Rückmeldung darüber, in welchem Block ein Fehler aufgetreten ist.

Der gesamte Kodierungskernel für beliebige ABFT-Blockgrößen und beliebige Anzahl Gewichte ist in Algorithmus A.3 dargestellt. Die hierbei verwendeten Hilfsfunktionen `rowWeight`(col, cs) und `colWeight`(row, cs) geben Zugriff auf die jeweiligen, durch $rowWeights[]$ und $colWeights[]$ übergebenen Gewichte.

Grenzen der Implementierung durch die Hardware

Für jeden Kernel gelten die Restriktionen, die durch die Ressourcen der verwendeten Hardware gestellt werden. Diese Implementierung ist in der Lage, bei Verwendung von zwei Prüfsummen und redundanter Prüfsummenberechnung ABFT-Blöcke bis zu einer Größe von 256×256 zu kodieren. Pro *Thread-Block* werden dabei $2 \cdot 2 \times 256 = 1024$ *Threads* eingesetzt, was genau der maximalen Anzahl *Threads* pro *Thread-Block* der aktuellen Generation der eingesetzten Hardware entspricht. Sollen mehr gewichtete Prüfsummen verwendet werden, dann kann maximal eine ABFT-Blockgröße von 128×128 verwendet werden.

6.4.2. Kernel zur Fehlerschwellwertbestimmung

Im Folgenden werden die entwickelten Kernel zur Fehlerschwellwertbestimmung nach dem Verfahren der vereinfachten Fehleranalyse und dem probabilistischen Verfahren unter Verwendung der betragsmäßig größten Elemente, sowie der lokalen und globalen Durchschnittsmetrik beschrieben. Basierend auf der Analyse des Einsatzschemas wurden diese

Methoden zur Fehlerschwellwertbestimmung in einen kontextunabhängigen Vorverarbeitungsschritt und einen kontextabhängigen Bestimmungsschritt unterteilt. Die meisten dieser Kernel sind abhängig von Blockgrößen BS_{ABFT}, BS_X, BS_Y oder BS_Z. BS_{ABFT} entspricht hierbei immer der ABFT-Blockgröße. BS_X, BS_Y und BS_Z beschreiben die Anzahl *Threads*, die in x-Richtung, y-Richtung und z-Richtung in einem *Thread-Block* verwendet werden. Diese Größen sind zumeist begrenzt durch die maximal verwendbaren Anzahl *Threads* pro *Thread-Block* oder der maximalen Kapazität des *Shared Memory*. Durch Einsatz dieser Parametrisierung sind diese Kernel auch für künftige Generationen der Hardware anpassbar. Die Hilfsfunktionen rows($\mathbf{A_{fc}}$) und columns($\mathbf{A_{fc}}$) geben die Anzahl an Reihen beziehungsweise Spalten der Matrix $\mathbf{A_{fc}}$ inklusive der zusätzlichen Prüfsummenvektoren zurück, die Hilfsfunktionen originalRows($\mathbf{A_{fc}}$) und originalColumns($\mathbf{A_{fc}}$) geben die Anzahl Reihen beziehungsweise Spalten ohne diese Prüfsummenvektoren zurück.

Fehlerschwellwertbestimmung für die vereinfachte Fehleranalyse

Bei diesem Verfahren werden Vektornormen für jede Reihe der Matrix $\mathbf{A_{fc}}$ und jede Spalte der Matrix $\mathbf{B_{fc}}$ inklusive der Prüfsummenvektoren benötigt. Im kontextunabhängigen Vorverarbeitungsschritt können diese Normen für jeden Reihen- und Spaltenvektor der Matrizen $\mathbf{A_{fc}}$ und $\mathbf{B_{fc}}$ bestimmt werden. Im Kernel `determineVectorNormsRow()`, dargelegt in Algorithmus A.2, werden diese Normen parallel für alle Reihen, aufgeteilt in mehrere *Threads* pro *Thread-Block* berechnet. Jeder *Thread* bearbeitet hierbei eine Reihe der Matrix $\mathbf{A_{fc}}$.

Im anschließenden kontextabhängigen Schritt kann basierend auf den so bestimmten Normen der Reihen- und Spaltenvektoren der Fehlerschwellwert für das jeweilige Prüfsummenelement berechnet werden. Dieses Vorgehen ist in Algorithmus A.4 exemplarisch für die Bestimmung der Fehlerschwellwerte der Reihenprüfsummen im Kernel `determineRowEpsilonSEA()` dargelegt. Die Abbildung 6.17 zeigt die für die Bestimmung der Fehlerschwellwerte der Reihenprüfsummen benötigten Daten.

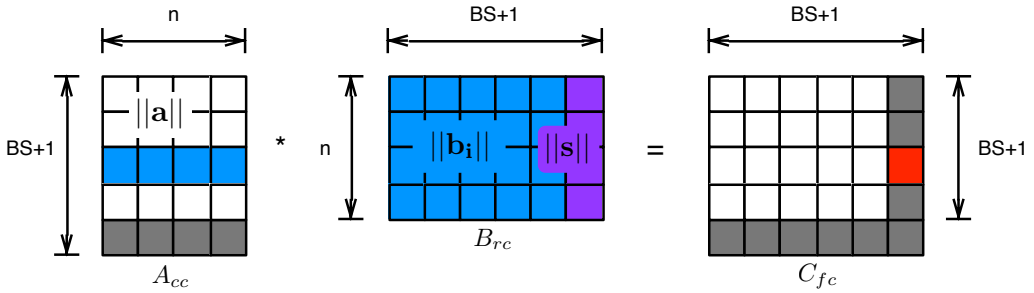


Abbildung 6.17.: Für die Bestimmung des Fehlerschwellwerts von Reihenprüfsummen benötigte Daten nach dem Verfahren der vereinfachten Fehleranalyse.

Für diesen Kernel wird hierbei ein *Thread-Block* in der x-Dimension mit BS_{ABFT} *Threads* in Übereinstimmung zur ABFT-Blockgröße gestartet. Diese Indizierung wird zum Einen

benutzt, um die BS_{ABFT} Normen der Spaltenvektoren ins *Shared Memory* zu laden und mittels einer Reduktion zu summieren, zum Anderen um mehrere Reihen und damit mehrere Reihenprüfsummen parallel zu berechnen. Die y-Dimension der Indizierung innerhalb eines *Thread-Blocks* wird darüber hinaus verwendet, um die Anzahl der berechneten Schwellwerte für die Reihenprüfsummen pro *Thread-Block* weiter zu erhöhen. Wird der Kernel also für eine ABFT-Blockgröße 32×32 verwendet, so ist die Anzahl *Threads* in der x-Dimension mit 32 festgelegt. Da maximal 1024 *Threads* pro *Thread-Block* von der Hardware unterstützt werden, können in y-Dimension nochmals 32 *Threads* eingesetzt werden. Dadurch ist ein *Thread-Block* in der Lage, für bis zu 1024 Reihen die Fehlerschwellwerte für die Reihenprüfsummen zu bestimmen.

Fehlerschwellwertbestimmung für das probabilistische Verfahren durch die betragsmäßig größten Elemente

Bei dieser Methode zur Fehlerschwellwertbestimmung müssen zunächst die $nMax$ betragsmäßig größten Elemente für jeden Reihenvektor und jeden Spaltenvektor einer Matrix $\mathbf{A_{fc}}$ bestimmt werden. Der Kernel `determineMaxElementsRow()` für diesen kontextunabhängigen Bestimmungsschritt ist in Algorithmus A.5 exemplarisch für die Bestimmung der betragsmäßig größten Elemente der Reihenvektoren dargelegt. Ein zweiter Kernel zur Bestimmung der betragsmäßig größten Elemente der Spaltenvektoren ist analog aufgebaut und daher nicht explizit beschrieben. Bei diesem Kernel wird eine Anzahl von BS_X Reihen parallel in einem *Thread-Block* bearbeitet. Die dazu benötigten Absolutwerte aus der Matrix $\mathbf{A_{fc}}$ werden in ein Feld im *Shared Memory* der Größe $BS_X \cdot BS_Y$ geladen, wobei BS_Y durch die Kapazität des *Shared Memory* begrenzt ist und angibt, wie viele Elemente pro Reihenvektor geladen werden können. Jeder *Thread* bestimmt iterativ die $nMax$ betragsmäßig größten Elemente und die zugehörigen Positionen innerhalb dieses Felds und setzt den dabei gefundenen Wert auf 0. Damit wird dieser bei der anschließenden Bestimmung des betragsmäßig zweitgrößten Elements nicht nochmals herangezogen. Sind alle $nMax$ Elemente bestimmt, dann wird der nächste $BS_X \cdot BS_Y$ Block von Absolutwerten der Matrix $\mathbf{A_{fc}}$ in das Feld im *Shared Memory* geladen, davon ebenfalls die $nMax$ betragsmäßig größten Elemente bestimmt und mit den Werten aus dem vorhergehenden Schritt vereinigt. Nachdem alle Elemente jeder Reihe betrachtet wurden, liegen für jede Reihe die $nMax$ betragsmäßig größten Elemente und die Information vor, an welcher Position innerhalb des Vektors sich diese befinden. Diese Informationen werden im globalen Speicher abgelegt und können im nachfolgenden kontextabhängigen Schritt verwendet werden.

Der Kernel `determineRowEpsilonMaxElements()` für den kontextabhängigen Schritt zur Bestimmung des Fehlerschwellwerts ist in Algorithmus A.7 exemplarisch für die Reihenprüfsummenelemente dargelegt. Die Bestimmung für die Spaltenprüfsummenelemente erfolgt analog und ist deshalb nicht explizit beschrieben. Bei diesem Kernel wird für jedes Reihenprüfsummenelement die Position und der Wert der $nMax$ betragsmäßig größten Elemente der entsprechenden Reihen- und Spaltenkombination in den lokalen Speicher geladen und nach dem in Kapitel 6.3.5 beschriebenen Verfahren zur probabilistischen Abschätzung des

Rundungsfehlers verwendet. Die Formel zur Bestimmung des Fehlerschwellwerts für das probabilistische Verfahren ist in Algorithmus A.6 dargelegt.

Fehlerschwellwertbestimmung für das probabilistische Verfahren durch lokale Durchschnittsmetrik

Für das probabilistische Verfahren zur Fehlerschwellwertbestimmung unter Verwendung der lokalen Durchschnittsmetrik müssen zunächst für jeden Block von 32 Elementen eines jeden Vektors die Durchschnittswerte und Bitvektoren für alle Kodierungsebenen generiert werden. Der hierfür eingesetzte Kernel `determineBitvectorsLocalAverage()` ist in Algorithmus A.8 exemplarisch für die Kodierung der Reihenprüfsummen dargestellt. Beim Laden der Absolutwerte für einen Block wird neben dem ersten Durchschnittswert in *avg* das Maximum dieser Werte bestimmt, das für die höchste Kodierungsebene anstelle des Durchschnittswerts gespeichert wird. In den nachfolgenden Schleifendurchgängen werden iterativ die Kodierungsebenen aufsteigend generiert. Dazu wird im Bitvektor *bitVec* die Stelle mit einer 1 kodiert, bei der das Element über dem letzten Durchschnittswert *avg* liegt. Diese Werte werden in *sum_gt_avg* aufsummiert und mit der Anzahl Elemente in *count* zur Bestimmung des nächsten Durchschnittswerts verwendet.

Im kontextabhängigen Schritt kann nun aus den Bitvektoren und zugehörigen Durchschnittswerten für jeden im Skalarprodukt kombinierten Block eine lokale obere Schranke bestimmt werden. Eine obere Schranke für das entsprechende Prüfsummenelement ergibt sich dann als das Maximum aller lokalen Schranken. Die Funktion `determineBlockBound()` für die Bestimmung einer lokalen oberen Schranke für einen Block ist in Algorithmus A.9 dargelegt. Dieses entspricht dem in Kapitel 6.3.5 für die lokale Durchschnittsmetrik beschriebenen Konzept. Der Operator & entspricht dabei einer bitweisen UND-Operation. Im Kernel `determineRowEpsilonBitvectorsLocalAverage()` in Algorithmus A.10 werden *Threads* in zwei Dimensionen pro *Thread-Block* eingesetzt, um diese lokalen Schranken zu berechnen und anschließend zu einer oberen Schranke für das Reihenprüfsummenelement zu kombinieren. Zum Einen wird dabei die x-Dimension mit 32 *Threads* entsprechend der Warp-Größe verwendet, um 32 Blöcke von 32 Elementen zu bearbeiten. Übersteigt die Länge der betrachteten Vektoren die damit verarbeitbare Größe von 1024 Elementen, so wird zusätzlich die z-Dimension des *Thread-Blocks* benutzt, um weitere Blöcke zu bearbeiten. Die y-Dimension des *Thread-Blocks* kann damit für die parallele Bearbeitung mehrerer Reihen verwendet werden. Nachdem die lokalen oberen Schranken für den jeweiligen Block bestimmt wurden, werden diese innerhalb eines Warps über *Shuffle Instructions* zu einer oberen Schranke über alle 32 Blöcke innerhalb dieses Warps reduziert. Im Anschluß wird dann die Schranke für den gesamten Vektor durch Reduktion in z-Richtung bestimmt. Diese Schranke wird zur Berechnung des Fehlerschwellwerts eingesetzt und in den globalen Speicher geschrieben.

Fehlerschwellwertbestimmung für das probabilistische Verfahren durch globale Durchschnittsmetrik

Bei Verwendung einer globalen Durchschnittsmetrik müssen im kontextunabhängigen Vorverarbeitungsschritt für jeden Reihen- und Spaltenvektor globale Durchschnittswerte verwendet werden. Der Kernel `determineBitvectorsGlobalAverage()` in Algorithmus A.11 zeigt dieses Vorgehen. Dieser Kernel ist dabei ähnlich aufgebaut wie der Kernel für den Vorverarbeitungsschritt der lokalen Durchschnittsmetrik. Hier wird jedoch eine Reduktion verwendet, um aus den lokalen Summen der Elemente, die über dem vorherigen Durchschnittswert liegen eine für den Vektor globale Summe zu generieren. Analog dazu wird die Anzahl Elemente, die über dem vorherigen Durchschnittswert zu einer globalen Anzahl reduziert. Basierend auf diesen Daten kann dann ein neuer Durchschnittswert berechnet werden. Das Maximum für einen Vektor wird ebenfalls nach dem gleichen Reduktionsschema berechnet.

6.4.3. Korrekturkernel für Prüfsummen mit einfacher Lokalisierung

Der Korrekturkernel für Prüfsummen mit einfacher Lokalisierung implementiert die in Kapitel 6.3.6 beschriebenen Korrekturmaßnahmen. Hierbei kann maximal ein fehlerhaftes Element pro ABFT-Block korrigiert werden. Dieser Kernel ist für eine beliebige Anzahl Prüfsummen *numWCS* ausgelegt, die erste Prüfsumme muss dabei jedoch immer die normale Prüfsumme sein. Die prinzipielle algorithmische Struktur des Korrekturkernels ist an den Kodierungskernel angelegt, die Einteilung von *Threads* in x-Richtung und y-Richtung erfolgt analog. Durch die Beschränkung auf eine einfache Lokalisierung kann maximal ein fehlerhaftes Element eines ABFT-Blocks korrigiert werden. Die Funktion `checkSingleCorrectable()` in Algorithmus A.13 überprüft, ob die Differenz von neu berechneter Prüfsumme und Referenzprüfsumme über dem zugehörigen Fehlerschwellwert liegt. Bei jeder auftretenden Überschreitung des Fehlerschwellwerts wird die zugehörige Reihe (oder Spalte) gespeichert. Dabei wird überprüft, ob in einer beliebigen anderen Reihe (oder Spalte) des ABFT-Blocks bereits ein Fehlerschwellwert überschritten wurde. In diesem Fall können Fehler nicht korrigiert werden. Die Verwendung von atomaren Funktionen verhindert hierbei, dass mehrere *Threads* in diese Position schreiben und so einen Fehler durch eine Wettlaufsituation (engl. *race condition*) maskieren. Die eigentliche Korrektur erfolgt nach erfolgreicher Lokalisierung durch die Reihen- und Spaltensyndrome. Dabei kann das Element nur korrigiert werden, wenn sowohl die Überprüfung der Reihen- als auch Spaltenprüfsummen genau eine fehlerhafte Reihe und genau eine fehlerhafte Spalte ergeben hat. Die zugehörige Funktion `correctSingleError()` ist in Algorithmus A.14 dargestellt. Die Korrektur von Reihenprüfsummenelementen erfolgt anhand der bereits in $\mathbf{C_{fc}}$ gespeicherten, neu berechneten Prüfsummen. Die Korrektur des Spaltenprüfsummenelements erfolgt dagegen direkt auf Registerebene, da diese Spaltenprüfsumme im Anschluss an den Aufruf dieser Funktion die Referenzprüfsumme in der Matrix $\mathbf{C_{fc}}$ speichert (vergleiche Algorithmus A.15).

6.4.4. Korrekturkernel für Prüfsummen mit erweiterter Lokalisierung

Der Korrekturkernel für Prüfsummen mit erweiterter Lokalisierung kann bis zu BS_{ABFT} Fehler in einem ABFT-Block korrigieren, wenn pro Reihenvektor und pro Spaltenvektor maximal ein fehlerhaftes Element auftritt. Das grundlegende Konzept für das Korrekturschema ist in Kapitel 6.3.6 beschrieben. Für die erweiterte Lokalisierung und die Überprüfung einer 1-zu-1 Zuordnung von lokalisierten Reihen und Spalten werden zwei Felder der Größe BS_{ABFT} im *Shared Memory* verwendet: Das Feld $locRow[BS_{ABFT}]$ und das Feld $locCol[BS_{ABFT}]$. In $locRow[row]$ wird die Information über die lokalisierte oder erwartete fehlerhafte Spalte gespeichert, in $locCol[col]$ die Information über die lokalisierte oder erwartete fehlerhafte Reihe gespeichert. Beide Felder werden zu Beginn des Kernels mit -1 initialisiert, was die Information über eine nicht vorliegende Lokalisierung kodiert. Zentrales Element für eindeutige Lokalisierung von fehlerhaften Elementen ist die in Algorithmus A.16 beschriebene Funktion `checkAndLocateError()`. Hier werden *Shuffle Instructions* benutzt, um die Information über das Überschreiten eines Schwellwerts an die *Threads* zu propagieren, die die gleiche Reihe (oder Spalte), jedoch ein anderes Gewicht bearbeiten. Bei der Überprüfung der Lokalisierung in dieser Funktion wird zunächst untersucht, ob für diese Reihe (Spalte) bereits eine Lokalisierung des fehlerhaften Elements in einer bestimmten Spalte (Reihe) erwartet wird. Dies ist dann der Fall, wenn im Feld $locSelf$ (bei Überprüfung der Reihenprüfsummen *locRow*, bei Überprüfung der Spaltenprüfsummen *locCol*) an der dem *Thread* zugehörigen Position ein anderer Wert als -1 steht. Stimmt die von dieser Reihe (Spalte) lokalisierte Spalte (Reihe) nicht mit der erwarteten Spalte (Reihe) überein, dann ist die Bedingung für maximal ein fehlerhaftes Element pro Vektor nicht mehr gegeben und eine 1-zu-1 Zuordnung kann nicht hergestellt werden. In diesem Fall wird dem Benutzer und der ABFT-BLAS-Bibliothek diese Information zur Verfügung gestellt, um beispielsweise die Neuberechnung des ABFT-Blocks zu initiieren. Ist diese Überprüfung erfolgreich oder wurden keine Erwartungen an die Lokalisierung gestellt, dann wird die Information über die Lokalisierung sowohl für die Reihe durch Speichern der lokalisierten Spalte in $locRow[row]$, als auch für die Spalte durch Speichern der zugehörigen Reihe in $locCol[col]$ gesichert. Dies dient dann als neue erwartete Lokalisierung bei der nächsten Überprüfung.

Die Funktion `correctMultiError()` in Algorithmus A.17 überprüft die Eindeutigkeit der Zuordnung über den Vergleich der Syndrome der ungewichteten Prüfsummen. Jeder *Thread* überprüft dabei für seine Spalte, ob eine fehlerhaftes Element in einer Reihe lokalisiert wurde. Ist dies der Fall, dann wird das ungewichtete Syndrom der Spaltenprüfsummen berechnet und an alle dieser Spalte zugewiesenen *Threads* mittels einer *Shuffle Instruction* propagiert. Über den zum jeweiligen Prüfsummenelement gehörenden Fehlerschwellwert kann entschieden werden, ob das Syndrom der ungewichteten Reihenprüfsummen und das Syndrom der ungewichteten Spaltenprüfsummen bis auf einen Rundungsfehler identisch sind. Ist diese Bedingung nicht gegeben, dann muss davon ausgegangen werden, dass in einer Reihe oder einer Spalte mehrere fehlerhafte Elemente aufgetreten sind. Die Eindeutigkeit der Lokalisierung ist in diesem Fall nicht gegeben und die Korrektur dieses ABFT-Blocks muss durch andere Maßnahmen durchgeführt werden. Es wird dazu eine entsprechende Information für den Benutzer und die ABFT-BLAS-Bibliothek gespeichert. Sind die Syndrome jedoch identisch, so kann das fehlerhafte Element eindeutig lokalisiert und damit korrigiert

werden. Neben der Korrektur des Datenelements in der Ergebnismatrix durch das Syndrom einer ungewichteten Prüfsumme müssen auch die Reihenprüfsummenelemente korrigiert werden. Diese Korrektur wird parallel durchgeführt, indem das ungewichtete Syndrom mit dem Gewicht für diese Spalte und für dieses Prüfsummenschema multipliziert und zum Prüfsummenelement addiert wird. Die Korrektur der Spaltenprüfsummen erfolgt durch einfaches Überschreiben der neu berechneten Prüfsummen in den lokalen Registern, da diese nach Aufruf dieser Funktion in die Matrix $\mathbf{C_{fc}}$ geschrieben werden. Der gesamte Algorithmus des Korrekturkernels mit erweiterter Lokalisierung ist in Algorithmus A.18 dargestellt.

6.4.5. Kernel zur Matrixmultiplikation mit Fehlerinjektion

Für die Durchführung von Experimenten mit Fehlerinjektion in der Matrixmultiplikation wurde ein Kernel implementiert, der an einen Multiplikationskernel aus der Literatur [48] angelehnt ist. Dieser Algorithmus ist in seiner einfachsten Version in A.19 dargestellt. Die Änderungen in den weiterentwickelten Versionen von [48] betreffen lediglich Speicheroperationen, eine grundlegende Änderung am Berechnungsteil des Algorithmus wird nicht vorgenommen. Bei diesem Algorithmus tritt die Addition von Daten an zwei Stellen auf: Bei der Akkumulation in der inneren Schleife, sowie beim Zusammenführen am Ende des Kernels. Die Multiplikation von Daten erfolgt nur in der inneren Schleife. Für die Implementierung der Fehlerinjektion kann die bitweise XOR-Operation ($\oplus$) verwendet werden. Dazu wird ein Fehlervektor *errorVec* definiert, der als Maske für die zu ändernden Bits dient. Bei der bitweisen XOR-Operation mit einem Datenwort *dataVec* der gleichen Länge werden die Werte des Bitvektors an den Positionen umgekehrt, an denen *errorVec* eine 1 enthält.

$$\begin{aligned} dataVec &= 0111101011000 \\ \oplus\, errorVec &= 0100000000001 \\ \hline result &= 0011101011001 \end{aligned}$$

Der angepasste Algorithmus für die Matrixmultiplikation mit Bit-Flip Fehlerinjektion ist in Algorithmus A.20 dargestellt. Diesem Kernel werden dabei folgende globale Parameter übergeben:

- *ProzessorID* des fehlerhaften Prozessors: Wählt den Prozessor, bei dessen Berechnung Fehler injiziert werden sollen.
- *Fehlertyp*: Wählt die Operation, bei der ein Fehler injiziert wird. Über diese Auswahl wird festgelegt, ob die Fehlerinjektion nach der Addition in der inneren Schleife, nach der Addition zum entsprechenden Block der Matrix C oder nach der Multiplikation in der inneren Schleife erfolgen soll.
- *ModulID*: Über diesen Parameter wird ausgewählt, welcher der $RX \cdot RY$ Addierer bzw. Multiplizierer betroffen sein soll.
- *Fehlervektor*: Maske als Bitvektor

7. Experimente

7.1. Versuchsaufbau

7.1.1. Verwendete Hardware

Die hier vorgestellten Ergebnisse wurden auf folgenden Rechnersystemen ermittelt:

- Compute Server A: 2x Intel Xeon E5-2650 mit 256 GB Ram und 2x NVIDIA Tesla K20c, 2x NVIDIA GTX480
- Compute Server B: 2x Intel Xeon X5680 mit 96 GB Ram und 4x NVIDIA Tesla C2070

Die NVIDIA Tesla K20c GPUs sind bestückt mit einem Kepler GK110 Grafikprozessor mit 2496 CUDA Kernen, sowie 5 GB DDR5 Speicher. Diese Karten haben eine maximale Rechenkapazität von 1.17 TFLOPS für Rechnungen in IEEE 754 *double precision*. Die NVIDIA Tesla C2070 und GTX480 GPUs sind Karten der Fermi Architektur. Die C2070 besitzt 448 CUDA Kerne und 6 GB DDR5 Speicher, die GTX480 besitzt ebenfalls 448 CUDA Kerne bei 2 GB DDR5 Speicher.

7.1.2. Generierung der Testdaten

Für die Experimente zur Qualitätsmessung und für die Fehlerinjektion wurden drei verschiedene Testreihen von Daten eingesetzt. Die ersten beiden Testreihen umfassen Matrizen mit gleichverteilten Zufallszahlen in einem vorgegebenen Bereich.

- Für die erste Testreihe T_{pos} wurden Matrizen mit gleichverteilten Zufallszahlen aus dem Bereich $[0, 10^i]$ mit $i \in \{0, \ldots, 5\}$ generiert.
- Die zweite Testreihe T_{full} umfasst einen Bereich, der auch negative Zahlen beinhaltet. Die Zufallszahlen wurden für diese Testreihe in einem Bereich von $[-10^i, 10^i]$ ebenfalls mit $i \in \{0, \ldots, 5\}$ erzeugt.
- Die dritte Testreihe T_{orth} benutzt randomisierte orthogonale Matrizen und generiert Testmatrizen basierend auf zwei randomisierten orthogonale Matrizen $\mathbf{U}$ und $\mathbf{V}$ nach der Regel [35]:

$$\mathbf{A} = 10^{\alpha} \cdot \mathbf{U} \cdot \mathbf{D}_{\kappa} \cdot \mathbf{V}^{\mathbf{T}}. \tag{7.1}$$

D_κ ist dabei eine Matrix mit gleichverteilten Zufallswerten in der Diagonalen so skaliert, dass der kleinste Eintrag $1/\kappa$, der größte Eintrag κ ist. Diese Testreihe weist den größten Dynamikbereich der betrachteten Testreihen auf. Der Parameter α wurde als globaler Skalierungsfaktor nicht variiert, für κ wurden folgende Werte verwendet: $\kappa \in \{2, 16, 128, 1024, 8192, 65526\}$. Als Matrixgrößen wurden quadratische Matrizen der Größe $n \times n$ mit $n \in \{512, 1024, 2048, 3072, 4096, 5120, 6144, 7168, 8192\}$ verwendet. Die Messungen wurden für ABFT-Blockgrößen 32, 64, 128 und 256 Elementen durchgeführt.

7.1.3. Betrachtete Verfahren

Bei den durchgeführten Experimenten wurden die Methode zur Fehlerschwellwertbestimmung nach der vereinfachten Fehleranalyse [36] (SEA), sowie verschiedene Versionen der probabilistischen Methode zur Fehlerschwellwertbestimmung (PEA) betrachtet. Die Varianten der probabilistischen Methode umfassen die Bestimmung des Fehlerschwellwerts durch die lokale Durchschnittsmetrik (PEA LOCAL), die Bestimmung des Fehlerschwellwerts durch die globale Durchschnittsmetrik (PEA GLOBAL) und die Bestimmung des Fehlerschwellwerts durch die Betrachtung der zwei, beziehungsweise acht betragsmäßig größten Elemente pro Vektor (PEA 2 und PEA 8). Das Verfahren nach Gunnels und Katz [33] wurde nicht als Vergleichsverfahren herangezogen. Dies begründet sich damit, dass bei diesem Verfahren keine Fehlerschwellwerte für einzelne Prüfsummenelemente erzeugt werden können. Der auf der Norm des Differenzvektors des durch die lineare Operation entstandenen Prüfsummenvektors und des Referenzprüfsummenvektors basierende Fehlerschwellwert bezieht sich auf einen gesamten ABFT-Block einer Matrix.

7.2. Messungen zur Qualität der Fehlerschwellwerte

Bei den Messungen zur Beurteilung der Qualität der Fehlerschwellwerte der eingesetzten Verfahren wurde eine Vielzahl von Matrixmultiplikationen durchgeführt und die Fehlerschwellwerte für die Prüfsummen der Ergebnismatrizen analysiert. Zum Vergleich wurden mit einer Multipräzisionsbibliothek für Gleitkommaarithmetik [49] die Prüfsummenelemente der Ergebnismatrix in 100-Bit Genauigkeit berechnet und als Referenz benutzt, um den absoluten Rundungsfehler für die Prüfsummenelemente zu bestimmen. Der absolute Rundungsfehler ergibt sich aus der Differenz des Werts in Maschinengenauigkeit (IEEE 754 *double precision*) $value_{fp}$ und dem in höherer Genauigkeit berechneten Wert $value_{mp}$:

$$err_{absolute} = |value_{fp} - value_{mp}|. \tag{7.2}$$

Im Folgenden werden die Ergebnisse der drei Testreihen T_{pos}, T_{full} und T_{orth} für die Beurteilung der Qualität der eingesetzten Verfahren dargestellt.

Für die Matrizen mit gleichverteilten Zufallszahlen im Bereich $[-1, 1]$ aus der Testreihe T_{full} ist der durchschnittliche absolute Rundungsfehler und die durchschnittlichen Fehlerschwellwerte der verschiedenen Verfahren für eine ABFT-Blockgröße von 32 in Tabelle 7.1 aufgeführt.

Matrixgröße $n \times n$	Rundungs-fehler	Fehler-schwellwert SEA	Fehler-schwellwert PEA LOCAL	Fehler-schwellwert PEA GLOBAL	Fehler-schwellwert PEA 2	Fehler-schwellwert PEA 8
512	2.27×10^{-14}	8.05×10^{-10}	1.38×10^{-11}	1.57×10^{-11}	1.67×10^{-11}	1.65×10^{-11}
1024	4.54×10^{-14}	3.07×10^{-9}	4.14×10^{-11}	4.80×10^{-11}	4.94×10^{-11}	4.91×10^{-11}
2048	9.10×10^{-14}	1.20×10^{-8}	1.23×10^{-10}	1.44×10^{-10}	1.46×10^{-10}	1.45×10^{-10}
3072	1.36×10^{-13}	2.67×10^{-8}	2.35×10^{-10}	2.78×10^{-10}	2.79×10^{-10}	2.78×10^{-10}
4096	1.81×10^{-13}	4.73×10^{-8}	3.65×10^{-10}	4.27×10^{-10}	4.28×10^{-10}	4.27×10^{-10}
5120	2.26×10^{-13}	7.38×10^{-8}	5.22×10^{-10}	6.14×10^{-10}	6.15×10^{-10}	6.14×10^{-10}
6144	2.71×10^{-13}	1.06×10^{-7}	6.95×10^{-10}	8.15×10^{-10}	8.15×10^{-10}	8.15×10^{-10}
7168	3.17×10^{-13}	1.44×10^{-7}	8.87×10^{-10}	1.04×10^{-9}	1.04×10^{-9}	1.04×10^{-9}
8192	3.62×10^{-13}	1.88×10^{-7}	1.10×10^{-9}	1.29×10^{-9}	1.29×10^{-9}	1.29×10^{-9}

Tabelle 7.1.: Durchschnittlicher absoluter Rundungsfehler und durchschnittliche absolute Fehlerschwellwerte bei einer ABFT-Blockgröße 32 und Matrizen aus der Testreihe T_{full} mit gleichverteilten Zufallszahlen aus dem Wertebereich $[-1, 1]$.

Die Fehlerschwellwerte, die durch die probabilistischen Verfahren PEA LOCAL, PEA GLOBAL, PEA 2 und PEA 8 ermittelt wurden, sind um etwa zwei Magnituden näher am gemessenenen Rundungsfehler als die Fehlerschwellwerte, die nach der vereinfachten Fehleranalyse (SEA) ermittelt wurden. Sowohl Rundungsfehler, als auch die Fehlerschwellwerte nehmen mit steigender Matrixgröße größere durchschnittliche Werte an. Die Verwendung der lokalen Durchschnittsmetrik (PEA LOCAL) erzielt hierbei die Fehlerschwellwerte, die den geringsten Abstand zum gemessenen Rundungsfehler zeigen. Bei Verwendung der globalen Durchschnittsmetrik und den betragsmäßig größten Elementen kann ab einer Matrixgröße von 6144 kein Unterschied der durchschnittlichen Fehlerschwellwerte festgestellt werden.

Um den Rundungsfehler und die Fehlerschwellwerte für verschiedene Datensätze, sowie Datensätze mit einer größeren Dynamik im Wertebereich in einer weiteren Analyse vergleichen zu können, werden im Folgenden relative Werte betrachtet. Dazu wird der gemessene absolute Rundungsfehler err_{abs} auf den absoluten Wert des Prüfsummenelements $|value|$ bezogen um so den relativen Rundungsfehler zu ermitteln:

$$err_{rel} = err_{abs} / |value|. \tag{7.3}$$

Ebenso werden die Fehlerschwellwerte auf den absoluten Wert des Prüfsummenelements bezogen:

$$\epsilon_{x,rel} = \epsilon_{x,abs} / |value|. \tag{7.4}$$

Die Tabelle 7.2 zeigt den durchschnittlichen relativen Rundungsfehler und die durchschnittlichen relativen Fehlerschwellwerte für die Testreihe mit dem größten Dynamikbereich T_{orth} für die Parameter $\alpha = 0$ und $\kappa = 65536$. Die durchschnittliche Fehlerschwellwert der probabilistischen Methode sind für alle Matrixgrößen um etwa zwei Magnituden näher am

gemessenen Rundungsfehler als der durchschnittliche Fehlerschwellwert der vereinfachten Fehleranalyse.

Matrixgröße $n \times n$	Rundungs-fehler	Fehler-schwellwert SEA	Fehler-schwellwert PEA LOCAL	Fehler-schwellwert PEA GLOBAL	Fehler-schwellwert PEA 2	Fehler-schwellwert PEA 8
512	2.62×10^{-15}	1.45×10^{-10}	3.22×10^{-12}	4.20×10^{-12}	5.28×10^{-12}	4.41×10^{-12}
1024	6.32×10^{-15}	6.71×10^{-10}	1.19×10^{-11}	1.60×10^{-11}	1.95×10^{-11}	1.69×10^{-11}
2048	1.89×10^{-14}	8.00×10^{-9}	1.38×10^{-10}	1.56×10^{-10}	1.81×10^{-10}	1.66×10^{-10}
3072	1.56×10^{-14}	4.20×10^{-9}	5.33×10^{-11}	9.11×10^{-11}	9.13×10^{-11}	8.04×10^{-11}
4096	1.06×10^{-14}	4.24×10^{-9}	4.73×10^{-11}	8.72×10^{-11}	8.50×10^{-11}	7.54×10^{-11}
5120	1.61×10^{-14}	9.61×10^{-9}	1.04×10^{-10}	1.77×10^{-10}	1.73×10^{-10}	1.55×10^{-10}
6144	2.27×10^{-13}	2.47×10^{-7}	2.19×10^{-9}	4.32×10^{-9}	4.22×10^{-9}	3.67×10^{-9}
7168	1.92×10^{-14}	1.23×10^{-8}	1.12×10^{-10}	2.08×10^{-10}	2.02×10^{-10}	1.81×10^{-10}
8192	1.59×10^{-14}	1.36×10^{-8}	1.17×10^{-10}	2.16×10^{-10}	2.11×10^{-10}	1.88×10^{-10}

Tabelle 7.2.: Durchschnittlicher relativer Rundungsfehler und durchschnittliche relative Fehlerschwellwerte bei einer ABFT-Bockgröße von 32 und Matrizen aus der Testreihe T_{orth} mit $\alpha = 0$, $\kappa = 2$.

Hierbei kann ein Abstand zwischen durchschnittlichem relativen Rundungsfehler und den relativen Fehlerschwellwerten von etwa 10^3 bis 10^4 für das probabilistische Verfahren, ein Abstand von etwa 10^5 bis 10^6 für die vereinfachte Fehleranalyse festgestellt werden. Dieses Verhalten findet sich in allen betrachteten Wertebereichen der Testreihen T_{full} und T_{orth} wieder. Eine Ausnahme stellt die Testreihe T_{pos} dar. Bei dieser Testreihe sind alle Fehlerschwellwerte im Durchschnitt um einen Faktor 10^2 näher am Rundungsfehler gelegen. Innerhalb der Varianten des probabilistischen Verfahrens können durch die Verwendung der lokalen Durchschnittsmetrik um fast einen Faktor 2 näher am Rundungsfehler gelegene Werte gemessen werden. Auch die Erhöhung der betrachteten betragsmäßig größten Elemente führt zu einem konstant besseren Fehlerschwellwert, der Zugewinn fällt jedoch nicht so groß aus wie bei Verwendung der lokalen Durchschnittsmetrik. Ab einer Matrixgröße von 4096 führt die Verwendung der globalen Durchschnittsmetrik zu höheren durchschnittlichen Fehlerschwellwerten als die Betrachtung der betragsmäßig größten Elemente.

7.2.1. Verteilung der Fehlerschwellwerte

Für eine genauere Untersuchung der Verteilung der Qualität der Fehlerschwellwerte wird im Folgenden die Relation zwischen Fehlerschwellwert $\epsilon_{x,abs}$ und Rundungsfehler err_{abs} betrachtet. Der dabei betrachtete Faktor $q = \epsilon_{x,abs} / err_{abs}$ ist unabhängig von der Größe des Prüfsummenelements:

$$q = \frac{\epsilon_{x,rel}}{err_{rel}} = \frac{\epsilon_{x,abs}/|val|}{err_{abs}/|val|} = \frac{\epsilon_{x,abs}}{err_{abs}}. \tag{7.5}$$

Ein Faktor $q > 1$ bedeutet, dass der Fehlerschwellwert über dem Rundungsfehler liegt. Je größer der Faktor q ist, desto größer ist der Abstand zwischen dem Rundungsfehler und dem Fehlerschwellwert. Treten Werte für q kleiner als 1 auf, so liegt der Fehlerschwellwert unter dem Rundungsfehler und es kommt mit einer hohen Wahrscheinlichkeit zu *False Positives*. Zur Untersuchung der Verteilung des Auftretens von Faktoren $q = \epsilon_{x,abs}/err_{abs}$ wird im Folgenden ein Histogramm verwendet, bei dem die für die Testdaten gemessene Faktoren in Blöcke eingeteilt werden. Der Bereich, in dem sich dieser Faktor bewegt, wird in 23 Blöcke mit exponentiell größer werdender Blockgröße unterteilt. Dadurch lässt sich ein weiter Bereich von Faktoren betrachten und interessantere Bereiche, da näher am Rundungsfehler gelegen, können mit höherer Auflösung betrachtet werden. In den ersten Block werden Faktoren mit $2^0 \leq q < 2^1$ gezählt, in den zweiten Block Faktoren mit $2^1 \leq q < 2^2$ usw. In den letzten Block fallen Faktoren zwischen Fehlerschwellwert und Rundungsfehler mit $2^{22} \leq q < 2^{23}$.

Abbildung 7.1 zeigt dieses Histogramm für die Testreihe T_{orth} über alle untersuchten Wertebereiche und alle ABFT-Blockgrößen. Hierbei kann festgestellt werden, dass für die probabilistische Methode zur Fehlerschwellwertbestimmung die Faktoren zwischen Fehlerschwellwert und Rundungsfehler mit der größten Auftrittshäufigkeit bei Faktoren zwischen 2^{11} und 2^{12} liegt. Für die bei der vereinfachten Fehleranalyse ermittelten Schwellwerte sind die am häufigsten auftretende Faktoren im Bereich zwischen 2^{18} und 2^{19} zu finden. Jeweils rund 20% der Faktoren liegen in diesen Bereichen. Eine weitere Beobachtung lässt sich bezüglich

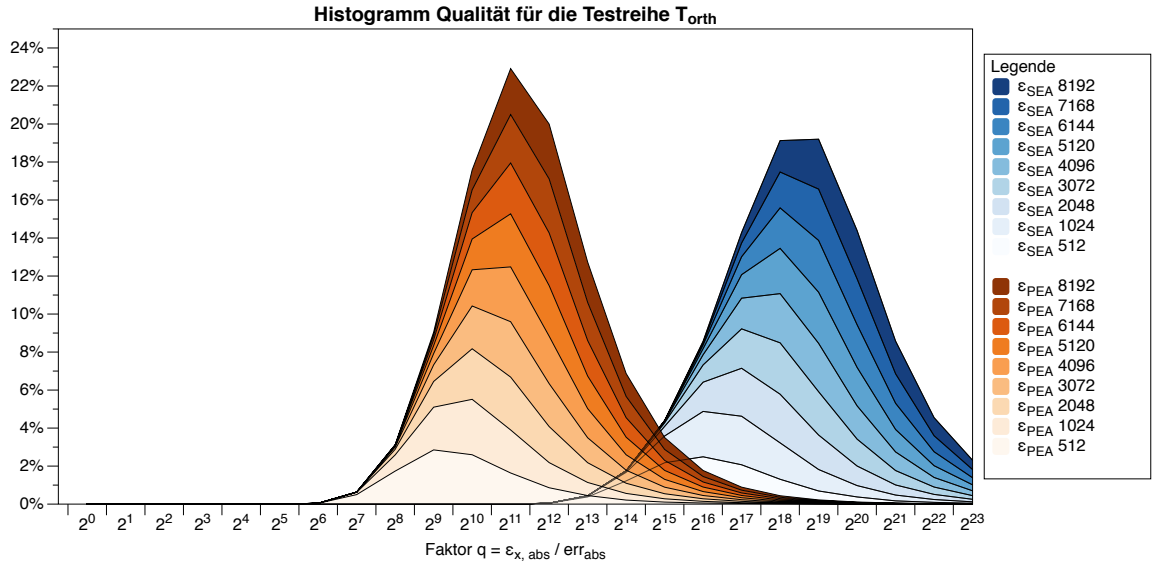


Abbildung 7.1.: Histogramm der Qualität der Fehlerschwellwerte für alle Wertebereiche von T_{orth} und alle ABFT-Blockgrößen durch Darstellung des Faktors $q = \epsilon_{x,abs}/err_{abs}$ aufgeschlüsselt nach verschiedenen Matrixgrößen.

der Abhängigkeit der Qualität der Fehlerschwellwerte von der Matrixgröße feststellen: Mit steigender Matrixgröße verschiebt sich der Punkt der größten Auftrittshäufigkeit für das probabilistische Verfahren um einen Faktor 4 von 2^9 zu 2^{11}, für die vereinfachte Fehleranalyse

fast um einen Faktor 8 von 2^{16} zu 2^{19}. Eine weitere Eigenschaft der Verfahren lässt sich an diesem Histogramm ablesen: Der Bereich, in dem sich der Großteil der Faktoren zwischen Fehlerschwellwert und Rundungsfehler bewegt, ist für das probabilistische Verfahren der Bereich zwischen 2^4 und 2^{15} und für die vereinfachte Fehleranalyse der Bereich zwischen 2^{15} und 2^{23}. Die Breite dieser Bereiche ist für das probabilistische Verfahren $2^{15} - 2^4 = 3.58 \times 10^4$, für die vereinfachte Fehleranalyse $2^{23} - 2^{15} = 8.36 \times 10^6$. Der Großteil der Faktoren fällt für das probabilistische Verfahren in einen enger gefassten Bereich. Für die Testreihe T_{full} ergibt sich ein nahezu identisches Bild für die Verteilung der Faktoren zwischen Fehlerschwellwert und Rundungsfehler. Bei Betrachtung der Testreihe T_{pos} ergibt sich das in Abbildung 7.2 dargestellte Histogramm.

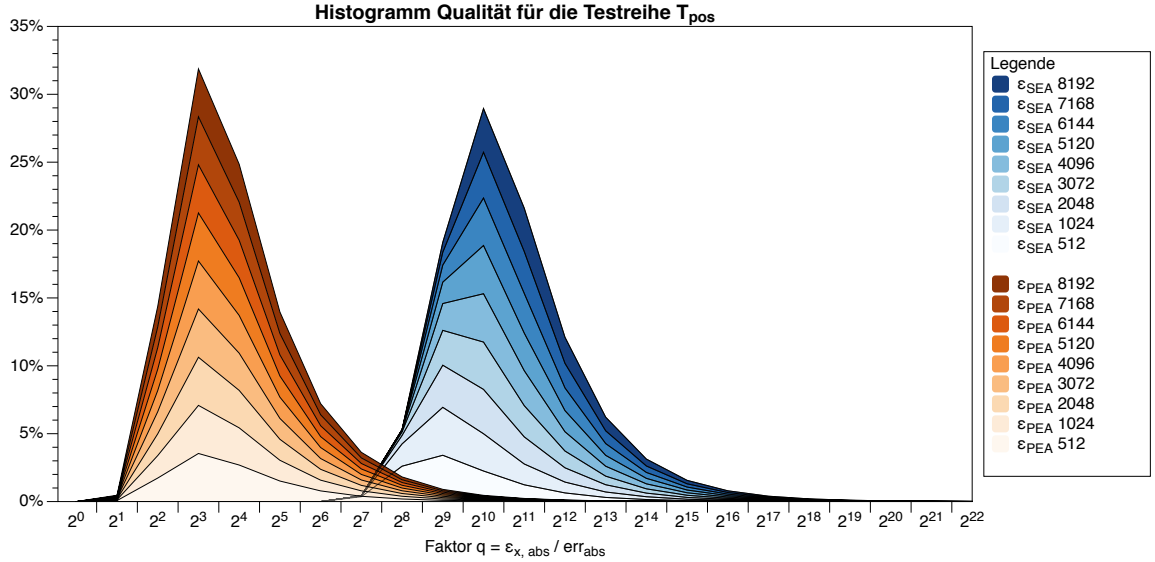


Abbildung 7.2.: Histogramm der Qualität der Fehlerschwellwerte für alle Wertebereiche von T_{pos} und alle ABFT-Blockgrößen durch Darstellung des Faktors $q = \epsilon_{x,abs} / err_{abs}$ aufgeschlüsselt nach verschiedenen Matrixgrößen.

Für diese recht homogene Testreihe sind die Bereiche, in denen Faktoren auftreten enger gefasst, zeigen eine weniger starke Abhängigkeit von der Matrixgröße und es werden insgesamt kleinere Faktoren gemessen.

7.2.2. Variation der ABFT-Blockgröße

Abbildung 7.3 stellt die Ergebnisse bei Variation der ABFT-Blockgröße exemplarisch für den Wertebereich $[-1, 1]$ dar. Dabei zeigt sich für das probabilistische Verfahren und die vereinfachte Fehleranalyse ein unterschiedliches Verhalten. Während der relative Fehlerschwellwert der vereinfachten Fehleranalyse bei zunehmender ABFT-Blockgröße ansteigt, folgt der relative Fehlerschwellwert des probabilistischen Verfahrens dem Verlauf des Run-

dungsfehlers. Sehr gut lässt sich dies bei einer Matrixgröße von 3072 beobachten. Diese Eigenschaft findet sich auch bei den Testreihen T_{orth} und T_{pos} wieder.

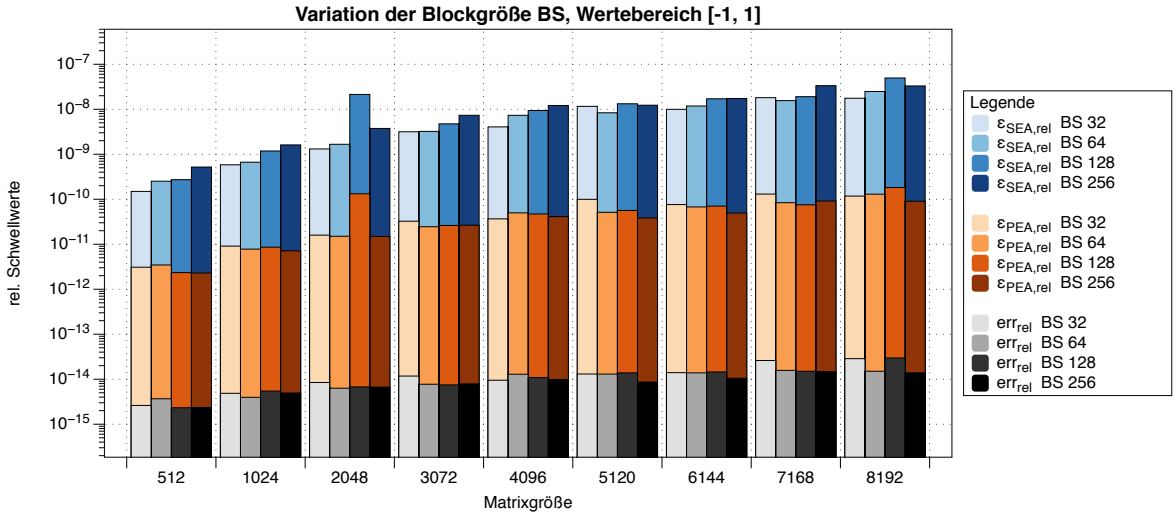


Abbildung 7.3.: Variation der ABFT-Blockgröße für die Testreihe T_{full}.

7.3. Fehlererkennung bei Fehlerinjektion

Für die Messungen zur Fehlererkennung bei Fehlerinjektion wurden pro Parameterkombination mehr als 10^4 Fehlerinjektionen durchgeführt. Dabei wurde der betroffene Prozessor, das betroffene Modul, der Zeitpunkt der Fehlerinjektion (k-ter Schleifendurchlauf), die Operation (*Multiplikation Innere Schleife*, *Addition Innere Schleife* oder *Addition Akkumulation*) und die Bitposition des zu ändernden Bits in der Mantisse randomisiert bestimmt. Bei Fehlerinjektion in Vorzeichen- und Exponentenbits wurden alle injizierten Fehler von allen eingesetzten Verfahren erkannt, weshalb von einer Betrachtung dieser Fehlerinjektionen im Folgenden abgesehen wird. Jede Fehlerinjektion wurde für alle Varianten der probabilistischen Methode (PEA LOCAL, PEA GLOBAL, PEA 2 und PEA 8), sowie für das Verfahren der vereinfachten Fehleranalyse (SEA) durchgeführt. Für jedes Verfahren wurde die Rückgabe des Korrekturkernels analysiert, um die Fehlererkennung in drei verschiedene Klassen einordnen zu können:

- Fehler nicht detektiert: Durch die verwendeten Fehlerschwellwerte wurde kein Fehler detektiert. Da in jeder Matrixmultiplikation ein Fehler injiziert wurde, entspricht dies einem *False Negative*.
- Fehlerhafter ABFT-Block lokalisiert: Es wurden Fehlerschwellwerte überschritten, der Fehler konnte jedoch nicht eindeutig in einem Element lokalisiert werden.

- Fehlerhaftes Element lokalisiert: Die überschrittenen Fehlerschwellwerte erlauben mit dem zugehörigen Korrekturverfahren eine Lokalisierung und damit die Korrektur des fehlerhaften Elements innerhalb eines ABFT-Blocks.

Für jede Fehlerinjektion wurde darüber hinaus die Position des fehlerhaften Elements über die Wahl von Prozessor und Modul berechnet und für dieses Element drei Werte bestimmt, anhand derer eine Einteilung der erkannten und nicht erkannten Fehler in signifikante Fehler erfolgen kann:

- Der absolute Rundungsfehler err_{abs}, der in dem von einem Fehler betroffenen Datenelement auftritt durch Einsatz einer Multipräzisionsbibliothek für Gleitkommaarithmetik.
- Der erwartete Rundungsfehler err_{erw}, der für das von einem Fehler betroffenen Datenelement durch Betrachtung der quadrierten Zwischensummen nach Gleichung 5.18 abgeschätzt wird.
- Der erwartete Rundungsfehler err_{prob}, der für das von einem Fehler betroffene Datenelement durch das probabilistische Verfahren abgeschätzt wird.

Für diese Schranken gilt $err_{abs} \leq err_{erw} \leq err_{prob}$.

7.3.1. Variation der ABFT-Blockgröße

Die Abbildung 7.4 zeigt die Fehlererkennungsraten der vereinfachten Fehleranalyse und des probabilistischen Verfahrens PEA LOCAL exemplarisch für die Matrixgrößen 512, 4096 und 8192 bei variierender ABFT-Blockgröße für die Testreihe T_{orth}, die den größten Dynamikbereich von Werten umfasst. Für die Bestimmung der Erkennungsraten wurden alle injizierten Fehler mit einbezogen, die eine Auswirkung größer als der Rundungsfehler err_{abs} für das vom Fehler betroffene Matrixelement zeigten. Die Darstellung ist aufgeteilt in die drei Operationen, bei denen Fehler injiziert wurden: Der Multiplikation in der inneren Schleife (links), der Addition dieses Ergebnisses auf das Zwischenergebnis in der inneren Schleife (mitte) und der Addition beim Zurückschreiben des Ergebnisses (rechts).

Hierbei ergeben sich für alle drei Operationen höhere Fehlererkennungsraten bei Verwendung des probabilistischen Verfahrens zur Bestimmung der Fehlerschwellwerte (PEA) gegenüber der Verwendung der Fehlerschwellwerte der vereinfachten Fehleranalyse (SEA). Mit steigender Matrixgröße nimmt dabei die Erkennungsrate für beide Verfahren ab. Am stärksten zeigt sich dies bei Betrachtung der Operation *Multiplikation Innere Schleife*, bei der die Erkennungsraten des probabilistischen Verfahrens um rund 10%, die Erkennungsraten der vereinfachten Fehleranalyse um rund 15% abnehmen. Bei Erhöhung der ABFT-Blockgröße von 32 auf 256 Elemente lässt sich für die vereinfachte Fehleranalyse eine Reduktion von rund 9%, für das probabilistische Verfahren eine Reduktion der Erkennungsraten von rund 4% feststellen. Für diese Operation hat zum Einen die Erhöhung der Matrixgröße, zum Anderen die Erhöhung der ABFT-Blockgröße für die vereinfachte Fehleranalyse einen größeren Einfluss auf die Erkennungsraten als für das probabilistische Verfahren. Die Fehlererkennungsraten sind für die beiden Operationen *Addition Innere Schleife* und *Addition Akkumulation* für beide Verfahren höher als für die Operation *Multiplikation Innere Schleife*. Die Differenz in der

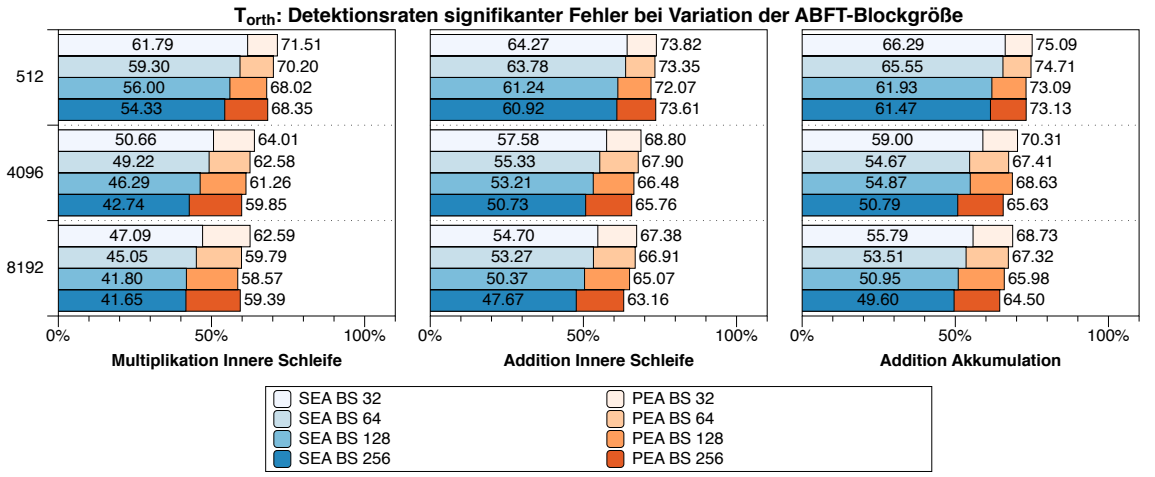


Abbildung 7.4.: Fehlererkennungsraten Testreihe T_{orth} für verschiedene Blockgrößen und Wahl der Schranke err_{abs}.

Anzahl Fehler, die von der vereinfachten Fehleranalyse im Vergleich zum probabilistischen Verfahren für diese Operationen erkannt wird, nimmt auch hier mit steigender Matrixgröße zu. Liegt diese bei einer Matrixgröße von 512 bei rund 10%, werden bei einer Matrixgröße von 8192 rund 14% weniger Fehler durch die vereinfachte Fehleranalyse erkannt. Auch die ABFT-Blockgröße hat Auswirkung auf die Fehlererkennungsrate: Eine Reduktion von rund 4% lässt sich für das probabilistische Verfahren bei Erhöhung der ABFT-Blockgröße von 32 auf 256 verzeichnen, für die vereinfachte Fehlerrate liegt diese Reduktion bei rund 5%. Bei Wahl einer ABFT-Blockgröße von 32 werden bei Verwendung der vereinfachten Fehleranalyse Fehlererkennungsraten zwischen 47% und 67% erreicht, bei Verwendung des probabilistischen Verfahrens werden Fehlererkennungsraten zwischen 62% und 76% erreicht.

7.3.2. Weitere Klassifizierung von Fehlern

Die Klassifizierung von auftretenden Fehlern als signifikante Fehler bei Überschreiten des absoluten Rundungsfehlers err_{abs} stellt eine diskussionswürdige Herangehensweise dar. Zum Einen kann bereits eine Änderung der Ausführungsreihenfolge von kommutativen Operationen eine Änderung im beobachteten Rundungsfehler auswirken, zum Anderen stellt dieser Rundungsfehler eine Grenze dar, der bei Verwendung von *double precision* für die hier betrachteten linearen Operationen sehr klein ist. Selbst für Datensätze mit einem großen Dynamikumfang lag der gemessene relative Rundungsfehler im Bereich von 10^{-14} (vgl. Tabelle 7.2). Bei Wahl einer alternativen Klassifizierung von signifikanten Fehlern ergeben sich für eine ABFT-Blockgröße von 32 die in Abbildung 7.5 dargestellten Fehlererkennungsraten. Bei Verwendung der Klassifizierung über die Abschätzung des Rundungsfehlers über die Betrachtung der quadrierten Zwischensummen err_{erw} werden

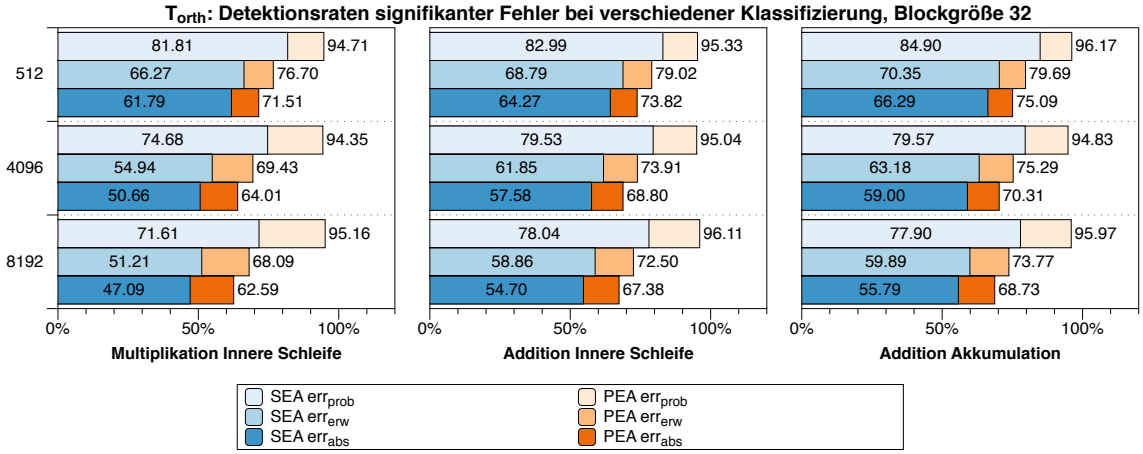


Abbildung 7.5.: Fehlererkennungsraten Testreihe T_{orth} aufgeschlüsselt nach den Schranken err_{abs}, err_{erw} und err_{prob}.

rund 4-5% der zuvor als signifikant eingestuften injizierten Fehler von der Bestimmung der Fehlererkennungsraten ausgeschlossen. Der Bereich der Fehlererkennungsraten erstreckt sich dadurch für die vereinfachte Fehleranalyse von 51% erkannten Fehlern bei einer Matrixgröße von 8192 und der Fehlerinjektion bei der Operation *Multiplikation Innere Schleife* bis 70% erkannten Fehlern bei einer Matrixgröße von 512 und der Fehlerinjektion bei der Operation *Addition Akkumulation*. Für das probabilistische Verfahren werden dabei 68% bis 80% der Fehler erkannt. Wird eine gröbere Abschätzung des Rundungsfehlers zur Klassifizierung von signifikanten Fehlern durch Verwendung einer probabilistischen Abschätzung err_{prob} angewendet, so ergeben sich für das probabilistische Verfahren Fehlererkennungsraten von über 94% und Fehlererkennungsraten zwischen 69% und 71% für die vereinfachte Fehleranalyse.

Für die beiden anderen Testreihen T_{full} und T_{pos} ergeben sich bei Klassifizierung signifikanter Fehler über den absoluten Rundungsfehler err_{abs} und den erwarteten Rundungsfehler err_{erw} höhere Fehlererkennungsraten als bei der Testreihe T_{orth}. Wird der probabilistisch abgeschätzte Rundungsfehler als Schranke zur Klassifizierung verwendet, werden für beide Testreihen weniger als signifikant eingestufte Fehler im Vergleich zur Testreihe T_{orth} erkannt, die Fehlererkennungsrate liegt dennoch bei über 85% für das probabilistische Verfahren, bei über 60% für die vereinfachte Fehleranalyse.

Zur Einordnung dieser Fehlererkennungsraten muss nun die Frage beantwortet werden, welche relativen Fehler denn dadurch maximal und durchschnittlich als nicht signifikant eingestuft werden. Für eine ABFT-Blockgröße von 32 sind die maximalen und durchschnittlichen als insignifikant eingestuften relativen Fehler in Tabelle A.1 für die Testreihe T_{orth} im Anhang dargestellt. Injizierte Fehler ohne Auswirkung wurden von der Bildung der Durchschnittswerte ausgeschlossen. Für die Verwendung der Schranke err_{erw} ergibt sich ein bei Variation der Matrixgröße in etwa konstanter maximaler als insignifikant eingestufter relativer Fehler im Bereich von 10^{-12}, auch der durchschnittliche Wert dieser insignifikant

eingestuften relativen Fehler liegt in etwa konstant bei 10^{-14}. Wird die probabilistische Abschätzung des Rundungsfehlers für die Klassifizierung von signifikanten Fehlern verwendet, so nimmt der durchschnittliche, insignifikant eingestufte relative Fehler mit steigender Matrixgröße zu und liegt im Bereich zwischen 10^{-13} und 10^{-11}. Der maximale auftretende insignifikant eingestufte relative Fehler liegt bei etwa 10^{-7}. Die Maschinengenauigkeit liegt im Vergleich dazu für *Double Precision* bei 1.10×10^{-16}, für *Single Precision* bei 6.00×10^{-8}.

Für die Testreihen T_{full} und T_{pos} sind die insignifikant eingestuften relativen Fehler bei Verwendung von err_{erw} und err_{prob} in den Tabellen A.2 und A.3 im Anhang dargestellt. Der maximale, als insignifikant eingestufte relative Fehler für die Schranke err_{erw} liegt bei der Testreihe T_{full} bei 10^{-11} und durchschnittliche bei 10^{-14}, für die Schranke err_{prob} bei maximal 10^{-8} und abhängig von der Matrixgröße durchschnittlich zwischen 10^{-13} bis 10^{-11}. Bei der Testreihe T_{pos} liegen die insignifikant eingestuften Fehler homogen für beide Schranken maximal und durchschnittlich zwischen 10^{-14} und 10^{-15}.

7.3.3. Varianten des probabilistischen Verfahrens

Bei den vorgestellten Ergebnissen zur Fehlerinjektion wurde bisher nur die Variante über die Verwendung einer lokalen Durchschnittsmetrik für das probabilistische Verfahren präsentiert. Die Abbildung 7.6 zeigt die durchschnittlichen Fehlererkennungsraten aller eingesetzter probabilistischer Methoden bei Verwendung der Schranken err_{abs} und err_{prob} zur Klassifizierung signifikanter Fehler. Hierbei ergeben sich Unterschiede von maximal 3% zwischen den Erkennungsraten der eingesetzten Varianten des probabilistischen Verfahren zur Fehlerschwellwertbestimmung.

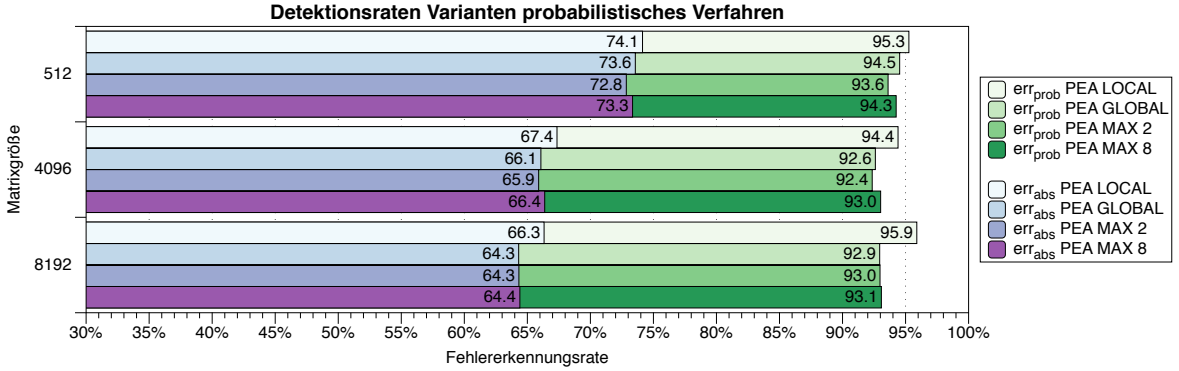


Abbildung 7.6.: Durchschnittliche Fehlererkennungsraten der eingesetzten Varianten des probabilistischen Verfahrens zur Fehlerschwellwertbestimmung für die Testreihe T_{orth} bei einer ABFT-Blockgröße von 32.

Diese Werte sind über alle betrachteten Testreihen, Matrixgrößen, Wertebereiche und ABFT-Blockgrößen konsistent.

7.4. Messungen zur Performanz

Die Messungen zur Performanz der Verfahren wurde auf dem Compute Server A auf einer NVIDIA Tesla K20C durchgeführt. Die hier dargestellten Laufzeiten sind Durchschnittswerte, bei denen für jeden Kernel mindestens 100 Läufe gemessen wurden.

7.4.1. Laufzeiten der Kodierungs- und Korrekturkernel

Die Tabelle 7.3 zeigt die durchschnittlichen Laufzeiten der Kernel zur Kodierung der Matrizen zu einer vollständigen Prüfsummenmatrix, der Matrixmultiplikation bei Verwendung des *Black-Box Schemas* und des Kodierungskernels für eine ABFT-Blockgröße von 32 und Verwendung einer Prüfsumme pro Reihen- und Spaltenvektor eines ABFT-Blocks. Diese Schritte sind unabhängig vom zur Bestimmung des Fehlerschwellwerts eingesetzten Verfahren.

Matrixgröße $n \times n$	FC-Kodierung	Matrixmultiplikation	Korrekturkernel
512	0.13 ms	0.49 ms	0.14 ms
1024	0.48 ms	2.62 ms	0.50 ms
2048	1.82 ms	17.71 ms	1.91 ms
3072	4.06 ms	60.93 ms	4.21 ms
4096	7.20 ms	139.99 ms	7.46 ms
5120	11.23 ms	276.52 ms	11.60 ms
6144	16.57 ms	471.15 ms	17.00 ms
7168	22.04 ms	752.58 ms	22.68 ms
8192	28.80 ms	1115.04 ms	29.72 ms

Tabelle 7.3.: Durchschnittliche Laufzeiten der zur Kodierung, Multiplikation und Überprüfung verwendeten Kernel bei einer ABFT-Blockgröße von 32 und Verwendung einer Prüfsumme.

Die Laufzeiten der Kernel zur Kodierung und zur Überprüfung der Ergebnisse unterscheiden sich nur marginal um maximal eine Millisekunde. Die Laufzeit der Matrixmultiplikation ist für jede Matrixgröße größer als die Summe aus der zur Kodierung und zur Überprüfung benötigte Zeit. Mit steigender Matrixgröße fällt die zur Matrixmultiplikation benötigte Zeit stärker ins Gewicht als die zur Kodierung und Überprüfung benötigte Zeit: Bei einer Matrixgröße von 512 ist die Laufzeit der Matrixmultiplikation um einen Faktor 3,6 größer als die zur Überprüfung benötigte Zeit, bei einer Matrixgröße von 8192 ergibt sich ein Faktor 37,5 zwischen Laufzeit der Matrixmultiplikation und des Korrekturkernels.

In Tabelle 7.4 sind die durchschnittlichen Laufzeiten bei Verwendung von zwei Prüfsummen und ebenfalls einer ABFT-Blockgröße von 32 dargestellt. Durch die Erhöhung der zur Kodierung verwendeten Prüfsummen ergibt sich eine längere Laufzeit der Matrixmultiplikation. Dies begründet sich darauf, dass durch die Erhöhung der Anzahl verwendeter Prüfsummen die Größe der zu multiplizierenden Matrix zunimmt. Auch die Kernel zur Kodierung und

Überprüfung der Prüfsummen zeigen eine längere Laufzeit. Die Differenz im Vergleich zur Verwendung einer Prüfsumme ist dabei für die Überprüfung größer als für die Kodierung. Während bei der Kodierung lediglich mehr *Threads* zur Berechnung der zusätzlichen Prüfsummen verwendet werden, wird für die Überprüfung ein anderer Kernel benötigt. Der zusätzliche Aufwand ergibt sich hierbei durch die Speicherung von Reihenprüfsummen und die Überprüfung von erwarteten Lokalisierungen von Fehlern (siehe Kapitel 6.3.6).

Matrixgröße $n \times n$	FC-Kodierung	Matrixmultiplikation	Korrekturkernel
512	0.14 ms	0.49 ms	0.19 ms
1024	0.51 ms	2.62 ms	0.67 ms
2048	1.93 ms	19.33 ms	2.53 ms
3072	4.31 ms	63.34 ms	5.62 ms
4096	7.65 ms	148.26 ms	9.95 ms
5120	11.96 ms	289.10 ms	15.52 ms
6144	17.22 ms	500.93 ms	22.33 ms
7168	23.46 ms	792.97 ms	30.38 ms
8192	30.62 ms	1181.34 ms	39.64 ms

Tabelle 7.4.: Durchschnittliche Laufzeiten der zur Kodierung, Multiplikation und Überprüfung verwendeten Kernel bei einer ABFT-Blockgröße von 32 und Verwendung zweier Prüfsummen.

Bei Variation der ABFT-Blockgröße ergeben sich für eine Matrixgröße von 8192 und Verwendung einer Prüfsumme die in Tabelle 7.5 dargestellten Werte, bei Verwendung zweier Prüfsummen die in Tablle 7.6 dargestellten Werte. Hierbei zeigt sich ein größerer Einfluss der ABFT-Blockgröße auf die Laufzeit der Kodierungskernel als auf die Laufzeit der Überprüfungskernel. Dabei kann ein größerer Zugewinn bei Verwendung einer Prüfsumme

ABFT-Blockgröße	FC-Kodierung	Matrixmultiplikation	Korrekturkernel
32	28.80 ms	1115.04 ms	29.72 ms
64	19.43 ms	1079.51 ms	22.59 ms
128	9.43 ms	1063.89 ms	16.05 ms

Tabelle 7.5.: Laufzeiten bei Variation der ABFT-Blockgröße bei Verwendung einer Prüfsumme und einer Matrixgröße von 8192.

festgestellt werden, die Laufzeit der Kodierungskernel wird nahezu um einen Faktor 3, die Laufzeit der Überprüfungskernel um einen Faktor 2 reduziert. Insgesamt wird bei einer ABFT-Blockgröße von 32 eine theoretische Laufzeit von 1173,59 ms, bei einer ABFT-Blockgröße von 128 eine theoretische Laufzeit von 1089,42 ms erreicht. Durch die Erhöhung der ABFT-Blockgröße kann eine Reduktion der Laufzeit um etwa 7% bei Verwendung einer Prüfsumme erreicht werden. Werden zwei Prüfsummen verwendet, kann die theoretische Gesamtlaufzeit um etwa 10% reduziert werden.

ABFT-Blockgröße	FC-Kodierung	Matrixmultiplikation	Korrekturkernel
32	30.62 ms	1181.34 ms	39.64 ms
64	22.14 ms	1115.04 ms	27.99 ms
128	12.96 ms	1079.51 ms	33.56 ms

Tabelle 7.6.: Laufzeiten bei Variation der ABFT-Blockgröße bei Verwendung zweier Prüfsummen und einer Matrixgröße von 8192.

7.4.2. Laufzeiten der Vorverarbeitungs- und Bestimmungskernel

Die durchschnittlichen Laufzeiten der Kernel des kontextunabhängigen Vorverarbeitungsschritts für die vereinfachte Fehleranalyse SEA und die Varianten des probabilistischen Verfahren PEA LOCAL, PEA GLOBA, PEA MAX 2 und PEA MAX 8 sind in Tabelle 7.7 dargestellt. Diese Werte wurden bei einer ABFT-Blockgröße von 32 ermittelt. Die Laufzeiten der Verfahren SEA und PEA MAX 2 zeigen ein sehr ähnliches Laufzeitverhalten bei zunehmender Matrixgröße. Die für diese beiden Verfahren gemessenen Laufzeiten entsprechen etwa den für die Kodierung und Überprüfung gemessenen Laufzeiten. Im Vergleich dazu zeigt die Variante PEA MAX 8 eine bei jeder Matrixgröße längere Laufzeit, die Variante PEA LOCAL eine bei jeder Matrixgröße kürzere Laufzeit. Die Laufzeit der Variante PEA GLOBAL nimmt mit steigender Matrixgröße stärker zu als bei allen anderen betrachteten Verfahren.

Matrixgröße $n \times n$	Vorverarbeitung SEA	Vorverarbeitung PEA LOCAL	Vorverarbeitung PEA GLOBAL	Vorverarbeitung PEA MAX 2	Vorverarbeitung PEA MAX 8
512	0.82 ms	0.16 ms	0.22 ms	0.72 ms	2.18 ms
1024	1.65 ms	0.50 ms	0.65 ms	1.40 ms	4.13 ms
2048	3.30 ms	1.68 ms	2.88 ms	3.28 ms	8.92 ms
3072	5.01 ms	3.72 ms	9.66 ms	4.90 ms	13.32 ms
4096	6.63 ms	6.50 ms	16.09 ms	7.96 ms	19.87 ms
5120	8.38 ms	10.13 ms	48.16 ms	12.15 ms	27.85 ms
6144	10.49 ms	14.49 ms	68.88 ms	14.79 ms	33.57 ms
7168	24.15 ms	19.77 ms	91.51 ms	20.77 ms	44.20 ms
8192	29.06 ms	25.72 ms	117.65 ms	28.32 ms	57.50 ms

Tabelle 7.7.: Durchschnittliche Laufzeit der Kernel zur kontextunabhängigen Vorverarbeitung bei einer ABFT-Blockgröße von 32.

Ein etwas anderes Bild zeichnet sich bei den Kerneln zum kontextabhängigen Bestimmungsschritt ab. Die Ergebnisse dieser Laufzeitmessung sind in Tabelle 7.8 dargestellt. Die vereinfachte Fehleranalyse zeichnet sich durch sehr geringe Laufzeiten von unter einer Millisekunde aus. Vergleichbare Werte werden beim probabilistischen Verfahren durch die Variante PEA MAX 2 erreicht. Bei Erhöhung der Anzahl betrachteter betragsmäßig größter Elemente wird durch die benötigte Kombinationsüberprüfung die Laufzeit um bis zu einem Faktor 5,5 erhöht. Dennoch liegt sie deutlich unter den Laufzeiten der zur Kodierung und

Überprüfung eingesetzten Kerneln. Für die blockbasierten Durchschnittsmetriken PEA LOCAL und PEA GLOBAL wurden Laufzeiten gemessen, die zum Teil deutlich über den für die Matrixmultiplikation benötigten Laufzeiten liegen.

Matrixgröße $n \times n$	Bestimmung SEA	Bestimmung PEA LOCAL	Bestimmung PEA GLOBAL	Bestimmung PEA MAX 2	Bestimmung PEA MAX 8
512	0.02 ms	0.63 ms	0.50 ms	0.03 ms	0.08 ms
1024	0.03 ms	4.25 ms	5.49 ms	0.06 ms	0.22 ms
2048	0.04 ms	31.63 ms	33.68 ms	0.14 ms	0.66 ms
3072	0.06 ms	122.56 ms	110.40 ms	0.27 ms	1.39 ms
4096	0.10 ms	274.48 ms	258.87 ms	0.46 ms	2.41 ms
5120	0.14 ms	596.08 ms	513.90 ms	0.70 ms	3.70 ms
6144	0.19 ms	970.76 ms	923.85 ms	0.98 ms	5.27 ms
7168	0.25 ms	1499.38 ms	1551.30 ms	1.32 ms	7.20 ms
8192	0.33 ms	2184.76 ms	2348.38 ms	1.71 ms	9.34 ms

Tabelle 7.8.: Durchschnittliche Laufzeit der Kernel zur kontextabhängigen Bestimmung der Fehlerschwellwerte bei einer ABFT-Blockgröße von 32.

7.4.3. Datendurchsatzraten bei der gesamten ABFT-Matrixmultiplikation

In Tabelle 7.9 sind die durchschnittlichen Datendurchsatzraten bei Einsatz der unterschiedlichen Verfahren zur Fehlerschwellwertbestimmung dargestellt. Diese Werte wurden ermittelt, indem zweimal direkt hintereinander die folgenden Multiplikationen ausgeführt wurden: $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$, $\mathbf{C} \cdot \mathbf{B} = \mathbf{A}$ und $\mathbf{C} \cdot \mathbf{A} = \mathbf{B}$. Für die letzten drei Multiplikationen wurde dabei die Zeit gemessen, die für die gesamte ABFT-Multiplikation bestehend aus der Kodierung, der Bestimmung der Fehlerschwellwerte, der Matrixmultiplikation und des Korrekturschritts benötigt wurde. Durch dieses Vorgehen wird zum Einen eine hohe Auslastung der Hardware erzeugt, zum Anderen wird die Hintereinanderausführung der Operationen gefordert. Dadurch wird der Hardware die Möglichkeit gegeben, die in der ABFT-BLAS-Bibliothek integrierte grobgranulare Parallelisierung zu nutzen, um beispielsweise entstehende Latenzen bei Lade- und Schreiboperationen zu verstecken. Aus diesen für die letzten drei Matrixmultiplikationen ermittelten Laufzeiten wurde für jedes Verfahren der Durchschnitt gebildet, auf die n^3 Operationen der Matrixmultiplikation bezogen und somit der Datendurchsatz in Milliarden Gleitkommaoperationen pro Sekunde (GFLOPS) für die Matrixmultiplikation bestimmt. Zum Vergleich wurde die bei der ungeschützten Matrixmultiplikation (CUBLAS) erreichte Durchsatzrate ebenfalls mit in die Tabelle aufgenommen.

Die beiden Verfahren PEA LOCAL und PEA GLOBAL haben, wie nach Betrachtung der zur Fehlerschwellwertbestimmung benötigten Zeit erwartet, einen großen Abstand zu den anderen Verfahren. Die vereinfachte Fehleranalyse (SEA) und das probabilistische Verfahren bei Betrachtung der zwei betragsmäßig größten Elementen (PEA MAX 2) weisen die höchsten Durchsatzraten der betrachteten ABFT-Methoden auf. Diese beiden Verfahren zeigen dabei

ähnliche Werte, ab einer Matrixgröße von 2048 ergeben sich höhere Werte für die vereinfachte Fehleranalyse, wobei der Abstand mit zunehmender Matrixgröße wieder kleiner wird. Für kleine Matrizen zeigt sich im Vergleich zur ungeschützten Matrixmultiplikation ein großer Abstand, die ABFT-geschützten Verfahren nähern sich jedoch schnell mit zunehmender Matrixgröße an die Durchsatzraten der ungeschützten Matrixmultiplikation an. Ab einer Matrixgröße von 2048 wird für SEA und PEA MAX 2 fast 75% der möglichen Leistung erreicht.

Matrixgröße $n \times n$	GFLOPS SEA	GFLOPS PEA LOCAL	GFLOPS PEA GLOBAL	GFLOPS PEA MAX 2	GFLOPS PEA MAX 8	GFLOPS CUBLAS
512	185.36	100.01	122.21	201.19	95.37	591.94
1024	451.71	106.53	172.36	468.21	276.70	1000.29
2048	751.32	106.89	202.71	739.33	560.48	1022.98
3072	828.85	95.15	205.86	820.16	699.15	1028.33
4096	888.53	107.97	211.17	871.47	774.98	1044.17
5120	903.03	87.98	205.66	881.55	784.24	1041.22
6144	928.57	97.02	206.24	908.66	822.10	1045.70
7168	918.76	103.23	197.78	914.75	847.55	1047.70
8192	934.37	109.02	200.58	931.28	876.33	1048.32

Tabelle 7.9.: Milliarden Gleitkommaoperationen pro Sekunde (GFLOPS) für die unterschiedlichen Verfahren und die ungeschützte Matrixmultiplikation (CUBLAS).

Insgesamt erreichten die ABFT-Methoden fast 90% der Datendurchsatzraten der ungeschützte Matrixmultiplikation.

7.5. Anwendung

Zur Untersuchung der Fehlerauswirkung in einer Anwendung wurde die QR-Zerlegung betrachtet. Dabei werden ausgehend von einer Matrix $\mathbf{A}$ zwei Matrizen $\mathbf{Q}$ und $\mathbf{R}$ berechnet, sodass $\mathbf{Q}$ eine orthogonale Matrix, und $\mathbf{R}$ eine obere Dreiecksmatrix ergibt. Eine Möglichkeit zur Zerlegung ist die Anwendung der *Householder*-Spiegelung. Für eine Größe $n \times n$ der Matrix $\mathbf{A}$ wird dazu eine Folge von *Householder*-Matrizen $\mathbf{H_i}$ erzeugt, sodass

$$\mathbf{Q} = \mathbf{H_1} \cdot \mathbf{H_2} \cdot \cdots \cdot \mathbf{H_{n-1}} \tag{7.6}$$

und

$$\mathbf{R} = \mathbf{H_{n-1}} \cdot \mathbf{H_{n-2}} \cdot \cdots \cdot \mathbf{H_1} \cdot \mathbf{A} \tag{7.7}$$

Dabei wird je eine Matrixmultiplikation zur Bestimmung der Matrix $\mathbf{H_i}$ durchgeführt, für die Berechnung der Matrix $\mathbf{R}$ werden n Matrixmultiplikationen benötigt. Für weitere Informationen wird an dieser Stelle auf [32] verwiesen.

Bei den Experimenten zur Fehlerinjektion innerhalb dieser Anwendung wurden die eingesetzten Verfahren einem ausgiebigen Test unterzogen, dabei wurde in jeder durchgeführten Matrixmultiplikation eine randomisierte Fehlerinjektion in ein Mantissenbit durchgeführt. Nach Berechnung der Matrizen $\mathbf{Q}$ und $\mathbf{R}$ wurde $\mathbf{A_{neu}} = \mathbf{Q} \cdot \mathbf{R}$ berechnet und mit einer Referenzmatrix $\mathbf{A_{ref}}$ verglichen, bei deren Berechnung keine Fehler injiziert wurden. Für die einzelnen Verfahren wurde jeweils die Differenzmatrix $\mathbf{D_x} = \mathbf{A_{ref}} - \mathbf{A_{x,neu}}$ berechnet und über diese Differenzmatrix die Frobenius-Norm mit

$$||\mathbf{A}||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{i,j}|^2}, \tag{7.8}$$

sowie die Zeilensummennorm mit

$$||\mathbf{A}||_\infty = \max_{i=1,\ldots,m}\sum_{j=1}^{n}|a_{i,j}| \tag{7.9}$$

für $m \times n$ Matrizen bestimmt. Als Eingangsdaten wurden dazu Matrizen aus der Testreihe T_{orth} verwendet, die aufgrund ihrer Orthogonalitätseigenschaft in einer QR-Zerlegung eingesetzt werden können. In Tabelle 7.10 werden die durchschnittlichen Frobenius- und Zeilensummennormen der Differenzmatrizen bei Fehlerinjektion bei Verwendung von ungeschützter Matrixmultiplikation (NoABFT), der vereinfachten Fehleranalyse (SEA) und des probablistischen Verfahrens (PEA) dargestellt. Bei dieser ausgedehnten Fehlerinjektion zeigt sich eine deutliche Reduktion des auftretenden Fehlers durch die Verwendung von ABFT im Vergleich zur ungeschützten Matrixmultiplikation. Der über Frobenius- und Zeilensummennorm der Differenzmatrizen gemessene Fehler fällt dabei bei Verwendung von ABFT um einen Faktor von 10^{10} bis 10^{12} geringer aus. Für beide Normen zeigt die vereinfachte Fehleranalyse einen durchschnittlich fast doppelt so großen Fehler wie das probabilistische Verfahren.

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

Tabelle 7.10.: Durchschnittliche Frobenius- und Zeilensummennormen der Differenzmatrizen bei Verwendung ungeschützter Matrixmultiplikation und durch algorithmenbasierte Fehlertoleranz geschützte Matrixmultiplikation.

In Abbildung 7.7 sind die Zeilensummennormen der Differenzmatrizen für je 100 durchgeführte Experimente aufgeführt. Hierbei zeigt sich, dass die probabilistische Methode in jedem Fall zu besseren, mindestens aber zu gleich guten Ergebnissen wie die vereinfachte

Fehleranalyse führt. Mit steigender Matrixgröße reduziert sich die Anzahl Tests, bei denen beide Verfahren zum gleichen Ergebnis geführt haben und es zeichnet sich ein etwa konstanter Abstand um einen Faktor 2 ab.

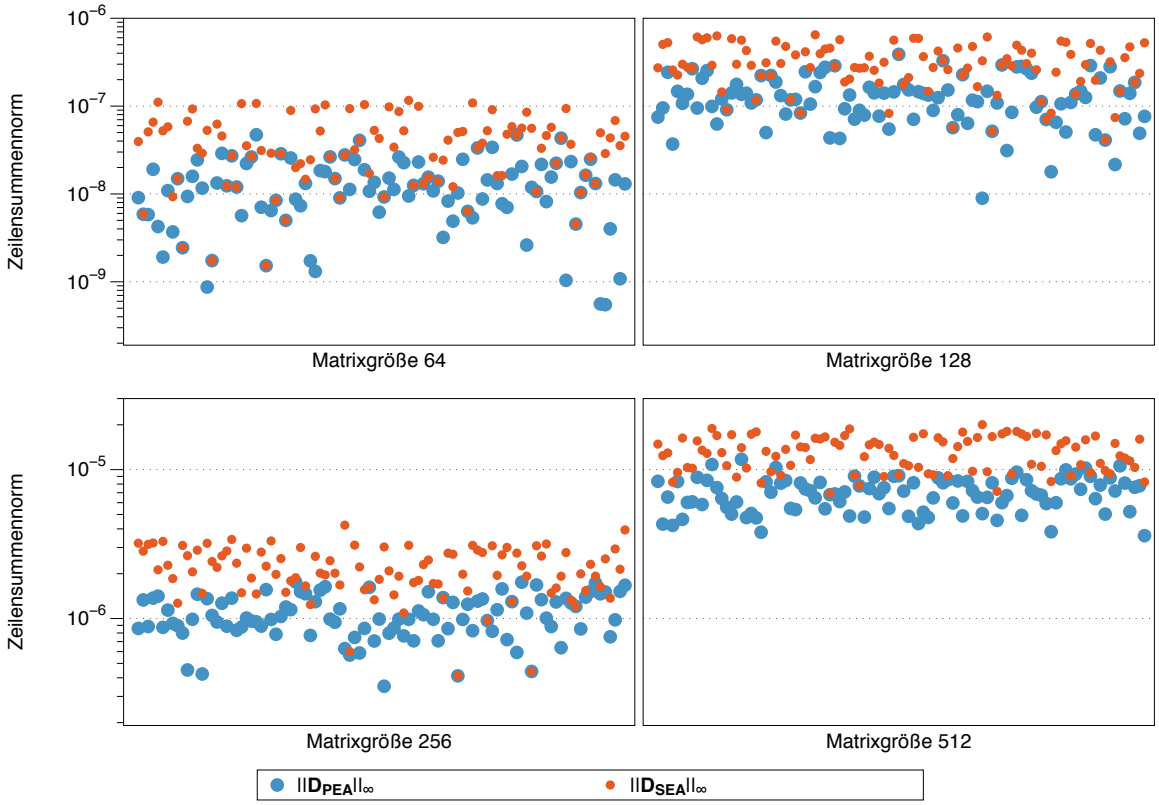


Abbildung 7.7.: Zeilensummennormen der Differenzmatrizen für je 100 Experimente pro Matrixgröße für die Verfahren PEA und SEA bei Fehlerinjektion in der QR-Zerlegung.

Für eine Matrixgröße von 128 ist der in den einzelnen Matrixelementen gemessene Fehler exemplarisch für einen Datensatz in Abbildung 7.8 für die vereinfachte Fehleranalyse (links) und das probabilistische Verfahren (rechts) dargestellt. Je heller dabei die Farbe im Matrixelement ist, um so geringer ist der gemessene Fehler.

Hierbei zeigt sich deutlich die Propagierung von Fehlern über mehrere Matrixelemente hinweg. Der durchschnittliche Fehler ist für die vereinfachte Fehleranalyse höher als für das probabilistische Verfahren, was sich durch eine insgesamt dunklere Matrix abzeichnet. Für das probabilistische Verfahren können zwei mit größerem Fehler behaftete Spalten ausgemacht werden, bei der vereinfachten Fehleranalyse sind diese Spalten ebenfalls sichtbar. Zusätzlich dazu lassen sich für die vereinfachte Fehleranalyse acht weitere mit größerem Fehler behaftete Spalten erkennen.

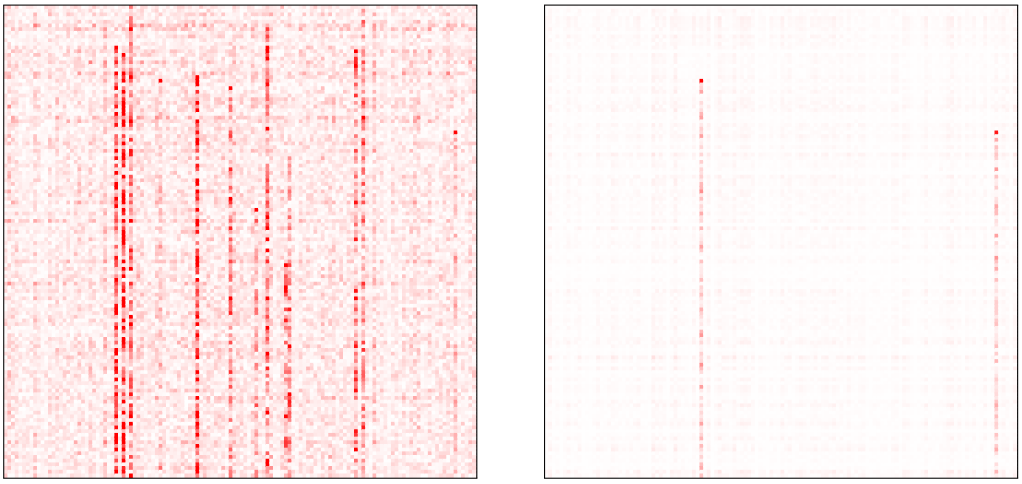


Abbildung 7.8.: Auswirkung der Fehlerinjektionen am Beispiel eines Datensatzes durch Verwendung der vereinfachten Fehleranalyse (links) und des probabilistischen Verfahrens (rechts).

7.6. Diskussion

Im experimentellen Teil dieser Arbeit wurde die Auswirkung von Fehlerinjektionen bei Verwendung von ungeschützter und durch algorithmenbasierte Fehlertoleranz geschützter Matrixmultiplikation auf GPUs untersucht. Der bei der Fehlerinjektion verwendete Multiplikationskernel basiert dabei auf einem für hochoptimierte BLAS-Bibliotheken publizierten Algorithmus [48] und wird vermutlich in einer ähnlichen Form auch in aktuellen Bibliotheken verwendet. Bei den Experimenten wurden mehrere Verfahren zur Bestimmung von Fehlerschwellwerten betrachtet: Die Bestimmung von Fehlerschwellwerten nach der vereinfachten Fehleranalyse und die Bestimmung von Fehlerschwellwerten durch einen probabilistischen Ansatz, für den mehrere Varianten untersucht wurden. Die Verfahren wurden dabei auf die Qualität der Fehlerschwellwerte, der Fehlererkennungsrate bei Fehlerinjektion und die Performanz bei Integration auf der GPU untersucht.

7.6.1. Algorithmenbasierte Fehlertoleranz auf GPUs

Die Untersuchungen zur Laufzeit zeigen, dass sich die algorithmenbasierte Fehlertoleranz für Matrixoperationen effizient auf der GPU implementieren lässt. Durch angepasste Kodierungs- und Korrekturkernel werden besonders bei großen Matrizen Durchsatzraten erreicht, die nur wenig von den Durchsatzraten der ungeschützten Matrixmultiplikation entfernt sind. Insgesamt wird dabei für die gesamte ABFT-geschützte Matrixmultiplikation mit allen zugehörigen Schritten fast 90% der Leistungsfähigkeit der ungeschützten Matrixmultiplikation erreicht. Diese Durchsatzraten sind mit über 931 GFLOPS sehr nahe an der theoretischen Leistungsfähigkeit der Hardware, die mit 1170 GFLOPS angegeben ist. In der Praxis werden meist nur Werte von 1040 GFLOPS erreicht.

7.6.2. Vergleich der eingesetzten Verfahren

Bei Betrachtung des durchschnittlichen relativen Rundungsfehlers und der durchschnittlichen relativen Fehlerschwellwerte ergeben sich für alle Testreihen und Wertebereiche um etwa zwei Magnituden näher am Rundungsfehler gelegene Fehlerschwellwerte für das probabilistische Verfahren gegenüber den Fehlerschwellwerten der vereinfachten Fehleranalyse. Bei einem durchschnittlichen relativen Rundungsfehler von 1.59×10^{-14} liegt der durchschnittliche relative Fehlerschwellwert des probabilistischen Verfahrens beispielsweise bei 2.11×10^{-10}, der relative Fehlerschwellwert der vereinfachten Fehleranalyse bei 1.36×10^{-8}. Dies bedeutet, dass bei Verwendung der Fehlerschwellwerte der vereinfachten Fehleranalyse Fehler nicht detektiert werden können, die durch das probabilistische Verfahren erkannt werden. Bei genauerer Betrachtung der Relation zwischen Rundungsfehler und Fehlerschwellwert anhand von Histogrammen zeigt sich, dass der Großteil der Fehlerschwellwerte des probabilistischen Verfahrens in einem enger gefassten Bereich liegt als die Fehlerschwellwerte der vereinfachten Fehleranalyse. Für die vereinfachte Fehleranalyse ergibt sich aus dieser Beobachtung, dass der Faktor zwischen Rundungsfehler und Fehlerschwellwert eine größere Variation aufweist, mit höherer Wahrscheinlichkeit also über dem als Durchschnitt ermittelten Fehlerschwellwert liegt. Die Fehlerschwellwerte des probabilistischen Verfahrens werden im Umkehrschluss präziser bestimmt als die Fehlerschwellwerte der vereinfachten Fehleranalyse.

Bei Variation der Matrixgröße und der ABFT-Blockgröße kann festgestellt werden, dass die relativen Fehlerschwellwerte der vereinfachten Fehleranalyse sich mit zunehmender Matrix- und ABFT-Blockgröße weiter vom relativen Rundungsfehler entfernen als die Fehlerschwellwerte des probabilistischen Verfahrens. Das probabilistische Verfahren liefert also auch hier präzisere Fehlerschwellwerte über einen weiten Bereich der variierten Parameter als die vereinfachte Fehleranalyse. Diese Eigenschaft spiegelt sich vor allem in den Ergebnissen zur Untersuchung der Auswirkung von Fehlerinjektionen wieder. Für die vereinfachte Fehleranalyse sinken die Fehlererkennungsraten bei Variation der Matrixgröße von 512 auf 8192 um bis zu 14%, für das probabilistische Verfahren nur um 10%. Bei Variation der ABFT-Blockgröße von 32 auf 256 Elemente pro Block werden durch die vereinfachte Fehleranalyse bis zu 9% weniger Fehler, durch das probabilistische Verfahren nur bis zu 4% weniger Fehler erkannt. Die Fehlererkennungsraten liegen dabei je nach Operation, bei der die Fehlerinjektion durchgeführt wurde, bei einer maximalen Matrixgröße für die vereinfachte Fehleranalyse bei 47% bis 56%, für das probabilistische Verfahren bei 63% bis 69%. Diese Erkennungsraten gelten für alle Fehler, die zu einer Auswirkung größer dem gemessenen Rundungsfehler führen. Wird der Bereich akzeptierbarer Fehler weiter gefasst, ergeben sich bei gleichen Parametern Fehlererkennungsraten von 72% bis 77% für die vereinfachte Fehleranalyse und 96% für das probabilistische Verfahren.

Bei Betrachtung der Varianten zur Fehlerschwellwertbestimmung für das probabilistische Verfahren über lokale und globale Durchschnittsmetriken, sowie die betragsmäßig größten Elemente kann kein großer Unterschied für die durchschnittlichen relativen Fehlerschwellwerte festgestellt werden. Auch die Fehlerinjektion zeigt keine großen Änderungen in der Fehlererkennungsrate für die verschiedenen Varianten. Die relativen Fehlerschwellwerte

unterscheiden sich dabei um weniger als einen Faktor 2, für die Fehlerinjektion variiert die Erkennungsrate um maximal 4%. Ein anderes Bild zeichnet sich bei den Messungen zur Performanz der Methoden ab. Die Verwendung von lokalen und globalen Durchschnittsmetriken führt dabei für den kontextabhängigen Bestimmungsschritt zu Laufzeiten, die fast doppelt so lang wie die zur Matrixmultiplikation benötigte Zeit sind. Diese auf Durchschnittsmetriken basierende Varianten des probabilistischen Verfahrens sind somit für eine effiziente ABFT-geschützte Matrixmultiplikation auf der GPU nicht attraktiv, da in der dafür benötigten Zeit prinzipiell auch eine doppelte Berechnung der gesamten Matrixmultiplikation mit anschließendem Vergleich der Elemente durchgeführt werden kann. Zurückzuführen ist dies auf die große, für diese Methode zur Fehlerschwellwertbestimmung benötigte Datenmenge. Für jedes Prüfsummenelement muss dafür pro Block, bestehend aus 32 Elementen der zur Bildung des Skalarprodukts eingesetzten Vektoren, rund 5 Bitvektoren und zugehörige Durchschnittswerte geladen und verglichen werden. Der Transfer dieser Datenmengen in Verbindung mit der großen Anzahl Prüfsummenelemente, für die diese Daten verarbeitet werden müssen, schränkt die Performanz des Bestimmungsschritts für diese Varianten ein. Die Variante der Bestimmung der Fehlerschwellwerte über die betragsmäßig größten Elemente des probabilistischen Verfahrens zeigt dagegen Laufzeiten, die unter den für die Kodierung und Überprüfung benötigten Zeit liegen und somit für eine effiziente ABFT-geschützte Matrixmultiplikation verwendet werden können.

Attraktive Verfahren bezüglich der Laufzeit sind die vereinfachte Fehleranalyse und das probabilistische Verfahren bei Betrachtung der zwei betragsmäßig größten Elemente. Die gemessenen Laufzeiten für den kontextunabhängigen Bestimmungsschritt liegen dabei im Bereich der für die Kodierung benötigten Zeit. Für den kontextabhängigen Bestimmungsschritt ergeben sich für beide Verfahren Laufzeiten von unter 1,7 Millisekunden bei einer Laufzeit der Matrixmultiplikation von über 2,3 Sekunden. Für die gesamte ABFT-Matrixmultiplikation werden dabei bei Verwendung der vereinfachten Fehleranalyse bis zu 934 GFLOPS, bei Verwendung des probabilistischen Verfahrens bis zu 931 GFLOPS erreicht.

Zusammenfassend bietet das probabilistische Verfahren bei Betrachtung der betragsmäßig größten Elemente pro Vektor eine Methode zur Fehlerschwellwertbestimmung, die bessere Ergebnisse bei Fehlerinjektion als die vereinfachte Fehleranalyse aufweist, durch näher am Rundungsfehler gelegene Fehlerschwellwerte überzeugen kann und Durchsatzraten auf der GPU liefert, die nur 10% von den Durchsatzraten einer ungeschützten Matrixmultiplikation auf GPU entfernt sind.

7.6.3. Ausblick

Für die probabilistische Methode zur Abschätzung des Rundungsfehlers besteht weiteres Potential für die Abschätzung des bei der Bildung des Skalarprodukts zweier Vektoren **a** und **b** auftretenden erwarteten Rundungsfehlers. Die ab Gleichung 5.18 durchgeführte Abschätzung der quadrierten Zwischensummen

$$\begin{aligned}\mathtt{Var}_{\mathtt{Sum}}(\Delta s_n) &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} (s_k^*)^2 \\ &\leq \frac{1}{8} \cdot 2^{-2t} \cdot \sum_{k=2}^{n} (k \cdot y)^2\end{aligned}$$

über eine obere Schranke mit $y \geq |a_i \cdot b_i|$ erlaubt es nicht, Ausreißer von der Abschätzung des Rundungsfehlers über die quadrierten Zwischensummen auszuschließen. Ein Extrembeispiel hierfür sind die Vektoren $\mathbf{a} = (10^5, 0, \ldots, 0)$ und $\mathbf{b} = (1, 0, \ldots, 0)$ mit Länge n. Für diese Vektoren nimmt der erwartete Rundungsfehler mit steigender Länge zu, obwohl bei der Bildung des Skalarprodukts nur ein Element ungleich 0 aufsummiert wird.

Des Weiteren wurde der Faktor, der für eine probabilistische Abschätzung definiert mit welcher Wahrscheinlichkeit Elemente in einem Konfidenzintervall zu finden sind, für alle Experimente mit $\omega = 3$ gewählt. Dadurch liegen theoretisch 99.7% aller Proben innerhalb des Konfidenzintervalls. Durch die vereinfachenden Annahmen bei der probabilistischen Abschätzungen des Rundungsfehlers wird der abgeschätzte Wert jedoch erhöht, das Konfidenzintervall wird dadurch entsprechend breiter gefasst, wodurch mehr Proben als eigentlich durch ω bestimmt in das Konfidenzintervall fallen. Die Variation und Anpassung dieses Parameters kann benutzt werden, um noch näher am Rundungsfehler gelegene Fehlerschwellwerte zu erhalten.

8. Zusammenfassung

Ziel dieser Arbeit war es, die Methoden der algorithmenbasierten Fehlertoleranz (ABFT) für grundlegende Operationen der linearen Algebra (BLAS) für den Einsatz auf Grafikprozessoren (GPUs) anzupassen. Das ABFT-Schema stellt dabei strikte Anforderungen, um eine Maskierung von auftretenden Fehlern durch fehlerhafte Prozessoren zu verhindern. Um den Einsatz von ABFT-geschützten Operationen für einen praktischen Einsatz attraktiv zu gestalten, muss das Rechenpotenzial der massiv-parallelen GPUs ausgenutzt werden und die BLAS-Operationen ohne großen Aufwand für den Benutzer in Anwendungen integrierbar sein. Die Verwendung von verschiedenen Architekturen mit physisch getrennten Speicherbereichen und asynchron ausgeführten Operationen macht es dabei erforderlich, dass die Integrität der Daten sichergestellt wird und für einen effizienten Einsatz die kostspielige Kommunikation von Daten zwischen den Architekturen minimiert wird. Die Zuverlässigkeit von ABFT-geschützten Operationen hängt stark davon ab, dass zum Einen die durch das ABFT-Schema gestellten Anforderungen eingehalten werden und zum Anderen alle signifikanten Fehler detektiert und korrigiert werden.

Bei den ABFT-geschützten Matrixoperationen werden zur Detektion und Korrektur von bei der Berechnung aufgetretenen Fehlern Prüfsummen ermittelt und verglichen, die mit einem Rundungsfehler behaftet sind. Die bei exakter Arithmetik durchgeführte Überprüfung auf Gleichheit muss bei Verwendung von Gleitkommaarithmetik deshalb durch den Test ersetzt werden, ob bei Bildung der Differenz beider Werte ein Fehlerschwellwert überschritten wird. Die Qualität dieser Fehlerschwellwerte hat großen Einfluss darauf, welche Fehler in welcher Größenordnung detektiert werden und welcher Mehraufwand durch die Verwendung der algorithmenbasierten Fehlertoleranz entsteht. Werden Fehlerschwellwerte zu hoch gewählt, werden signifikante Fehler nicht erkannt. Liegen die Fehlerschwellwerte unterhalb des Rundungsfehlers, kommt es mit hoher Wahrscheinlichkeit zu *False Positives* und es werden unnötigte Korrekturschritte durchgeführt, die die Performanz des Verfahrens verschlechtern.

In dieser Arbeit wurde eine probabilistische Methode zur Rundungsfehlerabschätzung für die Fehlerschwellwertbestimmung angepasst, weiterentwickelt, implementiert und mit einem deterministischen Referenzverfahren aus der Literatur verglichen. Durch die dabei ermittelten Fehlerschwellwerte konnten mehr als 94% der signifikanten Fehler bei Fehlerinjektion erkannt werden, ohne dass dabei *False Positives* auftraten. Das Vergleichsverfahren erreichte dagegen nur Fehlererkennungsraten zwischen 71% und 84% und zeigte stärkere Abhängigkeiten von den variierten Parametern und der Problemgröße. Die Überlegenheit des probabilistischen Verfahrens wurde durch Qualitätsmessungen bestätigt und zeigte auch in einer Anwendung mit Fehlerinjektion ein besseres Ergebnis.

Für eine performante Integration von ABFT auf GPUs wurde eine ausführliche Untersuchung der für die ABFT publizierten Methoden, sowie eine Betrachtung des Einsatzschemas von BLAS-Operationen durchgeführt. Basierend auf diesen Analysen wurde ein Konzept für eine ABFT-BLAS-Bibliothek entwickelt, bei der durch die partitionierte Kodierung und Verwendung von gewichteten Prüfsummen mehrere Fehler pro ABFT-Block erkannt werden können. Die zur Kodierung, Überprüfung und Korrektur von Matrizen entwickelten Kernel wurden so konzipiert, dass verschiedene ABFT-Blockgrößen und Prüfsummengewichte unterstützt werden und redundante Operationen bei mehreren aufeinanderfolgenden Operationen minimiert werden. Die Verfahren zur Schwellwertbestimmung wurden ebenfalls für den Einsatz auf GPUs angepasst und dabei auftretende Berechnungsschritte so aufgeteilt, dass zum frühest möglichen Zeitpunkt die für die Fehlerschwellwertbestimmung benötigten Informationen extrahiert werden können und so der Aufwand bei der Bestimmung der Fehlerschwellwerte reduziert wird.

Durch dieses Konzept, sowie eine effiziente Implementierung wurde der Mehraufwand minimiert, der durch die Verwendung von ABFT auf GPUs auftritt. Die ABFT-geschützte Matrixmultiplikation zeigte durch diese Optimierungsmaßnahmen bei Messungen zur Performanz eine Datendurchsatzrate von über 930 GFLOPS. Die Datendurchsatzrate der ungeschützten Matrixmultiplikation lag bei 1040 GFLOPS und die ABFT-geschützte Version damit nur rund 10% unter dem maximal erreichbaren Wert. Grundlegende Operationen wie die Matrixmultiplikation wurden bei der Konzeption der ABFT-BLAS-Bibliothek als *Black-Box Modell* aus der hochoptimierten, vom Hersteller bereitgestellten BLAS-Bibliothek verwendet. Dadurch kann auch bei zukünftigen Hardwaregenerationen die ABFT-BLAS-Bibliothek mit weiterhin hoher Performanz eingesetzt werden.

A. Anhang

A.1. Tabellen

Matrixgröße $n \times n$	Schranke err_{erw}		Schranke err_{prob}	
	Maximaler insignifikant eingestufter rel. Fehler	Durchschnittlicher insignifikanter rel. Fehler	Maximaler insignifikant eingestufter rel. Fehler	Durchschnittlicher insignifikanter rel. Fehler
512	3.35×10^{-12}	7.72×10^{-15}	1.29×10^{-10}	4.76×10^{-13}
1024	1.85×10^{-12}	5.92×10^{-15}	5.59×10^{-10}	1.24×10^{-12}
2048	8.90×10^{-13}	6.04×10^{-15}	6.70×10^{-9}	4.48×10^{-12}
3072	2.36×10^{-12}	9.52×10^{-15}	1.12×10^{-8}	6.73×10^{-12}
4096	2.02×10^{-12}	8.71×10^{-15}	5.95×10^{-8}	2.55×10^{-11}
5120	8.32×10^{-13}	7.49×10^{-15}	1.38×10^{-9}	4.78×10^{-12}
6144	4.91×10^{-12}	1.46×10^{-14}	4.67×10^{-9}	6.90×10^{-12}
7168	3.13×10^{-12}	1.18×10^{-14}	1.31×10^{-7}	1.92×10^{-11}
8192	1.63×10^{-12}	9.94×10^{-15}	8.37×10^{-9}	1.44×10^{-11}

Tabelle A.1.: Maximale und durchschnittliche nicht detektierte relativer Fehler bei Verwendung von err_{erw} und err_{prob} zur Klassifizierung signifikanter Fehler für die Testreihe T_{full}.

Matrixgröße $n \times n$	Schranke err_{erw}		Schranke err_{prob}	
	Maximaler insignifikant eingestufter rel. Fehler	Durchschnittlicher insignifikanter rel. Fehler	Maximaler insignifikant eingestufter rel. Fehler	Durchschnittlicher insignifikanter rel. Fehler
512	2.29×10^{-11}	1.21×10^{-14}	6.21×10^{-11}	1.76×10^{-13}
1024	1.64×10^{-12}	4.61×10^{-15}	1.34×10^{-9}	5.61×10^{-13}
2048	2.16×10^{-11}	1.28×10^{-14}	5.44×10^{-10}	7.44×10^{-13}
3072	6.03×10^{-12}	1.03×10^{-14}	2.21×10^{-9}	1.15×10^{-12}
4096	7.74×10^{-13}	6.41×10^{-15}	7.53×10^{-10}	9.65×10^{-13}
5120	1.11×10^{-12}	7.12×10^{-15}	5.71×10^{-10}	1.14×10^{-12}
6144	7.68×10^{-11}	2.91×10^{-14}	5.07×10^{-10}	1.19×10^{-12}
7168	6.54×10^{-12}	1.36×10^{-14}	3.24×10^{-8}	5.66×10^{-12}
8192	2.59×10^{-11}	1.95×10^{-14}	9.69×10^{-8}	1.49×10^{-11}

Tabelle A.2.: Maximale und durchschnittliche nicht detektierte relativer Fehler bei Verwendung von err_{erw} und err_{prob} zur Klassifizierung signifikanter Fehler für die Testreihe T_{full}.

Matrixgröße $n \times n$	Schranke err_{erw}		Schranke err_{prob}	
	Maximaler insignifikant eingestufter rel. Fehler	Durchschnittlicher insignifikanter rel. Fehler	Maximaler insignifikant eingestufter rel. Fehler	Durchschnittlicher insignifikanter rel. Fehler
512	3.13×10^{-15}	8.48×10^{-16}	1.33×10^{-14}	2.67×10^{-15}
1024	4.51×10^{-15}	1.07×10^{-15}	1.60×10^{-14}	3.02×10^{-15}
2048	6.27×10^{-15}	1.42×10^{-15}	2.59×10^{-14}	4.46×10^{-15}
3072	7.57×10^{-15}	1.54×10^{-15}	2.64×10^{-14}	4.88×10^{-15}
4096	7.69×10^{-15}	1.81×10^{-15}	3.10×10^{-14}	5.40×10^{-15}
5120	9.81×10^{-15}	2.04×10^{-15}	3.84×10^{-14}	6.20×10^{-15}
6144	1.02×10^{-14}	2.30×10^{-15}	4.06×10^{-14}	7.05×10^{-15}
7168	1.12×10^{-14}	2.37×10^{-15}	4.51×10^{-14}	7.51×10^{-15}
8192	1.23×10^{-14}	2.53×10^{-15}	5.01×10^{-14}	8.03×10^{-15}

Tabelle A.3.: Maximale und durchschnittliche nicht detektierte relativer Fehler bei Verwendung von err_{erw} und err_{prob} zur Klassifizierung signifikanter Fehler für die Testreihe T_{pos}.

A.2. Algorithmen

Algorithmus A.1: Funktion `shuffleAndCheck()`, die Werte zweier benachbarter *Threads* mittels *Shuffle Instructions* austauscht, diese vergleicht und bei einer Abweichung die Information speichert, in welchem ABFT-Block ein Fehler bei der Überprüfung aufgetreten ist.

```
Function shuffleAndCheck(sum, exception)
    Data: local register temp
    /* read value of register sum of threads calculating the same checksum and store
       into temp. The second argument of the shuffle instruction defines how data
       should be exchanged, in case of two checksums: 0->1, 1->0, 2->3, 3->2.      */
    temp ← cuda_shuffle_xor(sum, 0x00001, 2 · numWCS);
    /* compare values and set flag if not equal                                    */
    if temp ≠ sum then
        exception ← MODULE_BROKEN(blockID);
```

Algorithmus A.2: Kernel `determineVectorNormsRow()` zur Berechnung der Norm für jeden Reihenvektor einer Matrix.

```
Kernel determineVectorNormsRow()
    Input: encoded matrix A_fc
    Output: norm for every data and column checksum vectors in A_fc
    Launch Dimensions: BS_X threads per threadblock
    Launch Dimensions: rows(A_fc) / BS_X threadblocks
    Data: local register sum
    row ← blockIdx.x · BS_X + tid.x;
    if row ≥ rows(A_fc) then
        return;
    /* calculate the norm                                                          */
    sum ← 0;
    for i ← 1 to originialColumns(A_fc) do
        temp ← element of A_fc at position (row, i) ;
        sum ← sum + temp · temp;
    sum ← √sum;
    /* write back result                                                           */
    store norm of row row from sum in global memory;
```

Algorithmus A.3: Kernel `secureEncodingKernel()` zur Kodierung einer Matrix $\mathbf{A}$ zu einer vollständigen Prüfsummenmatrix $\mathbf{A_{fc}}$ mit beliebiger ABFT-Blockgröße und beliebiger Anzahl Gewichte, sowie doppelter Prüfsummenberechnung.

Kernel `secureEncodingKernel()`

Input: matrix $\mathbf{A}$, checksums weights $rowWeights[]$, $colWeights[]$
Output: encoded matrix $\mathbf{A_{fc}}$, feedback about errors

Launch Dimensions: $2 \cdot numWCS \times BS_{ABFT}$ threads per threadblock
Launch Dimensions: one threadblock for each $BS_{ABFT} \times BS_{ABFT}$ submatrix of $\mathbf{A}$

Data: shared $Asub[BS_X][BS_{ABFT}]$, $accum\_row[numWCS][BS_{ABFT}/BS_X][BS_X]$
Data: local registers $sumRow$, $sumCol$, $numStripes$, row, col

$numStripes \leftarrow BS_{ABFT}/BS_X$;
$sumCol \leftarrow 0$;

`/* loop over all stripes of rows with size` BS_X `*/`
for $curStripe \leftarrow 1$ **to** $numStripes$ **do**
  $sumRow \leftarrow 0$;
  $col \leftarrow tid.y$;

  `/* load current stripe to shared memory */`
  for $i \leftarrow 1$ **to** BS_X **do**
    load one element of submatrix of $\mathbf{A}$ into $Asub[i][col]$;

  sync;

  `/* each thread encodes one column throughout all processed stripes */`
  for $i \leftarrow 1$ **to** BS_X **do**
    $row \leftarrow rowOffset + i$;
    $sumCol \leftarrow sumCol + Asub[i][col] \cdot$ colWeight(row, cs);

  `/* encode the row */`
  $row \leftarrow tid.y$ modulo BS_X;
  $colBlock \leftarrow$ roundDown$(tid.y/BS_X)$;
  for $i \leftarrow 1$ **to** BS_X **do**
    $col \leftarrow colBlock \cdot BS_X + i$;
    $sumRow \leftarrow sumRow + Asub[row][col] \cdot$ rowWeight(col, cs);
  $accum\_row[cs][colBlock][row]$;
  sync;

  `/* accumulate to get the row checksum */`
  if $tid.y < BS_X$ **then**
    $sumRow \leftarrow 0$;
    for $i \leftarrow 1$ **to** BS_{ABFT}/BS_X **do**
      $sumRow \leftarrow sumRow + accumRow[cs][i][row]$;

    `/* check calculation with redundant encoding and store checksum */`
    shuffleAndCheck($sumRow$,$exception$);
    write back row checksum $sumRow$ into $\mathbf{A}$;
  $rowOffset \leftarrow rowOffset + BS_X$;

`/* check calculation with redundant encoding and store checksum */`
shuffleAndCheck($sumCol$,$exception$);
write back column checksum $sumCol$ into $\mathbf{A}$;

Algorithmus A.4: Kernel `determineRowEpsilonSEA()` zur Bestimmung des Fehlerschwellwerts nach der vereinfachten Fehleranalyse [36], basierend auf den Normen der Reihen- und Spaltenvektoren der Operanden der Matrixmultiplikation.

Kernel `determineRowEpsilonSEA()`

Input: vector norms of rows of $\mathbf{A_{fc}}$ and columns of $\mathbf{B_{fc}}$
Output: epsilon for all row checksums

Launch Dimensions: $BS_{ABFT} \times BS_Y$ threads per threadblock
Launch Dimensions: colChecksums($\mathbf{C_{fc}}$)$/numWCS \times$ rows($\mathbf{A_{fc}}$)$/BS_Y$ threadblocks

Data: shared $normsB[BS_{ABFT}]$
Data: local register *norm_a*, *norm_s*, *sum_norm_b*, *result*, *n*

$col \leftarrow BS_{ABFT} \cdot blockDim.x + tid.x$;
$row \leftarrow BS_{ABFT} \cdot (blockIdx.y \cdot BS_Y + tid.y) + tid.x$;
$n \leftarrow$ originalColumns($\mathbf{A_{fc}}$);

`/* load norms of` BS_{ABFT} `column vectors of entire ABFT-block of` $\mathbf{B_{fc}}$ `to shared memory and the norm of the row vector to local register */`
load norm of column vector *col* of $\mathbf{B_{fc}}$ into $normsB[tid.x]$;
load norm of row vector *row* of $\mathbf{A_{fc}}$ into *norm_a*;

`/* calculate the sum of all column vector norms of this ABFT-block via reduction */`
$k \leftarrow BS_{ABFT}/2$;
while $k > 32$ **do**
  if $tid < k$ **and** $threadIdx.y = 0$ **then**
    $normsB[tid.x] \leftarrow normsB[tid.x] + normsB[tid.x + k]$;
  sync;
  $k \leftarrow k/2$;

`/* reduce warp-intern */`
while $k > 1$ **do**
  if $tid < k$ **and** $threadIdx.y = 0$ **then**
    $normsB[tid.x] \leftarrow normsB[tid.x] + normsB[tid.x + k]$;
  $k \leftarrow k/2$;

`/* share this sum among all threads */`
sync;
$sum\_norm\_b \leftarrow normsB[0]$;

`/* now calculate the error bound for all weighted and unweighted row checksums */`
for $i \leftarrow 1$ **to** $numWCS$ **do**
  `/* load norm of checksum vector */`
  load norm of row checksum vector of $\mathbf{B_{fc}}$ into *norm_s*;
  `/* calculate the error bound */`
  $result \leftarrow (n + 2 \cdot BS_{ABFT} - 2) \cdot norm\_a \cdot sum\_norm\_b$;
  $result \leftarrow result + n \cdot norm\_s \cdot sum\_norm\_b$;
  $result \leftarrow result \cdot$ machineEpsilon();
  `/* write back error bound */`
  store error bound row *row* and ABFT-block in global memory;

Algorithmus A.5: Kernel `determineMaxElementsRow()`, der für jeden Reihenvektor die $nMax$ betragsmäßig größten Elemente bestimmt.

Kernel `determineMaxElementsRow()`

Input: encoded matrix $\mathbf{A_{fc}}$
Output: $nMax$ largest absolute values and their positions for every row of $\mathbf{A_{fc}}$

Launch Dimensions: $BS_X \times 1$ threads per threadblock
Launch Dimensions: rows($\mathbf{A_{fc}}$) $/BS_X$ threadblocks

Data: shared $Asub[BS_X][BS_Y]$
Data: local register $curMax[nMax]$, $curMaxID[nMax]$
Data: local register $finalMax[nMax]$, $finalMaxID[nMax]$

$numLoops \leftarrow$ originalCols($\mathbf{A_{fc}}$) $/BS_Y$;
$colOffset \leftarrow 0$;

$row \leftarrow blockIdx.x \cdot BS_X + tid$;

for $curLoop \leftarrow$ **to** $numLoops$ **do**
  `/* load block of absolute values of` $\mathbf{A_{fc}}$ `to shared memory */`
  for $j \leftarrow 1$ **to** BS_Y **do**
    $Asub[tid][j] \leftarrow$ abs($\mathbf{A_{fc}}[row][j + colOffset]$);

  `/* find` $nMax$ `elements with largest absolute value */`
  for $j \leftarrow 1$ **to** $nMax$ **do**
    $curMax[j] \leftarrow 0$;
    $curMaxID[j] \leftarrow 0$;
    for $i \leftarrow 1$ **to** BS_Y **do**
      if $Asub[tid][i] > curMax[j]$ **then**
        $curMax[j] = Asub[tid][i]$;
        $curMaxID[j] = i$;
    `/* set element in shared memory to 0 */`
    $Asub[tid][curMaxID[j]] \leftarrow 0$;

  `/* insert into list of previously found elements for this row */`
  if $curLoop = 1$ **then**
    copy $curMax$ and $curMaxID$ to $finalMax$ and $finalMaxID$;
  else
    insert elements of $curMax$ into sorted list of elements for this row $finalMax$;
    add $colOffset$ to the local positions in $curMaxID$ and store them in $finalMaxID$;
  $colOffset \leftarrow colOffset + BS_Y$;

`/* write elements to global memory */`
store $nMax$ elements from $finalMax$ for row row in global memory;
store corresponding positions from $finalMaxID$ for row row in global memory;

Algorithmus A.6: Funktion `calculateEpsilonPEA()` zur Berechnung des Fehlerschwellwerts für das probabilistische Verfahren.

Function `calculateEpsilonPEA(`n, $sigma$, $bound$`)`

Data: local register eps, $fact$

$fact \leftarrow \sqrt{\frac{n \cdot (n+1) \cdot (n+\frac{1}{2}) + 2 \cdot n}{24}}$;

$eps \leftarrow fact \cdot bound \cdot$ machineEpsilon();

return eps;

Algorithmus A.7: Kernel `determineRowEpsilonMaxElements()` zur Bestimmung des Fehlerschwellwerts bei Verwendung der probabilistischen Methode über die betragsmäßig größten Elemente.

Kernel `determineRowEpsilonMaxElements()`

Input: $nMax$ largest absolute values for every row of $\mathbf{A_{fc}}$ and every column of $\mathbf{B_{fc}}$, chosen $sigma$
Output: epsilon for all row checksums

Launch Dimensions: a total of $colCSVecs \times \text{rows}(\mathbf{A_{fc}})$ threads

Data: local register $A_{idx}[nMax]$, $B_{idx}[nMax]$, $maxA[nMax]$, $maxB[nMax]$
Data: local register max, eps

$bound, eps \leftarrow 0$;
$row \leftarrow blockIdx.y \cdot BS_Y + tid.y$;
$col \leftarrow blockIdx.x \cdot BS_X + tid.x$;

`/* load max elements and indices */`
for $i \leftarrow 1$ **to** $numMax$ **do**
  load position of i-th largest absolute value of row vector row of $\mathbf{A_{fc}}$ into $A_{idx}[i]$;
  load position of i-th largest absolute value of column vector col of $\mathbf{B_{fc}}$ into $B_{idx}[i]$;
  load i-th largest absolute value of row vector row of $\mathbf{A_{fc}}$ into $maxA[i]$;
  load i-th largest absolute value of column vector col of $\mathbf{B_{fc}}$ into $maxB[i]$;

`/* initialize bound */`
$bound \leftarrow \max(bound, maxA[nMax] \cdot maxB[1])$;
$bound \leftarrow \max(bound, maxA[1] \cdot maxB[nMax])$;

`/* check for other combinations */`
for $i \leftarrow 1$ **to** $nMax$ **do**
  for $j \leftarrow 1$ **to** $nMax$ **do**
    if $A_{idx}[i] = B_{idx}[j]$ **then**
      $bound \leftarrow \max(bound, maxA[i] \cdot maxB[j])$;

`/* calculate epsilon */`
$eps \leftarrow$ calculateEpsilonPEA(originalColumns($\mathbf{A}$), $sigma$, $bound$);

`/* store epsilon */`
store eps for row row and column checksum vector col in global memory;

Algorithmus A.8: Kernel `determineBitvectorsLocalAverage()` zur Bestimmung der *nLevels* Bitvektoren und Durchschnittswerte für jeden Block von 32 Elementen der Reihenvektoren.

Kernel `determineBitvectorsLocalAverage()`

Input: encoded matrix $\mathbf{A_{fc}}$
Output: *nLevels* of bitvectors and corresponding average values for every 32 elements of a vector

Launch Dimensions: one thread for each block of 32 column elements and each row

Data: local register $data[32]$, $maxVal$, $average$, $count$, sum_gt_avg
Data: local register $avgs[nMax]$, $bitVectors[nMax]$

$row \leftarrow blockIdx.y \cdot BS_Y + tid.y$;
$col_{start} \leftarrow blockIdx.x \cdot 1024 + tid.x \cdot 32$;
$block \leftarrow blockIdx.x \cdot BS_X + tid.x$;

`/* load 32 absolute values to local memory, determine maximum and first average value */`

$avg \leftarrow 0$;
for $i \leftarrow 1$ **to** 32 **do**
  $data[i] \leftarrow \text{abs}(\mathbf{A_{fc}}[row][col_{start} + i])$;
  $maxVal \leftarrow \max(maxVal, data[i])$;
  $avg \leftarrow avg + data[i]$;
$avg \leftarrow avg/32$;

`/* loop to generate the bitvectors and average values for the different encoding levels */`

for $level \leftarrow 0$ **to** $nLevels$ **do**
  `/* reset variables */`
  $sum\_gt\_avg \leftarrow 0$; $bitVec \leftarrow 0$; $count \leftarrow 0$;
  `/* determine bitvector */`
  for $i \leftarrow 0$ **to** 31 **do**
    if $data[i] > avg$ **then**
      $bitVec \leftarrow bitVec + \text{shiftLeft}(1, i)$;
      $sum\_gt\_avg \leftarrow sum\_gt\_avg + data[i]$;
      $count \leftarrow count + 1$;
  `/* store bitvector and average value */`
  $avgs[level] \leftarrow avg$;
  $bitVectors[level] \leftarrow bitVec$;
  `/* calculate average over all elements that are encoded in the last bitvector */`
  if $level < numLevels$ **then**
    if $count > 0$ **then**
      $avg \leftarrow sum\_gt\_avg/count$;
    else
      $avg \leftarrow maxVal$;
  else
    $avgs[level] = maxVal$;

`/* store bitvectors and corresponding average values */`
for $level \leftarrow 0$ **to** $nLevels$ **do**
  store average value of *block*, *row* and *level* from $avgs[level]$ in global memory;
  store bitvector of *block*, *row* and *level* from $bitVectors[level]$ in global memory;

Algorithmus A.9: Funktion `determineBlockBound()` zur Bestimmung der lokalen oberen Schranke mittels Bitvektoren und Durchschnittswerten.

```
Function determineBlockBound(bitVecA[], bitVecB[], avgA[], avgB[], nLevels)
    Data: local register levA, levB, curA, curB, bound
    bound ← 0;
    levA = nLevels;
    levB = 0;
    /* determine highest level of A where an element is encoded                    */
    while levA > 0 and bitVecA[levA] = 0 do
        levA ← levA − 1;
    curA ← bitVecA[levA];
    /* determine lowest level of B where elements between A and B are combined     */
    while levB < nLevels and (curA&bitVecB[levB]) ≠ 0 do
        levB ← levB + 1;
    bound ←max(bound, avgA[nLevels] · avgB[levB]);
    levB ← levB − 1;
    /* increase level of B and decrease level of A to find other combinations with a
       possibly larger bound                                                       */
    while levB < nLevels and levA ≥ 0 do
        levB ← levB + 1;
        levA ← min(levA + 1, nLevels);
        curA ← bitVecA[levA];
        curB ← bitVecB[levB];
        if curB ≠ 0 then
            while levA > 0 and (curA&curB) = 0 do
                levA ← levA − 1;
                curA ← bitVecA[levA];
            bound ← max(bound, avgA[levA + 1] · avgB[levB + 1]);
    return bound;
```

Algorithmus A.10: Kernel `determineRowEpsilonBitvectorsLocalAverage()` zur Bestimmung des Fehlerschwellwerts für Reihenprüfsummenelemente bei Verwendung der lokalen Durchschnittsmetrik für das probabilistische Verfahren.

Kernel `determineRowEpsilonBitvectorsLocalAverage()`

Input: max elements for every row of $\mathbf{A_{fc}}$ and every column of $\mathbf{B_{fc}}$
Output: epsilon for all row checksums of $\mathbf{C_{fc}}$

Launch Dimensions: $BS_X \times BS_Y \times BS_Z$ threads per threadblock
Launch Dimensions: rows($\mathtt{C_{fc}}$) $/BS_Y$ threadblocks

Data: shared $shBound[BS_Y][BS_Z]$
Data: local register $avgA[nLevels]$, $avgB[nLevels]$, $bitVecA[nLevels]$, $bitVecB[nLevels]$
Data: local register *bound*, *eps*, *block*

`/* load row data */`
for *level* $\leftarrow 0$ **to** *nLevels* **do**
  load bitvector of vector of $\mathbf{A_{fc}}$ for this *block* and *level* into *bitVecA*;
  load average value of vector $\mathbf{A_{fc}}$ for this *block* and *level* into *avgA*;

`/* loop through columns */`
for *col* $\leftarrow 1$ **to** *numChecksums* **do**
  `/* load data for column */`
  for *level* $\leftarrow 0$ **to** *nLevels* **do**
    load bitvector of checksum vector of $\mathbf{B_{fc}}$ for this *block* and *level*;
    load average value of checksum vector of $\mathbf{B_{fc}}$ for this *block* and *level*;
  `/* calculate local bound */`
  bound $\leftarrow$ determineLocalBound(*bitVecA*[], *bitVecB*[], *avgA*[], *avgB*[], *nLevels*);
  `/* reduce to warp-intern bound */`
  $i \leftarrow BS_{ABFT}/2$;
  while $i \geq 1$ **do**
    bound $\leftarrow$ cuda_shuffle_xor(*bound*, *i*, BS_{ABFT});
    $i \leftarrow i/2$;
  sync;
  `/* reduce to bound for checksum element */`
  $shBound[tid.y][tid.z] \leftarrow bound$;
  $i \leftarrow blockDim.z$;
  while $i \geq 1$ **do**
    if $tid.x = 0$ **and** $tid.z \geq i$ **then**
      $shBound[tid.y][tid.z] \leftarrow \max(shBound[tid.y][tid.z], shBound[tid.y][tid.z + i])$;
    sync;
  $bound \leftarrow shBound[tid.y][0]$;
  `/* calculate epsilon */`
  eps $\leftarrow$ calculateEpsilonPEA(originalColumns(**A**), *sigma*, *bound*);
  `/* store error bound in global memory and work on next column */`
  store epsilon for *row* and *col* in global memory;

Algorithmus A.11: Kernel `determineBitvectorsGlobalAverage()` zur Bestimmung der *nLevels* Durchschnittswerten pro Reihenvektor und Bitvektoren für jeden Block von 32 Elementen.

Kernel `determineBitvectorsGlobalAverage()`

Input: encoded matrix $\mathbf{A_{fc}}$
Output: *nLevels* of bitvectors and corresponding average values for every 32 elements of a vector

Launch Dimensions: one thread for each block of 32 column elements and each row

Data: local register $data[32]$, $maxVal$, $average$, $count$, sum_gt_avg
Data: local register $avgs[nMax]$, $bitVectors[nMax]$

$row \leftarrow blockIdx.y \cdot BS_Y + tid.y$;
$col_{start} \leftarrow blockIdx.x \cdot 1024 + tid.x \cdot 32$;

`/* load 32 absolute values to local memory, determine maximum and first average value */`
$avg \leftarrow 0$;
for $i \leftarrow 1$ **to** 32 **do**
  $data[i] \leftarrow \text{abs}(\mathbf{A_{fc}}[row][col_{start} + i])$;
  $maxVal \leftarrow \max(maxVal, data[i])$;
  $avg \leftarrow avg + data[i]$;
$avg \leftarrow avg/32$;

`/* loop to generate the bitvectors and average values for the different encoding levels */`
for $level \leftarrow 0$ **to** $nLevels$ **do**
  `/* reset variables */`
  $sum\_gt\_avg \leftarrow 0$; $bitVec \leftarrow 0$; $count \leftarrow 0$;
  `/* determine bitvector */`
  for $i \leftarrow 0$ **to** 31 **do**
    if $data[i] > avg$ **then**
      $bitVec \leftarrow bitVec + \text{shiftLeft}(1, i)$;
      $sum\_gt\_avg \leftarrow sum\_gt\_avg + data[i]$;
      $count \leftarrow count + 1$;
  `/* store bitvector locally */`
  $bitVectors[level] \leftarrow bitVec$;
  sync;
  if $level < nLevels$ **then**
    `/* calculate global sum and count by reduction */`
    $sh\_sum[tid.y][tid.x] \leftarrow sum\_gt\_avg$; $sh\_cnt[tid.y][tid.x] \leftarrow count$;
    sum up counts in *sh_cnt* and sums in *sh_sum* for each block of 32 elements;
    sum up the counts and sums for each block;
    $avg \leftarrow sh\_sum[tid.y][0]/sh\_cnt[tid.y][0]$;
    $avgs[level] \leftarrow avg$;
  else
    `/* calculate global max by reduction */`
    $sh\_sum[tid.y][tid.x] \leftarrow maxVal$;
    calculate max element for each block of 32 elements;
    calculate max element for the whole vector;
    $avgs[level] \leftarrow sh\_sum[tid.y][0]$;

`/* store bitvectors and corresponding average values */`
for $level \leftarrow 0$ **to** $nLevels$ **do**
  if $tid.x = 0$ **then**
    store average value of *row* and *level* from $avgs[level]$ in global memory;
  store bitvector of *block*, *row* and *level* from $bitVectors[level]$ in global memory;

Algorithmus A.12: Kernel `determineRowEpsilonBitvectorsGlobalAverage()` zur Bestimmung des Fehlerschwellwerts für Reihenprüfsummenelemente bei Verwendung der globalen Durchschnittsmetrik für das probabilistische Verfahren.

Kernel `determineRowEpsilonBitvectorsGlobalAverage()`

Input: max elements for every row of $\mathbf{A_{fc}}$ and every column of $\mathbf{B_{fc}}$
Output: epsilon for all row checksums of $\mathbf{C_{fc}}$

Launch Dimensions: $BS_X \times BS_Y$ threads per threadblock
Launch Dimensions: rows($\mathtt{C_{fc}}$) $/BS_X \times numChecksums/BS_Y$ threadblocks

Data: local register $avgA[nLevels]$, $avgB[nLevels]$, $bitVecA[nLevels]$, $bitVecB[nLevels]$
Data: local register $finalBound$, $bound$, eps

$numColBlocks \leftarrow$ originalColumns($\mathbf{A_{fc}}$) /32;
$row \leftarrow blockIdx.x \cdot BS_X + tid.x$;
$col \leftarrow blockIdx.y \cdot BS_Y + tid.y$;

`/* load averages */`
for $level \leftarrow 0$ **to** $nLevels$ **do**
    load average value of vector $\mathbf{A_{fc}}$ for this $level$ into $avgA[level]$;
    load average value of vector $\mathbf{B_{fc}}$ for this $level$ into $avgB[level]$;

`/* go through all blocks to determine bound */`
$finalBound \leftarrow 0$;
for $block \leftarrow 1$ **to** $numColBlocks$ **do**
    `/* load bitvectors for this` $block$ `*/`
    for $level \leftarrow 1$ **to** $nLevels$ **do**
        load bitvector of vector of $\mathbf{A_{fc}}$ for this $block$ and $level$;
        load bitvector of vector of $\mathbf{B_{fc}}$ for this $block$ and $level$;

    `/* calculate local bound for this` $block$ `*/`
    $bound \leftarrow$ determineLocalBound($bitVecA[]$, $bitVecB[]$, $avgA[]$, $avgB[]$, $nLevels$);

    `/* update` $finalBound$ `*/`
    $finalBound \leftarrow \max(finalBound, bound)$;

`/* calculate epsilon */`
$eps \leftarrow$ calculateEpsilonPEA(originalColumns($[A]$, $sigma$, $bound$);

`/* store error bound in global memory */`
store epsilon for row and col in global memory;

Algorithmus A.13: Die Funktion `checkSingleCorrectable()`, die die Differenz zweier Werte mit einem gegebenen Fehlerschwellwert vergleicht und bei Auftreten von mehreren Fehlern in den Reihen oder den Spalten Rückmeldung darüber gibt.

```
Function checkSingleCorrectable(position, value, refValue, epsilon, location, exception)
    Data: local register oldLocation
    /* check if syndrome is greater than given epsilon                          */
    if |value − refValue| > epsilon then
        /* store position of this thread (row or column) in location and load the old
           value into the local register oldLocation                            */
        oldLocation ← cuda_atomic_exch(location, position);
        /* check if there has already been an error in this row / column        */
        if oldLocation ≠ −1 and oldLocation ≠ position then
            exception ← DOUBLE_ERROR(blockID);
```

Algorithmus A.14: Die Funktion `correctSingleError()`, die anhand der Reihen- und Spaltenposition ein fehlerhaftes Element durch das Syndrom $refValue - value$ korrigiert.

```
Function correctSingleError(rowPos, colPos, sumCol, refCol, exception)
    Data: local register syndrome
    if exception ≠ −1 then
        return
    /* erronous element can be localized                                        */
    if rowPos ≠ −1 and colPos ≠ −1 then
        /* correct value in data elements                                       */
        syndrome ← refCol − sumCol;
        correct value of data element in matrix C by adding syndrome;
        /* correct value in row checksum elements                               */
        correct value of row checksum element in C by adding colWeight(rowPos, cs) ·syndrome;
        /* correct value in column checksum elements                            */
        sumCol ← sumCol+ rowWeight(colPos, cs) ·syndrome;
        /* provide feedback about correction                                    */
        exception ← CORRECTED(blockID, rowPos, colPos);
    /* erronous element can not be localized                                    */
    if rowPos ≠ −1 xor colPos ≠ −1 then
        exception ← NOT_LOCALIZABLE_ERROR(blockID);
```

Algorithmus A.15: Korrekturkernel für Prüfsummen mit einfacher Lokalisierung

Kernel `checkBlockAndCorrectSingleError()`

Input: matrix $\mathbf{C_{fc}}$, checksums weights *rowWeights*[], *colWeights*[]
Output: corrected matrix $\mathbf{C_{fc}}$, feedback about correction

Launch Dimensions: $2 \cdot numWCS \times BS_{ABFT}$ threads per threadblock
Launch Dimensions: one threadblock for each $BS_{ABFT} \times BS_{ABFT}$ submatrix of $\mathbf{C_{fc}}$

Data: shared $Asub[BS_X][BS_{ABFT}]$, $accum\_row[WC][BS_{ABFT}/BS_X][BS_X]$
Data: shared *locRow*, *locCol*, *exception*
Data: local registers *sumRow*, *sumCol*, *numStripes*

$numStripes \leftarrow BS_{ABFT}/BS_X$;
$locCol, locRow, exception \leftarrow -1$;

loadReferenceColumnChecksum(*tid.y*); loadColumnEpsilon(*tid.y*);

for $curStripe \leftarrow 1$ **to** *numStripes* **do**
    $sumRow \leftarrow 0$;
    `/* ...`
    `load data to shared memory, sum up column and row values like in encoding kernel`
    `... */`
    sync;
    `/* accumulate to get the row checksum */`
    if $tid.y < BS_X$ **then**
        loadReferenceRowChecksum(*row*); loadRowEpsilon(*row*);
        $sumRow \leftarrow 0$;
        for $i \leftarrow 1$ **to** BS_{ABFT}/BS_X **do**
            $sumRow \leftarrow sumRow + accumRow[cs][i][row]$;
        shuffleAndCheck(*sumRow*,*exception*);
        `/* compare row checksum with reference checksum using epsilon value */`
        checkSingleCorrectable(*row*, *sumRow*, *refRow*, *epsRow*, *locRow*, *exception*);
        `/* store new calculated checksum to prevent redundant encoding */`
        write back row checksum *sumRow* into $\mathbf{C_{fc}}$;
    $rowOffset \leftarrow rowOffset + BS_X$;
sync;
`/* compare column checksum with reference checksum using epsilon value */`
shuffleAndCheck(*sumCol*,*exception*);
$col \leftarrow tid.y$;
checkSingleCorrectable(*col*, *sumCol*, *refCol*, *epsCol*, *locCol*, *exception*);
`/* correct single error with` $2 \cdot numWCS$ `threads */`
if $tid.y = 0$ **then**
    correctSingleError(*locRow*, *locCol*, *sumCol*, *refCol*, *exception*);
`/* store new calculated checksum to prevent redundant encoding */`
write back column checksum *sumCol* into $\mathbf{C_{fc}}$;

Algorithmus A.16: Die Funktion `checkAndLocateError()`, die die Differenz zweier Werte mit einem gegebenen Fehlerschwellwert vergleicht, bei einer Abweichung den Fehler lokalisiert und bei mehreren Lokalisierungen für eine Spalte oder Reihe einen Fehler zurückgibt.

```
Function checkAndLocateError(value, refValue, epsilon, locSelf, locOther, exception)
    Data: local register syndrome, weightedSyndrome, error, weightedError, loc

    /* check if syndrome is greater than given epsilon                          */
    syndrome = value − refValue;
    if |syndrome| > epsilon then
        error ← 1;
    else
        error ← 0;

    /* exchange syndrome and error information with other threads that calculated the
       same row/column but for different weights                                */
    weightedSyndrome ← cuda_shuffle_xor(syndrome, numWCS, 2 · numWCS);
    weightedError ← cuda_shuffle_xor(error, numWCS, 2 · numWCS);

    if tid.x = 0 then
        /* locate error if at least one error occured or another localization has been
           the targeting this row/column                                        */
        if error = 1 or weightedError = 1 or locSelf[tid] ≠ −1 then
            /* locate error depending on weighting scheme                       */
            loc ← locateError(syndrome,weightedSyndrome);

            /* check if localization is a 1-to-1 relationship                   */
            if locSelf[tid] ≠ −1 and locSelf[tid] ≠ loc then
                /* another localization was expecting something different, raise error */
                exception ← DOUBLE_ERROR(blockID);

            if locOther[loc] = tid or locOther[loc] = −1 then
                /* e.g. store that row 1 located error in column 2              */
                locSelf[tid] ← loc;
                /* e.g. store that the localization in column 2 should be pointing to row
                   1                                                            */
                locOther[loc] ← tid ;
            else
                /* the previosly performed localization said something different */
                exception ← DOUBLE_ERROR(blockID);
```

Algorithmus A.17: Die Funktion `correctMultiError()`, die anhand der Reihen- und Spaltenposition ein fehlerhaftes Element durch das Syndrom korrigiert, wenn dieses eindeutig lokalisiert wurde.

```
Function correctMultiError(locRow, locCol, sumCol, refCol ,rowSyndromes[], exception)
    Data: local register colSyndrome, rowSyndrome, epsilon

    if exception ≠ −1 then
        return

    col ← tid.y;
    row ← locCol[col];

    /* correct error if error occured                                          */
    if row ≠ −1 then

        /* calculate column syndrome and propagate unweighted syndrome to all threads
           associated with this column                                          */
        colSyndrome ← refCol − sumCol;
        colSyndrome ← cuda_shuffle(colSyndrome, 0 ,2 · WC);
        rowSyndrome ← rowSyndromes[row];

        /* compare unweighted syndromes, needed: epsilon value                 */
        if |colSyndrome − rowSyndrome| > epsilon then
            /* if no match: return with error                                  */
            exception ← NOT_LOCALIZABLE_ERROR(blockID);
        else

            /* correct value in data element                                   */
            correct value of data element in matrix C by adding colSyndrome;

            /* correct value in row checksum elements                          */
            correct value of row checksum element in matrix C by adding
              rowWeight(col,cs) · colSyndrome;

            /* correct value in col checksum elements, these will be stored after this
               subfunction, so alter sumCol                                    */
            sumCol ← refCol;

            /* provide feedback about correction                               */
            exception ← CORRECTED(blockID);
```

Algorithmus A.18: Korrekturkernel für Prüfsummen mit erweiterter Lokalisierung

Kernel `checkBlockAndCorrectMultipleError()`

Input: matrix $\mathbf{C_{fc}}$, checksums weights *rowWeights*[], *colWeights*[]
Output: corrected matrix $\mathbf{C_{fc}}$, feedback about correction

Launch Dimensions: $2 \cdot numWCS \times BS_{ABFT}$ threads per threadblock
Launch Dimensions: one threadblock for each $BS_{ABFT} \times BS_{ABFT}$ submatrix of $\mathbf{C_{fc}}$

Data: shared $Asub[BS_X][BS_{ABFT}]$, $accum\_row[WC][BS_{ABFT}/BS_X][BS_X]$
Data: shared $linearSyndromeRow[BS_{ABFT}]$, $locRow[BS_{ABFT}]$, $locCol[BS_{ABFT}]$, *exception*
Data: local registers *sumRow*, *sumCol*, *numStripes*

$numStripes \leftarrow BS_{ABFT}/BS_X$;
$locCol[*], locRow[*], exception \leftarrow -1$;

loadReferenceColumnChecksum(*tid*); loadColumnEpsilon(*tid*);

`/* loop over all stripes */`
for $curStripe \leftarrow 1$ **to** *numStripes* **do**
  $sumRow \leftarrow 0$;
  `/* load data to shared memory, sum up column and row values like in encoding kernel ... */`
  sync;
  `/* accumulate to get the row checksum */`
  if $tid.y < BS_X$ **then**
    `/* load reference checksums and epsilon, accumulate ... */`
    shuffleAndCheck(*sumRow*,*exception*);
    store the unweighted syndrome of row i in $syndromeRow[i]$;
    `/* compare row checksum with reference checksum using epsilon value */`
    checkAndLocateError(*sumRow*,*refRow*,*epsRow*,*locRow*,*locCol*,*exception*);
    `/* store new calculated checksum to prevent redundant encoding */`
    write back row checksum *sumRow* into $\mathbf{C_{fc}}$;
  $rowOffset \leftarrow rowOffset + BS_X$;
sync;
`/* compare column checksum with reference checksum using epsilon value */`
shuffleAndCheck(*sumCol*,*exception*);
`/* notice that` *locCol* `and` *locRow* `are switched */`
checkAndLocateError(*sumCol*, *refCol*, *epsCol*, *locCol*, *locRow*, *exception*);
`/* correct error */`
correctMultiError(*locRow*, *locCol*, *sumCol*, *refCol*, *syndromeRow*, *exception*);
`/* store new calculated checksum to prevent redundant encoding */`
write back column checksum *sumCol* into $\mathbf{C_{fc}}$;

Algorithmus A.19: Kernel `matrixMult()` zur Matrixmultiplikation nach [48].

Kernel `matrixMult()`
Input: matrix **A**, matrix **B**
Output: matrix **C**

Data: shared memory $smA[BK][BM]$, $smB[BK][BN]$
Data: local registers $accum[RX \cdot RY]$, $rA[RX]$, $rB[RY]$

$accum[0..RX][0..RY] \leftarrow 0$;

`/* load first blocks of` **A** `and` **B** `to shared memory */`
load one $BM \times BK$ block of **A** into $smA[BK][BM]$;
load one $BK \times BN$ block of **B** into $smB[BK][BN]$;
sync;
while $K > 0$ **do**
  $K = K - 1$;
  for $ki \leftarrow 1$ **to** BK **do**
    `/* load data from shared memory to local registers */`
    load one column of **A** in smA into $rA[0..RX]$;
    load one row of **B** in smB into $rB[0..RY]$;
    `/* add local data to accumulated data */`
    $accum[0..RX][0..RY] \leftarrow accum[0..RX][0..RY] + rA[0..RX] \cdot rB[0..RY]$;
  `/* load the next block of` **A** `and` **B** `to shared memory */`
  load next $BM \times BK$ block of **A** into $smA[BK][BM]$;
  load next $BK \times BN$ block of **B** into $smB[BK][BN]$;
  sync;
Merge $accum[0..RX][0..RY]$ with $BM \times BN$ block of **C**

Algorithmus A.20: Kernel zur Matrixmultiplikation mit Fehlerinjektion

Kernel `matrixMultFaultInjected()`
Input: matrix **A**, matrix **B**, parameters for fault injection
Output: matrix **C**

Launch Dimensions: $BM \times BN$ threads, each threadblock processes one submatrix of **C**

Data: shared memory $smA[BK][BM]$, $smB[BK][BN]$
Data: local register $accum[RX \cdot RY]$, $rA[RX]$, $rB[RY]$
Data: local register $errorVecMul[RX][RY]$
Data: local register $errorVecAdd1[RX][RY]$, $errorVecAdd1[RX][RY]$

$accum[0..RX][0..RY] \leftarrow 0$;

load one $BM \times BK$ block of **A** into $smA[BK][BM]$;
load one $BK \times BN$ block of **B** into $smB[BK][BN]$;
sync;

[... Fortsetzung auf der nachfolgenden Seite ...]

Fortsetzung Kernel `matrixMultFaultInjected()`

[... Fortsetzung von der vorhergehenden Seite ...]

while $K > 0$ **do**
- $K = K - 1$;
- **for** $ki \leftarrow 1$ **to** BK **do**
  - `/* initialize local bit masks for inner loop fault injection */`
  - set $errorVecMult[0..RX][0..RY]$;
  - set $errorVecAdd[0..RX][0..RY]$;
  - load one column of **A** in smA into $rA[0..RX]$;
  - load one row of **B** in smB into $rB[0..RY]$;
  - `/* accumulation with inner loop fault injection */`
  - $accum[0..RX][0..RY] \leftarrow accum[0..RX][0..RY] +$ $((rA[0..RX] \cdot rB[0..RY]) \oplus errorVecMult[0..RX][0..RY])$;
  - $accum[0..RX][0..RY] \leftarrow (accum[0..RX][0..RY] \oplus errorVecAdd1[0..RX][0..RY])$;
- load next $BM \times BK$ block of **A** into $smA[BK][BM]$;
- load next $BK \times BN$ block of **B** into $smB[BK][BN]$;
- **sync**

`/* fault injection for merging results */`

set $errorVecAdd[0..RX][0..RY]$;

Merge $accum[0..RX][0..RY]$ with $BM \times BN$ block of **C**;

$BM \times BN$ block of **C** $= BM \times BN$ block of $C \oplus errorVecAdd2[0..RX][0..RY]$;

Literaturverzeichnis

[1] E. Winsberg, *Science in the Age of Computer Simulation*. University of Chicago Press, 2010. (Zitiert auf Seite 1)

[2] J. S. Vetter, *Contemporary High Performance Computing: From Petascale Toward Exascale*. Chapman & Hall/CRC, 2013. (Zitiert auf Seite 1)

[3] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. (Zitiert auf Seite 3)

[4] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010. (Zitiert auf Seite 3)

[5] J. Von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata studies*, 1956. (Zitiert auf Seite 3)

[6] R. W. Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950. (Zitiert auf Seite 3)

[7] D. A. Anderson and G. Metze, "Design of Totally Self-Checking Check Circuits for m-Out-of-n Codes," *IEEE Transactions on Computers*, no. 3, pp. 263–269, 1973. (Zitiert auf Seite 3)

[8] R. A. Frohwerk, "Signature Analysis: A new Digital Field Service Method," *Hewlett-Packard J.*, vol. 28, no. 9, pp. 2–8, May 1977. (Zitiert auf Seite 4)

[9] A. Mahmood and E. J. McCluskey, "Concurrent Error Detection using Watchdog Processors - A Survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, 1988. (Zitiert auf Seite 4)

[10] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, no. 2, pp. 220–232, 1975. (Zitiert auf den Seiten 4 und 5)

[11] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, no. 12, pp. 1491–1501, 1985. (Zitiert auf Seite 4)

[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2005, pp. 243–254. (Zitiert auf Seite 4)

[13] F. Cappello, "Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, Jul. 2009. (Zitiert auf Seite 5)

[14] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, Oct. 1998. (Zitiert auf Seite 5)

[15] X. Ouyang, S. Marcarelli, and D. K. Panda, "Enhancing Checkpoint Performance with Staging IO and SSD," in *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, ser. SNAPI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 13–20. (Zitiert auf Seite 5)

[16] K.-H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. 33, no. 6, Jun. 1984. (Zitiert auf Seite 6)

[17] F. T. Luk and H. Park, "An Analysis of Algorithm-Based Fault Tolerance Techniques," *Journal of Parallel and Distributed Computing*, vol. 5, no. 2, pp. 172–184, Apr. 1988. (Zitiert auf Seite 6)

[18] J. Y. Jou and J. A. Abraham, "Fault-Tolerant FFT Networks," *IEEE Transactions on Computers*, vol. 37, no. 5, pp. 548–561, 1988. (Zitiert auf Seite 6)

[19] M. Tsunoyama and S. Naito, "A Fault-Tolerant FFT Processor," *Digest of Papers of the Twenty-First International Symposium Fault-Tolerant Computing (FTCS-21)*, pp. 128–135, 1991. (Zitiert auf Seite 6)

[20] S. J. Wang and N. K. Jha, "Algorithm-Based Fault Tolerance for FFT Networks," *IEEE Transactions on Computers*, 1994. (Zitiert auf Seite 6)

[21] J. Y. Jou and J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," in *Proceedings of the IEEE*, 1986, pp. 732–741. (Zitiert auf den Seiten 9, 10 und 11)

[22] C. J. Anfinson and F. T. Luk, "A Linear Algebraic Model of Algorithm-Based Fault Tolerance," *IEEE Transactions on Computers*, vol. 37, no. 12, pp. 1599–1604, 1988. (Zitiert auf den Seiten 10 und 11)

[23] V. S. S. Nair and J. A. Abraham, "General Linear Codes for Fault-Tolerant Matrix Operations on Processor Arrays," *Digest of Papers of the Eighteenth International Symposium on Fault-Tolerant Computing (FTCS-18)*, pp. 180–185, 1988. (Zitiert auf den Seiten 11 und 50)

[24] 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*, 1988, pp. 631–635. (Zitiert auf den Seiten 11 und 51)

[25] J. Rexford and N. K. Jha, "Partitioned Encoding Schemes for Algorithm-Based Fault Tolerance in Massively Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 6, pp. 649–653, 1994. (Zitiert auf Seite 12)

[26] 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*, vol. 39, no. 9, pp. 1132–1145, 1990. (Zitiert auf den Seiten 13 und 18)

[27] A. Roy-Chowdhury and P. Banerjee, "Algorithm-Based Fault Location and Recovery for Matrix Computations on Multiprocessor Systems," *Digest of Papers of the Twenty-Fourth International Symposium on Fault-Tolerant Computing (FTCS-24)*, pp. 38–47, 1994. (Zitiert auf Seite 13)

[28] R. Banerjee and J. A. Abraham, "Bounds on Algorithm-Based Fault Tolerance in Multiple Processor Systems," *IEEE Transactions on Computers*, vol. 35, no. 4, Apr. 1986. (Zitiert auf Seite 13)

[29] D. Gu, D. J. Rosenkrantz, and S. S. Ravil, "Design and Analysis of Test Schemes for Algorithm-Based Fault Tolerance," *Digest of Papers of the 20th International Symposium Fault-Tolerant Computing (FTCS-20)*, pp. 106–113, 1990. (Zitiert auf Seite 13)

[30] D. Goldberg, "What Every Computer Scientist Should Know About Floating-point Arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, Mar. 1991. (Zitiert auf Seite 16)

[31] V. Balasubramanian, "The Analysis and Synthesis of Efficient Algorithm-Based Error-Detection Schemes for Hypercube Multiprocessors," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Jan. 1991. (Zitiert auf Seite 18)

[32] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Edition)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996. (Zitiert auf den Seiten 18 und 80)

[33] J. A. Gunnels, D. S. Katz, E. S. Quintana-Orti, 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)*, 2001, pp. 47–56. (Zitiert auf den Seiten 18, 19 und 66)

[34] 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)*, 2000, pp. 107–116. (Zitiert auf Seite 19)

[35] M. Turmon, R. Granat, D. S. Katz, and J. Z. Lou, "Tests and Tolerances for High-Performance Software-Implemented Fault Detection," *IEEE Transactions on Computers*, vol. 52, no. 5, pp. 579–591, 2003. (Zitiert auf den Seiten 19 und 65)

[36] A. Roy-Chowdhury and P. Banerjee, "Tolerance Determination for Algorithm-Based Checks Using Simplified Error Analysis Techniques," *Digest of Papers of the Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23)*, 1993. (Zitiert auf den Seiten 19, 21, 66 und 93)

[37] S. Dutt and F. T. Assaad, "Mantissa-Preserving Operations and Robust Algorithm Based Fault Tolerance for Matrix Computations," *IEEE Transactions on Computers*, vol. 45, no. 4, pp. 408–424, 1996. (Zitiert auf Seite 20)

[38] J. H. Wilkinson, "Rounding errors in algebraic processes," 1994. (Zitiert auf Seite 21)

[39] J. L. Barlow and E. H. Bareiss, "Über Rundungsfehlerverteilungen bei Gleitkomma- und logarithmischer Arithmetik," *Computing*, vol. 34, no. 4, pp. 325–347, Dec. 1985. (Zitiert auf den Seiten 21, 23, 24 und 25)

[40] F. Benford, "The Law of Anomalous Numbers," *Proceedings of the American Philosophical Society*, vol. 78, no. 4, pp. 551–572, 1938. (Zitiert auf Seite 22)

[41] B. Rauch, M. Göttsche, G. Brähler, and S. Engel, "Fact and Fiction in EU-Governmental Economic Data," *German Economic Review*, vol. 12, no. 3, pp. 243–255, 2011. (Zitiert auf Seite 22)

[42] B. F. Roukema, "A first-digit anomaly in the 2009 Iranian presidential election," *Journal of Applied Statistics*, vol. 41, no. 1, pp. 164–199, 2014. (Zitiert auf Seite 22)

[43] R. W. Hamming, "On the Distribution of Numbers," *Bell System Technical Journal*, vol. 49, 1970. (Zitiert auf Seite 23)

[44] R. S. Pinkham, "On the Distribution of First Significant Digits," *The Annals of Mathematical Statistics*, vol. 32, no. 4, pp. 1223–1230, 12 1961. (Zitiert auf Seite 23)

[45] C. Braun, S. Halder, and H.-J. Wunderlich, "A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units," in *to appear in Proc. of The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, 2014. (Zitiert auf Seite 23)

[46] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the AFPIS Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. (Zitiert auf Seite 30)

[47] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, "An Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, pp. 135–151, 2001. (Zitiert auf Seite 37)

[48] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on Fermi GPU," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011, pp. 1–11. (Zitiert auf den Seiten 64, 83 und 106)

[49] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6th ed., 2014, http://gmplib.org/. (Zitiert auf Seite 66)

Alle URLs wurden zuletzt am 15. 04. 2014 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschift