

Goethe-Universität Frankfurt am Main

Fachbereich 12
Informatik und Mathematik



Masterarbeit

**Analyse und Optimierung der parallelen
Prozessierung des Unpackings, ausgehend vom
STS-Detektor am mCBM-Experiment**

Eingereicht bei

Dr. habil. Andreas Redelbach

Eingereicht von

Sebastian Heinemann

6586667

Eingereicht am

12.10.2023

Zusammenfassung

Ein wichtiger Teil des Rekonstruktionsalgorithmus für kollidierte Teilchen des CBM-Experiments an der GSI (Gesellschaft für Schwerionenforschung) in Darmstadt ist der Unpacker-Prozess (Unpacker). Er steht am Anfang der Rekonstruktionsprozessierung und sorgt für eine Umwandlung der Detektor-Rohdaten in C++ Datenstrukturen, welche anschließend von den Folgeprozessen verwendet werden. Am CBM-Experiment soll die Auswertung der Daten Online, also in Echtzeit direkt nach der Erfassung durch die Detektoren stattfinden. Da die anfallenden Datenmengen sehr hoch sind, ist eine schnelle Ausführungszeit des Unpackers und damit die effiziente Ressourcennutzung unerlässlich. Aus diesem Grund wurde er bereits im Rahmen anderer Arbeiten angepasst und verbessert. Um weitere Fortschritte in Richtung der Online Nutzung des Unpackers zu erzielen wird in dieser Arbeit speziell die parallele Prozessierung der Daten des STS-Detektors betrachtet.

In einem ersten Schritt wird die parallele Ausführung des Unpackers auf CPU Basis optimiert, sodass ein Speedup von 4-5 erzielt wird. Anschließend folgt eine Analyse des FairMQ Frameworks, um dessen Nutzbarkeit für den Parallelisierungsprozess bewerten zu können.

Zuletzt wird der Einsatz von Grafikkarten zur effizienteren Verarbeitung der Daten betrachtet. In diesem Rahmen entsteht eine Optimierung auf Ebene des Unpacker-Algorithmus, die einen Speedup von etwa 20 erreicht. Die gleichzeitige Verlangsamung des Unpacker-Tasks sorgt allerdings für eine insgesamt höhere Gesamtlaufzeit des Unpackers, wodurch dieser noch nicht einsatzbereit für die CBM-Software ist. Der GPU-basierte Ansatz zeigt aber, dass noch Optimierungspotenzial besteht und bildet eine solide Basis für zukünftige Anpassungen.

Inhaltsverzeichnis

1	Einleitung	1
2	Grundlagen	3
2.1	CBM-Experiment	3
2.1.1	mCBM-Experiment	4
2.1.2	Silicon Tracking System	4
2.2	FairMQ	5
2.3	Rekonstruktion der Teilchenflugbahnen	6
2.3.1	Task-basierte Umsetzung	8
2.3.2	MQ-basierte Umsetzung	9
2.4	Parallele Programmierung	10
2.4.1	OpenMP	10
2.4.2	GPUs	11
2.4.3	XPU-Framework	13
3	STS-Unpacker	15
3.1	CbmStsDigi	15
3.2	Timeslice Konzept	16
3.3	Unpacker-Algorithmus	17
3.4	Aktueller Stand des Unpackers	19
3.4.1	Task-basiert sequenziell	19
3.4.2	Task-basiert parallel	19
3.4.3	MQ-basiert	21
3.4.4	Task-basiert GPU	21
3.5	Testumgebung	22
3.5.1	Hardware	22
3.5.2	Testdaten	24
3.5.3	Hinweis zu Laufzeitmessungen	26

4	Optimierung und Analyse des parallelen Unpackers	27
4.1	Optimierung des Task-basierten parallelen Unpackers	27
4.1.1	Parallelisierung über Microslices	27
4.1.2	Optimierung der Resize-Funktion	29
4.1.3	Laufzeitanalyse	31
4.2	Analyse der MQ-basierten Parallelisierung	33
4.2.1	Anpassung des Codes	33
4.2.2	Laufzeitanalyse	34
4.3	Single-Thread GPU	37
4.3.1	Konzept	37
4.3.2	Laufzeitanalyse Unpacker-Algorithmus	39
4.3.3	Laufzeitanalyse Unpacker-Task	41
4.4	Multi-Thread GPU	43
4.4.1	Konzept	44
4.4.2	Laufzeitanalyse	47
5	Validierung	49
6	Zusammenfassung und Ausblick	51
7	Anhang	56
7.1	Pseudocode des ursprünglichen Unpacker-Algorithmus	56
7.2	Pseudocode des sequenziellen Unpacker-Tasks	58
7.3	Pseudocode des parallelen Unpacker-Tasks	59
7.4	Pseudocode des optimierten parallelen Unpacker-Tasks	60
7.5	Code NoInitAllocator	61
7.6	Pseudocode des Single-Thread GPU Unpacker-Algorithmus	62
7.7	Pseudocode des Multi-Thread GPU Unpacker-Algorithmus	64
7.8	Glossar	66

Abbildungsverzeichnis

2.1	Layout der FAIR Beschleunigeranlage	4
2.2	mCBM Aufbau	5
2.3	Prozesskette der Online STS-Rekonstruktion	7
2.4	Task-basierte Steuerung der Online STS-Prozesskette	8
2.5	MQ-basierte Steuerung der Online STS-Prozesskette	9
2.6	OpenMP join-fork-Modell	11
2.7	Aufbau einer CUDA GPU	12
3.1	Timeslice Struktur	16
3.2	Größenverteilung der Timeslices	24
4.1	Laufzeitvergleich CPU Varianten	31
4.2	Laufzeitvergleich MQ-basiert vs. Task-basiert	35
4.3	Kernellaufzeitvergleich Single-Thread GPU vs. Grundlagen	39
4.4	Laufzeitvergleich Unpacker-Task Single-Thread GPU	42
4.5	Präfixsummenbeispiel	45
4.6	Kernellaufzeitvergleich aller Ansätze	47

1 Einleitung

Neutronensterne sind Sterne mit außergewöhnlichen Eigenschaften wie einer extrem hohen Temperatur und Dichte. Ihre Dichte ist so hoch, dass Protonen und Elektronen zu Neutronen verdichtet werden. Es wird vermutet, dass im Zentrum eines Neutronensterns, wo die Bedingungen am extremsten sind, sogar die Neutronen in ihre Bestandteile, Quarks und Gluonen, zerlegt werden. Mit dem heutigen Stand der Technik ist es nicht möglich, Neutronensterne direkt zu untersuchen, um das Verhalten dieser kleinsten Teilchen unter solch extremen Bedingungen zu beobachten. Eben dieses Ziel verfolgt deshalb das Compressed Baryonic Matter Experiment (CBM-Experiment) [1]. Durch die Kollision von schweren Atomkernen können die Bedingungen, die im Inneren eines Neutronensterns herrschen, hier auf der Erde in einer kontrollierten Umgebung nachgestellt werden. Da bei einer solchen Kollision nur sehr wenig Materie entsteht und diese außerdem nur für einen äußerst kurzen Zeitraum existiert, kann dieses Ereignis nicht direkt selbst beobachtet werden. Im CBM-Experiment werden deshalb leistungsstarke und hochauflösende Detektoren eingesetzt, die jene Teilchen aufzeichnen, welche vom Zentrum einer Kollision weggeschossen werden. Der Fokus dieser Arbeit liegt im Speziellen auf dem STS-Detektor (Silicon Tracking System) und der dazugehörigen Datenauswertung. Wird ein Teilchen von ihm detektiert, ist es anschließend mit Hilfe von Algorithmen möglich, dessen Flugbahn (*Track*) zu berechnen und auf den Kollisionspunkt zurückzuführen. So lassen sich Informationen über Kollisionsereignisse sammeln, die später analysiert werden können. Die anfallenden Datenraten im CBM-Experiment werden auf über 1TB/s geschätzt [2].

Damit die Rekonstruktion der Tracks algorithmisch geschehen kann, müssen die Rohdaten aus dem STS-Detektor zunächst in eine C++ Datenstruktur umgewandelt werden. Dies ist die Aufgabe des Unpacker-Prozesses (Unpacker). Um eine schnelle Weiterverarbeitung der Daten zu ermöglichen, muss der Unpacker die großen Datenmengen möglichst effizient mit den vorhandenen Ressourcen verarbeiten können. Die vorliegende Arbeit widmet sich deshalb der Analyse und Optimierung der parallelen Prozessierung des STS-Unpackers. Kapitel 2 beginnt mit einer Einführung in die Grundlagen des Unpacker Konzepts sowie einer Erklärung des CBM-Projekts und der in dieser Arbeit verwendeten

Software-Frameworks.

Im Folgenden werden in Kapitel 3 die Details der verschiedenen vorhandenen Implementierungen des Unpackers besprochen, während in Kapitel 4 die vorgenommenen Optimierungen und Analysen behandelt werden. Abschließend folgen noch Anmerkungen zur Validierung der Änderungen sowie eine umfassende Beschreibung der noch ausstehenden Arbeiten.

2 Grundlagen

In diesem Kapitel wird der Aufbau des CBM-Experiments beschrieben und die in dieser Arbeit verwendeten Frameworks erklärt.

2.1 CBM-Experiment

Das CBM-Experiment ist ein sich im Bau befindendes Experiment der FAIR (Facility for Antiproton and Ion Research) an der GSI (Gesellschaft für Schwerionenforschung) in Darmstadt. Es soll voraussichtlich bis 2025 [3] fertiggestellt werden und an den ebenfalls im Bau befindlichen Teilchenbeschleuniger SIS100 angeschlossen werden. Die Fertigstellung des SIS100 ist zeitgleich mit dem CBM-Experiment im Jahr 2025 [4] geplant. Er wird die vorhandene FAIR Beschleunigeranlage erweitern und somit das Beschleunigen von Teilchen auf bis zu 99% der Lichtgeschwindigkeit ermöglichen [5]. Dazu wird er, wie in Abbildung 2.1 zu erkennen ist, an den bereits vorhandenen SIS18 Beschleuniger angeschlossen. Vom SIS100 werden die Teilchen dann zu den verschiedenen Experimenten geleitet, zu denen auch das CBM-Experiment zählt. Dessen Ziel ist es, das Quantenchromodynamik-Phasendiagramm im Bereich hoher baryonischer Dichten zu erforschen [6]. Dies soll Aufschluss über verschiedene kosmische Rätsel geben, wie etwa die Struktur im Inneren von Neutronensternen beschaffen ist oder welche Vorgänge bei der Kollision zweier Neutronensterne zu erwarten sind.

Um die Bedingungen für diese Forschungen kreieren zu können, werden schwere Ionen mit dem SIS100 auf 99% der Lichtgeschwindigkeit beschleunigt. Anschließend werden sie zur Kollision mit einem Ziel aus ebenfalls schweren Ionen gebracht, wobei ein Quark-Gluon-Plasma erzeugt wird, das kurze Zeit später wieder zerfällt. Hinter der Kollisionsstelle wird das CBM-Experiment mit seinen Detektoren aufgebaut sein, um die Ereignisse während der Kollision studieren zu können. Es wird eine Kollisionsrate von 10MHz angestrebt, sodass mit Datenraten von bis zu 1TB/s zu rechnen ist [2].

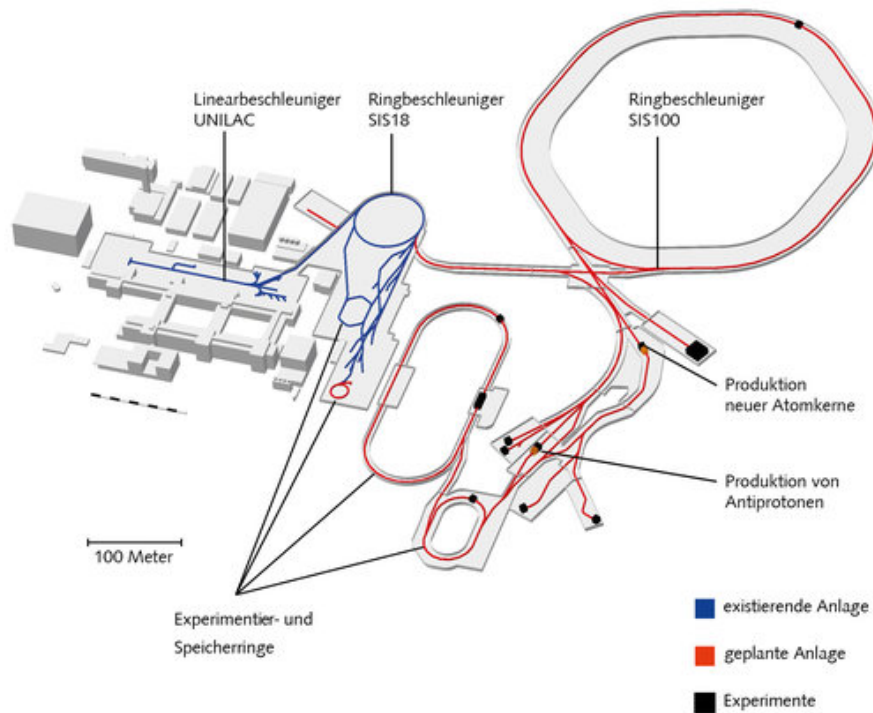


Abbildung 2.1: Layout der FAIR Beschleunigeranlage [5]

2.1.1 mCBM-Experiment

Bis zur Fertigstellung des CBM-Experiments wird ein Testaufbau der Detektoren mit dem SIS18 betrieben. Das sogenannte mCBM-Experiment ermöglicht das Testen und Weiterentwickeln der Detektoren von Elektronik und Hardware, bis hin zur gesamten Datenauswertungskette (erklärt in Kap. 2.3) unter realen Bedingungen [7]. Die für das mCBM-Experiment genutzten Detektoren sind Prototypen jener, die später im CBM-Experiment eingesetzt werden. Die Detektoren des mCBM-Experiment tragen zur Kennzeichnung ein *m* im Namen. So ist beispielsweise der mSTS-Detektor eine herunterskalierte Version des STS-Detektors. Der Testaufbau ist in Abbildung 2.2 zu sehen. Alle in dieser Arbeit verwendeten Testdaten stammen aus Versuchsläufen des mCBM-Experiments.

2.1.2 Silicon Tracking System

Das Silicon Tracking System (STS-Detektor) gehört zu den Kernsystemen des CBM-Experiments. Es wird dazu genutzt, die Flugbahnen und den Impuls der Teilchen zu bestimmen, die bei der Kollision der schweren Ionen entstehen [9]. Dafür werden Silizium-Mikrostreifen genutzt, die in Modulen in einer Entfernung zwischen 30cm und 100cm

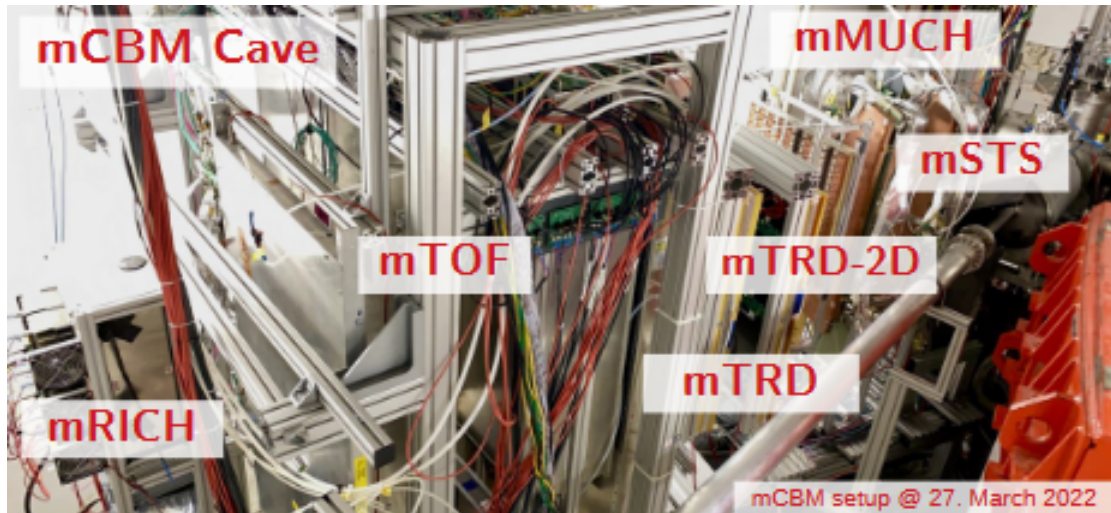


Abbildung 2.2: Aufbau der mCBM Detektor-Prototypen in der mCBM-Cave [8]

hinter dem Ziel installiert sind. Jeder Streifen ist einem Readout-Channel zugeteilt, der zu dessen Adressierung dient. Der im mCBM verbaute mSTS-Detektor Prototyp ist aktuell mit 11 Modulen ausgestattet [10]. Da sie sowohl nebeneinander als auch hintereinander im Raum verteilt sind, wird ein Teilchen beim Flug durch den Detektor also mehrfach registriert. Aus den Registrierungspunkten können somit dreidimensionale Koordinaten erstellt werden, die die Basis für die Berechnung der Flugbahnen bilden. Da der Detektor selbst allerdings keine direkte Berechnungsfunktion dafür besitzt, muss dies von der Auswertungssoftware auf dem Compute Cluster erledigt werden. Der genaue Ablauf der Flugbahnrekonstruktion ist in 2.3 beschrieben.

2.2 FairMQ

FairMQ ist eine von der FairRootGroup entwickelte C++ Bibliothek zur Vereinfachung von Arbeitsabläufen mit sehr hohem Datendurchsatz [11]. Das MQ im Namen steht für *Message Queue*, wobei es sich hier um eine asynchrone Umsetzung davon handelt: Die Daten werden in Form von Messages in asynchronen Queues übertragen. Das hat den Vorteil, dass Prozesse bzw. Algorithmen unabhängiger voneinander arbeiten können. Ist ein Prozess schneller als ein anderer, werden die Daten in den Queues zwischengespeichert, bis sie zur Bearbeitung des Folgeprozesses benötigt werden. Außerdem bringt die Datenübertragung in Form von Messages mit sich, dass diese unabhängig von zugrundeliegenden Datentypen operieren.

In FairMQ werden Prozesse (Tasks) in sogenannte *Devices* eingebettet. Die Aufgabe der Devices ist primär die Organisation des Datenflusses. Entsprechend ist eine der wichtigsten Funktionen eines Devices die Kommunikation mit anderen Devices. Diese basiert auf dem zeroMQ Framework [11]. Ein Device kann grundsätzlich mit mehreren Devices kommunizieren und braucht dafür entsprechende Endpunkte, die *Channels* genannt werden. Es gibt verschiedene Typen von Channels, die deren Funktionsweise bestimmen, z.B. Push-Pull oder Request-Reply. Ein Channel muss immer mit mindestens einem anderen Channel verbunden und die Kommunikationsrichtung eindeutig definiert sein. Dies wird erreicht, indem immer Eingangschannels mit Ausgangschannels gekoppelt werden. Ein Channel ist auch über das Netzwerk erreichbar, da ihm eine feste IP-Adresse und Port zugewiesen werden können [12]. Gerade im Umfeld von Projekten mit hohem Datendurchsatz ist dies unerlässlich, da dort meist auf verteilten Systemen gearbeitet wird. Die FairMQ Prozesssteuerung basiert auf einem Shellsript, über welches das Programm ausgeführt wird. In diesem werden alle Devices samt ihrer Channels definiert. Für jedes Device wird festgelegt, wo und wie es seinen Input übergeben bekommt und der Output hingeschickt wird. Die Devices selbst werden in eigenen Klassen definiert, die durch das Shellsript initialisiert werden.

2.3 Rekonstruktion der Teilchenflugbahnen

Bevor das mCBM-Experiment in Betrieb genommen wurde, lag der Schwerpunkt der CBM-Software auf der Simulation von Daten und der qualitativ hochwertigen Rekonstruktion von Kollisionsevents. Als das mCBM-Experiment die ersten echten Testdaten lieferte, rückte aber das eigentliche Ziel des CBM-Experiments wieder in den Vordergrund: Die Auswahl von physikalisch relevanten Daten in Echtzeit [13]. Hierfür soll die gesamte Datenauswertungssoftware Online, also in Echtzeit, während des Betriebs mitlaufen. Vor der Nutzung des mCBM-Experiments fand diese Offline statt, weshalb die Notwendigkeit einer schnellen Ausführungszeit noch keine hohe Priorität hatte. Doch der Anspruch an eine Online-fähige Prozessierung führt nun dazu, dass die vorhandenen Algorithmen analysiert und optimiert werden müssen.

Um die in dieser Arbeit vorgenommenen Optimierungen besser erklären zu können, ist es sinnvoll, zunächst einen Blick auf die Prozesskette der Flugbahnrekonstruktion vom STS-Detektor zu werfen. Der in Abbildung 2.3 gezeigte Prozessablauf ist nur für Daten

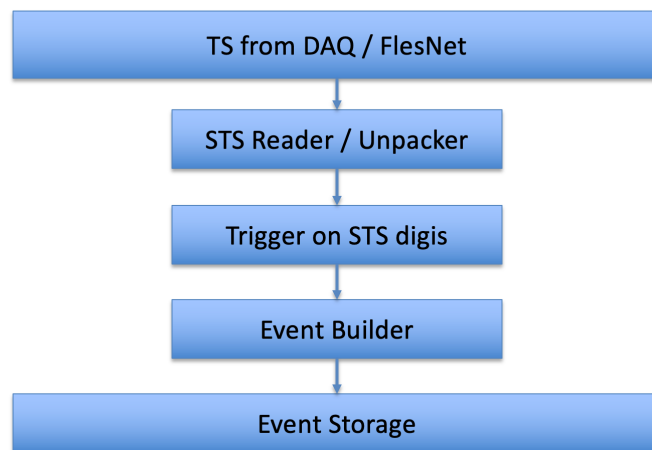


Abbildung 2.3: STS-spezifischer Teil der Online Rekonstruktionsprozesskette [13]

des STS-Detektors ausgelegt. Abgebildet ist aber nicht der gesamte Ablauf der Rekonstruktion, da zunächst nur die Zusammenarbeit dieser speziell ausgewählten Prozesse getestet werden sollte. Es folgt eine Erklärung der abgebildeten Schritte sowie der Folgeprozesse, die für die vollständige Rekonstruktion der Teilchenflugbahnen notwendig sind.

Zu Beginn werden die Rohdaten aus dem STS-Detektor zu sogenannten Timeslices (TS) zusammengefügt. Der genaue Aufbau einer Timeslice wird in Kapitel 3.2 erläutert. Anschließend werden diese vom Unpacker verarbeitet, welcher die Rohdaten in C++ Datenstrukturen, sogenannte *Digis* (Erklärung in Kap. 3.1), umwandelt. Diese Digis werden nun als Grundlage für das Finden von Triggern genutzt. Dieser Schritt dient dazu, das Hintergrundrauschen herauszufiltern und interessante Digis zu selektieren. Das Filtern des Rauschens beruht auf der Annahme, dass im Falle einer Kollision mehr Teilchen vom Detektor registriert werden und somit eine zeitlich abhängige Häufung von Digis zu erkennen ist. Entsprechend werden die Digis anhand ihrer Zeitstempel nach Clustern in der Zeit durchsucht. Ein solcher Cluster wird als *Trigger* bezeichnet.

Der *Event Building* Schritt, der auf das Finden der Trigger folgt, beschreibt die Umwandlung der Digis eines Clusters in die Datenstruktur *CbmStsEvent* (Event). Laut Abbildung 2.3 folgt in der aktuellen Konfiguration nur noch das Schreiben der Events in den Speicher. Damit ist die Rekonstruktion allerdings noch nicht vollständig abgeschlossen. Bisher wurden nur Digis aussortiert, aber noch keine Flugbahnen errechnet.

Bevor die Flugbahnen letztlich bestimmt werden können, muss noch der sogenannte *Hitfinder* Algorithmus angewendet werden. Dieser untersucht alle Digis eines Events und

berechnet daraus, wo genau im STS-Detektor ein echtes Teilchen registriert wurde. Der Hitfinder wandelt die vorhandenen Daten in ein 4-dimensionales Objekt um, einen sogenannten *Hit*. In einem Hit sind Informationen über die Position (X-Achse, Y-Achse, Z-Achse) im Detektor sowie der Zeitpunkt des aufgezeichneten Teilchens zusammengefasst. Der Hitfinder kann sowohl mit Events, als auch einzelnen, ungefilterten Digis genutzt werden. Er ist der letzte STS-spezifische Algorithmus in der Prozesskette.

Obwohl das Tracking der Teilchen die primäre Aufgabe des STS-Detektors ist, sind auch andere Detektoren dazu in der Lage Hits aus ihren Daten zu produzieren. Nachdem alle Hits gefunden wurden, kann der *Track Reconstruction* Algorithmus die Flugbahnen der Teilchen bestimmen. Für eine möglichst präzise Berechnung, werden alle verfügbaren Hits verschiedener Detektoren verwendet.

Die Wichtigkeit des Unpackers wird in Betracht aller beteiligten Algorithmen klar ersichtlich. Der Triggerfinder sowie die nachfolgenden Prozessschritte können erst dann genutzt werden, wenn Digis vorliegen. Hat der Unpacker einen niedrigeren Datendurchsatz als die folgenden Algorithmen, wird er zum Flaschenhals der Prozesskette.

2.3.1 Task-basierte Umsetzung

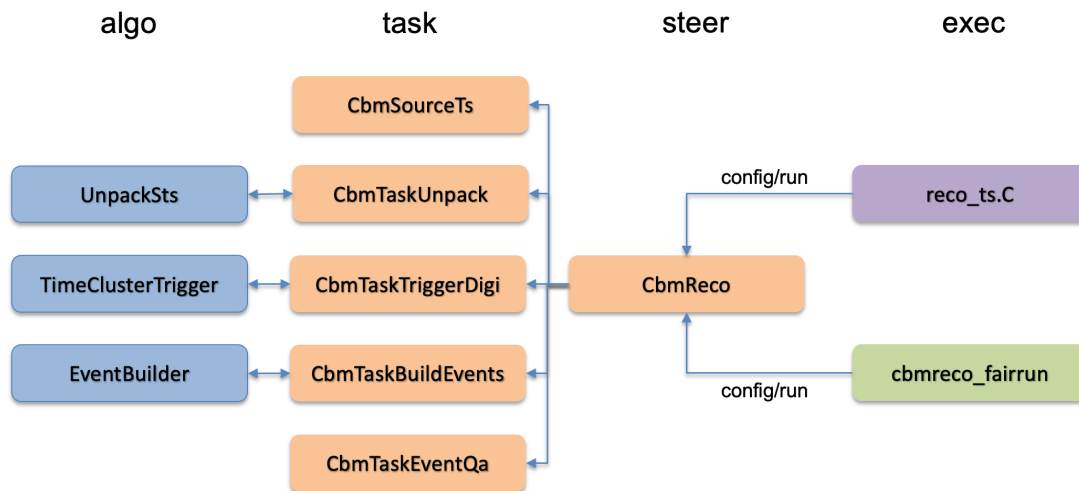


Abbildung 2.4: Task-basierte Steuerung der Online STS-Prozesskette [13]

Im Verlauf dieser Arbeit werden aufgrund verschiedener Faktoren verschiedene Ansät-

ze von Laufzeitmessungen genutzt. Um diese Faktoren besser nachvollziehen zu können, bietet Abbildung 2.4 einen Überblick über die Implementierung Prozesskette. Hier wird die Task-basierte Steuerung der Algorithmen bis zum Event Building beschrieben. Die Algorithmen des Unpackers, Triggerfinders und Event Builders sind so abgekapselt, dass sie immer auf den gleichen, standardisierten Eingabeparametern funktionieren. Für den Unpacker-Algorithmus ist dies eine Microslice, die in Abschnitt 3.2 genauer erklärt wird. Wie oben erwähnt benötigt der Triggerfinder Digis, die aktuell in Form eines C++ STL-Vectors übergeben werden. Der Event Builder bekommt ebenfalls eine gewisse Menge an Digis in einem STL-Vectors übergeben, welche er dann in ein *CbmStsEvent* umwandelt.

In der realen Anwendung wird allerdings nie nur eine einzige Microslice vom Unpacker verarbeitet. Folglich entsteht auch nicht nur ein Digivector, der weitergegeben wird und so weiter. Daher gibt es für jeden Algorithmus einen Task, der die vorliegenden Daten so aufteilt, dass die Algorithmen ihren Input im richtigen Format übergeben bekommen. Um alle Tasks miteinander zu verbinden, existiert die *CbmReco* Klasse. Diese steuert alle Abläufe, indem sie nacheinander die Tasks aufruft und die Datenübergabe zwischen ihnen regelt. Ein alternativer Steuerungsansatz wird in Kapitel 2.3.2 besprochen. Grundsätzlich ist in dieser Arbeit von der Nutzung der Task-basierten Steuerung auszugehen, wenn nicht explizit anders erwähnt.

2.3.2 MQ-basierte Umsetzung

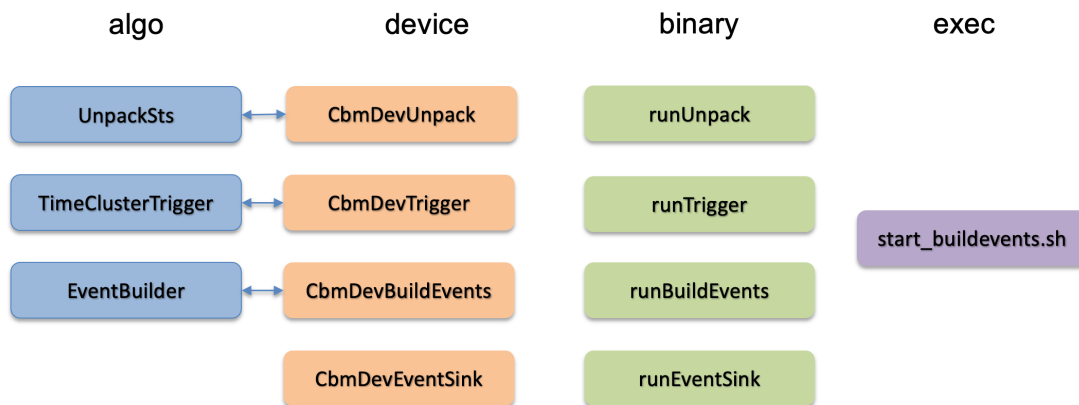


Abbildung 2.5: MQ-basierte Steuerung der Online STS-Prozesskette [13]

Das FairMQ Framework kann genutzt werden, um den Prozessablauf der in Abschnitt

2.3 vorgestellten Prozesskette zu steuern. FairMQ übernimmt dabei die Aufgabe der Steuerung, welche in der Task-basierten Implementation von der CbmReco Klasse ausgeht. Abbildung 2.5 zeigt die Unterschiede zur Task-basierten Variante. Die Tasks werden in der MQ-basierten Umsetzung durch Devices ersetzt. Die Funktionalität des zugrundeliegenden Codes wird dabei nicht verändert. Es werden lediglich Funktionen zum Versenden und Empfangen von Messages in den Device Klassen hinzugefügt. Die Steuerung durch die CbmReco Klasse wird durch das Shellscript ersetzt, welches alle Devices und zugehörigen Kommunikationswege definiert.

2.4 Parallele Programmierung

Die hohen erwarteten Datenraten von bis zu 1TB/s [2] im CBM-Experiment stellen hohe Ansprüche an die verwendete Hard- und Software. Um die Verarbeitung solcher Datenmengen bewerkstelligen zu können, wird High-End Hardware eingesetzt. Eine Schätzung von 2018 prognostiziert, dass für die Online Verarbeitung der Daten etwa 1600 Intel Xeon E7-4870 CPUs benötigt werden [14]. Der aktuelle Stückpreis für den Prozessor beträgt etwa 4400\$ [15]. Aus der Schätzung und den aktuellen Kosten errechnet sich eine Gesamtsumme von 7 mio \$ allein für die CPUs. Fortschritte in der Technik seit der Veröffentlichung von [14] vor 5 Jahren haben zwar einen kostensenkenden Effekt, die finanziellen Anforderungen bleiben dennoch ähnlich signifikant.

Um Kosten einzusparen, ist es daher von großem Interesse die Hardware so effizient wie möglich zu nutzen und auszulasten. Das Abstimmen der Software auf die vorhandene Hardware ist entsprechend von großer Bedeutung. Bei den oben beschriebenen Prozessoren handelt es sich um Multicore CPUs. Algorithmen, die auf der CPU ausgeführt werden, sollten also in der Lage sein, alle Kerne der CPU nutzen zu können. Dafür müssen sie soweit wie möglich parallelisiert werden.

2.4.1 OpenMP

OpenMP (Open Multi-Processing) ist eine Programmierschnittstelle, die unter anderem mit C++ genutzt werden kann. Über Compilerdirektiven ermöglicht OpenMP die Nutzung von Multithreading auf Systemen mit geteiltem Speicher. Durch die *#pragma* Direktiven kann im Code sehr genau kontrolliert werden, welche Programmabschnitte parallel ausgeführt werden sollen. Dabei ist die Funktionsweise von OpenMP hauptsächlich auf die Parallelisierung von Schleifen ausgerichtet. Die Anzahl der Threads, die genutzt werden sollen, kann über die Umgebungsvariable *OMP_NUM_THREADS* bestimmt werden.

Abbildung 2.6 zeigt das fork-join-Modell, auf dem die Funktionsweise von OpenMP ba-

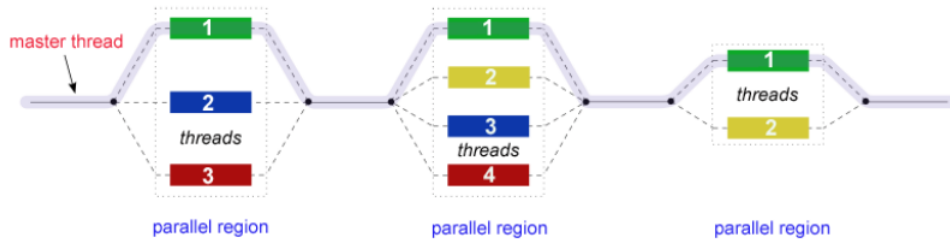


Abbildung 2.6: Das fork-join-Modell von OpenMP [16]

siert. Ein Programm wird mit einem Masterthread gestartet, welcher sequenziell läuft, bis er eine Parallelisierungsdirektive erreicht. Für die vorliegende parallele Region wird ein Team aus Threads gebildet, auf die die Arbeitslast verteilt wird (fork). Ist die Region fertig abgearbeitet wird das Multithreading beendet und der Masterthread führt sequenziell fort (join). Gibt es mehrere parallele Regionen in einem Programm wird dieser Ablauf wiederholt.

2.4.2 GPUs

Hardware

Ein anderer Ansatz gegenüber der Parallelisierung von CPU Prozessen zur effizienten Datenverarbeitung ist die Nutzung von GPUs. Grafikkarten sind darauf optimiert, viele parallele Prozesse zu verarbeiten. Dementsprechend können sie bei der Verwendung großer Datenmengen bessere Laufzeiten erreichen als CPUs. Damit dieser Vorteil im Bezug auf die hohen erzeugten Datenmengen des STS-Detektors genutzt werden kann, beschäftigt sich diese Arbeit unter anderem auch mit der Umsetzung des Unpackers auf GPUs. Um die entsprechenden Zusammenhänge später nachvollziehen zu können, wird hier die grundlegende Funktionalität einer Grafikkarte beschrieben.

Einer der großen Unterschiede zur CPU ist, dass eine GPU deutlich mehr Prozessoren besitzt. Ermöglicht wird das durch eine geringere Komplexität der Kerne. CPU Kerne sind konzipiert, um jegliche Art von Aufgaben gut durchführen zu können, während GPU Kerne hingegen deutlich kleiner und weniger Leistungsstark sind. Ihre Stärke liegt in der Ausführung einfacher Instruktionen mit einem hohen Maß an Parallelisierung, so dass die Arbeitslast auf möglichst viele Kerne verteilt werden kann. Für die effiziente Nutzung dieses Konzepts spielt die Verteilung der Daten sowie die richtige Auslastung der Kerne eine große Rolle.

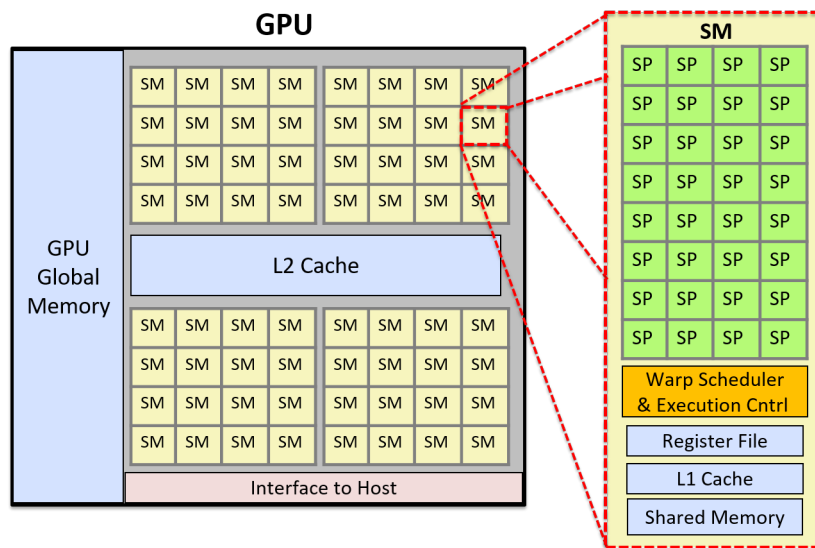


Abbildung 2.7: Beispielhafter Aufbau einer CUDA GPU [17]

Zur Veranschaulichung zeigt Abbildung 2.7 einen beispielhaften Aufbau einer CUDA GPU. Die im Folgenden beschriebene Struktur von NVIDIA (CUDA) und AMD GPUs ist grundsätzlich ähnlich, allerdings unterscheiden sich die Begrifflichkeiten. Eine Grafikkarte besteht aus vielen Streaming-Multiprozessoren (SM, NVIDIA) bzw. Compute Units (CU, AMD) Diese setzen sich wiederum aus mehreren Scalar-Prozessoren (SP) zusammen. Außerdem besitzen sie einen Scheduler zur Steuerung der SP sowie ein Shared Memory, auf das alle SP einer SM/CU geteilten Zugriff haben. Innerhalb einer SM/CU sind die SP in Warps (CUDA) bzw. Wavefronts (AMD) unterteilt. Warps und Wavefronts agieren im Lockstep, ähnlich wie Vektorregister auf einer CPU. Alle SPs in einer solchen Struktur führen also immer zur gleichen Zeit die gleichen Instruktionen aus. Warps bestehen immer aus genau 32 Threads, während es bei der AMD Architektur 64 Threads pro Wavefront sind. Die Anzahl der SM/CU sowie die Warps pro SM bzw. Wavefronts pro CU variieren von GPU zu GPU.

Software

Wenn Code auf einer Grafikkarte ausgeführt werden soll, gibt es dabei zwei wichtige Teile in einem Programm.

Das Herzstück eines Grafikkarten Programms ist der *Kernel*. Der Kernel ist die Funktion, welche die auf der GPU ausgeführten Prozesse enthält. Um die Tiefe der Parallelisierung

zu steuern, werden in der Software Threads zu Blöcken organisiert. Die Blockgröße wird vor Programmstart definiert und gibt vor wie viele Threads in einem Block zusammengefasst werden. Beim Kernelaufruf wird dann festgelegt, wie viele Blöcke den Kernelcode ausführen sollen, wodurch ein sehr präzises Ressourcenmanagement möglich ist. Innerhalb des Kernels müssen Speicherzugriffe abhängig vom Blockindex gemacht werden, damit die Blöcke unabhängig voneinander und somit parallel arbeiten können. Ein Programm was auf einer GPU ausgeführt werden soll kann beliebig viele Kernel besitzen, allerdings benötigt es mindestens einen. Dieser Teil des Codes wird auch als *Devicecode* bezeichnet, da er auf dem Device, in diesem Fall also der Grafikkarte läuft.

Das Gegenstück dazu ist der sogenannte *Hostcode*. Dieser wird auf der CPU ausgeführt und sorgt für die Steuerung des Devicecodes. Er beinhaltet die Vorbereitung, Nachbereitung, das Transferieren der Daten zum Device sowie den Kernelaufruf selbst. Für den Datenaustausch zwischen Host und Device muss GPU-Speicher allokiert und eine Kopierfunktion ausgeführt werden. Diese Aufgabe wird im unten beschriebenen XPU-Framework durch die *Buffer* Datenstruktur übernommen. Ein Buffer kann wie ein statisches Array genutzt werden, welches die Daten zwischen Host und Device verschieben kann. Dafür verfügen sie über zwei Seiten, eine für den Zugriff vom Host und eine für das Device. Die Hostseite arbeitet im CPU-Hauptspeicher, während die Deviceseite im Nutzungsfall von Grafikkarten auf dem GPU-Hauptspeicher liegt. Um die Daten Daten zwischen den Seiten auszutauschen ist ein expliziter Kopiervorgang nötig. Das Kopieren ist ein sehr teurer Schritt, der viel Zeit in Anspruch nimmt.

Im Kontext dieser Arbeit ist die Funktionsweise des Hostcodes mit dem Unpacker-Task gleichzusetzen. Die des Kernels entspricht dem Unpacker-Algorithmus.

2.4.3 XPU-Framework

Für die Programmierung von GPU Code gibt es einige verschiedene Plattformen. Die bekanntesten und am weitesten verbreiteten sind CUDA von NVIDIA und HIP von AMD. Während CUDA nur zur Programmierung von NVIDIA Grafikkarten genutzt werden kann, ist es mit HIP Code nicht nur möglich, AMD GPUs anzusteuern, sondern auch Karten von NVIDIA. Das hat den Vorteil, dass der gleiche Code auf der Hardware beider Hersteller genutzt werden kann.

Um Entwicklern mehr Unabhängigkeit von besagten Herstellern bieten zu können, wird das XPU-Framework von Felix Weiglhofer entwickelt. XPU ist ein leichtgewichtiges Framework, was die Funktionsweisen von HIP, CUDA und SYCL vereint. Dies ermöglicht es Programmierern nicht nur einen Code für verschiedene GPU Hardware zu nutzen, son-

dern diesen Code auch auf CPUs zu verwenden. XPU soll im CBM-Experiment als Basis für die GPU Prozessierung genutzt werden und ist zum Zeitpunkt dieser Arbeit weiterhin in Entwicklung [18]. Die hier eingesetzte Version ist der Masterbranch vom 24.08.2023 (commit cbe0a2c).

3 STS-Unpacker

Da alle Detektoren im CBM-Experiment unterschiedlich aufgebaut sind, sind auch die ausgelesenen Daten verschieden. Entsprechend benötigt fast jeder Detektor einen individuellen Unpacker-Algorithmus, um die Rohdaten in Digis umzuwandeln. Der Fokus dieser Arbeit liegt dabei auf dem Unpacker für den STS-Detektor. Da sich die Struktur der Daten vom mSTS zum STS voraussichtlich nicht mehr stark verändern wird, kann der Unpacker für den STS-Detektor anhand der Daten des mCBM-Experiments optimiert werden.

Nachdem alle wichtigen Grundlagen besprochen wurden, gilt es nun also, den Unpacker des STS-Detektors genauer zu betrachten. Dessen Aufgabe ist es, die Rohdaten des Detektors in Digis umzuwandeln. Dieses Kapitel beschäftigt sich damit, die Funktionsweise des Unpacker-Algorithmus und der verschiedenen Unpacker-Tasks im Detail zu erläutern.

3.1 CbmStsDigi

Das *CbmStsDigi* ist die Abstraktion eines im Detektor registrierten Teilchens in einer C++ Datenstruktur. Sie bildet die Grundlage für die Folgeprozesse des Unpackers und speichert die wichtigsten Informationen, die aus dem Detektor ausgelesen werden. Die folgenden 4 Parameter sind in einem Digi enthalten [19]:

- *wint16_t fAddress*: Die Adresse des Detektormoduls, welches das Teilchen registriert hat.
- *wint16_t fChannelAndCharge*: Hält die Adresse des Channels innerhalb eines Detektormoduls, welcher das Teilchen registriert hat sowie die Ladung des Teilchens.
- *wint32_t fTime*: Zeitstempel in ns.

Channel und Charge sind in ein 16-Bit Feld komprimiert, damit das Digi auf eine Gesamtgröße von 8 Byte reduziert werden kann.

Die Erstellung eines Digis findet im Unpacker-Algorithmus statt. Die Parameter werden dabei aus den Daten einer Hit-Message sowie einigen Konstanten errechnet.

3.2 Timeslice Konzept

Die Struktur der vorliegenden Rohdaten ist von großer Bedeutung für die Arbeitsweise des Unpackers. Wie bereits in Abschnitt 2.3.1 beschrieben besteht der Unpacker aus zwei Teilen, nämlich dem Unpacker-Task und dem Unpacker-Algorithmus. Einfachheitshalber wird in dieser Arbeit die Beschreibung *Unpacker* für die Kombination der beiden Teile verwendet. Der Unpacker-Algorithmus ist so standardisiert, dass er immer eine Microslice als Input bekommt, welche im Folgenden erklärt wird. Der Task ist dafür zuständig die Eingabedaten so aufzuspalten, dass der Algorithmus immer mit genau einer Microslice ausgeführt wird.

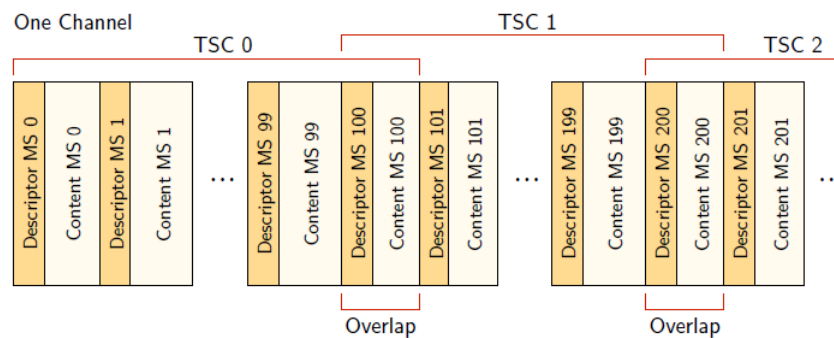


Abbildung 3.1: Inhaltliche Struktur einer Timeslice. Zeigt die Aufteilung in Timeslice Components (TSC) und Microslices (MS) [20].

Die Struktur der Rohdaten ist in Abbildung 3.1 beispielhaft gezeigt. Die Daten sind in einer verschachtelten Struktur verpackt. Die kleinste Einheit dieser Struktur nennt sich *Microslice* (MS). Der Content einer STS-Microslice enthält sogenannte *stsxyter Messages*. Für den STS-Detektor gibt es verschiedene Typen von Messages. Die wichtigsten sind die *Hit-Message* und die *TsMsb-Message*. Eine *Hit-Message* entsteht, wenn der STS-Detektor ein Teilchen registriert. Dabei wird aufgenommen, an welcher Stelle (Channel und Adresse), zu welcher relativen Zeit und mit welcher Ladung ein Teilchen aufgezeichnet wurde. Die dabei ausgelesene Zeit ist abhängig vom Clock Cycle des Detektors und der aktuellen Epoche. Die aktuelle Epoche wird dabei von *TsMsb-Message*s festgelegt, welche immer dann vom Detektor erstellt werden, wenn eine neue Epoche beginnt. In der aktuellen Umsetzung enthält eine Microslice die Daten für einen fest vorgegebenen Zeitraum. Entsprechend hängt die Größe einer Microslice von der Menge der registrierten Daten ab.

Neben dem Content besitzt eine Microslice noch einen Descriptor, der allgemeine Informationen enthält.

Die nächst größere Organisationsstruktur nach den Microslices sind Timeslice Components (TSC), später nur noch *Components* genannt. Diese bestehen aus einer festgelegten Anzahl von zeitlich aufeinander folgenden Microslices. Um einen Verlust von Daten auszuschließen, enthalten aufeinanderfolgende Components immer eine Microslice doppelt. Ansonsten werden die Microslices so in Components aufgeteilt, dass sie in der richtigen zeitlichen Reihenfolge bleiben. Components werden nur mit Microslices eines Subsystems gefüllt, damit es keine Vermischung von Daten verschiedener Detektoren innerhalb einer Component gibt. Da Microslices unterschiedlicher Subsysteme andere Unpacker erfordern, trägt dies zum einfacheren Entpacken bei.

Weiterhin folgt die Zusammensetzung der Components zu Timeslices. Auch hier wird eine festgelegte Anzahl an Components zu einer Timeslice zusammengefügt. Timeslices wiederum werden zu Timeslice-Archive-Files (TSA-Files) kombiniert, die im Speicher abgelegt werden können. Hierbei ist die Dateigröße des TSA-Files der ausschlaggebende Faktor, wie viele Timeslices zusammengeführt werden. Ein TSA-File hat meist eine Dateigröße von ca. 10GB. Die dabei abgedeckte Zeitspanne ist abhängig von der Menge der aufgezeichneten Messages, also der Größe der Microslices. Variationen in der Anzahl der Timeslices innerhalb eines TSA-Files können nur auf Microslice Level entstehen, da das Zusammenfügen in die anderen Strukturen einheitlich definiert ist. In der aktuellen Konfiguration enthält eine Timeslice 26 Components mit jeweils 101 Microslices, wobei nur 5 Components mit Daten des mSTS-Detektors gefüllt sind. Bis zum Start des CBM-Experiments kann diese Struktur noch weiterhin verändert werden, sodass Anforderungen von Hardwareseite der Detektoren erfüllt werden können.

Im Blick auf die Task-basierte Umsetzung des Unpackers entsteht der folgende Zusammenhang: Beim Start der Rekonstruktionskette wird das CbmReco Steuerungsprogramm aufgerufen und ein oder mehrere TSA-Files an dieses übergeben. CbmReco liest die einzelnen Timeslices aus dem TSA-File und übergibt diese nacheinander an den Unpacker-Task. Dort werden die Timeslices so entpackt, dass der Algorithmus korrekt mit individuellen Microslices ausgeführt wird.

3.3 Unpacker-Algorithmus

In diesem Abschnitt soll ein Überblick über den Unpacker-Algorithmus vermittelt werden. Zu Beginn dieser Arbeit existierten zwei verschiedene Implementierungen des Algo-

rithmus mit der gleichen Funktionsweise. Die Standardimplementierung wird in diesem Kapitel besprochen, während auf die angepasste Variante in Abschnitt 3.4.4 eingegangen wird. Alle im Folgenden erklärten Ansätze, abgesehen von 3.4.4, nutzen die hier erklärte Implementierung.

Der Unpacker-Algorithmus ist so aufgebaut, dass er eine Microslice entgegennimmt, diese sequenziell abarbeitet und anschließend einen STL-Vector mit den entstandenen Digis zurückgibt. Der Algorithmus ist in Form von Pseudocode in 7.1 vereinfacht dargestellt.

Das Prinzip des Unpacker-Algorithmus ist gradlinig und einfach. Er geht nacheinander alle Messages der gegebenen Microslice durch und verarbeitet diese je nach Messagety. Liegt eine Hit-Message vor, wird ein Digi erstellt und die 4 Parameter Adresse, Channel, Zeit und Ladung errechnet. Dazu werden verschiedene Daten benötigt. Adresse und Channel werden aus Hardwareinformationen errechnet, die im Unpacker-Task definiert werden. Die Ladung hingegen ist eine Information, die in der Hit-Message selbst enthalten ist. Der Zeitstempel setzt sich aus vielen verschiedenen Faktoren zusammen und ist die komplizierteste Berechnung. Die Grundlage bilden die Startzeit der Time- und Microslice sowie der Zeitstempel der Message selbst. Zusätzlich spielt die aktuelle Epoche, wie in Abschnitt 3.2 beschrieben, eine Rolle in der Kalkulation des Digi Zeitstempels. Der Start einer neuen Epoche wird mit einer TsMsb-Message festgehalten. Wird ein Digi aus einer Hit-Message erstellt, so ist der Zeitparameter immer abhängig von der aktuellen Epoche, also der letzten vorhergehenden TsMsb-Message. Epochen werden in Cycles gezählt, sodass nicht nur die aktuelle Epoche, sondern auch der aktuelle Cycle essenziell ist. Ein neuer Cycle beginnt immer dann automatisch, wenn die Epoche einer TsMsb-Message kleiner ist als die der vorherigen TsMsb-Message. Daraus folgt, dass bei der Erstellung eines Digis nicht nur eine Abhängigkeit zur letzten TsMsb-Message, sondern zu allen vorher verarbeiteten TsMsb-Message besteht. Diese Abhängigkeiten erschweren die Thematik der Parallelisierung des Unpacker-Algorithmus unterhalb der Microslice Ebene (innerhalb einer Microslice). Epochen sowie Cycles spielen nur innerhalb einer Microslice eine Rolle, sodass die Abhängigkeiten bei der Parallelisierung auf Microslice Ebene nicht von Relevanz sind. Damit es immer eine Basisepoche gibt, mit der die ersten Digis erstellt werden können, gibt die Struktur der Microslices vor, dass immer eine TsMsb-Message an zweiter Stelle in der Microslice steht. Diese wird daher noch vor Beginn der Message-Schleife verarbeitet. Epochen und Cycle sowie einige zusätzliche Informationen zum Berechnen des Zeitstempels sind in einer C++ Structure *time* zusammengefasst, welche allen verarbeiteten Messages zur Nutzung und Anpassung zur

Verfügung steht. Ein erstelltes Digi wird in den Output STL-Vector eingefügt, der nach Verarbeitung aller Messages vom Algorithmus an den Task zurückgegeben wird.

3.4 Aktueller Stand des Unpackers

In diesem Kapitel wird ein Überblick zur Implementierung des Unpackers zum Zeitpunkt vor Beginn dieser Arbeit gegeben. Der Unpacker wird bereits seit einiger Zeit weiterentwickelt und optimiert. Durch die ständigen Fortschritte sind verschiedene Implementierungen des Unpackers entstanden, die jeweils unterschiedliche Herangehensweisen aufweisen. Für die ersten drei Varianten ist der Unpacker-Algorithmus identisch, es variiert lediglich die Umsetzung des Tasks.

3.4.1 Task-basiert sequenziell

Die Task-basierte sequenzielle Umsetzung des Unpacker-Tasks ist der historisch älteste und gradlinigste Ansatz. Die Aufgabe des Tasks ist, die ihm übergebene Timeslice so zu entpacken, dass alle enthaltenen Microslices vom Unpacker-Algorithmus verarbeitet und die entstandenen Digis in einem STL-Vector gesammelt werden. Die Kernfunktionalität der sequenziellen Implementierung ist in Form von Pseudocode in 7.2 abgebildet. Der Ansatz ist sehr intuitiv. Es wird mit einer for-Schleife über alle Components in der gegebenen Timeslice iteriert. Mit einer weiteren for-Schleife werden alle Microslices, die in den jeweiligen Components enthalten sind, durchlaufen. In der inneren Schleife wird für jede Microslice der Unpacker-Algorithmus aufgerufen. Der Algorithmus gibt einen STL-Vector mit den erstellten Digis zurück, welcher anschließend in einen anderen Vector integriert wird, sodass alle Digis der Timeslice aufeinanderfolgend in einem 1-dimensionalen Vector gespeichert sind. Dieser Vector wird vom Task zurückgegeben.

Wie der Name bereits sagt, wird dieser Ansatz sequenziell ausgeführt, sodass nur ein Thread alle Arbeit übernimmt. Dadurch wird die Hardware nicht optimal ausgelastet und die Laufzeit des Unpackers ist zu hoch für eine Online Verarbeitung der Daten.

3.4.2 Task-basiert parallel

Um das Ziel der Online Prozessierung erreichen zu können, wurde der Unpacker ausgehend von der Basis des sequenziellen Ansatzes optimiert. Zuletzt stellte Maximilian Zick die neuesten Verbesserungen in seiner Bachelorarbeit aus dem Jahr 2022 vor. In [21] geht er auf die bereits in Abschnitt 2.4 erwähnten Vorteile einer Parallelisierung ein und im-

plementiert diese für den Unpacker. Die grundlegende Idee ist, sowohl die Verarbeitung der Components als auch die der Microslices innerhalb der Components zu parallelisieren. So kann beispielsweise ein Thread pro Component oder ein Thread pro Microslice genutzt werden. Dies ist generell dadurch möglich, dass die Microslices untereinander keine zeitlichen Abhängigkeiten besitzen.

In 7.3 ist der Hauptbestandteil der Implementierung aus [21] in Form von Pseudocode gezeigt. Die grundlegende Struktur der verschachtelten for-Schleifen bleibt in dieser Variante erhalten. Allerdings wird mit Hilfe des OpenMP Frameworks nun eine Parallelisierung der beiden Schleifen eingefügt. Die beiden Pragma Direktiven weisen den Compiler an, die Schleifen mit einer vorher festgelegten Zahl an Threads auszuführen. Wenn mehrere Threads gleichzeitig auf die gleiche Speicheradresse zugreifen, kommt es entweder zu Raceconditions oder die Speicherzugriffe müssen im Code geregelt werden. Dafür kann z.B. der *OMP_ATOMIC* Befehl genutzt werden, der einem Thread garantiert, dass er alleine auf eine Speicheradresse schreibt. Dadurch können allerdings Wartezeiten entstehen, die den parallelen Ablauf aufhalten. Deshalb sind parallele Threads genau dann sinnvoll, wenn jeder Thread unabhängig und ungehindert von anderen Threads Daten schreiben kann. Dies ist hier dadurch ermöglicht, dass ein extra STL-Vector (space) als Zwischenspeicher angelegt wird. Der space Vector enthält Vektoren von Digis und zwar genau so viele, wie es Threads gibt. So kann jeder Thread seine Ergebnisse mit Hilfe seiner ThreadID in den space Vector schreiben, ohne dass es zu Raceconditions oder Verzögerungen kommt. Anschließend müssen die Daten noch vom Zwischenvector in den Output Vector kopiert werden, welcher am Ende vom Unpacker-Task zurückgegeben wird. Dafür wird zunächst die Größe des Output Vectors an die Anzahl der entstandenen Digis angepasst. Dazu wird die *std::vector<T,Allocator>::resize* Funktion genutzt. Anschließend werden Offsets berechnet und die Daten entsprechend der Offsets kopiert.

Zwar fällt insgesamt durch den zusätzlichen Kopierschritt mehr Arbeit an, allerdings gewinnt die Parallelisierung des Unpackings signifikant mehr Zeit, als dieser Schritt benötigt. Entsprechend ist die parallele Umsetzung des Unpackers bereits ein guter Schritt in Richtung der Online Prozessierung im CBM-Experiment.

Der Ausblick in [21] beschreibt einen möglichen Ansatz zur weiteren Optimierung der Implementierung. Laufzeitmessungen haben ergeben, dass das *Resize* des Output Vectors einen Großteil der Gesamtlaufzeit einnimmt. Kann dafür eine Alternative gefunden werden, können nochmals erhebliche Verbesserungen der Laufzeit erzielt werden. Daher wird der gegebene Vorschlag in Abschnitt 4.1 als erste Optimierung im Rahmen dieser

Arbeit untersucht.

3.4.3 MQ-basiert

Eine andere Möglichkeit zur Parallelisierung des Unpackings basiert auf dem FairMQ Framework. FairMQ wird dabei anstelle von OpenMP genutzt. Das Framework ist nicht primär auf Parallelisierung auf Thread-Ebene ausgelegt, sondern wird eher dazu verwendet, Arbeitsabläufe in verteilten Systemen zu steuern. Die Parallelisierung eines einzelnen Prozesses ist allerdings dennoch möglich.

Da FairMQ bereits im ALICE Projekt am CERN eingesetzt wird [22] und das CBM-Experiment softwaretechnisch viele Ähnlichkeiten mit ALICE besitzt, wurde bereits der in Abbildung 2.3 dargestellte Teil des Rekonstruktionsprozesses mit FairMQ umgesetzt. Für jeden Prozess der Kette muss mindestens ein Device erstellt werden. Die Funktionalität der Devices ist im jeweiligen Devicecode festgelegt. Zusätzlich zu den in Abbildung 2.5 dargestellten Devices gibt es noch ein Sampler Device, welches die Timeslices aus den TSA-Files entnimmt und für deren Bereitstellung sorgt. Grundsätzlich können beliebig viele Devices erstellt werden. In der Grundkonfiguration des Shellscripts werden aber immer gleich viele Unpack, Trigger und BuildEvents Devices erstellt, sodass immer eine 1 zu 1 Beziehung zwischen den Tasks besteht. Der Devicecode für die MQ-basierte Variante ist nahezu identisch mit dem in Abschnitt 3.4.1 beschriebenen Task-basierten sequenziellen Code. Der Unpackercode wird lediglich durch einige Funktionen zur Kommunikation mit Messages (Senden und Empfangen) ergänzt.

Diese Variante des Unpackers wurde bisher noch nicht näher im Sinne der Unpacker Optimierung untersucht. Entsprechend gibt es keine Laufzeitvergleiche zwischen der MQ-basierten und den Task-basierten Methoden.

3.4.4 Task-basiert GPU

Zu Beginn dieser Arbeit existiert außerdem eine Implementierung des Unpackers, die zur Ausführung auf GPUs ausgelegt ist. Es ist der erste Ansatz, der die Rechenleistung von GPUs für den Unpacker Prozess zu nutzen versucht. Der Code basiert auf dem XPU-Framework, welches vorher nur beim Hitfinder und den zugehörigen Sortierverfahren eingesetzt wurde. Der Fokus dieser Umsetzung liegt hauptsächlich darauf, eine funktionierende Basis für einen GPU Unpacker zu bilden.

Die grundlegende Funktionsweise gleicht der des parallelen Task-basierten Ansatzes. Allerdings ist sie entsprechend des XPU-Frameworks angepasst, um die Ressourcen der GPUs nutzen zu können. Dazu gehört z.B. ein Wechsel der Datenstrukturen vom STL-Vector zu XPU Buffern, damit die Daten auf den Speicher der GPU kopiert werden können.

Der Hostcode dieser Umsetzung befasst sich mit der Erstellung und Befüllung der Buffer, um der GPU bei der Kernelausführung den Zugriff auf die Daten zu ermöglichen. Der Kernel besteht hier aus dem Unpacker-Algorithmus. Die Funktionsweise des Kernels gleicht der des CPU Unpacker-Algorithmus. Der Code ist allerdings so angepasst, dass die ThreadIDs für einen Indexzugriff auf die Daten in den Buffern genutzt werden. Dies ermöglicht Parallelität, da so jeder Thread mit individuellen Daten arbeiten kann. Die Parallelität ist allerdings nur in eingeschränkter Form vorhanden. Der Kernelaufruf bestimmt nämlich, dass nur so viele Threads wie Microslices verwendet werden sollen. Essenziell ist dies also ebenfalls eine Parallelisierung auf Microslice Ebene.

Das bedeutet, dass dieser Ansatz die Rechenleistung der Grafikkarte nicht effizient ausnutzt, da er nicht auf für die Nutzung einer GPU optimiert ist. Daher befassen sich Kapitel 4.3 und 4.4 mit genau dieser Problemstellung und passenden Lösungsansätzen.

3.5 Testumgebung

In diesem Kapitel werden die für Messungen genutzte Hardware sowie die Testdaten beschrieben.

3.5.1 Hardware

Nicht alle Laufzeitmessungen dieser Arbeit wurden auf dem gleichen Server vorgenommen. Hardware und Auslastung können zwischen Testservern variieren, weshalb keine serverübergreifenden Vergleiche betrachtet werden. Da die eingesetzte Hardware dennoch eine entscheidende Rolle bei Laufzeitvergleichen spielt ist im Folgenden beschrieben, welche Hardware in den jeweiligen Servern eingesetzt ist. An allen Ergebnissen dieser Arbeit wird vermerkt, auf welchem Server die Messungen durchgeführt wurden.

FLES Node9

Die in dieser Arbeit vorgenommenen Laufzeitmessungen fanden auf zwei verschiedenen Servern statt. Zu Beginn der Arbeit wurden die Laufzeiten des Task-basierten Unpackers auf dem zum CBM-Experiment gehörigen FLES (First-Level-Event-Selector) Cluster ge-

messen. Im Speziellen wurde Node9 des Clusters genutzt. Dort verbaut sind zwei *Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz* mit jeweils 32 Threads.

Asustest04

Als später der MQ-basierte Code getestet werden sollte, stellte sich heraus, dass FairMQ in der Standardausführung eine Grafikweiterleitung per X11-forwarding benötigt. Da FLES Node9 keine X11 Unterstützung bietet, konnte dieser Server nicht für die Tests genutzt werden. Stattdessen wurde ab diesem Zeitpunkt der zur Arbeitsgruppe von Prof. Dr. Lindenstruth gehörige Asustest04 zum Messen der Laufzeiten genutzt. Im Asustest04 ist folgende Hardware verbaut:

- Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
- AMD Radeon VII
- NVIDIA GeForce RTX 2080 Ti

Der Server besitzt 2 CPU Sockel, auf denen jeweils eine Intel Xeon CPU sitzt. Jede der beiden CPUs besitzt 16 physische Kerne mit jeweils 2 Threads. Insgesamt stehen dem Server also 64 Threads zur Verfügung.

	NVIDIA GeForce RTX 2080 Ti	AMD Radeon VII
SM / Compute Units	68	60
Speichergröße	11 GB	16 GB
Speicherbandbreite	616 GB/s	1024 GB/s

Tabelle 3.1: Technische Daten der Test-GPUs [23][24]

Tabelle 3.1 kann entnommen werden, dass der Radeon VII im Gegensatz zur RTX 2080 Ti ein größerer Speicher sowie eine höhere Speicherbandbreite zur Verfügung stehen, welche allerdings nur bei voller Auslastung einen Vorteil bringen. Um im späteren Verlauf der Arbeit die Effizienz der Grafikkarten einschätzen zu können, ist es wichtig die Anzahl der verfügbaren Threads zu bestimmen. Diese errechnet sich aus der Anzahl SM/CU, Anzahl an Warps pro SM [25] bzw. Wavefronts pro CU [26] und der Größe der Warps/Wavefronts.

$$AnzahlThreadsRadeon = CU * WavefrontsProCU * WavefrontGröße$$

$$\text{AnzahlThreadsRadeon} = 60 * 40 * 64 = 153.600$$

$$\text{AnzahlThreads2080} = SM * WarpsProSM * WarpGröße$$

$$\text{AnzahlThreads2080} = 68 * 32 * 32 = 69.632$$

Die RTX 2080 Ti hat also 69.632 Threads zur Verfügung, die Radeon VII hingegen 153.600.

3.5.2 Testdaten

Zum Testen der Ergebnisse werden in dieser Arbeit echte Daten zweier mCBM Runs genutzt. Zum einen handelt es sich dabei um Run 2391, bei dem Nickelionen mit einer durchschnittlichen Kollisionsrate von 400 kHz kollidierten. Zum anderen wurde Run 2456 gewählt, bei dem Goldionen mit einer etwas niedrigeren durchschnittlichen Kollisionsrate von 300-400 kHz aufeinander trafen. Bei einem Run über Minuten bis zu mehreren Stunden fallen sehr hohe Datenmengen an. Zur einfacheren Verarbeitung der Detektordaten, werden diese deshalb auf mehrere TSA-Files aufgeteilt. Aufgrund langer Laufzeiten und damit verbundenem Mehraufwand werden in dieser Arbeit nur einzelne TSA-Files zum Testen verwendet. Entsprechend werden nicht die gesamten Daten eines Runs zum Testen genutzt, sondern nur ein kleiner Teil.

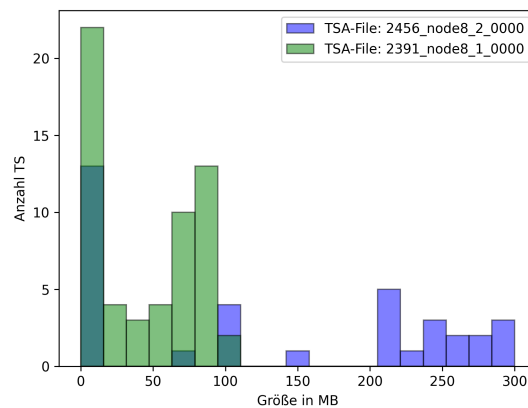


Abbildung 3.2: Größenverteilung der Timeslices in TSA-Files 2456_node8_4_0000.tsa und 2391_node8_1_0000.tsa

Der Name eines TSA-Files enthält die Bezeichnung des FLES-Entry-Nodes, von welchem es erstellt wurde sowie eine Nummerierung, die angibt, das wievielte TSA-File dieses Nodes es ist. Für diese Arbeit wurde für Run 2391 das TSA-File *2391_node8_1_0000.tsa*

und für Run 2456 *2456_node8_4_0000.tsa* gewählt. Im weiteren Verlauf ist der Verweis auf einen Run mit dem auf das gegebene TSA-File gleichzusetzen.

Testdaten aus zwei Runs zu nutzen ist sinnvoll, da die Daten verschiedener Runs sich stark unterscheiden können. So war der Gedanke bei der Wahl der Runs, dass bei Run 2391 aufgrund der höheren Kollisionsrate auch eine höhere Dichte der Daten vorliegt. Für die gewählten TSA-Files ist dies aber nicht der Fall. Dies kann an verschiedenen Faktoren liegen, wie z.B. dem Zeitpunkt im Run, den sie aufgezeichnet haben. Die beiden gewählten TSA-Files sind dennoch interessant im Bezug auf die vorliegende Datendichte. Sie sind etwa 10 GB groß, unterscheiden sich allerdings stark in der Menge enthaltener Timeslices. Run 2391 enthält 58 TS, Run 2456 hingegen nur 35. Die Größenverteilung der Timeslices in den beiden Runs ist in Abbildung 3.2 zu sehen. Da beide TSA-Files ähnlich groß sind, aber unterschiedlich viele Timeslices enthalten, variiert folglich die Größe der Timeslices. Wie in Abschnitt 3.2 erklärt, ist die Anzahl der Timeslices in einem TSA-File abhängig von der Größe der enthaltenen Microslices. Die Microslices in Run 2456 müssen also deutlich größer sein und daher mehr Daten enthalten als die von Run 2391. Dies ist eine gute Eigenschaft, um das Verhalten der Algorithmen unter verschiedensten Bedingungen testen zu können.

Da der Fokus dieser Arbeit aber speziell auf dem STS-Unpacker liegt, ist es wichtig zu verifizieren, welche Daten der TSA-Files auch von diesem verarbeitet werden. Messungen ergeben, dass Run 2391 etwa 2,7 GB an STS-Daten in Form von Microslices enthält, während es in Run 2456 etwa 4,8 GB sind. In jeder Timeslice sind genau 5 STS-Components mit jeweils 101 Microslices enthalten. Die durchschnittliche Größe der STS-Microslices in den beiden vorliegenden TSA-Files lässt sich daher wie folgt errechnen:

$$AnzahlStsMsInTsa = AnzahlTs * AnzahlStsCompInTs * AnzahlMsInComp$$

$$AnzahlStsMsIn2391 = 58 * 5 * 101 = 29.290$$

$$AnzahlStsMsIn2456 = 35 * 5 * 101 = 17.675$$

$$AvgStsMsGrößeTsa = \frac{StsDatenInTsa}{AnzahlStsMsInTsa}$$

$$AvgStsMsGröße2391 = \frac{2.700.000KB}{29.290} \approx 92KB$$

$$AvgStsMsGröße_{2456} = \frac{4.800.000KB}{17.675} \approx 272KB$$

Die durchschnittliche Größe der vom STS-Unpacker verarbeiteten Microslices in Run 2456 ist also das dreifache von Run 2391. Die Varietät der vorliegenden Daten ist also eine gute Basis zum Testen der Algorithmen.

3.5.3 Hinweis zu Laufzeitmessungen

Es ist bei allen Laufzeitmessungen zu beachten, dass diese auf Testservern durchgeführt werden, die auch in Benutzung anderer Entwickler sind. Vereinzelt kann es daher zu abweichenden Laufzeiten kommen, wenn die Kapazitäten des Servers während eines Tests bereits ausgelastet sind. Für die Laufzeitmessungen in dieser Arbeit wurde versucht, dies zu vermeiden, indem mit dem Linux Befehl *htop* die aktuelle Auslastung der Ressourcen überprüft und Laufzeiten immer mehrfach gemessen wurden.

4 Optimierung und Analyse des parallelen Unpackers

In diesem Kapitel werden alle vorgenommenen Verbesserungen und Analysen erörtert. Die dabei entstandenen Änderungen am Code sind im Github Verzeichnis unter <https://github.com/fweig/cbmroot> gesammelt.

4.1 Optimierung des Task-basierten parallelen Unpackers

Durch die in [21] beschriebenen Verbesserungen am Task-basierten Unpacker ist es sinnvoll, den dort entstandenen parallelisierten Code als Ausgangspunkt für diese Arbeit zu nutzen. Zur Vereinfachung werden die hier betrachteten Unpacker Varianten in diesem Abschnitt nicht weiter als *Task-basiert* beschrieben. Wie bereits in Abschnitt 3.4.2 erklärt ist ein erster Ansatz der weiteren Optimierung, die effizientere Umsetzung der Resize-Funktion zur Anpassung der Größe des Output Vectors. Bevor dieser Teil des Codes genauer betrachtet wird, ist es jedoch angebracht, noch einen Blick auf die Idee hinter der Umsetzung des parallelen Unpackers zu werfen.

4.1.1 Parallelisierung über Microslices

Problem

In [21] wird beschrieben, warum eine Parallelisierung auf Ebene der Microslices effizienter ist, als auf Component Ebene. Der Grund dafür ist, dass nur 5 der 26 Components mit Daten für den STS-Unpacker gefüllt sind, während jede Component hingegen 101 Microslices enthält, wie bereits in Abschnitt 3.2 diskutiert. Wenn ausschließlich über die Component Schleife parallelisiert wird, werden effektiv nur 5 Threads beschäftigt, während bis zu 21 nichts zu tun haben. Jeder der 5 Threads würde die 101 Microslices entpacken, die in der entsprechenden Component enthalten sind. Dies führt bei Systemen mit mehr als 5 Kernen zu einer schlechten Auslastung. Zusätzlich kann es passieren, dass die

Größe der Components stark variiert. Die Gesamtlauzeit des Programms ist bei der Parallelisierung über Components abhängig von der größten Component. Bei einer geringen Anzahl von Components, kann dadurch schnell ein Ungleichgewicht entstehen, bei dem im Endeffekt nur ein Thread den Großteil der Arbeit leistet. Wird hingegen über die innere (Microslice) Schleife parallelisiert, ist die Gesamtlauzeit nicht mehr so anfällig dafür, dass ein Thread höheren Arbeitsaufwand hat als die anderen. In einer Timeslice gibt es aktuell 5 Components mit jeweils 101 Microslices, also insgesamt 505 Microslices zu verarbeiten. Wird das Unpacking auf einem Server mit 64 Threads durchgeführt, gibt es auf jeden Fall weniger Threads als Microslices. Wenn ein Thread eine große Microslice bearbeitet, können Threads, die ihre Microslice schneller entpackt haben, einfach mit einer anderen Microslice weitermachen und müssen nicht auf den langsamen Thread warten.

In der aktuellen Implementierung von Maximilian Zick (siehe Pseudocode 7.3) aus [21] ist die Parallelisierung über die Microslices zusätzlich zur Parallelisierung über Components gegeben. Zeile 7 initiiert die Parallelisierung der Components, Zeile 10 die der Microslices. An dieser Stelle tritt ein Problem auf, welches mit der korrekten Nutzung von OpenMP zu tun hat. Und zwar ist verschachteltes Parallelisieren in OpenMP standardmäßig deaktiviert [27]. In [21] gibt es keine Hinweise darauf, dass die benötigte Option `omp_set_nested` für die Verschachtelung aktiviert wurde. Die ersten Tests auf FLES Node9 zeigen, dass dort die Einstellung ebenfalls deaktiviert ist. Um diesen Code so zu nutzen, wie er vorgesehen ist, müsste also vor dem Start des Programms immer die Aktivierung der Einstellung sichergestellt werden. Ist die Verschachtelung nicht aktiviert, läuft die Verteilung der Threads wie folgt ab: Die äußere Parallelisierung bildet ein Team aus allen verfügbaren Threads, um die Arbeit der Schleife aufzuteilen. Jeder Thread stößt folglich auf die innere Parallelisierung und versucht erneut ein Team zu bilden. Da aber bereits alle verfügbaren Threads verteilt sind kann hier keine Parallelisierung mehr stattfinden. Die Arbeit der inneren Schleife wird also von dem Thread durchgeführt, der von der äußeren Parallelisierung bereitgestellt wurde. Die gewünschte Parallelisierung auf Microslice Ebene ist somit mit dieser Implementierung nicht gegeben. Stattdessen findet sie hier auf Component Ebene statt.

Lösung

Es gibt verschiedene Ansätze, die OpenMP Direktiven so umzuwandeln, dass das gewünschte Ziel erreicht werden kann. Der für diese Arbeit gewählte Ansatz entfernt die Direktive zur Parallelisierung der äußeren Schleife. Zusätzlich dazu muss der umliegende Code angepasst werden. Die Größe des space Vectors wird der Anzahl der Microslices

angegeben, sodass ein von der Microslice abhängiger Index errechnet werden kann, mit dem der space Vector befüllt wird. Durch diese Anpassung ist eine echte Parallelisierung über Microslices erreicht. Zusätzlich dazu entfällt der durch die erste Direktive erzeugte Overhead der OpenMP Parallelisierung. Der Pseudocode für diese Optimierung ist in 7.4 zu sehen.

Die gewählte Umsetzung ist einfach und gradlinig, hat allerdings einen Nachteil. Die äußere Schleife wird jetzt nicht mehr parallel, sondern sequenziell ausgeführt. Das spiegelt jedoch keinen großen Laufzeitverlust wieder, da in diesem Teil des Programms nur einige nicht kostenintensive Verzweigungen mit konstanter Laufzeit zu finden sind. Trotzdem ist es möglich, die Gesamtlaufzeit zu optimieren, indem dieser Teil ebenfalls parallelisiert wird. Die OpenMP *collapse* Funktion ermöglicht das Aufteilen der Arbeit verschachtelter Schleifen auf alle verfügbaren Threads.

Diese Optimierung wird interessant, sobald mehr CPU Kerne als Microslices in einer Component zur Verfügung stehen. In der aktuellen Umsetzung werden immer nur maximal 101 Threads benötigt um alle Microslices in einer Component zu verarbeiten. Sollten mehr Threads zur Verfügung stehen, werden diese nicht ausgelastet, da die Components nicht parallelisiert sind. Die Nutzung der Collapse Funktion ermöglicht hingegen die Parallelisierung auf Microslice und Component Ebene. Dadurch ist es theoretisch möglich bis zu 505, statt 101 Threads, gleichzeitig zu nutzen.

Aufgrund der Hardwarelimitierungen der aktuellen FLES Nodes und des Asustest Testservers ist es nicht möglich die theoretischen Vorteile dieser Optimierung zu verifizieren. Da zusätzlich ungewiss ist, ob in Zukunft CPUs mit mehr als 64 Kernen im CBM-Experiment eingesetzt werden und die Nutzung der Collapse Funktion unter den aktuellen Voraussetzungen vermutlich keinen Laufzeitgewinn einbringt, wird der Ansatz in dieser Arbeit nicht weiter verfolgt.

4.1.2 Optimierung der Resize-Funktion

Nicht insignifikant hingegen sind die Kosten der Resize-Funktion. Tests ergeben, dass sie alleine bis zu 25% der Gesamtlaufzeit des auf Microslice Ebene parallelisierten Unpackers ausmachen kann. Der Anteil steigt dabei mit höherem Parallelisierungsgrad, da das Resize selbst nicht parallelisierbar ist. Der Code drumherum kann also schneller werden, während das Resize eine konstante Laufzeit hat.

Optimierung

Der Grund für die langsame Laufzeit liegt im Prozess der Speicherzuweisung des STL-Vectors. Wird der Vector in seiner Standardkonfiguration resized, so wird Speicher für jedes geplante Element allokiert. Zusätzlich wird eine Instanz des Objekts initialisiert, für welches der Speicher allokiert wird. Dieses Verhalten ist vom standardmäßig genutzten Allocator des STL-Vectors vorgegeben und normaler Teil des Allokierungsprozesses. Die Initialisierung des CbmStsDigis nimmt in diesem Fall aber sehr viel Zeit in Anspruch und ist nicht nötig, da die Objekte, die in den Vector kopiert werden sollen, bereits existieren.

Eine Lösung für das Problem ist, einen Allocator anzulegen, der diesen überflüssigen Schritt auslässt. In 7.5 ist der entsprechende Code für ein Allocator Template zu sehen, in dem die Initialisierung des Objekts ausgelassen wird. Das Verhalten der Speicherzuweisung und -freigabe wird von der Nutzung der *New()* und *Delete()* Funktionen auf elementare Allocationsfunktionen reduziert, während die *construct* Funktion leer gelassen wird, um die Erstellung eines Objektes zu vermeiden. Diese Lösung basiert auf [28] und Kommentaren von Felix Weiglhofer.

Um diesen Allocator nutzen zu können, muss die Definition des Output Vectors entsprechend lauten:

```
1 std::vector<CbmStsDigi, NoInitAlloc<CbmStsDigi>> outDigis;
```

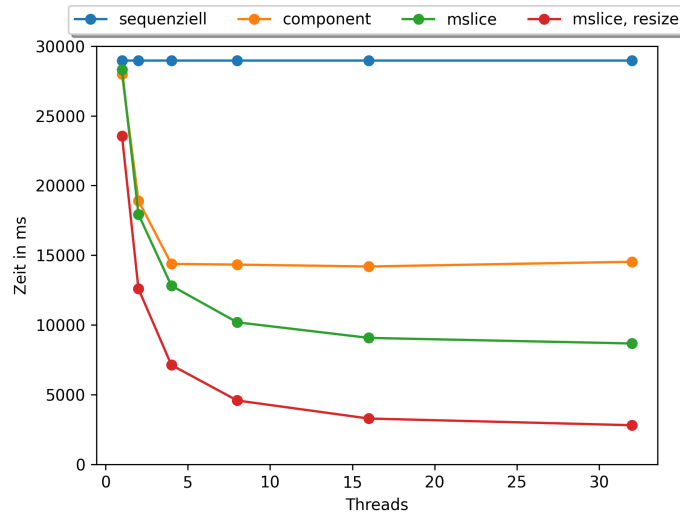
Listing 4.1: Initialisierung des NoInit STL-Vectors

Folgeproblem

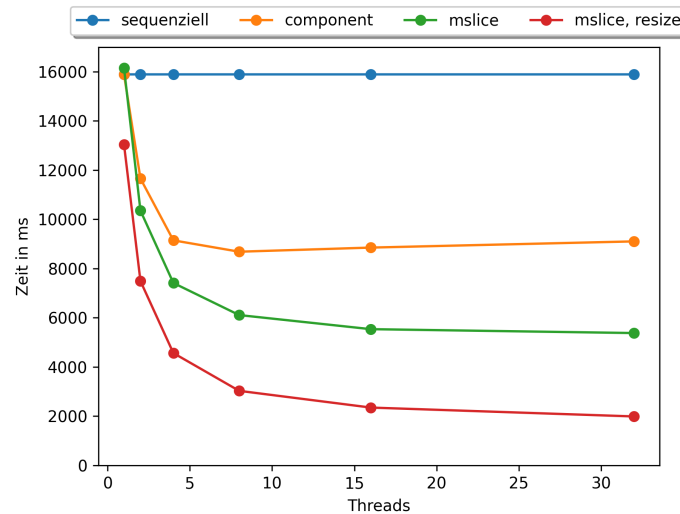
Bei dieser Lösung entsteht allerdings ein Problem, welches mit der restlichen CBM Software zu tun hat. In C++ werden STL-Vektoren mit verschiedenen Allocatoren im Template-Argument als unterschiedliche Typen behandelt. Die Folgeprozesse des Unpackers sind so implementiert, dass sie einen STL-Vector mit Standardallocator als Eingabe erwarten. Ist die Ausgabe des Unpackers also ein abgewandelter Vector, so ist sie nicht kompatibel mit dem erwarteten Datentypen. Damit die Optimierung der Resize-Funktion eingebracht werden kann, muss also nicht nur der Unpacker Code, sondern der Code der gesamten STS-Rekonstruktionskette entsprechend angepasst werden. Die Änderung des Datentypen wurde bereits eingeleitet, ist aber nicht Teil dieser Arbeit und wird deshalb nicht weiter besprochen.

4.1.3 Laufzeitanalyse

Um den Laufzeitgewinn der Optimierungen klar gegenüberstellen zu können, wird die Gesamtlaufzeit des Unpacker-Tasks gemessen. Dies geschieht, indem ein Timer innerhalb des Tasks die benötigte Zeit misst, was also der reinen Arbeitszeit des Tasks für alle Microslices des TSA-Files entspricht.



(a) TSA-File: 2456_node8_4_0000.tsa



(b) TSA-File: 2391_node8_4_0000.tsa

Abbildung 4.1: Laufzeitvergleich der CPU Varianten auf Asustest04

Da der STL-Vector mit dem angepassten Allocator nicht vom Task zurückgegeben werden kann, muss ein zusätzlicher Kopierschritt vorgenommen werden, der allerdings außerhalb der Laufzeitmessung stattfindet.

Abbildung 4.1 zeigt die Ergebnisse der oben beschriebenen Optimierungen. Alle Laufzeiten wurden für 1, 2, 4, 8, 16 und 32 Threads getestet. Dabei wurde jede Variation jeweils 3 mal ausgeführt und ein Mittelwert gebildet, um Schwankungen im Millisekundenbereich der Laufzeiten auszugleichen.

In der Abbildung sind die Laufzeiten der sequenziellen, Component-basierten und Microslice-basierten Implementierungen sowie der Microslice-basierten Variante mit der Resize Optimierung zu sehen. Die sequenzielle Umsetzung des Unpackers ist lediglich als grobe Orientierung mit aufgezeichnet, da der eigentliche Ausgangspunkt dieser Arbeit der auf Component Ebene parallelisierte Unpacker aus [21] ist. Bei der Component-basierten Variante ist zu erkennen, dass es ab 5 Threads keine Verbesserung der Laufzeit mehr zu verzeichnen gibt, da nur 5 STS-Components zur Verfügung stehen. Bei der Microslice-basierten Parallelisierung ohne Resize Optimierung lässt sich ein Anstieg der Performance bis 32 Threads beobachten. Die Resize Optimierung ändert nichts am Verlauf der Kurve, sondern zieht lediglich einen niedrigeren Startpunkt der Laufzeit nach sich. Der Grund dafür ist, dass die Resize Optimierung einen konstanten Anteil der Gesamtlaufzeit einspart, der unabhängig von der Threadzahl ist.

In Run 2456 bringt die Optimierung der Parallelisierung von Component auf Microslice Ebene bei 32 Threads einen Speedup von 1,65. Der Speedup der Microslice Variante durch die Resize Optimierung beträgt 3,09. Durch beide Optimierungen ergibt sich ein Speedup von 5,13 von der Component basierten Variante zur Microslice Parallelisierung mit Resize Optimierung. In Run 2391 ist der Gesamtspeedup mit 4,37 etwas geringer. Eine Erklärung dafür kann die in Abschnitt 4.1.1 angesprochene Variation in der Größe der Components, oder die unterschiedlichen Microslicegrößen sein.

Es wurden außerdem Tests mit 64 Threads durchgeführt, allerdings waren alle Laufzeiten signifikant langsamer als mit 32 Threads. Die Begründung dafür liegt darin, dass vermutlich die Speicherbandbreite des Servers bereits mit 32 Threads vollständig ausgelastet ist, sodass weitere Threads nur zur Verlangsamung des Gesamtsystems beitragen.

4.2 Analyse der MQ-basierten Parallelisierung

Nach der Verbesserung des Task-basierten CPU Unpackers gilt es nun einen anderen Ansatz zu analysieren. Wie in Abschnitt 2.3.2 erklärt ist es möglich, das FairMQ Framework zum Parallelisieren von Prozessen zu nutzen. Der Ausgangspunkt dieser Analyse liegt in einer abgewandelten Form der in Abschnitt 3.4.3 erwähnten Implementierung.

4.2.1 Anpassung des Codes

Der Fokus dieser Arbeit liegt ausschließlich auf dem STS-Unpacker und nicht auf der gesamten STS-Rekonstruktionskette. Die vorliegende Umsetzung ist jedoch so konfiguriert, dass ein Device pro Prozess in der Kette initialisiert wird. Da die anderen Prozessschritte allerdings für die Unpacker Analyse nicht relevant sind, gilt es diese abzuschalten. Dazu werden im Shellscript die Aufrufe der nicht benötigten Devices deaktiviert. Die Anzahl der Unpacker Devices kann nun beim Start des Shellscripts festgelegt werden. Dieses wird im Terminal mit dem Befehl `./startUnpack.sh X` aufgerufen, wobei X die Anzahl der Devices bestimmt.

Jedes Unpacker Device ist über einen Request-Channel mit dem Sampler Device verbunden, um eine Timeslice anzufordern, wenn es dazu bereit ist. Anschließend entpackt ein Device die erhaltene Timeslice und sendet die entstandenen Digis zum Triggerfinder Device, mit dem es per Push-Channel verbunden ist. In der vorliegenden Konfiguration handelt es sich um einen Channel mit einer 1:1 Beziehung. So müssten also genauso viele Triggerfinder Devices wie Unpacker Devices erstellt werden, damit jeder Unpacker Prozess seine Ergebnisse weitergeben kann. Dies ist für Performance Tests des Unpackers nicht wünschenswert, weil jedes zusätzliche Device weitere Rechenleistung in Anspruch nimmt. Da die anderen Unpacker Varianten ebenfalls gänzlich isoliert getestet wurden, wäre dies ein unfairer Nachteil für die MQ-basierte Variante. Daher wird im Shellscript die Channelverbindung von 1:1 auf n:1 zwischen Unpacker und Trigger Device umgestellt. So muss lediglich ein Trigger Device existieren, zu dem der Output der Unpacker Devices gepusht werden kann, wodurch nur ein zusätzlicher Prozess beschäftigt wird.

Um einen fairen Vergleich mit der Task-basierten Variante zu ermöglichen, sollten die Laufzeiten möglichst nach dem gleichen Prinzip gemessen werden. Da die Ansätze aber grundlegend verschieden sind, ist dies nicht ohne weiteres umzusetzen. Die Unpacker Devices arbeiten gänzlich unabhängig voneinander, sodass eine geteilte Zeitmessung nur umständlich über den Einsatz von Messages realisierbar wäre. Daher ist die einfachs-

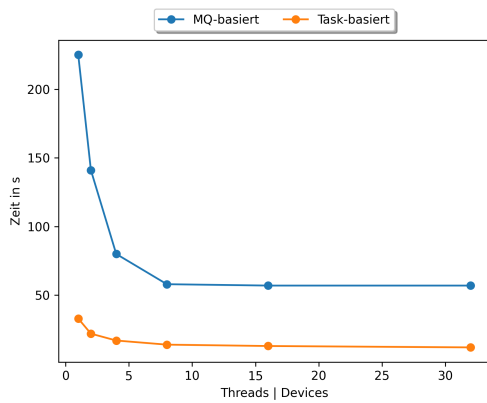
te Möglichkeit ein Logeintrag jedes Devices, sobald es die letzte Timeslice verarbeitet hat. Da ein Device allerdings nicht weiß, ob die vorliegende Timeslice die letzte ist, wird einfach nach jeder verarbeiteten Timeslice ein Logeintrag gemacht, der die Endzeit der Prozessierung festhält. Die Laufzeit eines Unpacker Devices errechnet sich aus dem Initialisierungszeitpunkt des Devices und dem Endzeitpunkt der letzten Timeslice. Um die Gesamtlaufzeit des MQ-basierten Unpackers herauszufinden, wird ein Pythonscript genutzt, welches die Logs aller Devices nach der längsten Devicelaufzeit durchsucht. Da die Devices parallel arbeiten, gibt die längste Devicelaufzeit auch die Gesamtlaufzeit des Unpackers an.

Der Vergleich dieser Zeitmessung zur in Abschnitt 4.1.3 beschriebenen wäre allerdings noch nicht ganz fair. In 4.1.3 wurde lediglich die Zeit gemessen, die innerhalb des Unpacker-Tasks benötigt wurde. Die MQ-basierte Zeitmessung berücksichtigt hingegen auch noch die Verteilung der Timeslices auf die Unpacker Prozesse. Dieser Schritt geschieht in der Task-basierten Variante auf der Steuerungsebene, also der CbmReco Klasse. Um die Zeiten der Task-basierten und MQ-basierten Umsetzungen vergleichen zu können, muss die Task-basierte Zeitmessung daher auf CbmReco Ebene stattfinden. Da ein Device immer eine ganze Timeslice verarbeitet, findet die MQ-Parallelisierung also auf Timeslice-Ebene statt, entgegen der Parallelisierung über Microslices des Task-basierten Unpackers.

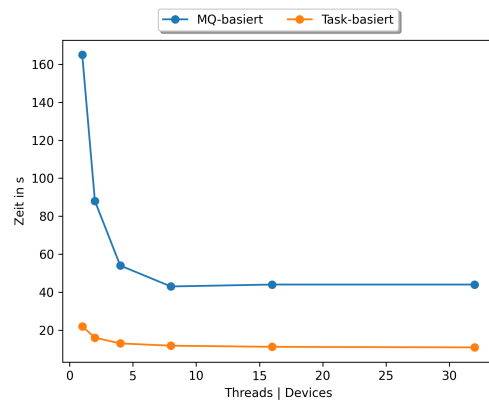
4.2.2 Laufzeitanalyse

In Abbildung 4.2 ist der Laufzeitvergleich zwischen der MQ-basierten und Task-basierten Variante für Runs 2391 und 2456 zu sehen. 4.2a sowie 4.2b zeigen dabei die Laufzeiten im realen Verhältnis, während die Achsen in 4.2c und 4.2d normalisiert dargestellt sind.

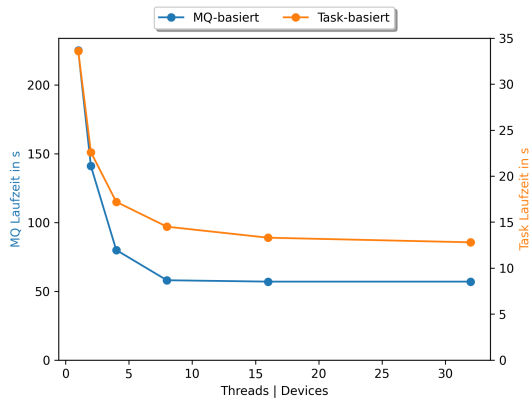
Die besten Ergebnisse erzielt die MQ-basierte Variante mit 8 Devices. Mehr Devices bringen keinen weiteren Speedup mehr. Die Auswertung der Logs zeigt, dass bei der Nutzung von mehr als 8 Devices die Unpack-Zeit einer Timeslice gleich bleibt, allerdings die Zeit zwischen zwei Timeslices in einem Device ansteigt. Mit steigender Device Anzahl steigt also die Zeit zum Beziehen einer neuen Timeslice innerhalb der Devices. Der Grund dafür konnte nicht exakt festgestellt werden. Vermutlich ist die Speicherbandbreite des Servers ausgelastet, oder die Verbindung der Unpacker Devices zum Sampler Device. Um darüber genaueren Aufschluss zu erhalten, sind weitere Tests notwendig.



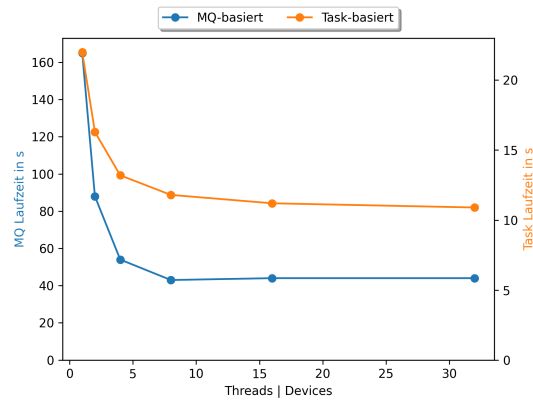
(a) TSA-File: 2456_node8_4_0000.tsa



(b) TSA-File: 2391_node8_4_0000.tsa



(c) TSA-File: 2456_node8_4_0000.tsa
Skaliert, um Speedup pro Thread zu vergleichen.



(d) TSA-File: 2391_node8_4_0000.tsa
Skaliert, um Speedup pro Thread zu vergleichen.

Abbildung 4.2: Laufzeitvergleich MQ-basiert vs. Task-basiert auf Asustest04

Beim Vergleich der Spitzenwerte ist der Task-basierte Ansatz für Run 2456 um etwa einen Faktor 2,7 und für Run 2391 etwa einen Faktor 3,9 schneller. Diese starken Abweichungen lassen sich durch die Unterschiede in den Daten erklären. Wie in Abschnitt 3.5.2 besprochen sind in Run 2391 29.290 Microslices mit STS-Daten enthalten, während es bei Run 2456 nur 17.675 sind. Logisch ist die Schlussfolgerung, dass die Parallelisierung auf Ebene der Microslices vergleichsweise besser abschneidet, wenn mehr Microslices vorhanden sind. Mehr Microslices entspricht besserer Parallelisierung und weniger Overhead. Bei der Parallelisierung über Timeslices spielt die Anzahl der Microslices hingegen keine Rolle, da sie sequenziell abgearbeitet werden.

Ein vollständigeres Bild für den Vergleich der beiden Ansätze liefern 4.2c und 4.2d. Hier ist erkenntlich, welchen Einfluss die Anzahl der Threads / Devices auf die Laufzeit haben. Der Speedup von einem zu 8 Devices in der MQ-Variante ist 3,9 (Run 2456), bzw. 3,8 (Run 2391), während der Speedup von einem Thread zu 32 bei der Task-basierten Variante nur 1,5 (Run 2456) bzw. 2 (Run 2391) beträgt. Entsprechend ist der Speedup durch mehr Threads bei MQ höher als beim Task-basierten Unpacker, obwohl dieser insgesamt deutlich schneller ist. Der Grund dafür ist in der Struktur des Unpackers zu finden. In der Task-basierten Version ist ausschließlich die Microslice Schleife parallelisiert, wohingegen in der MQ-basierten Variante die gesamte Timeslice Verarbeitung parallel ist. Messungen ergeben, dass der parallele Anteil der Task-basierten Variante etwa 19,8s für Run 2456 und 12s für Run 2391 bei einem Thread in Anspruch nimmt. Damit beträgt dessen Anteil an der Gesamtlaufzeit 62% (Run 2456), bzw. 55% (Run 2391). Der Anteil bei einer Parallelisierung über Timeslices liegt hingegen bei 100%.

Wenn die Parallelisierung über Timeslices einen so viel besseren Speedup hat, stellt sich allerdings die Frage, warum sie insgesamt so viel langsamer ist. Ein Teil der Antwort ist der Overhead durch das FairMQ Framework. Dinge wie Devices und Channel müssen vor der Nutzung zunächst noch initialisiert werden. Dies ist allerdings ein einmaliger Arbeitsaufwand, der nicht stark ins Gewicht fällt. Ein viel größeres Problem ist, dass das FairMQ Framework eigentlich auf verteilte Systeme optimiert ist. Entsprechend ist es durch die Nutzung von Messages auf die Kommunikation im Netzwerk ausgelegt. Die Umwandlung der Daten in Messages beim Senden und Empfangen ist ein kostspieliger Kopierschritt, den die Task-basierte Variante nicht benötigt.

Um das Potenzial der MQ-basierten Implementierung ausschöpfen zu können, muss festgestellt werden, warum die Anfrage neuer Timeslices ab 8 Devices so viel Zeit kostet. Allerdings ist auch dann der Einsatz von FairMQ zur Parallelisierung auf Timeslice

Ebene nicht der beste Ansatz. Sollte FairMQ in Zukunft im CBM-Experiment zur Prozesssteuerung eingesetzt werden, ist es sinnvoller die Implementierung des Unpacker Devices zu optimieren. Beispielsweise könnte ein Device GPU-basierten Unpacker Code enthalten, sodass die Parallelisierung nicht über FairMQ, sondern innerhalb des Devices gesteuert wird. In diesem Falle könnten so viele Unpacker Devices verwendet werden, wie GPUs im Server verbaut sind.

4.3 Single-Thread GPU

Mit Abschluss der Optimierung und Analyse der vorgestellten CPU Varianten bleibt noch die Untersuchung eines gänzlich anderen Ansatzes. Die Nutzung von GPUs für den STS-Unpacker wurde bisher nur oberflächlich in der in Abschnitt 3.4.4 vorgestellten XPU-Implementierung getestet. Da diese allerdings nicht auf die zugrunde liegende Hardware angepasst wurde, bestehen an dieser Stelle noch einige Optimierungsmöglichkeiten. Im Folgenden wird der bereits bestehende Code als *XPU-Ansatz* bezeichnet. Die für diese Arbeit erstellten GPU Varianten basieren zwar ebenfalls auf dem XPU-Framework, sind allerdings auf GPUs optimiert.

4.3.1 Konzept

Um eine effiziente Auslastung einer Grafikkarte zu erzielen, ist es wichtig, stark parallelisierbaren Code zu verwenden. Aufgrund der in Abschnitt 3.3 besprochenen Abhängigkeiten von Messages innerhalb einer Microslice ist die Wahl für diesen ersten Ansatz auf eine Parallelisierung auf Microslice Ebene gefallen. Die Funktionalität ähnelt der des XPU-Ansatzes, ist aber aufgrund von Updates im XPU-Framework komplett neu geschrieben. Der Hostcode beinhaltet das Bereitstellen und Kopieren der Daten auf das Device, den Kernelaufruf sowie die Prozessierung des Outputs. Die Funktionalität des Hostcodes ist somit identisch mit dem des XPU-Ansatzes.

Der Unterschied liegt in der Verarbeitung des Unpacker-Algorithmus, also im Kernel. Im XPU-Ansatz und der Task-basierten CPU Variante wird immer genau ein Thread zur sequenziellen Verarbeitung einer Microslice genutzt. Das bedeutet, dass beim Entpacken einer Timeslice nur 505 Threads in Anspruch genommen werden können. In Betracht der knapp 70k Threads Gesamtkapazität der RTX 2080 Ti, bzw. 150k Threads der Radeon VII fällt die Auslastung durch 505 Threads nur sehr gering aus.

Die ungenutzten Threads können allerdings unterstützend genutzt werden. Der zeitaufwändigste Schritt bei der Verarbeitung einer Message ist der Zugriff auf den Haupt-

speicher. Ein CPU Thread liest die Daten selbst aus, erstellt ggf. ein Digi und schreibt dieses zurück in den Hauptspeicher. Der Lese- und Schreibprozess kann auf der GPU ausgelagert werden. So werden zwar weiterhin alle Messages innerhalb einer Microslice sequenziell abgearbeitet, aber die Arbeitslast wird aufgeteilt. Dies ist möglich durch die Nutzung des Shared Memory, welches sich alle Threads eines Blocks teilen. Jeder Streaming-Multiprozessor bzw. jede Compute Unit besitzt einen eigenen On-Chip Speicher, der in L1-Cache und Shared Memory aufgeteilt ist. In der Standardkonfiguration ist das Shared Memory 64 KB groß [25], hat aber eine Lese- und Schreibgeschwindigkeit wie der L1-Cache [29].

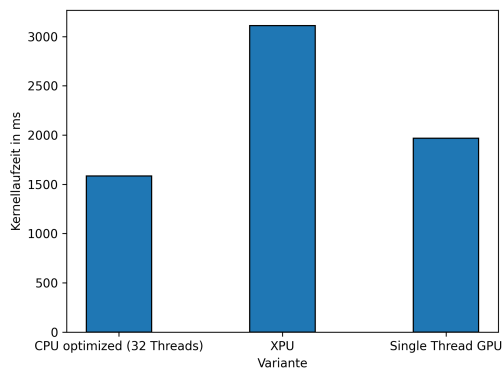
Entsprechend ist die grundlegende Idee dieses Ansatzes, dass der Thread, der die Messages sequenziell abarbeitet, die Daten nicht direkt vom Hauptspeicher bezieht, sondern vom Shared Memory. Hilfsthreads können dazu genutzt werden, parallel die Daten vom Hauptspeicher ins Shared Memory und zurück zu kopieren. Aufgrund der besonderen Architektur der Warps und Wavefronts in einem Block, kann dies nur innerhalb dieser effizient passieren. Sollen Daten vom Hauptspeicher ins Shared Memory kopiert werden, kann der Kopiervorgang nur dann gleichzeitig von mehreren Threads durchgeführt werden, wenn die Daten als Batch in der gleichen Reihenfolge von Hauptspeicher zu Shared Memory transferiert werden [29]. Ist dies nicht gegeben, werden die Speicherzugriffe sequenziell verarbeitet und die Parallelität geht verloren. Das bedeutet, dass die Anzahl an Hilfsthreads pro Microslice die Warp- / Wavefrontgröße - 1 beträgt. Beim Kernelaufruf wird die Parallelisierung auf Microslice Ebene durch die Nutzung von einem Block pro Microslice erreicht. Jeder Block hat dabei die Größe einer Warp (32 Threads) auf der RTX 2080 Ti, bzw. die Größe einer Wavefront (64 Threads) auf der Radeon VII.

Die Umsetzung dieses Ansatzes ist im Pseudocode 7.6 vereinfacht dargestellt. Der größte Unterschied zur Standardumsetzung (gezeigt in 7.1) liegt wie bereits erwähnt in der Message-Schleife. Es werden alle Messages der Microslice in Batches durchlaufen, die abhängig von der Größe des allokierten Shared Memories sind. Für jeden Batch werden zunächst alle Threads im Block (Blockgröße = Warpgröße in diesem Fall) tätig und kopieren Messages ins Shared Memory. Das Shared Memory wird in zwei Buffer aufgeteilt. Der eine enthält die aus dem Hauptspeicher kopierten Messages, der andere die daraus entstandenen Digis. Thread 0 entpackt anschließend die Messages direkt aus dem Input Buffer, während erstellte Digis in den Output Buffer des Shared Memories geschrieben werden. Sobald alle Messages von Thread 0 verarbeitet sind, werden alle Threads im Block aktiv und schreiben die Digis vom Output Buffer in den Hauptspeicher. Sind alle

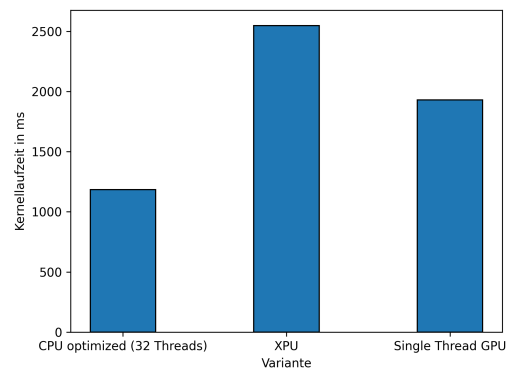
Message Batches der Microslice durchlaufen, können die Digis aus dem Hauptspeicher an den Unpacker-Task zurückgegeben werden. Im Folgenden wird dieses Konzept als die *Single-Thread GPU Variante* bezeichnet.

4.3.2 Laufzeitanalyse Unpacker-Algorithmus

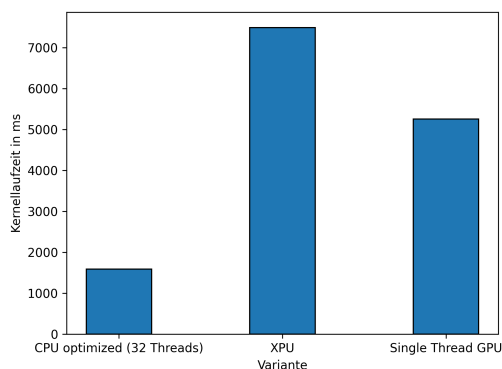
In diesem Abschnitt wird die gemessene Laufzeit des vorgestellten Konzepts analysiert. Dazu ist in Abbildung 4.3 ein Vergleich zur XPU sowie der optimierten parallelen Task-basierten Variante (folgend nur noch CPU Variante genannt) dargestellt.



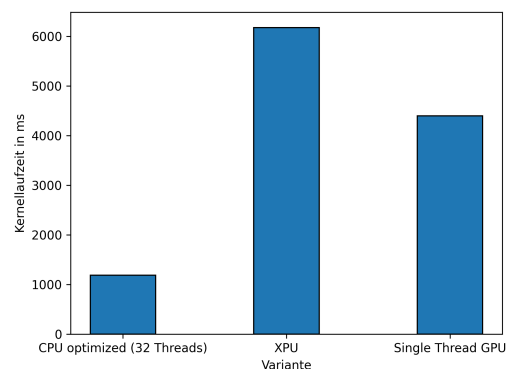
(a) Run 2456 auf NVIDIA RTX 2080 Ti



(b) Run 2391 auf NVIDIA RTX 2080 Ti



(c) Run 2456 auf AMD Radeon VII



(d) Run 2391 auf AMD Radeon VII

Abbildung 4.3: Kernellaufzeitvergleich Single-Thread GPU auf Asustest04

Um die Laufzeiten sinnvoll gegenüberzustellen, muss auch hier festgelegt werden, wie die Zeitmessung stattfinden soll. Der Fokus der vorgestellten Optimierung liegt auf dem

Unpacker-Algorithmus. Dieser unterscheidet sich nun maßgeblich von denen der XPU und CPU Varianten. Allerdings ist der zugehörige Unpacker-Task der Single-Thread GPU Variante in der Funktionsweise identisch mit dem der XPU Variante, während die des CPU Ansatzes eine gänzlich andere Rolle spielt. Da die verschiedenen Unpacker-Tasks verschieden starken Einfluss auf die Gesamtlaufzeit des Unpackers haben, gilt es also ausschließlich die Kernellaufzeit, also die des Unpacker-Algorithmus zu messen. Die Unterschiede im Unpacker-Task werden im folgenden Abschnitt 4.3.3 beschrieben und analysiert.

Gemessen wurden die Laufzeiten der XPU und GPU Umsetzungen für Run 2456 und 2391 auf den Grafikkarten AMD Radeon VII und NVIDIA RTX 2080 Ti. Gegenübergestellt ist die aktuell schnellste CPU Variante mit 32 Threads. Wie erwähnt ist die Funktionsweise des Unpacker-Algorithmus von CPU und XPU Variante gleich. Beide Algorithmen sind auf Microslice Ebene parallelisiert, nur ist der Code an die Hardware-spezifischen Datenstrukturen angepasst. Die allerdings stark abweichenden Laufzeiten stammen zum einen vom Overhead, der durch die angepassten Datenstrukturen im XPU-Algorithmus entsteht. Zum anderen ist ein CPU Kern aufgrund seiner Architektur deutlich schneller als ein einzelner GPU Kern, wenn sie beide die gleiche Arbeit verrichten. Die Abbildung verrät außerdem, dass die GPU Variante gegenüber der XPU Variante in allen Fällen einen Speedup zwischen 1,3 und 1,6 hat. Dies ist ein logisches Ergebnis, da die Single-Thread GPU Umsetzung konzeptionell eine Optimierung der XPU Variante ist. Die Tests zeigen allerdings auch, dass sie langsamer ist als die optimierte CPU Variante.

Weiterhin sind die stark abweichenden Laufzeiten gleicher Runs auf den beiden Grafikkarten auffällig. Für beide Runs ist die RTX 2080 Ti um etwa einen Faktor 2 schneller als die Radeon VII, obwohl die Radeon VII mehr Ressourcen zur Verfügung hat. Der Vorteil kann bei dieser Implementierung allerdings nicht vollständig ausgenutzt werden, da weder der volle Speicher, noch alle zur Verfügung stehenden Threads genutzt werden. In diesem Fall scheint die RTX 2080 Ti aufgrund ihrer Architektur effizienter mit den Daten umzugehen. Die genauen Gründe für dieses Verhalten sind unbekannt und müssten bei Interesse in Zukunft durch detaillierte Tests in Erfahrung gebracht werden.

Die Einführung der Hilfstreads zur Optimierung der Speicherzugriffe bringt also zwar insgesamt einen Vorteil, dieser ist aber nicht signifikant genug, um die langsame Geschwindigkeit der einzelnen GPU Kerne auszugleichen.

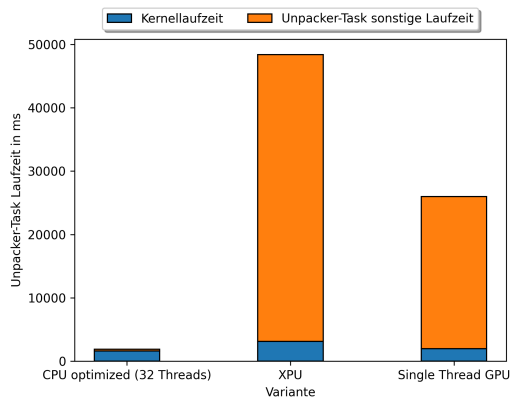
4.3.3 Laufzeitanalyse Unpacker-Task

Im vorherigen Abschnitt wurde die Laufzeit des Unpacker-Algorithmus isoliert betrachtet, um einen fairen Vergleich bei unterschiedlichen Bedingungen zu ermöglichen. Es ist allerdings wichtig, das Hauptziel dieser Arbeit im Auge zu behalten, nämlich den Unpacker Prozess für die Online Datenverarbeitung zu optimieren. Entsprechend muss der Unpacker als Ganzes detailliert untersucht werden. Dafür wird in diesem Abschnitt der angepasste Unpacker-Task des Single-Thread GPU Unpackers untersucht.

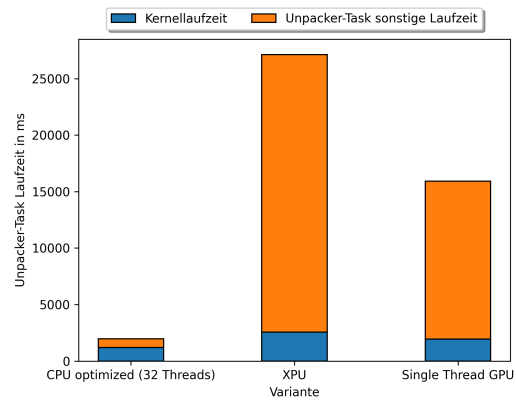
Der Unpacker-Task beinhaltet den Hostcode des GPU Unpackers. Neben dem Kernelaufruf ist seine Hauptaufgabe der Datentransfer zwischen Host und Device. Dafür werden XPU Buffer genutzt, die zunächst mit den Daten aus dem Hostspeicher gefüllt werden müssen. Dazu gehören die Hardwareparameter zur Errechnung des Digi Zeitstempels sowie die Daten der Microslices und einiger Offsets für den Zugriff auf diese. Bevor der Kernel ausgeführt werden kann, müssen die Daten im Buffer noch von Host zu Device kopiert werden, damit sie dort im Speicher verfügbar sind. Anschließend kann der Unpack-Kernel seine Aufgabe durchführen und die Microslices auf der GPU verarbeiten.

Aufgrund der Struktur der vorliegenden Daten ist es vor dem Entpacken einer Microslice nicht ohne weiteren Arbeitsaufwand möglich zu wissen, wie viele Digis beim Entpacken entstehen werden. Dies führt dazu, dass die statischen Buffer zunächst zu groß angelegt werden und Lücken im Outputbuffer zwischen den Digis der einzelnen Microslices entstehen. Da die Folgeprozesse allerdings einen eindimensionalen STL-Vector mit aufeinanderfolgenden Digis erwarten, müssen diese Lücken zunächst noch entfernt werden.

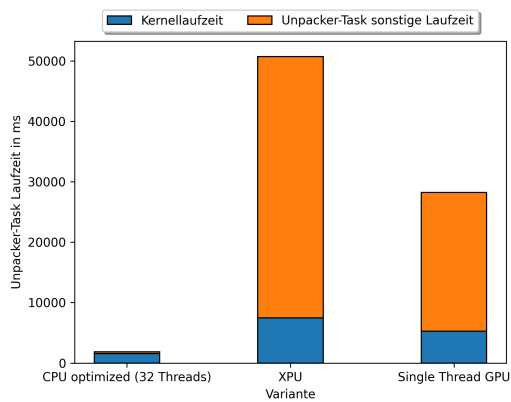
Diese Aufgabe kann auf dem Host erledigt werden, nachdem die Daten vom Device zurück transferiert wurden. Dabei werden beim Kopieren der Daten vom Buffer in den Output Vector die Lücken ausgelassen. Eine Alternative dazu ist die Nutzung eines zweiten Kernels (Copy-Kernel), der diese Aufgabe auf der GPU selbst erledigt. Dafür werden die Daten des Outputbuffers nicht vom Device zum Host kopiert, sondern erst durch den zweiten Kernel bereinigt. Dies ist in diesem Schritt möglich, da zwischen Unpack-Kernel und Copy-Kernel ein Buffer mit der korrekten Größe erstellt wird, in den die Digis aneinandergereiht kopiert werden. Die Nutzung dieser Variante ist vor allem von Vorteil, wenn der Folgeprozess ebenfalls auf der GPU arbeitet. In Zukunft kann es damit möglich sein, dass der Hitfinder die auf der GPU befindlichen Daten direkt weiter verwertet und damit zeitintensive Kopierschritte einspart. Aber auch beim Übertragen der Daten zurück



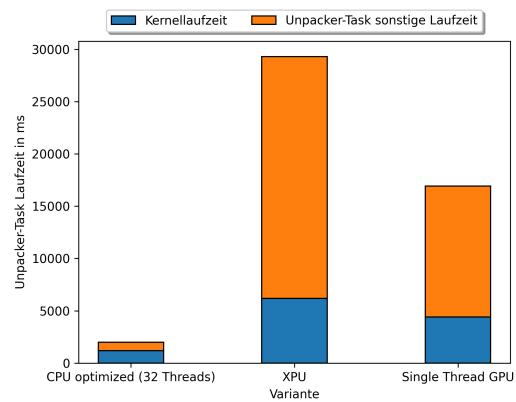
(a) Run 2456 auf NVIDIA RTX 2080 Ti



(b) Run 2391 auf NVIDIA RTX 2080 Ti



(c) Run 2456 auf AMD Radeon VII



(d) Run 2391 auf AMD Radeon VII

Abbildung 4.4: Laufzeitvergleich Unpacker-Task Single-Thread GPU auf Asustest04

an den Host, ist die Nutzung des Kernels, aufgrund der guten Parallelisierbarkeit des Problems, sinnvoll. In der aktuellen Umsetzung des Unpacker-Tasks ist der Copy-Kernel integriert und wird genutzt. Hinterher müssen die Daten dennoch zum Host transferiert und dort in einen STL-Vector kopiert werden.

Zusammengefasst durchläuft der Unpacker-Task für jede Timeslice die folgenden Schritte:

- Kopieren der Hardwareparameter zu Buffer (Hostseite)
- Kopieren der Microslice Daten und Offsets zu Buffer (Hostseite)

- Kopieren der Bufferdaten von Host zu Device
- Ausführung Unpack-Kernel
- Erstellen des Buffers mit korrekter Größe
- Ausführung Copy-Kernel
- Kopieren der Bufferdaten von Device zu Host
- Kopieren der Daten des Outputbuffers zu STL-Vector

Der CPU Unpacker-Task hingegen iteriert einmal über alle Microslices und kopiert anschließend nur noch die Daten vom Zwischenvector in den Outputvector. Die vielen zusätzlichen Kopiervorgänge mit großen Datenmengen im GPU Unpacker-Task sorgen daher für einen signifikanten Anstieg der Laufzeit gegenüber der CPU Version. Abbildung 4.4 zeigt die Laufzeiten der drei Umsetzungen, wobei der Fokus auf der Tasklaufzeit ohne der Kernellaufzeit liegt. Für die CPU Variante beträgt die Tasklaufzeit 0,3s für Run 2456 und 0,8s für Run 2391, während die beiden XPU-basierten Unpacker-Tasks um Faktoren zwischen 80 und 140 langsamer sind. Der Unterschied zwischen XPU und der Single-Thread GPU Variante liegt in einigen bereits vorgenommenen Optimierungen des GPU Tasks, wie z.B. parallelisierter Kopiervorgänge.

Die Analyse der Gesamtlaufzeit zeigt deutlich, dass der Flaschenhals der GPU Ansätze im Unpacker-Task und nicht im Unpacker-Algorithmus zu liegen scheint. Diese können allerdings theoretisch in zukünftiger Arbeit auf die Geschwindigkeit des CPU Unpacker-Tasks optimiert werden. Kopierschritte können durch Änderungen am XPU-Framework im Hintergrund parallelisiert und somit eliminiert werden. Die Kopiervorgänge nach Ausführung der Kernels sind aktuell ausschließlich aufgrund der Strukturierung des Rekonstruktionsprozessablaufs nötig. Eine Optimierung der Parameterübergabe zwischen den Prozessen, beispielsweise eine Übergabe der Daten auf GPU-Ebene würde signifikante Vorteile im Ablauf der gesamten Rekonstruktion bringen. Aufgrund dessen wird die Laufzeit des Unpacker-Tasks im weiteren Verlauf dieser Arbeit vernachlässigt, sodass nur noch die Kernellaufzeiten betrachtet werden.

4.4 Multi-Thread GPU

Der Single-Thread GPU Ansatz zeigt, wie wichtig die effiziente Auslastung beim Einsatz von Grafikkarten ist. Um diese erreichen zu können, muss ein Weg gefunden werden,

mehr Threads für eine bessere Parallelisierung zu nutzen. Eine Parallelisierung auf Microslice Ebene reicht daher nicht mehr aus. Stattdessen wird in diesem Abschnitt die Parallelisierung des Unpacker-Algorithmus, also innerhalb einer Microslice, auf der GPU betrachtet. Die Umsetzung dieses Ansatzes ist als Pseudocode in 7.7 vereinfacht dargestellt.

4.4.1 Konzept

Wie in Abschnitt 3.3 erklärt bestehen innerhalb einer Microslice Abhängigkeiten auf Ebene der Messages. Für die Erstellung eines Digi aus einer Hit-Message muss der Zeitstempel aus mehreren Faktoren errechnet werden, unter anderem der aktuellen Epoche und dem Epochen Cycle. Die zuletzt verarbeitete TsMsb-Message gibt dabei die Epoche vor und bestimmt, basierend auf der vorhergehenden Epoche, ob ein neuer Cycle beginnt. In der sequenziellen Variante werden die Werte von Epoche und Cycle in Variablen gespeichert und verwendet, sobald sie wieder benötigt werden. In einer parallelen Umsetzung kann es aber passieren, dass Hit-Messages und TsMsb-Messages zeitgleich verarbeitet werden. Dabei dürfen die Abhängigkeiten für eine korrekte Berechnung des Digi Zeitstempels nicht vernachlässigt werden. Die einzige Möglichkeit, dies zu erreichen, ist durch Kommunikation der Threads untereinander. Für den Austausch der Informationen kann das Shared Memory genutzt werden.

In diesem Ansatz wird die Message Schleife gleichmäßig auf alle zur Verfügung stehenden Threads aufgeteilt. Die Messages werden in Batches abgearbeitet, die so groß sind wie die Blockgröße n . Obwohl die n Messages gleichzeitig bearbeitet werden, spielt deren ursprüngliche Reihenfolge eine wichtige Rolle. Daher bekommen die Threads mit aufsteigender ThreadID die Messages in entsprechend aufsteigender Reihenfolge zugeteilt, sodass Thread 0 Message 0 und Thread $n-1$ Message $n-1$ bearbeitet.

Der wichtigste Teil der Kommunikation zwischen den Threads ist die Scan-Funktion [30] des XPU-Frameworks. Diese erstellt für jeden Thread eine Präfixsumme, die die Summe eines spezifischen Wertes aller vorhergehenden Threads enthält. Beispielsweise kann damit geprüft werden, wie viele Threads mit kleinerer ThreadID eine Hit- bzw. TsMsb-Message halten, um Abhängigkeiten zu bestimmen. In diesem Beispiel wird dafür die Präfixsumme über einen Integerwert gebildet, der nur 0 oder 1 annimmt, da nur zwei Arten von Messages vorhanden sein können. Die Scan-Funktion kann aber auch mit größeren Werten für den Austausch anderer Informationen genutzt werden. Die Grundlage dafür bildet das Shared Memory mit seinen schnellen Zugriffszeiten. Eine andere Metho-

de zum Austausch einzelner Werte ist das Broadcasten. Dafür wird ausschließlich Thread $n-1$ genutzt, da er der einzige ist, der durch einen Scan die Informationen aller anderen Threads mitgeteilt bekommt. Für den Broadcast bestimmter Werte ist im Shared Memory Speicher reserviert, in den Thread $n-1$ schreiben und jeder andere Thread lesen kann.

Thread Parameter Beispiel

Thread ID	0	1	2	3	4	5	6	7	...	n
Message Typ	H	H	T_0	H	H	H	T_1	H	...	H
TsMsb Scan / Buffer Index	0	0	1	1	1	1	2	2	...	x

Shared Memory TsMsb-Message Buffer

Buffer Index	0	1	2	3	4	5	...	n
Epoche	Baseline	T_0	T_1	T_2	T_3	T_4	...	-

Abbildung 4.5: Beispiel des TsMsb Scans und Indexberechnung, wobei x die Summe aller TsMsb-Messages darstellt.

Die Aufgabe eines Threads hängt vom vorliegenden Messagetyp ab. Zu Beginn der Bearbeitung des Message Batches liest jeder Thread die ihm zugewiesene Message aus und die Scan-Funktion wird auf dem Ergebnis ausgeführt. Nach dem Scan weiß jeder Thread, wie viele TsMsb-Messages gleichzeitig von Threads mit kleinerer ID bearbeitet werden, also wie viele der parallelen Messages relevant für seine eigenen Berechnungen sind. Um jedem Thread den Zugriff auf die für ihn relevanten Epocheninformationen zu ermöglichen, wird im Shared Memory ein Buffer definiert, der die Epochenwerte aller aktuell zu verarbeitenden TsMsb-Messages sowie eine Baseline Epoche halten kann. Threads mit TsMsb-Message schreiben ihre Epochenwerte in den Buffer, während Threads mit Hit-Message die Informationen von dort auslesen. Damit dies ohne Zugriffskollisionen geschehen kann, wird das Ergebnis der Scan-Funktion für die Indexierung genutzt. In Abbildung 4.5 ist ein Beispiel für die Präfixsumme sowie dem daraus entstehenden Shared Memory Buffer aufgezeichnet. Hit-Message Threads, denen im aktuellen Batch keine TsMsb-Messages vorangegangen sind, haben das Scan Ergebnis 0 und greifen somit auf Index 0 des Buffers zum Auslesen ihrer Epoche zu. Dieser enthält eine Baseline Epoche, die zu Beginn des Programms von der 1. Message bzw. später von vorhergehenden Batches gegeben ist. Enthält ein Thread eine TsMsb-Message, wird die Präfixsumme für

diesen und alle nachfolgenden Threads um 1 inkrementiert. Der TsMsb-Thread schreibt seine Message in den errechneten Index des Buffers und alle folgenden Hit-Threads greifen entsprechend auf diesen zu. Durch dieses Verfahren ist sichergestellt, dass alle Digis mit der korrekten Epoche erstellt werden. Der Buffer muss $n+1$ Einträge enthalten können, da theoretisch alle Messages in einem Batch TsMsb-Messages sein könnten und die Baseline ebenfalls mit enthalten sein muss. Je nach Datenmenge in der Microslice kann das Verhältnis zwischen Hit- und TsMsb-Message stark schwanken. In den meisten Fällen werden allerdings nur die ersten Einträge des Buffers überhaupt benötigt.

Ist eine TsMsb-Message vorhanden, muss außerdem noch überprüft werden, ob ein neuer Cycle begonnen hat. Cycles sind nur innerhalb einer Microslice gültig, haben zu Beginn also immer den Wert 0. Um eine Inkrementierung des Cycles festzustellen, muss die Epoche einer TsMsb-Message mit der letzten vorhergehenden Epoche verglichen werden. Dies kann ebenfalls anhand des TsMsb Buffers im Shared Memories erledigt werden. Wichtig ist allerdings, dass eine Cycle Erhöhung allen anderen Threads durch einen Scan mitgeteilt werden muss.

Anschließend können anhand der geteilten Informationen Digis aus den Hit-Messages erstellt werden. Bevor diese in den GPU-Hauptspeicher geschrieben werden können, muss ein letzter Scan durchgeführt werden, um festzustellen, wie viele Digis in diesem Batch erstellt wurden. Das Ergebnis definiert den Index, an den ein Thread das erstellte Digi in den Output Buffer schreibt. Zuletzt werden von Thread $n-1$ noch alle nötigen Broadcasts für den nächsten Batch durchgeführt. Dazu gehört das setzen der Baseline Epoche, oder die Weitergabe der Cycle Inkrementierungen. Vor jedem der drei Scans werden alle Threads durch Barrieren angehalten und gesammelt, um Raceconditions zu vermeiden. Die beschriebene Vorgehensweise wird wiederholt, bis alle Messages verarbeitet sind.

Auch in dieser Implementierung wird ein Block pro Microslice genutzt. Allerdings ist die Effizienz des Programms nicht mehr abhängig von parallelisierten Kopierschritten, weshalb die Blockgröße nicht mehr gleich der Warp- / Wavefrontgröße sein muss. Tests ergeben, dass für beide vorliegenden GPUs die optimale Blockgröße 512 beträgt. Bei paralleler Ausführung aller 505 Microslices ergibt sich dadurch ein Anspruch von 258.560 Threads, wodurch die Grafikkarten vollständig ausgelastet sind. Die Blockgröße muss in Zukunft beim Einsatz anderer GPUs potenziell angepasst werden.

4.4.2 Laufzeitanalyse

Abbildung 4.6 zeigt den Laufzeitvergleich zwischen dem vorgestellten und den bereits in 4.3.2 besprochenen Varianten für Run 2456 auf der AMD Radeon VII. Aufgrund des hohen Speedups ist die Darstellung logarithmisch skaliert und es wurde auf die Abbildung der anderen Setups (Run 2391 und NVIDIA GPU) verzichtet. Stattdessen sind alle Kernellaufzeiten detailliert in 4.1 aufgezeigt.

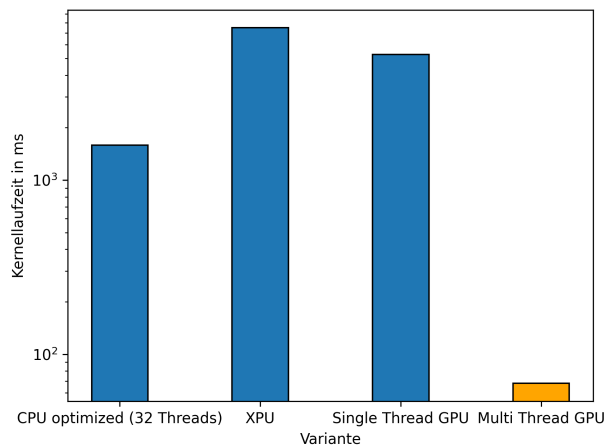


Abbildung 4.6: Logarithmisch skaliertes Kernellaufzeitvergleich aller vorgestellten Ansätze. Gemessen wurde Run 2456 unter Nutzung der AMD Radeon VII auf Asustest04.

Run	CPU	XPU		Single-Thread GPU		Multi-Thread GPU	
		NVIDIA	AMD	NVIDIA	AMD	NVIDIA	AMD
2391	1185,67	2550,33	6177,67	1931,33	4399,33	78,61	58,67
2456	1584,00	3112,67	7494,00	1968,77	5259,00	99,80	68,83

Tabelle 4.1: Kernellaufzeiten aller vorgestellten Ansätze in ms auf Asustest04

Die Multi-Thread GPU Implementierung erzielt gegenüber der bisher schnellsten Variante (CPU 32 Threads) einen Speedup von 23 für Run 2456 und 20 für Run 2391. Auffallend bei den gemessenen Zeiten ist die Performanz der Grafikkarten. Während für den XPU und Single-Thread GPU Ansatz die NVIDIA RTX 2080 Ti immer über einen Faktor 2 schneller war als die Radeon VII, ist es hier nun umgekehrt, wenn auch nicht so stark. Die AMD Grafikkarte ist in beiden Runs um etwa einen Faktor 0,5 schneller. Letztendlich

werden dadurch die Erwartungen widergespiegelt, die durch die höhere Speicherkapazität, -bandbreite und Threadzahl gestellt wurden, wobei die Speicherkapazität noch nicht ausgereizt werden konnte. Sie beträgt jeweils 11 GB und 16 GB, allerdings werden in Run 2456 nur 4,8 GB und in Run 2391 2,7 GB STS-Daten entpackt. Die unterschiedliche Ressourcenauslastung der anderen Faktoren zwischen den Varianten hat dennoch signifikante Auswirkungen auf die gemessenen Laufzeiten.

Der Unterschied zwischen Single-Thread und Multi-Thread GPU unterstreicht, wie wichtig die effiziente Ressourcennutzung und Auslastung bei der Verwendung von Grafikkarten ist. Insgesamt zeigen die Ergebnisse, dass eine weitere Verbesserung des Unpackers möglich ist. Die Multi-Thread GPU Variante kann allerdings vorerst noch nicht in die CBM-Software integriert werden, da seine Gesamtlaufzeit bei etwa 21,5s, verglichen mit 1,8s des CPU Unpackers liegt. Der Grund dafür befindet sich im Unpacker-Task, der bisher noch nicht auf die Nutzung von GPUs optimiert wurde. Hinweise auf Verbesserungsmöglichkeiten des Tasks wurden bereits in 4.3.3 besprochen und können in einer zukünftigen Arbeit getestet werden.

5 Validierung

Zur Verifizierung der korrekten Funktionalität des Unpackers nach einer Anpassung gibt es zwei Möglichkeiten, die abhängig von der Art der Änderung sind. Unterschieden wird dabei in Optimierungen am Unpacker-Task und Unpacker-Algorithmus. Wird lediglich der Task verändert, reicht es aus die Anzahl der erstellten Digis zu überprüfen. Dazu wurde als Kontrollwert die Anzahl der produzierten Digis des sequenziellen CPU Unpackers gewählt, da dieser grundsätzlich als Ausgangspunkt für alle Unpacker Optimierungen gilt. Wenn keine Änderungen am Algorithmus vorliegen, ist davon auszugehen, dass die Parameter der erstellten Digis korrekt berechnet werden und nicht überprüft werden müssen. Da der Unpacker-Task nur bestimmt, wie die Verarbeitung der Microslices abläuft, reicht es aus zu überprüfen, ob alle Hit-Messages der Microslices umgewandelt wurden. Dies ist in der Anzahl der Digis am Ende einer Timeslice widergespiegelt. Die Digianzahl sollte auf Basis der gleichen Daten auch immer das gleiche Ergebnis liefern, da der Unpacker deterministisch ist. Entsprechend wurde für diese Arbeit nach jeder Optimierung geprüft, ob die Anzahl der erstellten Digis für beide Runs konstant bleibt. Diese Metrik konnte häufig genutzt werden, um Fehler aufzudecken und auszubessern.

Anpassungen am Unpacker-Algorithmus hingegen erfordern eine detailliertere Betrachtung der Digis, da die Berechnung der Digi Parameter direkt betroffen ist. Die Korrektheit der Parameter spiegelt sich nicht nur in der Anzahl erzeugter Digis wieder. Stattdessen können die Ergebnisse des Hitfinders zur Unterstützung herbeigezogen werden. Dieser gibt für jede Timeslice aus, wie viele Hits und Cluster er aus den Digis errechnet hat. Da diese Berechnungen auf den Zeitstempeln der Digis basieren und auch der Hitfinder Algorithmus deterministisch ist, können diese Zahlen als Kontrollwert genutzt werden. Die Überprüfung der GPU-Varianten zeigt geringe Abweichungen im Bereich von 0,0001% für gefundene Hits, allerdings keine für die Clustersuche. Dies ist dadurch zu erklären, dass die Präzision im Umgang mit Floating Points sich zwischen CPU und GPU unterscheidet. Da der Kontrollwert aus der sequenziellen CPU Variante stammt, führt dies also zu leichten Unterschieden in der Berechnung der Zeitstempel. Die resultierenden Abweichungen sind allerdings weder signifikant, noch auf die Änderungen am Algorith-

mus zurückzuführen.

Im Verlauf der Arbeit wurden Schwankungen im einstelligen Millisekunden Bereich bei wiederholten Laufzeitmessungen mit gleichen Daten festgestellt. Der Grund dafür liegt in der allgemeinen Nutzung des Testservers. Da im Hintergrund stetig Prozesse (z.B. eingehende SSH-Verbindungen) von anderen Entwicklern mitlaufen, kann nie die vollständige Auslastung der verfügbaren Ressourcen erfolgen. Obwohl dies in den meisten Fällen weniger als 0,1% der Laufzeit ausmacht, wurde für jedes Ergebnis immer das Mittel aus 3 Messungen genutzt, um ein möglichst genaues Resultat zu erzielen.

6 Zusammenfassung und Ausblick

Das Hauptziel dieser Arbeit war die Optimierung des Unpackers in der STS-Rekonstruktionskette. Optimierungen am Unpacker-Task der CPU Variante durch Anpassung der Parallelisierungstiefe sowie der Resize-Funktion erzielten einen Speedup von 4-5. Teile der Optimierungen sind bereits in die CBM-Software integriert und wurden erfolgreich in einer weitreichenden Datachallenge zum Testen der Online Prozessierung eingesetzt.

Im Anschluss an diese Optimierungen, wurde der Einsatz des FairMQ Frameworks zur Parallelisierung getestet. Dessen Funktionalität ist allerdings auf das High-Level Konzept der Prozesssteuerung fokussiert statt der Low-Level Parallelisierung. Für den isolierten Unpacker-Prozess ergibt sich daher keine Verbesserungen der Parallelisierung durch FairMQ. Low-Level Optimierungen können stattdessen in die Devicestruktur des Frameworks integriert werden, falls es zu einem späteren Zeitpunkt in der CBM-Software genutzt werden sollte.

Um weitere Erfolge im Bezug auf die Online-Fähigkeit der Rekonstruktionssoftware erzielen zu können, wurde die Realisierbarkeit eines GPU-basierten Unpacker Prozesses untersucht. Zunächst wurden mit der Single-Thread GPU Variante die Konzepte des erfolgreichen CPU Unpackers beibehalten. Durch die Nutzung von Hilfstreads zum schnelleren Speicherzugriff erreichte die Umsetzung eine ähnliche, allerdings etwas langsamere Kernellaufzeit als die CPU Implementierung. Die volle Auslastung der GPU Rechenleistung konnte letztlich durch den Multi-Thread GPU Ansatz erreicht werden. Eine auf die Hardware angepasste Softwarelogik führte zu einem Speedup der Kernellaufzeit von etwa 20 im Vergleich zur CPU Variante. Die Unterschiede zwischen Single-Thread und Multi-Thread GPU Laufzeiten unterstreichen die Wichtigkeit einer effizienten Ressourcennutzung.

Obwohl die Nutzung der Grafikkarte einen stark beschleunigten Unpacker-Algorithmus erzielt, ist die Gesamtlaufzeit durch den noch ineffizienten Unpacker-Task um etwa einen Faktor 10 zu langsam für die Nutzung im Live-Projekt. Durch Optimierungen am XPU-

Framework sowie Parallelisierung und Eliminierung von Kopiervorgängen sollte es in zukünftiger Arbeit aber möglich sein, den Unpacker-Task so zu beschleunigen, dass die Nutzung des GPU Unpackers ermöglicht wird. In diesem Zusammenhang kann unter anderem auch verfolgt werden, ob und wie der Betrieb mehrerer GPUs in einem Server zur besseren Performanz genutzt werden kann.

Sollte sich ergeben, dass weiterhin die CPU Variante die beste Wahl für den Unpackerprozess bleibt, ist es noch möglich, einen leichten Laufzeitgewinn durch die Nutzung der *OMP_COLLAPSE* Funktion zu erzielen.

Literaturverzeichnis

- [1] CBM Collaboration. Cbm - im Inneren eines Neutronensterns. https://www.gsi.de/forschungbeschleuniger/fair/forschung/cbm_im_innenen_eines_neutronensterns. Letzter Zugriff am 03.07.2023.
- [2] V. Lindenstruth J. de Cuveland, D. Hutter. CBM First-level Event Selector Data Management Developments. <https://core.ac.uk/download/pdf/52556966.pdf>, 2013. Letzter Zugriff am 03.10.2023.
- [3] N. Herrmann. Status and Perspectives of the CBM experiment at FAIR. https://www.epj-conferences.org/articles/epjconf/pdf/2022/03/epjconf_sqm2021_09001.pdf, 2022. Letzter Zugriff am 17.07.2023.
- [4] GSI Darmstadt. FAIR SIS100 - FEATURES AND STATUS OF REALISATION. <https://accelconf.web.cern.ch/ipac2017/papers/wepva030.pdf>. Letzter Zugriff am 10.07.2023.
- [5] GSI Darmstadt. Die Beschleunigeranlage. https://www.gsi.de/forschungbeschleuniger/fair/die_maschine. Letzter Zugriff am 10.07.2023.
- [6] Partha Pratim Bhaduri. The physics goals of the CBM experiment at FAIR. https://indico.cern.ch/event/985460/contributions/4264612/attachments/2211433/3743016/ppbhaduri_CPOD2021_CBM_summary.pdf. Letzter Zugriff am 10.07.2023.
- [7] The CBM Collaboration. A CBM full system test-setup for high-rate nucleus-nucleus collisions at GSI / FAIR. <https://cbm-wiki.gsi.de/pub/Public/Documents/mcbm-proposal2GPAC-fullVersion.pdf>, 6 2017. Letzter Zugriff am 09.10.2023.
- [8] Adrian Weber. The mCBM experiment at SIS18 of GSI/FAIR - a CBM precursor and demonstrator. https://indico.cern.ch/event/895086/contributions/4724533/attachments/2421031/4144021/QM2022_poster_AdrianWeber.pdf. Letzter Zugriff am 10.07.2023.

- [9] The CBM Collaboration. Technical Design Report for the CBM. <http://repository.gsi.de/record/54798/files/GSI-Report-2013-4.pdf?version=2>, 2013. Letzter Zugriff am 17.07.2023.
- [10] CBM Collaboration. CBM Progress Report 2021. https://repository.gsi.de/record/246663/files/cbm_pr2021_final.pdf, 2021. Letzter Zugriff am 22.08.2023.
- [11] FairRootGroup. FairMQ. <https://github.com/FairRootGroup/FairMQ>, 2023. Letzter Zugriff am 24.07.2023.
- [12] Pieter Hintjens et al. ØMQ - The Guide. <https://zguide.zeromq.org>. Letzter Zugriff am 24.07.2023.
- [13] J. de Cuveland V. Friese. Towards an online-capable software stack. <http://repository.gsi.de/record/246663/files/?ln=de>, 2022. Letzter Zugriff am 17.07.2023.
- [14] Volker Friese. Estimate of the CBM computing requirements for operation at SIS-100 CBM-CN-18001, 5 2018. Kontakt: v.friese@gsi.de.
- [15] Intel Corporation. Intel® Xeon® Prozessor E7-4870. <https://ark.intel.com/content/www/de/de/ark/products/53579/intel-xeon-processor-e74870-30m-cache-2-40-ghz-6-40-gts-intel-qp.html>. Letzter Zugriff am 18.08.2023.
- [16] Martin Lanser Stephanie Friedhoff. Einige Grundlagen zu OpenMP. https://numerik.uni-koeln.de/sites/numerik/uebungen/hpc_16/Grundlagen_OpenMP.pdf, 6 2016. Letzter Zugriff am 27.09.2023.
- [17] Dive into Systems LLC. Heterogeneous Computing: Hardware Accelerators, GPGPU Computing, and CUDA. <https://diveintosystems.org/book/C15-Parallel/gpu.html>, 2020. Letzter Zugriff am 08.08.2023.
- [18] Felix Weiglhofer. XPU. <https://github.com/fweig/xpu>. Letzter Zugriff am 22.08.2023.
- [19] GSI. CbmStsDigi Class Reference. <http://computing.gitpages.cbm.gsi.de/cbmroot/classCbmStsDigi.html>. Letzter Zugriff am 03.10.2023.
- [20] Dirk Hutter. *An Input Interface for the CBM First-level Event Selector*. PhD thesis, Goethe-Universität Frankfurt am Main, 2020.

- [21] Maximilian Zick. Parallele Prozessierung des Unpackers des STS-Detektors am mCBM-Experiment - Skalierung bei OpenMP. Bachelorarbeit, Goethe-Universität Frankfurt am Main, 10 2020.
- [22] Mohammad Al-Turany. ALFA: A framework for building distributed applications. https://indico.desy.de/event/40024/attachments/83205/110117/Alfa_punchlunch.pdf, 7 2023. Letzter Zugriff am 27.09.2023.
- [23] www.techpowerup.com. GPU Specs Database. <https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305>. Letzter Zugriff am 13.09.2023.
- [24] www.techpowerup.com. GPU Specs Database. <https://www.techpowerup.com/gpu-specs/radeon-vii.c3358>. Letzter Zugriff am 13.09.2023.
- [25] NVIDIA Corporation. TUNING CUDA APPLICATIONS FOR TURING. https://docs.nvidia.com/cuda/archive/11.0/pdf/Turing_Tuning_Guide.pdf, 2020. Letzter Zugriff am 18.09.2023.
- [26] Van Oostrum et al. Frontier Application Readiness Kick-Off Workshop. https://www.olcf.ornl.gov/wp-content/uploads/2019/10/ORNL_Application_Readiness_Workshop-AMD_GPU_Basics.pdf, 2019. Letzter Zugriff am 18.09.2023.
- [27] OpenMP Architecture Review Board. OpenMP 4.5 API C/C++ Syntax Reference Guide. <https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf>, 2015. Letzter Zugriff am 16.08.2023.
- [28] cppreference. std::allocator. <https://en.cppreference.com/w/cpp/memory/allocator>. Letzter Zugriff am 03.10.2023.
- [29] Mark Harris. Using Shared Memory in CUDA C/C++. <https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/>. Letzter Zugriff am 13.09.2023.
- [30] Felix Weiglhofer. Xpu Device.h. <https://github.com/fweig/xpu/blob/master/src/xpu/device.h>. Letzter Zugriff am 03.10.2023.

7 Anhang

7.1 Psuedocode des ursprünglichen Unpacker-Algorithmus

```
1  MsUnpacker::Operator(std::vector<CbmStsDigi> outDigis ,
2  std::vector<stsxyter::Message> messages){
3      time = ...
4
5      if (message[1].GetMessType() != stsxyter::MessType::TsMsb) {return;}
6      ProcessTsmsbMessage(message[1], time);
7
8      for (messageNr = 2; messageNr < messages.size(); messageNr++){
9
10         // Verarbeitung der Messages je nach Messagetyp
11         switch (messages[messageNr].GetMessType()) {
12
13             // Hitmessage wird zu Digi umgewandelt
14             case stsxyter::MessType::Hit: {
15                 ProcessHitMessage(..., time, messages[messageNr], outDigis);
16                 break;
17             }
18             // TsMsb Message wird für Epochenupdate genutzt
19             case stsxyter::MessType::TsMsb: {
20                 ProcessTsmsbMessage(messages[messageNr], time);
21                 break;
22             }
23         }
24     }
25 }
26
27 MsUnpacker::ProcessHitMessage(...){
28     // Nutzung der Daten zum bestimmen der 4 Digi Parameter
29     ...
30     outDigis.emplace_back(address, channel, messageTime, charge);
31 }
32
33 MsUnpacker::ProcessTsmsbMessage(...){
34     // Update der Epoche und ggf. EpochenCycle
```

```
35     if (aktuelleEpoche < letzteEpoche){Cycle++;}  
36     update time;}
```

Listing 7.1: Pseudocode der Kernfunktionalität des Unpacker-Algorithmus

7.2 Pseudocode des sequenziellen Unpacker-Tasks

```
1
2 MsUnpackChain(...){
3     std::vector<CbmStsDigi> outDigis;
4     for (compID = 0; compID < nrCompsInTs; compID++){
5         ...
6         for (msID = 0; msID < nrMsInComp; msID++){
7             ...
8             result = UnpackerAlgo(ms(msID));
9             outDigis.insert(result);
10        }
11    }
12    ...
13    return outDigis;
14 }
```

Listing 7.2: Pseudocode der Kernfunktionalität des sequenziellen Unpacker-Tasks

7.3 Pseudocode des parallelen Unpacker-Tasks

```
1 MsUnpackChain(...){
2     // Anlegen des Zwischenvektors für paralleles Schreiben
3     std::vector<std::vector<CbmStsDigi>> space (numThreads);
4     std::vector<CbmStsDigi> outDigis;
5
6     // Ausführen des Unpacker Algos
7     #pragma omp parallel for
8     for (compID = 0; compID < nrCompsInTs; compID++){
9         ...
10        #pragma omp parallel for
11        for (msID = 0; msID < nrMsInComp; msID++){
12            ...
13            result = UnpackerAlgo(ms(msID));
14            space[get_thread_num()].insert(result);}
15        }
16
17        // Anpassen der Größe des Output Vectors
18        outDigis.resize(numDigis);
19
20        // Kopieren von Zwischenvektor zu Output
21        for(i = 0; i < numThreads; i++){
22            // Berechnung der Offsets für den Kopiervorgang
23            partialSum = 0;
24            for(x = 0; x < i; x++){
25                partialSum += space[x].size();
26            }
27            // Kopieren der Daten
28            for(x = 0; x < space[i].size(); x++){
29                outDigis[partialSum + x] = space[i][x];
30            }
31        }
32        return outDigis;
33    }
```

Listing 7.3: Pseudocode der Kernfunktionalität des parallelen Unpacker-Tasks

7.4 Pseudocode des optimierten parallelen Unpacker-Tasks

```
1 MsUnpackChain(...){
2     // Anlegen des Zwischenvectors für paralleles Schreiben
3     std::vector<std::vector<CbmStsDigi>> space (numMS);
4
5     // Ausführen des Unpacker Algos
6     for (compID = 0; compID < nrCompsInTs; compID++){
7         if (comp(compID) != STS){
8             rejectedComps++;
9             continue;
10        }
11        ...
12        #pragma omp parallel for
13        for (msID = 0; msID < nrMsInComp; msID++){
14            ...
15            result = UnpackerAlgo(ms(msID));
16            index = (compID - rejectedComps) * numMsInComp + msID;
17            space[index].insert(result);
18        }
19    }
20    ...
21    return outDigis;
22 }
```

Listing 7.4: Pseudocode der Kernfunktionalität des optimierten sequenziellen Unpacker-Tasks

7.5 Code NoInitAllocator

```
1  template<class T>
2  class NoInitAlloc {
3      public:
4      using value_type = T;
5
6      T* allocate(size_t size) {
7          return static_cast<T*>(std::malloc(size * sizeof(T))); }
8      void deallocate(T* p_t, size_t) { std::free(p_t); }
9
10     template<class U, class... Args>
11     void construct(U*, Args&&...){}
12 };
```

Listing 7.5: Code des NoInitAllocators für einen STL-Vector

7.6 Pseudocode des Single-Thread GPU Unpacker-Algorithmus

```
1 MsUnpacker::Operator(xpu::Buffer<stsxyter::Message> messages ,
2 xpu::Buffer<CbmStsDigi> outDigis , ...){
3     time = ...
4
5     if (message[1].Typ != TsMsb) {return;}
6     ProcessTsmsbMessage(message[1], time);
7
8     for (messageNr = 2; messageNr < messages.size(); messageNr +=
9         messageBufferSize){
10        ...
11        messageIndex = ... + messageNr
12
13        // Kopieren der Daten zum Smem (alle Threads)
14        for(i = threadID; i < SmemSize; i += blockSize){
15            // Zugriff auf Messages muss abhängig von threadID sein
16            smem.messageBuffer[i] = messages[messageIndex + i]
17        }
18
19        // Sequenzielles entpacken der Messages durch Thread 0
20        if (threadID == 0){
21            // Pointer auf Outputabschnitt des Smem
22            CbmStsDigi* smemDigis = smem.digiBuffer[0];
23            ...
24            for (j = 0; j < messageBufferSize; j++){
25                // Verarbeitung der Messages je nach Messagetyt
26                switch (smem.messageBuffer[j].Typ) {
27
28                    // Hitmessage wird zu Digi umgewandelt
29                    case Hit: {
30                        ProcessHitMessage(..., time, smem.messageBuffer[j], smemDigis);
31                        break;
32                    }
33                    // TsMsb Message wird für Epochenupdate genutzt
34                    case TsMsb: {
35                        ProcessTsmsbMessage(smem.messageBuffer[j], time);
36                        break;
37                    }
38                }
39            }
40        }
41    }
```

```

42     // Kopieren der Output Daten von Smem in Hauptspeicher (alle Threads)
43     for (k = threadID; k < numDigis; k += blockSize){
44         index = ...
45         outDigis[index] = smem.digiBuffer[k];
46     }
47 }
48 ...
49 }
50
51 MsUnpacker::ProcessHitMessage(...){
52     // Nutzung der Daten zum bestimmen der 4 Digi Parameter
53     ...
54     smemDigis[...] = CbmStsDigi(address, channel, messageTime, charge);
55 }
56
57 MsUnpacker::ProcessTsmsbMessage(...){
58     // Update der Epoche und ggf. EpochenCycle
59     if (aktuelleEpoche < letzteEpoche){Cycle++;}
60     ...
61     update time;
62 }

```

Listing 7.6: Pseudocode der Kernfunktionalität des Single-Thread GPU Unpacker-Algorithmus

7.7 Pseudocode des Multi-Thread GPU Unpacker-Algorithmus

```
1 MsUnpacker::Operator(xpu::Buffer<stsxyter::Message> messages,
2                       xpu::Buffer outDigis, ...){
3     ...
4
5     if (message[1].Typ != TsMsb) {return;}
6     ProcessTsmsbMessage(message[1], time);
7
8     // Präparieren des Smem für ersten Durchlauf
9     if (threadId == 0){
10        ctx.smem().bcastBlockDigiOffset = 0; // = numDigis
11        ctx.smem().tsMsbMsg[0] = secondMessage.epochs;
12        ctx.smem().bcastNCycles = 0;
13    }
14
15    for (messageNr = 2 + threadID; messageNr < messages.size(); messageNr +=
16         blockGröße){
17        ...
18        // festlegen des eigenen Message Typen
19        int isTsMsbMsg = currentMsg.Typ == TsMsb;
20        int isHitMsg = currentMsg.Typ == Hit;
21
22        // Bilden der Prefixsumme über TsMsb-Messages
23        tsMsbIdx = scan(isTsMsbMsg);
24
25        cycleWrap = 0;
26        if (isTsMsbMsg){
27            // Schreiben der eigenen Epoche ins Shared Memory
28            smem.tsMsbBuffer[tsMsbIdx] = currentEpoch;
29
30            // Cycle Inkrement prüfen
31            if (currentEpoch < smem.tsMsbBuffer[tsMsbIdx - 1]){cycleWrap++;}
32        }
33        // Sammeln der Threads
34        xpu::barrier();
35
36        // Scan über alle Cycle Inkrements in diesem Batch
37        nCycleWraps = scan(cycleWraps);
38
39        // Entpacken der Digis
40        if (isHitMsg){
41            currentEpochTime = ((... + nCycleWraps + smem.bcastNcycles) * ...)
```

```

42     digi = ProcessHitMessage(currentMsg, currentEpochTime, ...)
43 }
44
45 // Sammeln der Threads
46 xpu::barrier();
47
48 // Bilden des Digi Index, zum Schreiben in den Hauptspeicher
49 digiIndex = scan(isHitMsg) - 1;
50
51 // Schreiben des Digis
52 if (isHitMsg){
53     outDigis[... + digiIndex + smem.bcastBlockDigiOffset] = digi;
54 }
55
56 // Broadcasten der Smem Werte durch den letzten Thread
57 if (threadID == (blockGröße - 1)){
58     smem.bcastBlockDigiOffset += digiIndex + 1;
59     smem.bcastNCycles += nCycleWraps;
60     if (tsMsbIdx > 0){
61         smem.tsMsbBuffer[0] = smem.tsMsbBuffer[tsMsbIdx];
62     }
63 }
64 xpu::barrier();
65 }
66 ...
67 }
68
69 MsUnpacker::ProcessHitMessage(...){
70     // Nutzung der Daten zum bestimmen der 4 Digi Parameter
71     ...
72     return CbmStsDigi(address, channel, messageTime, charge);
73 }

```

Listing 7.7: Pseudocode der Kernfunktionalität des Multi-Thread GPU Unpacker-Algorithmus

7.8 Glossar

- *Hit*: Punkt mit 3 Koordinaten, an dem ein Teilchen den Detektor passiert.
- *STS-Detektor*: Silicon Tracking System; Detektor zur Bestimmung von Teilchenflugbahnen.
- *Track*: Flugbahn eines Teilchens, die aus den Daten des STS-Detektors errechnet wird.
- *Online Verarbeitung*: Datenverarbeitung in Echtzeit während des Experiments.
- *Message*: Datenstruktur, die Rohdaten des Detektors enthält.
- *ALICE*: A Large Ion Collider Experiment am CERN.
- *Shared Memory*: Ein Chipspeicher zum schnellen Austausch von Daten zwischen GPU Threads.
- *Warp*: Kleinste Organisationseinheit einer NVIDIA GPU. Fasst 32 Threads zusammen, die im Lockstep arbeiten.
- *Wavefront*: Kleinste Organisationseinheit einer AMD GPU. Fasst 64 Threads zusammen, die im Lockstep arbeiten.
- *Block*: Variable Organisationseinheit für zusammenarbeitende Threads mit Shared Memory. Die Blockgröße bestimmt Threadzahl und sollte möglichst ein Vielfaches der Warp-/Wavefrontgröße sein.

Erklärung zur Abschlussarbeit

gemäß § 34, Abs. 16 der Ordnung für den Masterstudiengang Informatik
vom 17. Juni 2019

Hiermit erkläre ich

Heinemann, Sebastian
(Nachname, Vorname)

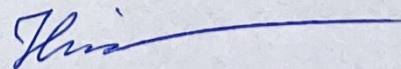
Die vorliegende Arbeit habe ich selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst.

Ebenso bestätige ich, dass diese Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

Zudem versichere ich, dass die von mir eingereichten schriftlichen gebundenen Versionen meiner Masterarbeit mit der eingereichten elektronischen Version meiner Masterarbeit übereinstimmen.

Frankfurt am Main, den

09.10.2023



Unterschrift der/des Studierenden