Fachbereich Informatik und Mathematik
Institut für Informatik

Lehrstuhl für Architektur von Hochleistungsrechnern

Masterarbeit

Kilian Hunold

Thema:	Optimierung der lokalen Rekonstruktion beim STS-Detektor des CBM-Experiments zur effizienten Prozessierung auf GPUs
Matrikelnr.:	6045577
Version vom:	25. Februar 2022
Betreuer:	PD. Dr. Andreas Redelbach

Abstract

Das Collision Baryobinc Matter (CBM) Experiment wird vom Facility for Antiproton and Ion Research (FAIR) durchgeführt. Mittels Teilchenbeschleunigern werden hochenergetische Zusammenstöße von Schwerionen erzeugt, um die neu entstehenden Teilchen mittels des Quantenchromo dynamik-Phasendiagramms (QCD- Phasendiagram) zu erforschen.
Eigenschaften des durch die Kollision entstandenen Plasmas werden mithilfe einer Kette von Detektoren aufgezeichnet und anschließend untersucht.
Einer der Detektoren des Experiments ist das so genannte Silicone Tracking System (STS), welches den lokal gesehen ersten Detektor in der Reihe der unterschiedlichen Messgeräte darstellt. Es soll den Verlauf der Teilchen durch Messung der Ladung und Position des Eintreffens rekonstruieren. Diese Messungen werden im Nachhinein im First Level Event Selector (FLES) weiterverarbeitet. Er stellt die zentrale Station für die Berechnung und Verarbeitung der Daten dar, um diese direkt auszuwerten und nicht alle gemessenen Werte speichern zu müssen. Dies ist notwendig, da eine solche Masse von bis zu 1 TB/s an gemessenen Daten nur schwerlich gespeichert werden kann.
Daher muss der FLES eine möglichst optimale Verarbeitung der Daten vorweisen. Gerade die zeitliche Performance ist hier der entscheidende Faktor. Aus diesem Grund wird der FLES mit einer Vielzahl an Hardware ausgestattet und sollte dementsprechend mit Software zum Laufen gebracht werden, die die Hardware möglichst optimal ausnutzt.
Graphics Processing Units (GPUs) bieten sich für die Verarbeitung solcher Daten an, da sie aufgrund ihrer Bauweise hoch parallel arbeiten können. In dieser Arbeit angesprochene Algorithmen optimieren die lokale Rekonstruktion des Silicone Tracking Systems im FLES. Duch die Zuhilfenahme von GPUs und dem XPU-Framework (basierend auf CUDA) werden bereits bestehende Algorithmen für eine parallele Abarbeitung der Rekonstruktionsschritte abgewandelt.
Dabei wurde durch alle Anpassungen und Optimierungen ein Speedup von 21 der Echtzeitverarbeitung erzielt. Im Einzelnen wurden bei der Rekonstruktion das Sortieren, das Finden von Clusterstrukturen und die anschließende Berechnung der Hits angepasst, um diesen Speedup zu erzeugen.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Einst wurde angenommen, dass Atome die kleinsten existierenden Teilchen seien, aus denen unsere gesamte Welt zusammengesetzt ist. Durch die Grundlagenforschungen konnte dies jedoch widerlegt werden. Die uns nun bekannten kleinsten Teilchen sind Quarks. Die Teilchenphysik ist die Wissenschaft, die sich mit Thematiken wie dieser auseinandersetzt, um das grundlegende Wissen zu erweitern. Hierzu werden Theorien aus naturwissenschaftlichen Gebieten aufgestellt und erforscht, um diese anschließend bestätigen oder widerlegen zu können. Dazu werden Experimente an Universitäten und Forschungseinrichtungen ins Leben gerufen.
Solche Experimente liefern häufig große Datenmengen, welche nur schwer in der geforderten Geschwindigkeit mit derzeitigen Methoden auf Langzeitspeicher übertragen werden können. Daher hat nicht zuletzt die hoch performante Datenverarbeitung hier einen gravierenden Stellenwert. Daten, die sonst sicher gespeichert werden müssten, werden nun durch Hochleistungsrechner und dem darauf laufenden Code effizient verarbeitet und somit sind es nur die (Teil-)Ergebnisse der Forschung, die auf Dauer gespeichert werden müssen.
Einen solchen Teilchenbeschleuniger befindet sich derzeit im Bau am GSI Helmholzzentrum für Schwerionenforschung [1]. Ein erstes Experiment ist hier am FAIR (Facility for Antiproton and Ion Research) das Compressed Baryonic Matter Experiment (CBM), bei dem sich mit den unterschiedlichen Stadien, Phasen stark verdichteter Materien und deren Eigenschaften auseinandergesetzt wird [2].
Grundlage hierbei ist, wie zuvor erwähnt, die Kollision von Schwerionen, bei der die Quark-Struktur im Inneren des Ions gelöst und so ein Quark-Gluon-Plasma Aggregatzutand erzeugt wird. Die Forschung soll Aufschlüsse über das grundlegende Verhalten von Quarks geben, um zum Beispiel herauszufinden, warum keine isolierten Quarks existieren, wie deren Masse zustande kommt und um Rückschlüsse über das Verhalten von Neutronensternen zu ziehen. Des Weiteren sollen erstmals Messungen über Elektron-Positron-Paare, Myonenpaare oder schwere Quarks getätigt werden. Dazu werden Eigenschaften wie Masse, Energie, Anzahl, Zusammensetzung und das Ausbreitungsverhalten der Teilchen gezielt ermittelt und analysiert. Im CBM-Experiment werden ca. zehnmillionen Kollisionen pro Sekunde erzeugt, um auch seltene Ereignisse häufiger hervorzurufen. Beispielsweise werden manche Teilchen nur kurz während des Zerfalls von Ionen messbar

und verschwinden anschließend wieder. Es wird dazu eine Kette von Detektoren eingesetzt, die das unterschiedliche Verhalten der Teilchen messen sollten und Hochleistrungsrechner, die die gemessenen Daten anschließend weiterverarbeiten sollen. Die Detektoren nutzen dazu unterschiedlichste Verfahren und arbeiten dabei mit einer Datenrate von ca. 10Mhz [3] [4]. Man kann also davon ausgehen, dass das Experiment während der Ausführung bis zu 1 TB/s an Daten pro Sekunde liefern wird.
Diese Arbeit widmet sich der Optimierung von einem Programmiercode, der von einer CPU-Ausführung in eine parallele GPU-Ausführung überführt werden soll. Es wird sich im Genauen mit der lokalen Rekonstruktion von Detektordaten des STS (Silicone Tracking System) auseinandergesetzt. Da der Detektor der erste in der Kette der Detektoren ist, und somit direkt hinter dem Target steht, bewirkt die Codeoptimierung nicht nur eine höhere Performanz für den STS selbst, sondern auch für das gesamte Experiment.

1.2 Problemstellung

Zielstellung der hier vorliegenden Arbeit ist die Optimierung der lokalen Rekonstruktion der Daten des Silicone Tracking Systems. Aus der hohen Wechselwirkungsrate des Detektors resultiert ein hoher Datendurchsatz für die folgenden Verarbeitungsschritte [4]. Daher müssen die vom Detektor gelieferten Daten schnell verarbeitet werden, sodass nachfolgende Ereignisse keine langen Wartezeiten haben. Die Optimierung könnte sowohl durch mehr, beziehungsweise bessere Hardware geschehen, doch durch hohe Kosten und Platzbedarf ist diese Möglichkeit weniger praktikabel als die Optimierung der Algorithmen, zumal die derzeitige Hardware noch nicht komplett ausgelastet wird. Die aktuell genutzten Algorithmen der lokalen Rekonstruktion arbeiten oft nur linear oder teilparallel auf Prozessoren. Daher bietet der Code noch viel Potential für weitere Optimierungen. Sie kann nicht nur statt auf einer CPU auf einer schnelleren GPU laufen, sondern auch auf dieser höchst parallel ausgeführt werden.
Natürlich bestünde auch die Option, den Code, wie auch jetzt schon möglich, parallel auf CPUs laufen zu lassen, aber durch die höhere Skalierbarkeit und Geschwindigkeit von Grafikkarten, bietet sich diese zwar komplexere, aber performantere Lösung an.
Um eine parallele Lösung für Grafikkarten zu entwickeln reicht es an dieser Stelle aber nicht aus, nur den bestehenden Code auf eine GPU zu überführen. Es müssen zudem Algorithmen und neue Rechenvorschriften zur Problemlösung entwickelt und implementiert werden, die zu demselben Ergebnis führen wie der lineare Code, da für die Auswertung der Daten eine hohe Genauigkeit unabdingbar ist. In Anbetracht dieser Faktoren widmet sich diese Arbeit der Optmierung der Algorithmen auf Grafikkaten unter Zuhilfenahme des von Nvidia bereitgestellten Cuda-Frameworks (siehe 2.6), welches einem ermöglicht, einzelne Böcke und die darin befindlichen Threads anzusprechen.
Man könnte dabei vermuten, dass die Algorithmen nur auf Nvidia-Hardware getestet würden, doch diese Arbeit zeigt unter anderem auch den Vergleich der Optimierungen auf unterschiedlicher Hardware. Dabei werden sowohl eine CPU, Nvidia als auch Karten des Herstellers AMD verwendet.

2 Grundlagen

2.1 Compressed Baryonic Matter (CBM)

Das Compressed Baryonic Metter Experiment ist Teil des vom FAIR sich im Bau befindlichen Teilchenbeschleunigers SIS100, welcher an den bereits besteheden SIS18 angeschlossen wird. In Auftrag gegeben wurde das Experiment vom GSI Helmholzzentrum für Schwerionenforschung in Darmstadt und dient der Erforschung von Teilchen von hoher baryonischer Dichte des QCD-Phasendiagramms. Der SIS100 soll 2026 [5] fertiggestellt werden und ist auf der schematischen Abbildung der Beschleunigeranlage (Abbildung 1) dargestellt. Das CBM-Experiment (siehe Abbildung 2) wird mehrere Konfigurationen haben. Beispielsweise kann dabei der RICH Detektor (Ring Imaging Cherenkov) und der MUCH Detektor (Muon Chamber) ausgetauscht werden. Alle anderen Detektoren bleiben ansonsten bei den derzeitigen Konfigurationen bestehen. Darunter befindet sich auch das in dieser Arbeit thematisierte Silicone Tracking System. Bei dem Experiment selbst werden Ionen beschleunigt und auf einem Target zur Kollision gebracht. Daraus entsteht ein Quark-Ionen-Plasma, wovon anschließend unterschiedlichste Eigenschaften von den Detektoren untersucht und später analysiert werden.

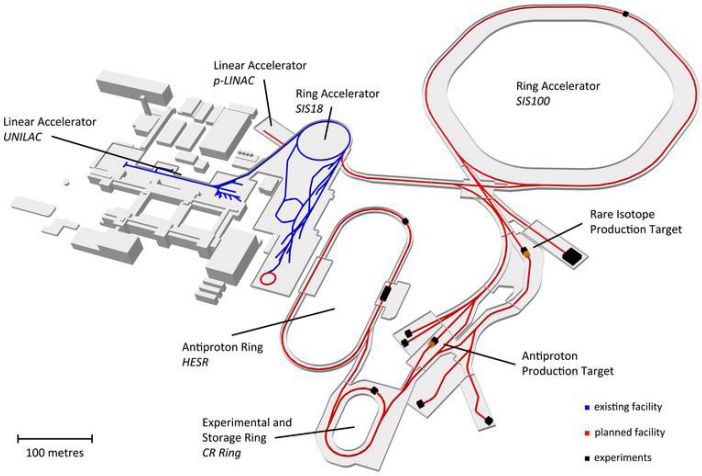

Abbildung 1: Layout Beschleunigeranlage [3]

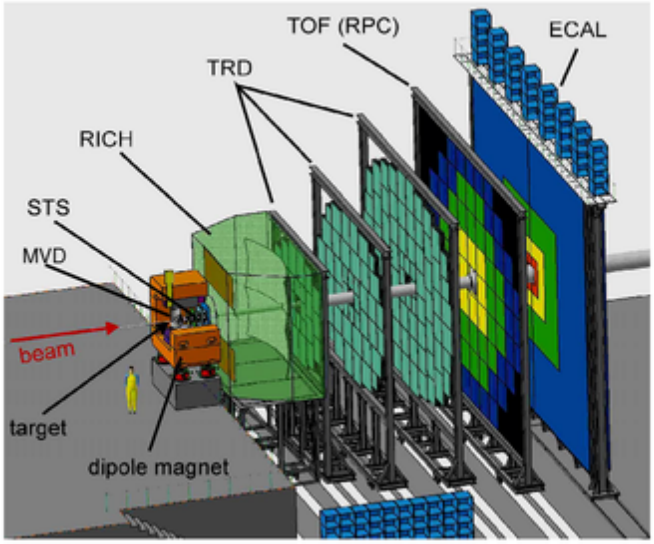

Abbildung 2: Layout CBM-Experiment [3]

2.2 Silicone Tracking System

Das Silicone Tracking System ist ein eigens vom GSI entwickelter Detektor für das CBM-Experiment und untersucht die Flugbahnen und das Momentum von geladenen Partikeln, entstanden aus der Schwerionenkollision [4]. Es ist der erste in einer Reihe von Detektoren, welche die unterschiedlichen Eigenschaften des entstandenen Plasmas messen sollen. Das STS arbeitet mit einer Rate von bis zu 10Mhz [4] [3], um die bis zu 1000 geladenen Teilchen pro Sekunde erfassen zu können. Gemessen werden die Teilchen auf 8 Sensorplatten, welche im Abstand von 30cm bis 100cm hinter dem Target aufgestellt werden.
Jede Sensorplatte besteht aus etwa 900 Microstrip-Sensoren (später als Module bezeichnet), welche sowohl auf der Vor- als auch auf der Rückseite der Module platziert sind, um Ausschläge auf den Sensoren später zu einem Hit zusammenfügen zu können. Jeder Microstrip-Sensor beinhaltet ca. 2000 300 µm-Kanäle in einem Abstand von ca. 58 µm zueinander.
Dabei kommt es vor, dass trotz des Aufschlags eines Teilchens an einer bestimmten räumlichen Position auch mehrere Kanäle des Sensors ausschlagen. Ein solcher multipler Ausschlag von benachbarten Sensoren durch ein Teilchen wird als Cluster interpretiert.

2.3 First Level Event Selector (FLES)

Die zentrale Auswertung der gemessenen Daten des CBM-Experiments übernimmt der First Level Event Selector (FLES). Dieser ist ein mit einer Vielzahl von unterschiedlichen Recheneinheiten bestückter Hochleistungsrechner. Dazu gehören in einem solchen heterogenen System nicht nur Central Processing Units (CPUs), sondern auch nicht zuletzt Field Programmable Gate Arrays (FPGAs) und GPUs, um eine effiziente Berechnung zu gewährleisten [6].
Ausgestattet ist das System mit etwa 1000 Input-Knoten, 60.000 Kernen auf den Rechenknoten und einer damit einhergehenden Netzwerk-Infrastruktur, um alle Knoten miteinander zu verbinden [6].
Die an dieses System übergebenen gemessenen Daten, sind maßgeblich über einen Zeitstempel, die räumliche(geometrische) Position und die gemessene Ladung zu identifizieren. Die in Microslices an den FLES gelieferten Daten werden dort anhand des Zeitstempels aufsteigend sortiert in sogenannte Timeslices aufgeteilt, um eine geordnete Übergabe zu gewährleisten. Damit Informationen dennoch parallel abgearbeitet werden können, werden die Timeslices überlappend generiert, sodass keine Digis (gemessenes Datum eines Detektors) und daraus resultierende Datenzusammenhänge verloren gehen. Wie groß dabei ein Zeitintervall ist, wird durch empirisches Ausprobieren ermittelt. Hilfreich sind die Timeslices nicht nur für die Organisation der Daten, sondern auch für die optimale Auslastung der Rechenknoten, da durch die Organisationseinheiten die Rechenlast unter den unterschiedlichen Rechenknoten gleichmäßig aufgeteilt werden kann.

2.4 CbmRoot

CBMRoot [7] ist das im CBM Experiment grundlegend verwendete Software-Paket. Es basiert auf FairRoot [8], einem Framework, das die Basis für alle FAIR Experimente der GSI darstellt. FairRoot basiert wiederum auf dem Root Framework des CERN. Es dient der Simulation, Analyse und Verarbeitung von Daten in den Experimenten des FAIR. Das Framework ist eine Ansammlung von Tools und Funktionen, die für die Nutzung innerhalb der Projekte bereitgestellt werden, um die Analyse, Verarbeitung, Speicherung und Virtualisierung von großen Datenmengen zu gewährleisten. CBMRoot selbst beinhaltet Funktionen und Datenstrukturen, die innerhalb des CBM-Experiments ihre Anwendung finden. Die Speicherung und Verarbeitung soll hier möglichst effizient passieren und exakt auf die Bedürfnisse des Experiments abgestimmt werden. So lassen sich beispielsweise für den hier behandelten STS Daten in Digis speichern,

die später in Cluster und Hits verarbeitet werden. Diese Klassen und Datenstrukturen bieten alle Funktionalitäten, die bei der Analyse später genutzt werden.

2.5 Graphics Processing Unit (GPU)

Eine Graphics Processing Unit ist eine Recheneinheit, die speziell für die Berechnungen von Grafiken optimiert ist. In modernen Computern ist sie häufig zusätzlich zur Central Processing Unit (CPU) verbaut, um Aufgaben wie beispielsweise die Berechnung von Darstellungen und Grafiken zu übernehmen [9]. Dabei werden Prozesse und die dazugehörigen Daten von der CPU auf die GPU ausgelagert und dort verarbeitet.
Der Grafikprozessor ist mit einem eigenen Speicher und eigenen Recheneinheiten ausgestattet, die grundlegend für die Verarbeitung von Vektor- und Matrixoperationen optimiert sind. Er nutzt dabei eine Vielzahl von Threads, die untergeordnet in Blöcke aufgeteilt sind (siehe Abbildung 3). Alle genutzten Threads durchlaufen dabei grundsätzlich erst einmal den gleichen Programmcode, können aber innerhalb der Blöcke einzeln angesprochen werden, um den Ablauf je nach Anwendung zu ändern. Dadurch können eine Vielzahl von unterschiedlichen Aufgaben gleichzeitig von unterschiedlichen Threads abgearbeitet werden (siehe SIMD). Durch den auf der GPU liegenden Speicher können die einzelnen Threads schnell auf die dort liegenden Daten zugreifen und müssen dabei nicht den Umweg über die CPU zum Hauptspeicher des Rechners gehen. Nicht zu vernachlässigen ist dabei die große Anzahl an Threads, die eine moderne GPU im Gegensatz zu einer CPU bietet. Die hier zu Benchmark-Zwecken verwendete CPU bietet beispielsweise 32 Threads. Die dabei genutzte GPU kann schon pro Threadblock eine maximale Anzahl von 1024 Threads verwalten und ist daher schon von den Spezifikationen der stärkere Kandidat, was eine parallele Abarbeitung angeht.

2.6 Cuda Framework

Um die in Abschnitt 2.3 behandelten GPUs anzusprechen werden Programmierschnittstellen benötigt, die für eine Kommunikation zwischen dem Programm und der Recheneinheit sorgen. Cuda ist eine solche Programmierschnittstelle, die vom Grafikchip-Hersteller Nvidia zur Entwicklung von Programmen auf Grafikkarten entwickelt wurde. Mit CUDA können Programme auf mehreren logischen Blöcken gleichzeitig ausgeführt werden. Die Block-Thread-Struktur ist in Abbildung 3 gezeigt.
Der Darstellung ist zu entnehmen, dass das Grid in mehrere Blöcke aufgeteilt

ist. Jeder Block beinhaltet die gleiche Anzahl an Threads, welche über das Cuda-Framework einzeln angesprochen werden können. Von allen Blöcken und somit den darin enthaltenen Threads wird derselbe Code ausgeführt und hat somit eine SIMD Struktur.
Mithilfe von Cuda lassen sich somit Berechnungen auf der GPU ausführen. Doch auch der Speicher kann durch CUDA auf der Grafikkarte verwaltet werden. So ist es möglich den einzelnen Threads oder Blöcken einen eigenen Speicher zuzuweisen, um zeitgleiches Schreiben und Lesen auf denselben Daten zu vermeiden. Zudem bietet das Framework atomare Operationen, um auch bei hoher Threadanzahl die Datenkonsistenz zu gewährleisten.

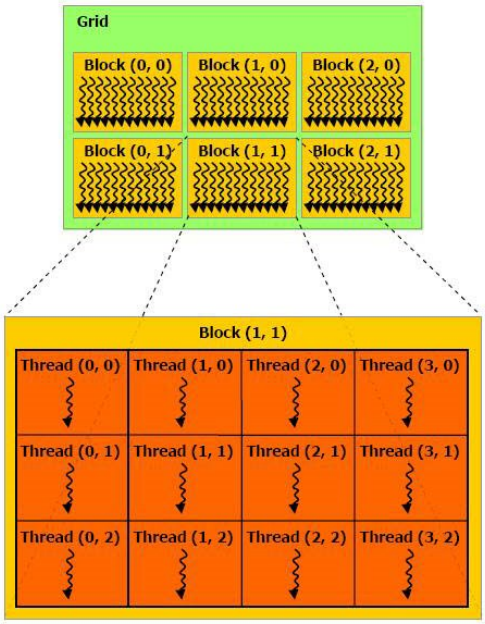

Abbildung 3: Schematische Thread-Block-Struktur einer GPU [10]

2.7 XPU Framework

Das XPU-Framework [11] ist eine von dem GSI selbst entwickelte Macroschnittstelle zur CPU und GPU-Programmierung. Die grundlegende Aufgabe besteht darin eine Programmierschnittstelle zu schaffen, die es ermöglicht, den gleichen Code im Hintergrund auf beliebiger Hardware auszuführen.
Da das CBM-Experiment in Zukunft auf unterschiedlichster Hardware ausgeführt werden soll und derzeit noch unbekannt ist, muss das Framework die Möglichkeit bieten, mit jeder Art von Hardware kommunizieren zu

können. Auch Arbeitsgruppen, die derzeit keinen Zugriff auf eine GPU haben, müssen dabei in der Lage sein, den Code auf einer CPU auszuführen. Dieser Ansatz ist nicht neu, denn sowohl openCL als auch eine eigens beim Alice-Experiment entwickelte Schnittstelle verfügen bereits über diese Möglichkeiten. Da aber openCL wenig Unterstützung von den allgemeinen Hardwareherstellern erhält und das in Alice verankerte Framework nur schwerlich davon zu trennen ist, war die Idee an dieser Stelle, ein neues Framework zu implementieren, welches auf modernen Schnittstellen wie CUDA, HIP und openMP aufsetzt.
XPU soll dabei nicht nur über die Option verfügen unterschiedliche Hardware anzusprechen, sondern bereits derart grundlegende Funktionalitäten beinhalten, die in den Projekten gebraucht werden. Beispiel dafür ist das in 3.1 vorgestellte Sortierverfahren welches im XPU bereitgestellt werden soll.
Die Größe der Threadblocks und Anzahl der genutzten Threads auf der unterschiedlichen Hardware muss pro genutztem Kernel am Ende noch empirisch ermittelt werden, da dies natürlich hardwarespezifisch ist, aber schon zur Compilezeit des Programms bekannt sein muss.

2.8 Genutzte Hardware

Die in den folgenden Abschnitten gezeigten Plots wurden alle auf derselben Hardware durchgeführt. Dabei wird in den Darstellungen unter anderem der Unterschied der Performance der Algorithmen auf unterschiedlichen Grafikkarten aufgezeigt. In der unten gezeigten Tabelle wird die genutzte Hardware vorgestellt und verglichen.

Type	**ComputeUnits**	**Clock**	**Performance (SP)**	**Bandwidth**
Intel(R) Xeon(R) Gold 6130 (x2)	16 Cores (32Threads)	2.1 GHz		119 GB/s
Nvidia RTX 2080Ti	48 CU (32 Thrd/Blk)	1635 MHz	13.8 TFLOPs	616 GB/s
Amd Vega 20 (Radeon VII)	60 CU (64 Thrd/Blk)	1400 MHz	14,2 TFLOPS	1024 GB/s
Amd Mi100	120 CU (256 Thrd/Blk)	1502 MHz	23,1 TFLOPS	1228 GB/s

2.9 Simulierte Daten und Messungen

Die für die Benchmarks genutzten Daten sind aus bereits bestehenden Projekten des CBM-Experiments simuliert, da noch keine Realdaten für die

Testung von Algorithmen vorliegen.
Die Daten wurden basierend auf einer zentralisierten Kollision von Goldteilchen mit einer Energiespanne von 10 Gigaelektronenvolt (GeV) generiert. In den Benchmarks werden Simulationen mit unterschiedlichen Anzahlen von Kollisionen verwendet, um ein breiteres Bild über die Performance der Algorithmen und deren Anpassungen zu erzeugen. Es wurden Anzahlen von 100 bis 2000 Kollisionen verwendet, um sowohl ein geringes, als auch ein hohes Aufkommen von Messungen in den Benchmarks abzudecken. Dabei werden Digis (Messwerte der Detektoren) in der Größenordnung von ca. 2 Mio. bis 30 Mio. erzeugt.
Die Messungen der optimierten Algorithmen wurden auf Basis der Ausführzeiten der Kernel des XPU-Frameworks (siehe 2.7) angefertigt. Die Messungen des originalen Algorithmus basieren hingegen auf der 'TStopwatch'-Klasse des Root-Frameworks.
Jeder Datenpunkt in den gezeigten Statistiken basiert auf 50 gemessenen Durchläufen. Die Schwankungen sind genau einmal die Standardabweichung der gemessenen Daten.
Wie in 2.8 dargestellt gibt es bei der genutzten Harware 2 CPUs mit jeweils 32 Threads. Somit könnten bei den Messungen insgesamt 64 Threads verwendet werden. Da es bei den Messungen aber immer wieder zu Schwankungen außerhalb der Norm kam, sowohl bei der Ausführung des in der Arbeit gezeigten als auch beim bereits vorhandenen Algorithmus, wurde sich dazu entschlossen die Messungen innerhalb der Arbeit auf 32 CPU-Threads zu beschränken.
In Abbildung 4, welche basierend auf dem in Abschnitt 3.2.3.4 erläuterten Algorithmus generiert wurde, lässt sich gut erkennen, dass die 64-Thread-Variante nicht nur gravierende Schwankungen in der Standardabweichung, sondern auch im Mittelwert aufweist. Zudem lässt sich der Grafik entnehmen, dass die Algorithmen zeitlich stabiler laufen, je weniger Threads verwendet werden.

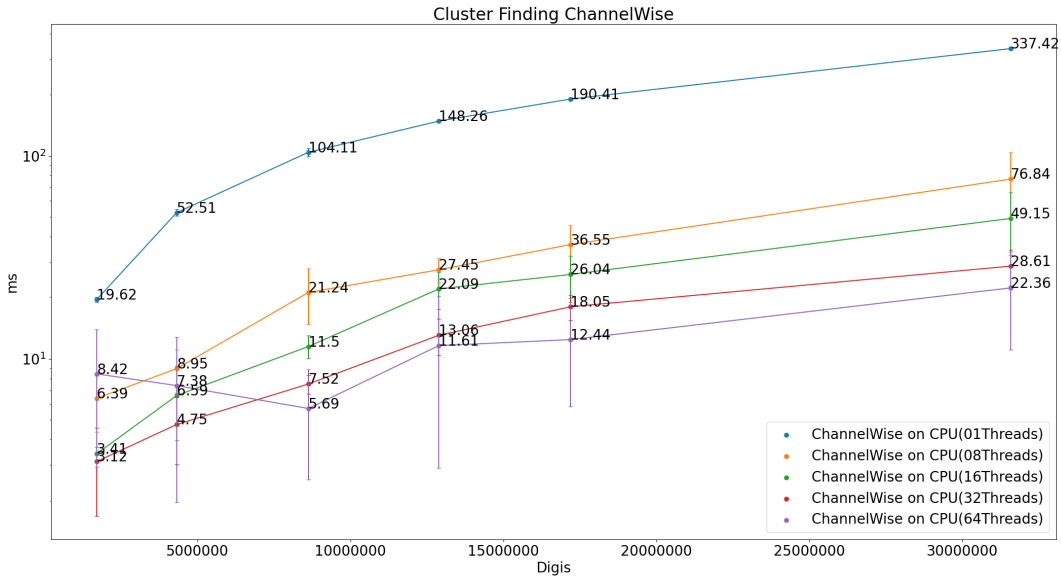

Abbildung 4: Vergleich der Zeitmessung unterschiedlicher CPU-Threadzahlen

3 Optimierung der Lokalen Rekonstruktion

Der Algorithmus zur Berechnung von Hits besteht aus mehreren einzelnen Schritten, welche nur linear nacheinander abgearbeitet werden. Zuerst werden die Digis in Gruppen (sogenannte Cluster) zusammengefasst. Ein Cluster sind auf ein Teilchen zurückzuführende zusammengehörige Detektorausschläge. Dazu müssen die Digis je nach Algorithmus vorher jedoch anhand des zeitlichen Einschlags sortiert werden. Im Anschluss an das Finden werden die Cluster noch einmal zeitlich sortiert. Abschließend müssen die berechneten Cluster auf der Vor- und Rückseite des Detektors (siehe 2.2) verglichen und überlappende Cluster zu einem Hit zusammengefasst werden. Veranschaulicht wird der allgemeine Ablauf des Algorithmus in Grafik 5.

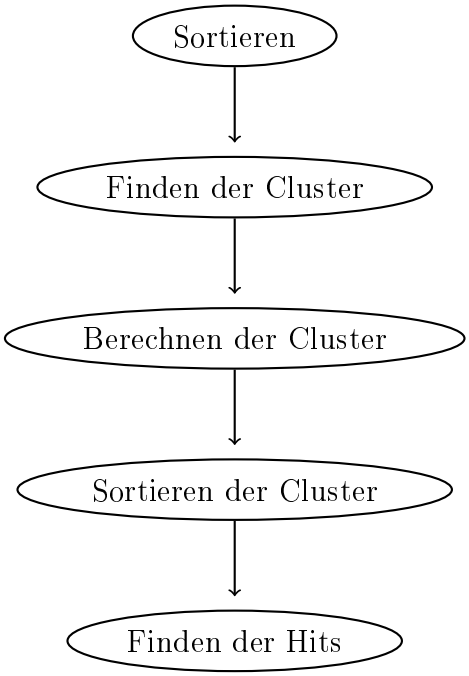

Abbildung 5: Ablauf der lokalen Rekonstruktion des STS

3.1 Digis Sortieren

Der erste Schritt der lokalen Rekonstruktion des STS ist das aufsteigende Sortieren der Digis anhand ihres Zeitstempels. Um diesen Schritt zu optimieren wird dem XPU-Framework eine neue Funktionalität hinzugefügt, welche optimaler sortiert als die derzeit genutzten Algorithmen. Bisher wurde die in C++ vorhandene STL-Funktionalität(Teil der Standard-Bibliothek aus C++) genutzt, um Arrays zu sortieren. Somit bietet es sich an dieser Stelle an, einen Algorithmus vorzustellen, welcher die Leistung einer Grafikkarte ausnutzt und durch Parallelität eine bessere Performance bekommt.
Im folgenden Abschnitt wird eine dem XPU-Framework beigefügt, parallele Variante des Mergesorts vorgestellt. Somit ist der Algorithmus nicht nur für den STS, sondern auch für alle anderen Abschnitte des CBM-Experiments zugänglich, die das XPU-Framework (siehe 2.7) als Grundlage verwenden. Implementiert wird dabei ein paralleler Mergesort.
Der Mergesort als stabiler Algorithmus, wird zum sortieren von Daten verwendet, die in einem eindimensionalen Array gespeichert werden. Der Algorithmus arbeitet dabei nach dem Teile und Herrsche- (Divide and Conquer-) Prinzip und wurde erstmals von John von Neumann vorgestellt. Der Mergesort-Algorithmus lässt sich durch seine Eigenschaften sehr gut parallelisieren und somit ideal auf Grafikkarten verarbeiten.

3.1.1 Funktionsweise des Mergesort

Wie in Abbildung 6 veranschaulicht geht der Algorithmus nach dem Teile und Herrsche Prinzip vor. Zuerst werden die Daten in die kleinstmöglichen Strukturen aufgeteilt und aufsteigend sortiert. Nach dem Sortiervorgang werden jeweils zwei der gebildeten Elemente zusammengefügt (Merge). Durch dieses Prinzip sind in jedem Schritt immer 2 sortierte Arrays zusammenzufügen. Dies wird wiederholt bis schlussendlich alle Daten in einem Array zusammengefasst wurden.

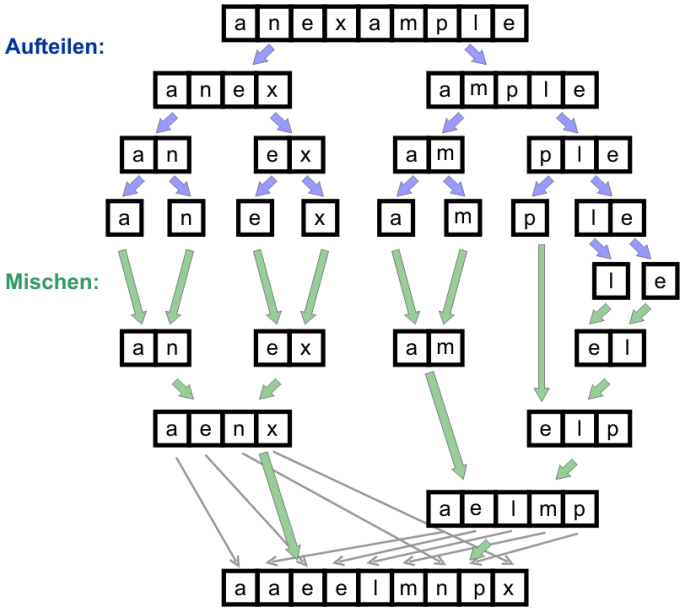

Abbildung 6: Schematisches Vorgehen des Mergesort [12]

3.1.2 Paralleles Mergen

Basis des hier in das XPU-Framework überführten Algorithmus ist die Arbeit von Green, Oded and Odeh, Saher and Birk, Yitzhak [13], bei dem 2 aufsteigend sortierte Arrays zu einem aufsteigend sortierten Array zusammengefügt werden. Dabei werden die Arrays in einer zweidimensionalen Matrix veranschaulicht (Abbildung 7), wobei Array A die x-Achse und Array B die y-Achse beschreibt.
Der Algorithmus arbeitet sich einen Pfad durch das Grid, bei welchem die im Abschnitt liegenden Zahlen verglichen werden. Gegeben ein $|A| \times |B|$ Grid, wobei A und B zwei zu zusammenfügende sortierte Arrays von Zahlen darstellen, wird ein Weg durch das Grid gesucht, bei dem es entweder nur

nach unten oder nach rechts geht. Der Pfad wird immer in die Richtung des kleineren Elements fortgesetzt. Durch die vorherige Sortierung und der Bedingung nur den Pfad nach rechts oder unten fortführen zu dürfen wird gewährleistet, dass $A[i] < B[j]$, auch $A[i] < B[j']$ für alle $j' > j$ ist [14].
Parallelisieren lässt sich dieser Algorithmus hervorragend, wie ebenfalls in Abbildung 7 rechts dargestellt. Das Grid wird durch die eingezeichneten Diagonalen in äquidistante Felder unterteilt. Dadurch kann jeder Thread einen der nun gebildeten Abschnitte übernehmen und komplett autonom bearbeiten. Vorteil dieser Methode ist ebenfalls, dass jeder Thread hier auf seinem eigenen Speicher arbeitet und somit Kollisionen, beim Lesen und Schreiben der Daten vermieden werden können. Dies schließt Wartezeiten auf parallel arbeitende Threads gänzlich aus.
Zusammenfassend lässt sich sagen, dass die Komplexität des Algorithmus bei einer Eingabegröße von $|A| + |B| = N$ liegt und p Prozessoren bei $O(N/p + log(N))$ und einer Arbeitskomplexität von $O(N + p \cdot log(N))$ liegt. Somit ist schlusszufolgern, dass der Algorithmus für $p < N/log(N)$, optimal ist [13]. Der in das XPU-Framework überführte Code basiert maßgeblich auf der Arbeit von Sean Baxter [15] und wurde durch kleine Anpassungen in das Framework eingebunden.

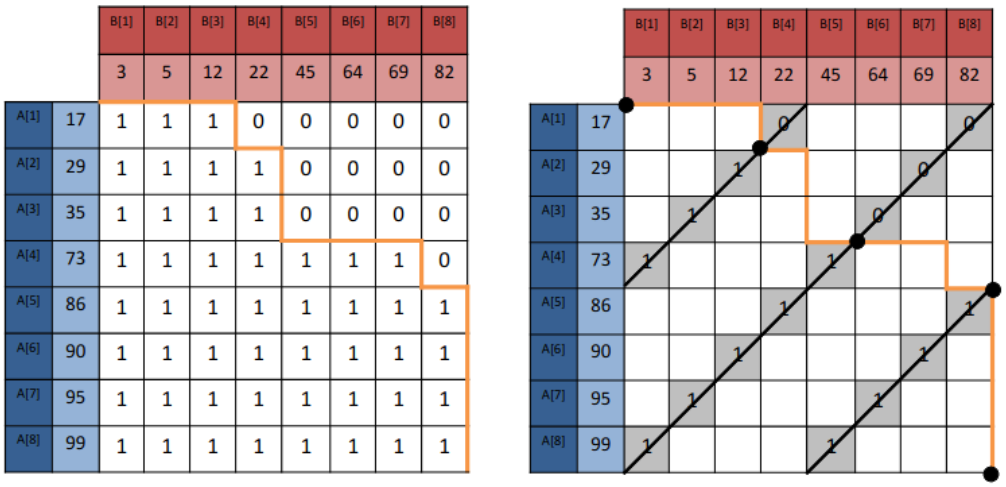

Abbildung 7: Mergepath durch 2 aufsteigend sortierte Arrays [13]

Abbildung 8 zeigt die Ergebnisse des Mergesorts im Vergleich zweier unterschiedlicher Grafikkarten und einer CPU. Die GPUs nutzen dabei den in Abschnitt 3.1.2 beschriebenen Algorithmus. Die CPU (Intel Xeon) hingegen nutzte den standard Sortieralgorithmus der STL.
Sortiert wurden hierbei STS Digis anhand eines Longfloats (32Bit). Die

Digis sind zudem bereits auf die Module aufgeteilt (aufgrund der Praxisnähe angelehnt and den Usecase), sodass nicht auf einem, sondern auf mehreren Arrays sortiert wird. Bei einer Eingabegröße von mehr als 30.000.000 Digis ist ein Geschwindigkeitsgewinn von etwa 15 bei dem Vergleich zum parallelen Sortieren über die STL Funktion zum GPU-Algorithmus zu erkennen. Verglichen wurden hierbei nur die Zeiten, die der Algorithmus zur Berechnung der Ergebnisse benötigt hat. Der IO wurde bei diesem Vergleich nicht betrachtet.

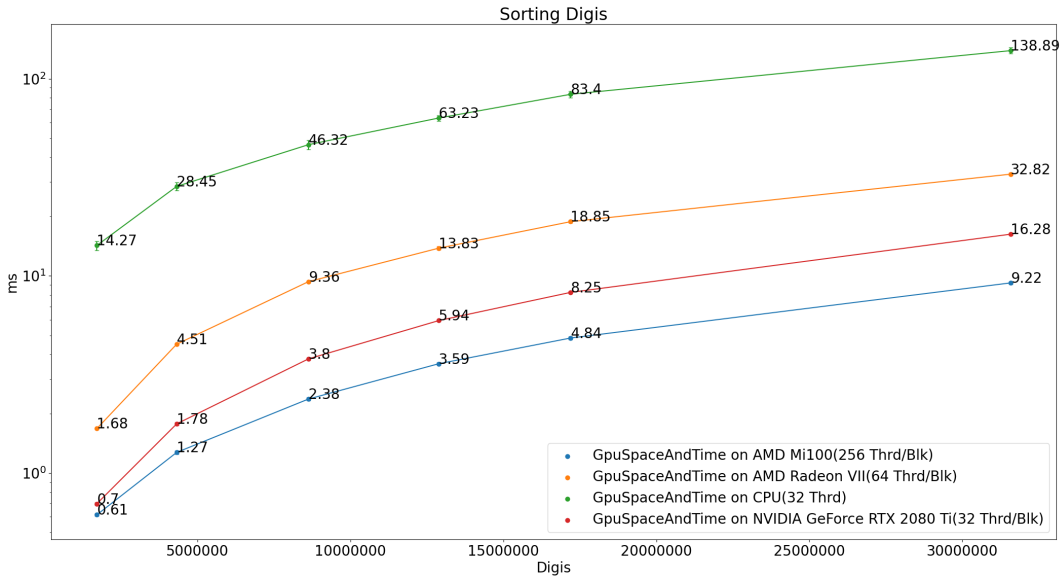

Abbildung 8: Ergebnisse des Mergsort der aufsteigenden Sortierung von eingegebenen Digis bei der lokalen Rekonstruktion des STS

3.2 Finden der Cluster

Auch wenn der Detektor die Digis bereits zeitlich sortiert übergibt, werden zu Beginn alle Daten noch einmal sortiert, um mögliche Fehler in den folgenden Prozessierungsschritten auszuschließen.
Um herauszufinden, ob zwei Digis zu einem Cluster gehören, müssen diese paarweise genau zwei Bedingungen erfüllen.

1. Die Digis müssen nebeneinander liegen. Das bedeutet, dass die Position der beiden Digis sich genau um 1 unterscheiden dürfen. Liegen zwei Digis im gleichen Kanal, so können sie nicht zu dem gleichen Cluster gehören. Dies lässt sich darauf zurückführen, dass der Detektor an einem Kanal zur gleichen Zeit nur einen Wert messen kann. Zwei Digis im selben Kanal resultieren also immer aus zwei unterschiedlichen Teilchen.
   Ist die Distanz größer eins, so wären die Digis nicht auf dasselbe Teilchen zurückzuführen, da der Ausschlag auf den Sensoren kontinuierlich ist.

2. Die Digis müssen einen maximalen zeitlichen Abstand von einem δ haben [16]. Ist die zeitliche Distanz größer als δ, so lassen sich die Ausschläge auf zwei unterschiedliche Teilchen zurückführen. Ist sie geringer, so gehören sie demselben Cluster an, solange auch Bedingung 1 erfüllt ist.

3.2.1 Ausgangsalgorithmus des Clusterfinders

Der Pseudocode in Codebeispiel 1 zeigt das Vorgehen des grundlegenden Algorithmus. Definitionen für genutzte Begrifflichkeiten finden sich in Tabelle 1.
Der Ausgangsalgorithmus iteriert zu Beginn über alle Digis. Für jeden Digi wird nun im Statusarray überprüft, ob der Kanal des gerade betrachteten Digis bereits belegt ist. Hat das Statusarray in diesem Kanal noch keinen Eintrag, so wird der Index des Digis dort vermerkt.
Anschließend wird überprüft, ob Nachbarn im Statusarray zum betrachteten Digi existieren und ob diese in einer zeitlichen Differenz von höchstens δ zueinander stehen. Trifft dies zu, so wird das nächste Digi verarbeitet.
Sollte letzteres nicht zutreffen und einer der Nachbarn außerhalb des Zeitraums δ liegen, so kann man davon ausgehen, dass in Zukunft keine weiteren Digis existieren, welche die zeitliche Bedingung für ein Cluster erfüllen, da jedes noch zu behandelnde Digi einen größeren Zeitstempel als seine Vorgänger besitzt. Somit werden alle benachbarten Digis im

Name	**Beschreibung**
Digi	ein Ausschlag eines Kanals innerhalb eines Moduls Bestehend aus: Zeit, Kanal, Ladung
Status	Array in welchem temporär Indices von Digis abgelegt werden um diese anschließend zu einem Cluster hinzuzufügen. Bestehend aus: Indices
Modul	siehe 2.2
Kanal	Spalte/Index in einem Modul

Tabelle 1: Definitionen häufig verwendeter Begrifflichkeiten

Statusarray überprüft und anschließend zu einem Cluster zusammengefügt, da diese alle sowohl die räumliche als auch die zeitliche Bedingung für einen Cluster erfüllen. Es werden in diesem Schritt sowohl die rechts als auch die links benachbarten Digis behandelt. Im Anschluss wird mit dem nächsten Iterationsschritt fortgefahren.
Ist ein Kanal im Statusarray bereits belegt, an welcher ein neues Digi eingefügt werden soll, so werden auch hier die Nachbarn überprüft, zu Clustern zusammengefügt, die zu einem Cluster zugehörigen Kanäle im Statusarray auf einen neutralen Wert (-1, nicht als Index interpretierbar) gesetzt und anschließend der neue Index eingefügt.
Abbildung 9 zeigt in Abschnitt 1 - 5.2 exemplarisch das Vorgehen des Algorithmus, wobei mit einem δ von 0.021ns gearbeitet wird. Die im oberen Teil jedes Schrittes gezeigten Daten stellen die Liste an sortierten Digis dar. Darunter befindet sich das Statusarray.

```
bool isInDelta(int &timeA, int &timeB){
        //checks if timeA and timeB have
        //a maximum difference of delta
}

//Also exists in form: checkLeftNeighbour()
void checkRightNeighbour(short channel, Cluster &cluster){
    Digi currDigi = digi[status[channel]];
    Digi nextDigi = digi[status[channel+1]];
    cluster.add(currDigi);

    while(isInDelta(currDigi.time, nextDigi.time)){
        cluster.add(nextDigi);
        status[channel+1] = -1;
        channel++;
        currDigi = digi[status[channel]];
        nextDigi = digi[status[channel+1]];
    }
}

void findClusters(){
    for (auto digi: digis){
        Cluster cluster{};
        if(!status[digi.channel].isSet()){
            status[digi.channel] = digi.index;

            if(isInDelta(digi[status[digi.channel+1]].time,
            digi.time)){
                continue;
            }
        } else {
            checkRightNeighbour(digi.channel, cluster);
            checkLeftNeighbour(digi.channel, cluster);
            cluster.save();
            continue;
        }
        checkRightNeighbour(digi.channel+1, cluster);
        cluster.save();
        cluster.clear();
        checkLeftNeighbour(digi.channel-1, cluster);
        cluster.save();
        cluster.clear();
    }
}
```

Codebeispiel 1: Pseudocode zum bestehenden Algorithmus des Clusterfinders (Originalcodeenthalten in [17])

Abbildung 9: Beispiel für das Vorgehen des alten Clusterfinders und Clusterdefinition (5.2) incl. Definitionsabweichung (5.1) ($\delta = 0.021$ns)

3.2.2 Definitionsabweichung des Clusterfinders

Das Vorgehen des Clusterfinders wird exemplarisch in Abbildung 9 dargestellt. In Abschnitt 1-6 sieht man das grundlegende Vorgehen des originalen Algorithmus, doch in gewissen Randfällen kommt dieser zu Ergebnissen, welche der Definition eines Clusters zwar nicht widersprechen, aber bei der Digis die laut Definition zu einem Cluster gehören, nicht zusammenfügt würden.
Ausschlaggebend ist hier das Vorgehen des Algorithmus in Abschnitt 5.1 der Abbildung. Das Digi mit dem Index 4 wird dem Statusarray hinzugefügt und überprüft, ob der linke/rechte Nachbar im gegebenen Deltabereich liegt. Da dies nicht der Fall ist, werden die hier links liegenden Daten zu einem Cluster zusammengefasst, weil angenommen wird, dass mit dem ersten Element, welches nicht mehr im Deltabereich zu seinem Nachbarn ist,

kein Weiteres folgen kann, das diese Eigenschaft für diesen Cluster erfüllt. Überprüft wird hier aber nur eine Seite des Clusters. Die andere Seite wird hier vernachlässigt.
Betrachtet man die Daten genauer, so fällt auf, dass der Cluster aber weitere zugehörige Digis hätte. Durch das Hinzufügen des Digis mit Index 5, hätte der Cluster, wie in 5.2 dargestellt, zwei weitere dazugehörige Elemente, die sowohl die Lokalitäts- als auch die Temporärbedingung erfüllen.
Da dieses Problem nur in wenigen Fällen auftritt - nämlich genau dann, wenn das neu hinzugefügte Digi einen Cluster beendet, aber noch ein weiteres Digi existiert, das den Cluster zur anderen Seite erweitern würde - so ist dies bisher als vernachlässigbar eingestuft worden.
Die in Abbildung 10 dargestellen Ergebnisse zeigen den Unterschied der berechneten Cluster in Prozent. Es ist zu erkennen, dass der Anteil der unterschiedlich erkannten Cluster selbst bei wachsenden Clusterzahlen bei ca. 0,02% bleibt.
Wichtig ist dieser Punkt dennoch, da die in folgenden Abschnitten gezeigten Algorithmen mit den Bedingungen aus Abschnitt 3.2 arbeiten. Daher sei an dieser Stelle gesagt, dass die verglichenen Ergebnisse sich marginal unterscheiden, die Unterschiede aber im Bereich weit unter einem Prozent liegen und sich somit auch die Ausführzeiten der unterschiedlichen Algorithmen noch erfolgreich in Relation setzen lassen.

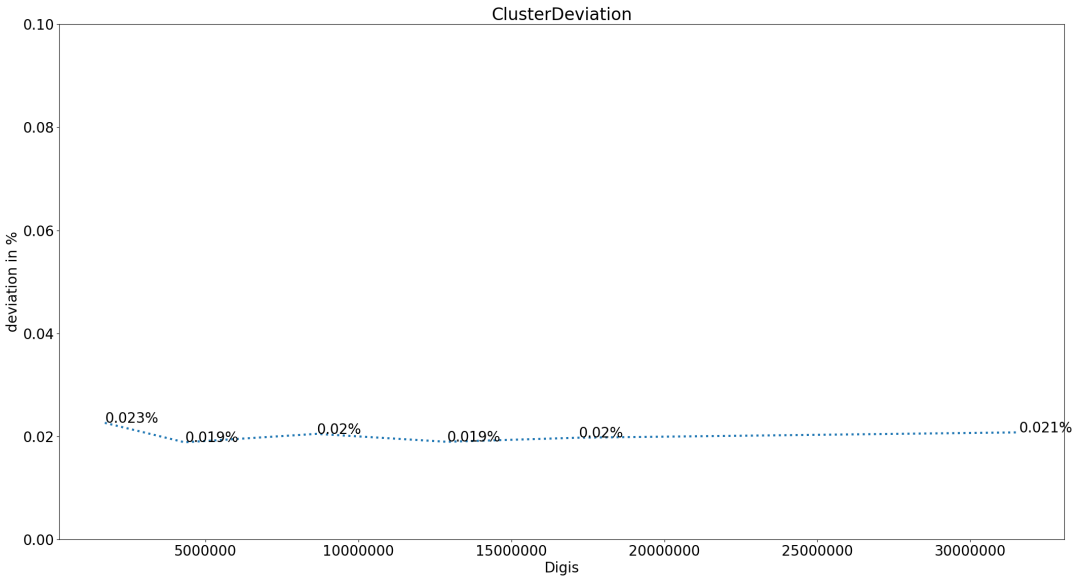

Abbildung 10: Abweichungen der gefundenen Cluster des Ausgangsalgorithmus zur vorgestellten Optimierung

3.2.3 Optimierung: Kanalorientiert

Der Ausgangsalgorithmus des Clusterfinders lässt sich nicht direkt parallelisieren, da die zugrundeliegende Idee auf einer iterativen Vorgehensweise basiert. Es ist hierbei wichtig die zeitliche Reihenfolge der Digis beizubehalten, damit der Algorithmus terminiert und korrekte Ergebnisse hervorbringt. Daher ist es für eine Parallelisierung unabdingbar einen neuen Algorithmus einzuführen. Im folgenden Abschnitt wird dieser vorgestellt. Der Algorithmus wird anfangs als einfache Grundvariante betrachtet und im Nachhinein weiter optimiert.
Der Algorithmus wird parallel auf unterschiedlichen Blöcken der Grafikkarte ausgeführt. Jeder Block (siehe 2.6) arbeitet auf den Daten eines Moduls (siehe 2.2) des Detektors. So kann eine hoch parallele Abarbeitung der Daten gewährleistet werden. Der im folgenden behandelte Algorithmus beschreibt nur die Abarbeitung eines Moduls des Clusters und betrachtet exemplarisch nur einen Block der Grafikkarte.

3.2.3.1 Sortieren der Daten

Um das Finden der Cluster zu parallelisieren werden die Digis zu Anfang sortiert. Erst werden die Digis nach Ort (Kanal) und anschließend innerhalb der Kanäle nach Zeit sortiert. Dies passiert in einem Schritt, wodurch die Zeit- und Ortsdaten in einem Datentyp zusammengefasst werden. Es wird ein 64 Bit Integer reserviert, in welchem die 32 most significant Bits den Ort und die 32 least significant Bits die Zeit beschreiben. Sortiert werden die Digis mit dem in Abschnitt 3.1 beschriebenen Mergesort. Da bei dem vorherigen Algorithmus auch im Vorhinein zeitlich sortiert wurde entsteht hier nur ein marginaler Geschwindigkeitsverlust, da nun mit größeren Datentypen sortiert wird.

3.2.3.2 Datenstrukturen

Das ClusterConnectorArray beschreibt ein Array mit einer Größe von der Anzahl der eingegebenen Digis eines Moduls. Es beinhaltet Einträge der Klasse ClusterConnector (Abbildung 12). Die ClusterConnector Klasse beschreibt die Verbindung zwischen zwei Digis eines Clusters. Zugrunde liegt hier ein eigens definierter 32 Bit Datentyp, der im nullten Bit einen Boolean und in den restlichen 31 Bits einen Index (Abbildung 11).
Der Boolean wird benötigt um festzustellen, ob der Kanal eines Digis der Niedrigste im zugehörigen Cluster ist. Der Index sagt aus, dass der gerade betrachtete Digi einen weiteren Digi rechts von ihm kennt, der zum selben Cluster gehört. Daraus ergibt sich schlussendlich eine verkettete Liste an Indices und somit Digis mit einem fest definierten Startindex.
Durch die Begrenzung auf 31 Bits für die Indices wird im Umkehrschluss auch die Anzahl der Digis in einem Modul auf 2.147.483.648 (2^{31}) begrenzt. Sollten mehr Digis in zukünfigen Experimenten eingegeben werden, so kann der Datentyp angepasst werden. Da auf den getesteten Grafikkarten aber nicht ausreichend Speicher für größere Datentypen zur Verfügung stand, wurde der Wert auch aufgrund der ausreichenden Eingabemenge auf 31 Bits festgelegt.
Sowohl die Funktion SetNext(), als auch SetHasPrevious() müssen atomar gestaltet werden, da ansonsten zwei Threads gleichzeitig auf den selben Datensatz zugreifen könnten. Da im Algorithmus dies nicht ausgeschlossen werden kann, wäre es möglich, dass es hier zu Race Conditions kommt. Durch die Atomarität dieser Operationen wird ein solcher Fall ausgeschlossen.

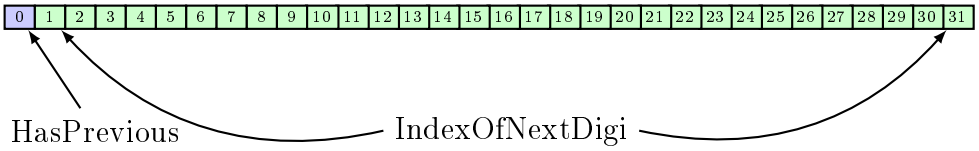

Abbildung 11: Bitstruktur HasPreviousAndNext

ClusterConnector
hasPreviousAndNext: int
SetNext(int): void GetNext(): int HasNext(): bool SetHasPrevious(bool): void HasPrevious(): bool

Abbildung 12: Diagramm der verwendeten ClusterConnector-Klasse

3.2.3.3 Berechnung der Offsets

Um herauszufinden, wo jeder Kanal innerhalb des sortierten Eingabearrays beginnt, wird schon zu hier parallel gearbeitet.
Der Algorithmus (siehe Codebeispiel 2) geht hierbei blockweise vor und lässt bei jedem Iterationsschritt jeden Thread genau ein Digi bearbeiten. Dabei wird überprüft, ob sich der Kanal des vom Thread betrachteten Digis zum nächsten Digi unterscheidet. Trifft die Bedingung zu, so wird der Index in einem Array (channelOffsets) an der Stelle des jeweiligen Kanals gespeichert. Eine Besonderheit ist ein Kanal, welcher kein Digi besitzt. Dieser Sonderfall wird behandelt, indem jeder Kanal zwischen zweier sich unterscheidenden Digis ebenfalls mit dem gefundenen Index in das Array übernommen wird. Die Assertion überprüft im Durchlauf noch einmal, ob die Daten wirklich sortiert vorliegen und wirft eine Exception, sollte dies nicht der Fall sein. Ansonsten kann nicht gewährleistet werden, dass die folgenden Algorithmen korrekt funktionieren.

```
void calculateChannelOffsets(Digi *digis, unsigned int *channelOffsets,
int nDigis)
{
  channelOffsets[0] = 0;
  for(int pos = xpu::thread_id::x(); pos < nDigis-1;
  pos+=xpu::block_dim::x()){
    auto const currChannel = digis[pos].fChannel;
    auto const nextChannel = digis[pos+1].fChannel;
    if(currChannel != nextChannel){
      for(int i = currChannel+1; i <= nextChannel; i++){
        channelOffsets[i] = pos+1;
      }
    }
    XPU_ASSERT(digis[pos].fChannel <= digis[pos+1].fChannel);
  }
}
```

Codebeispiel 2: Berechnung der Offsets(beginn von Kanälen) beim optimierten Clusterfinden

In Abbildung 13 sind die Ergebnisse der Berechnung der Offsets dargestellt. Im Plot ist zu erkennen, wie die unterschiedliche Hardware hierbei abschneidet. Relevant wird die Dauer der Offsetberechnung noch für die Berechnung der gesamten Zeiten, da bei dem in 3.2.1 beschriebenen Algorithmus keine Offsets berechnet werden mussten. Setzt man die Berechnung der Offsets in Relation zu dem zuvor angesprochenen Sortieren der Digis, fällt auf, dass selbst eine Ausführung auf der CPU der Offsetberechnung keinen großen Einfluss auf die Laufzeit der allgemeinen Performance hat. Selbst bei einer großen Anzahl von Digis hat die Berechnungszeit der Offsets eine sehr flache Steigung. Nicht zuletzt basiert dieses Verhalten auf der parallelen Berechnung und Ausnutzung aller der Threads auf den Devices.

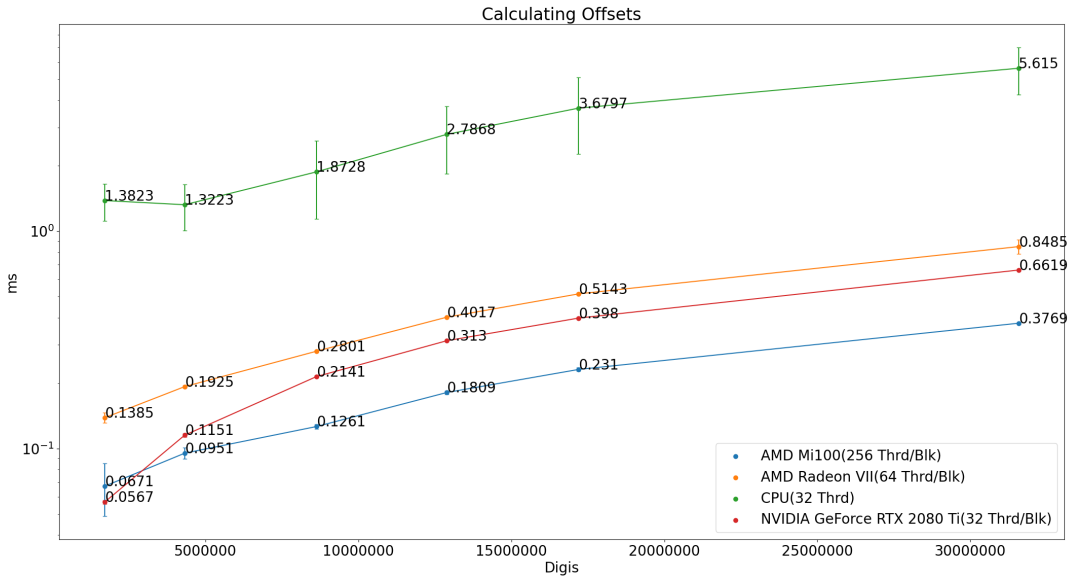

Abbildung 13: Ergebnisse von unterschiedlicher Hardware bei der Offsetberechnung

3.2.3.4 Paralleler Algorithmus

Die Digis liegen nun sortiert in einem Array vor. Dieses wird in den Shared-Memory der Grafikkarte geladen, sodass alle Threads eines Blocks auf dieses zugreifen können.
Zu Anfang werden die Offsets für jeden Kanal innerhalb des Eingabearrays berechnet (siehe 3.2.3.3).

Nachdem die Threads synchronisiert wurden, arbeitet jeder Thread mit den Digis aus genau einem Kanal. Ist ein Kanal abgearbeitet und es gibt noch unbearbeitete Kanäle, so bekommt jeder Thread einen weitere Kanal, bis alle Kanäle durchsucht wurden. Dabei müssen nur die Digis bis zu dem vorletzten Kanal überprüft werden, da alle im letzten Kanal liegenden Digis keinen Nachfolger mehr haben können. Ein Codebeispiel für den Algorithmus findet sich in Codebeispiel 3.
Jeder Thread iteriert über die Digis in seinem Kanal ("Channel") und beginnt dabei mit dem Ersten und somit dem Digi mit dem niedrigsten Zeitstempel. Das aktuell betrachtete Digi im eigenen Kanal wird als "currentDigi" bezeichnet und mit dem index "iCurr" im Beispiel beschrieben. Nun wird im nächsten, rechts liegenden Kanal (Beginn des Channels "nextChannelBegin") nach einem die Temporärbedingung erfüllenden Digi gesucht. Das betrachtete

Digi im nextChannel wird als "nextDigi" bezeichnet. Dabei läuft ein Iterator vom niederigsten zum höchsten Digi und vergleicht währenddessen die Zeitstempel vom currentDigi und nextDigi. Ist der Zeitstempel des nextDigi (index "iNext") kleiner als der des currentDigis - δ, so wird im nextChannel weiter iteriert. Ist der Zeitstempel des nextDigis größer als der des currentDigis + δ, so wird der nächste Digi im currentChannel als Vergleichsdigi genutzt, da nun kein Digi mehr kommen kann, das die Temporärbedingung erfüllen kann.
Wird ein Digi gefunden, das im Zeitintervall liegt, so wird der next-Wert des currentDigi auf den Index des gefundenen Digis gesetzt. Außerdem wird der hasPrevious-Wert des gefundenen Digis auf true gesetzt. So wird eine verkettete Liste erzeugt, die zu Beginn false im hasPrevious-Wert stehen und für alle weiteren Werte einen Zeiger auf den nächsten Digi desselben Clusters hat, welcher sich im rechts benachbarten Kanal befindet.

```
XPU_D void CbmStsGpuHitFinder::findClusterConnectionsChannelWise(
Digi *digis, DigiConnector *digiConnector,
unsigned int *channelOffsets, int const iModule, int const threadId) const{
  for (int channel = threadId; channel < nChannels-1; channel+=blocksize) {
    auto const currChannelBegin = channelOffsets[channel];
    auto const nextChannelBegin = channelOffsets[channel+1];
    auto const nextChannelEnd = (channel+2 < (unsigned int)nChannels)
        ? channelOffsets[channel + 2] : getNDigis(iModule);
    auto iNext = nextChannelBegin;

    //Check if first Digi of Channel belongs to Channel
    if(channel != digis[currChannelBegin].channel){ continue; }

    float const delta = calculateDelta(iModule, channel);

    //Calculate DigiConnections
    for(auto iCurr = currChannelBegin;
    iCurr < nextChannelBegin; iCurr++ ){

      while((iNext < nextChannelEnd)
      && ((digis[iCurr].time + delta) >= digis[iNext].time)){

          if(digis[iCurr].channel>=digis[iNext].channel){continue;}
          if(digis[iCurr].time+delta<digis[iNext].time){iNext++; break;}
          if(digis[iCurr].time-delta>digis[iNext].time){iNext++; continue;}

          digiConnector[iCurr].setNext(iNext);
          digiConnector[iNext].setHasPrevious(true);
          iNext++;
          break;
      }
    }
  }
}
```

Codebeispiel 3: Clusterfinden mit kanalorientiertem Vorgehen

Das kanalorientiere Finden der Cluster verhält sich auf allen genutzten

Devices ähnlich. Einzig und allein die CPU weicht, wie erwartet, etwas von den Zeiten der GPUs ab. Dennoch lässt sich an dieser Stelle schon grob sehen, wie sich die Performance der CPU im Vergleich zum originalen Algorithmus verhalten wird. Der neue parallele Algorithmus sorgt dafür, dass selbst die CPU eine nahezu ähnlich gute Berechnungsgeschwindigkeit (ca. 30ms) erreicht wie die GPU (ca. 23ms), auch wenn dies unter gewissen Schwankungen geschieht, welche die GPUs allesamt nicht aufweisen. Es ist zu erwähnen, dass hier zum ersten mal das Nvidia-Device schwächer abschneidet als das äquivalente AMD-Device.

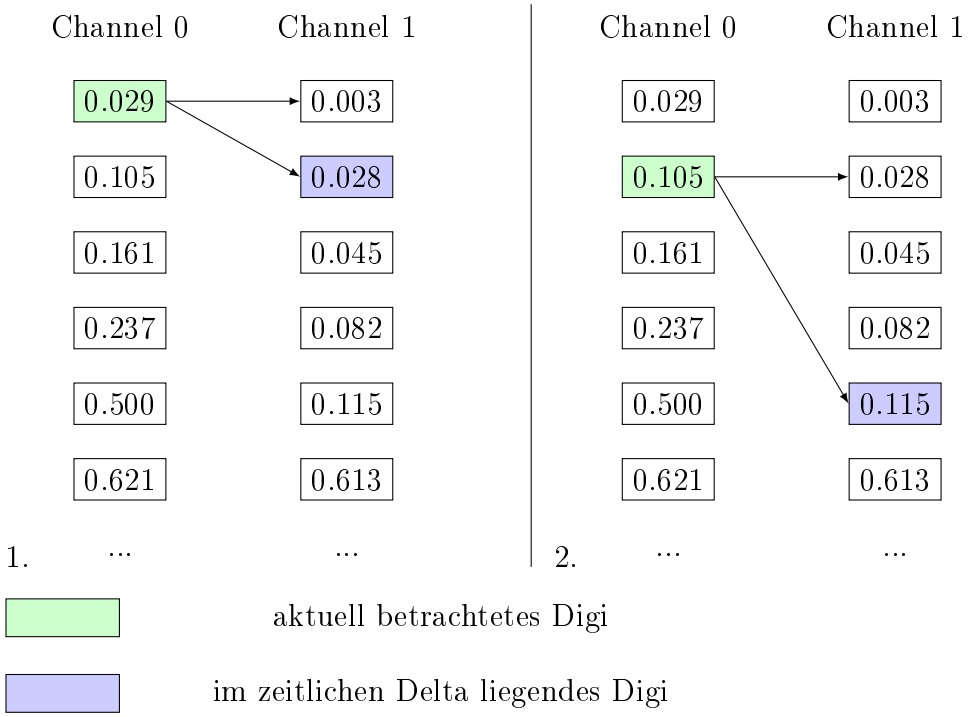

Abbildung 14: Vorgehen beim kanalorientieren Clusterfinden am Beispiel eines Kanals anhand von Zeiten in ns in 2 Schritten ($\delta = 0.021$ns)

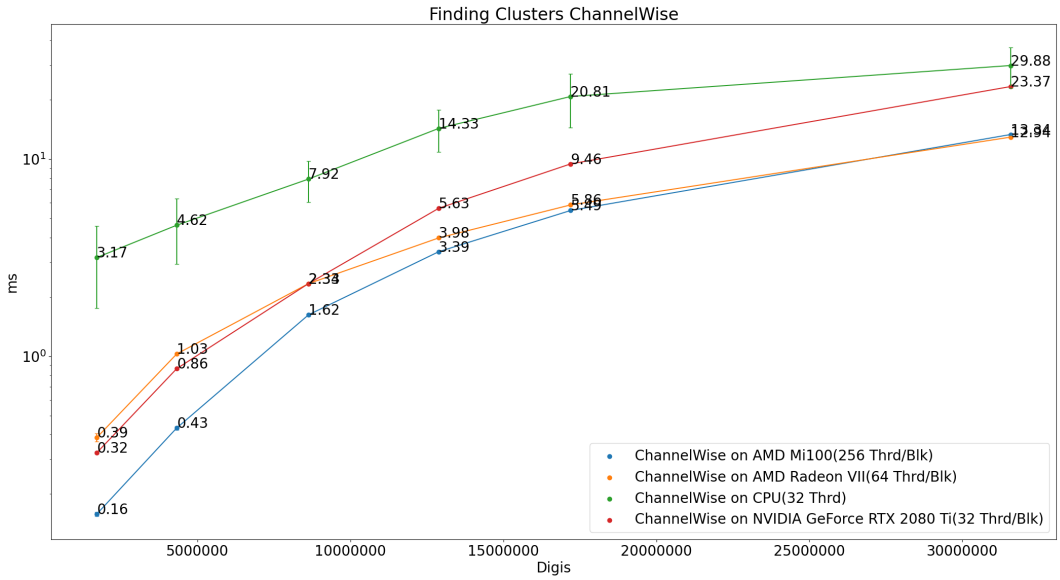

Abbildung 15: Ergebnisse von unterschiedlicher Hardware beim Finden der Cluster und kanalorientiertem Vorgehen

3.2.4 Optimierung: Digiorientiert

Da der Cache von Grafikkarten ähnlich zu dem von CPUs begrenzt ist, bietet es sich an diesen optimal auszunutzen. Jede Computeunit hat einen eigenen L1 Cache, der durch benötigte Daten gefüllt wird. Wenn ein Datum aus dem Shared-memory oder dem RAM in den Cache geladen wird, passiert dies nicht nur mit einem Datum, sondern alle lokal benachbarten Daten werden ebenfalls in einem koaleszierten Speicherzugriff in den Cache geladen. Dies geschieht zum Einen, weil Datensätze immer in Blöcken geladen werden und zum Anderen, weil es sehr wahrscheinlich ist, dass benachbarte Daten im Speicher zeitlich nacheinander genutzt werden.
Um dieses Prinzip auszunutzen, wird der in Abschnitt 3.2.3 beschriebene Algorithmus angepasst, sodass nicht jeder Thread seinen eigenen Kanal abarbeitet, sondern jeder Thread nun einen Digi betrachtet. Hat jeder Thread einen Digi abgearbeitet, so werden alle Threads verschoben und nehmen sich den nächsten Digi vor. Dadurch können Digis von unterschiedlichen Threads bearbeitet werden, die nebeneinander im Speicher liegen.
Leider wird durch diese Änderung nicht nur Performance gewonnen. Dadurch, dass jeder Thread nicht mehr mitbekommt bei welchem Digi der vorherige Thread terminiert ist, geht die Information über die aktuelle

Iteratorposition verloren. So muss jeder Thread immer den gesamten nächsten Kanal durchsuchen und kann nicht, wie im Ausgangsalgorithmus, die bereits behandelten Digis ausschließen. Abbildung 17 zeigt exemplarisch das Vorgehen der Threads und der zu kontrollierenden Digis des nächsten Kanals.
Der Codebeispiel 4 zeigt den Code zur Optimierung. Der Unterschied liegt hier in der ersten Schleife, die nun blockweise über die Digis läuft. Zusätzlich muss noch eine Überprüfung eingebaut werden, die nachschaut, ob der vorletzte Kanal behandelt wird, da das Ende des letzten Kanals nicht im Offset-Speicher vertreten ist.
Auch das digiorientierte Finden der Cluster schneidet zeitlich sehr gut ab (siehe Abbildung 16). Ähnlich wie bei der kanalorientierten Variante sind die Zeiten der GPU-Devices sehr nah beieinander, wobei sie in diesem Fall noch ähnlicher abschneiden. Gerade die Zeiten für hohe Digianzahlen unterscheiden sich nur noch marginal. Setzt man den Algorithmus jedoch in den Vergleich zu dem kanalorientierten fällt auf, dass sich die Zeiten bei einzelnen Devices fast schon verdoppeln. Dieses Verhalten ist auf die zuvor beschriebenen Faktoren zurückzuführen. Die CPU schneidet aufgrund der genannten Nachteile hier ebenso weit schlechter ab als bei der kanalorientierten Vatiante und ist dabei sogar vier mal langsamer als die GPUs.

```
XPU_D void CbmStsGpuHitFinder::findClusterConnectionsDigiWise(
Digi *digis, DigiConnector *digiConnector,
unsigned int *channelOffsets, int const iModule, int const threadId,
unsigned int const nDigis) const {

  for(unsigned int iCurr = threadId; iCurr < nDigis; iCurr += blocksize){
    auto const digi = digis[iCurr];
    auto const channel = digi.channel

    if (channel == nChannels-1){ break; }

    float const delta = calculateDelta(iModule, channel);

    auto const nextChannelEnd = (channel+2 < nChannels)
      ? channelOffsets[channel + 2]
      : nDigis;

    //Calculate DigiConnections
    for(auto iNext = channelOffsets[channel + 1];
    iNext < nextChannelEnd; iNext++){
      if(digis[iCurr].channel>=digis[iNext].channel){continue;}
      if(float(digis[iCurr].time+delta)<float(digis[iNext].time)){break;}
      if(float(digis[iCurr].time-delta)>float(digis[iNext].time)){continue;}

      digiConnector[iCurr].setNext(iNext);
      digiConnector[iNext].setHasPrevious(true);
      break;
    }
  }
}
```

Codebeispiel 4: Clusterfinden mit digiorientiertem Vorgehen

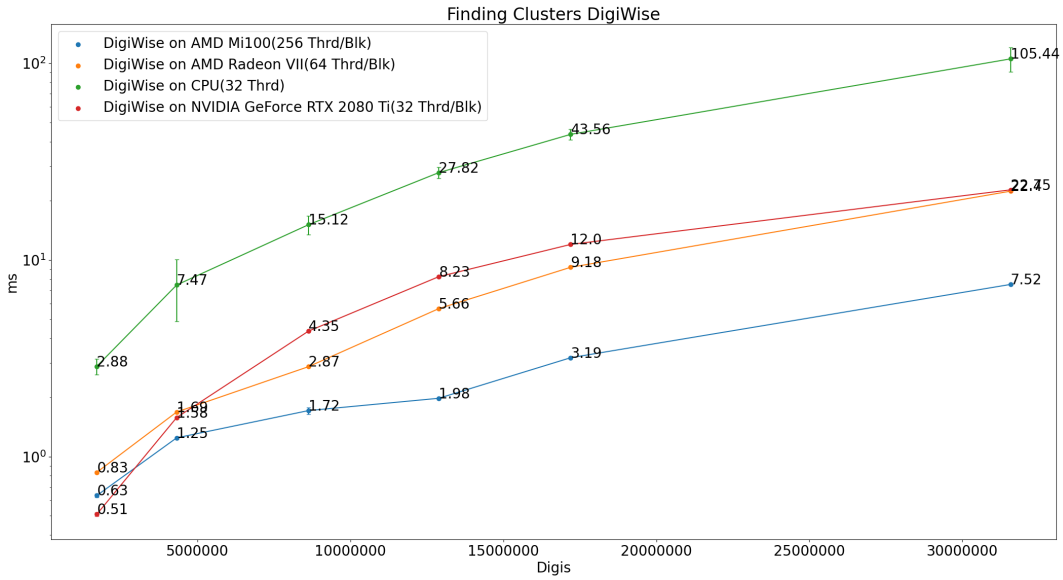

Abbildung 16: Ergebnisse von unterschiedlicher Hardware beim Finden der Cluster und digiorientiertem Vorgehen

3.3 Berechnen der Cluster

Nachdem alle Cluster gefunden wurden, verbleiben die berechneten und vorhandenen Daten im Speicher der Grafikkarte. Es sind hier keine unnötigen Kopieroperationen von GPU und CPU notwendig und es kann direkt mit der Berechnung der Cluster fortgesetzt werden.

3.3.1 Optimierung: Kanalorientiert

Ein Cluster, welcher derzeit nur eine Ansammlung von Digis ist, muss nun in einen Cluster, bestehend aus sechs zentralen Werten (Ladung, Größe, Zeit, Zeitfehler, Position und Positionsfehler), zusammengefasst werden. Die Berechnung wird dabei in 3 Komplexitätsklassen unterteilt, bei denen die Cluster entweder aus einem, zwei oder n Digis bestehen.
Die grundlegende Berechnung der einzelnen Cluster wurde nicht abgeändert. Der Geschwindigkeitszuwachs wurde hier durch eine Parallelisierung auf der GPU erreicht, indem die Abarbeitung der einzelnen Cluster nicht linear auf der CPU erfolgt.
Codebeispiel 5 beschreibt das Vorgehen. Die Parallelisierung erfolgt hier

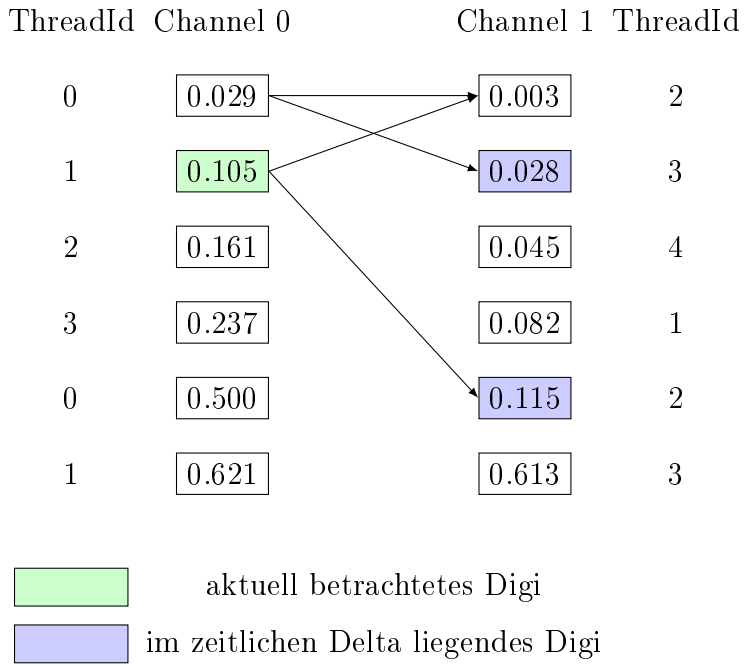

Abbildung 17: Vorgehen beim kanalorientieren Clusterfinden am Beispiel eines Kanals, anhand von Zeiten in ns ($\delta = 0.021$ms)

analog zu dem Algorithmus, erläutert in 3.2.3. Jeder Thread arbeitet dabei die zu einem Kanal gehörendem Cluster ab. Dabei sei gesagt, dass nur Cluster als einem Kanal zugehörig interpretiert werden, die den Beginn auch in diesem Kanal haben. Dies ist wichtig, da die verkettete Liste an Digis keine Rückwärtskanten besitzt und somit nur in eine Richtung abgearbeitet werden kann. Bei diesem Vorgehen ist die Anzahl der zugewiesenen Kanäle pro Thread gleich verteilt ($\frac{|\text{Kanäle}|}{|\text{Threads}|}$ Kanäle pro Thread). Es kann hierbei also vorkommen, dass ein Thread einen Kanal zugeteilt bekommt, welcher keinen Beginn eines Clusters beinhaltet. Da jeder Thread aber mehrere Kanäle abarbeitet reduziert dies die Wahrscheinlichkeit, dass ein Thread keinen Cluster berechnet.
Betrachtet man die Ergebnisse (Abbildung 18) der kanalorientierten Clusterberechnung, lässt sich hier auf den unterschiedlichen Devices eine ähnliche Performance erkennen. Sowohl AMD als auch Nvidia unterscheiden sich nur im Bereich von maximal 8ms. Bei der genutzten CPU treten, wie auch schon zuvor bei dem Finden der Cluster hohe Schwankungen bei den Berechnungszeiten auf. Hier ist schon zu erkennen, dass sich sowohl das Finden als auch das Berechnen der Cluster in einem ähnlichen Berechnungzeitraum abspielen.

```
void calculateClustersChannelWise(Digi *digis, DigiConnector *connector,
unsigned int *channelOffsets, int iModule, int threadId) {
  for (int channel = threadId; channel < nChannels; channel+=blocksize) {

    unsigned int begin = channelOffsets[channel];
    unsigned int end = (channel+1 < (unsigned int)nChannels)
    ? channelOffsets[channel + 1] : nDigis;

    //Check if first Digi of Channel belongs to Channel
    if(channel != digis[begin].fChannel){ continue; }

    for(auto currIter = begin; currIter < end; currIter++ ){
      if(!connector[currIter].hasPrevious()){
        if(!connector[currIter].hasNext()){
          //if Cluster has 1 element
          createClusterFromConnectors1(iModule, digis, currIter);
        } else if(!digiConnector[connector[currIter].next()].hasNext()){
          //if Cluster has 2 elements
          createClusterFromConnectors2(iModule,digis,connector,currIter);
        } else {
          //if Cluster has N elements
          createClusterFromConnectorsN(iModule,digis,connector,currIter);
        }
      }
    }
  }
}
```

Codebeispiel 5: Clusterberechnung mit kanalorientiertem Vorgehen

3.3.2 Optimierung: Digiorientiert

Eine faire Aufteilung nach Kanälen scheint anfangs eine gute Vorgehensweise zu sein. Da aber unklar ist, wie viele Anfänge von Clustern innerhalb der Kanäle liegen, ist es jedoch wahrscheinlich, dass Threads existieren, die keinerlei Cluster berechnen müssen, da in ihren Kanälen nur Digis liegen, die kein Startpunkt eines Clusters sind. Daher ist es sinnvoll ebenso einen Code zu implementieren, der nicht die Cluster den Threads anhand der Kanäle zuordnet, sondern jeden Thread blockweise nur einen Digi überprüfen zu lässt. Dies begünstigt auch, wie schon in Abschnitt 3.2.4 beschrieben, das Prinzip der Lokalität beim Laden des Speichers. Wie in Codebeispiel 6 gezeigt, berechnen die Threads nun blockweise die Cluster über die Digis.

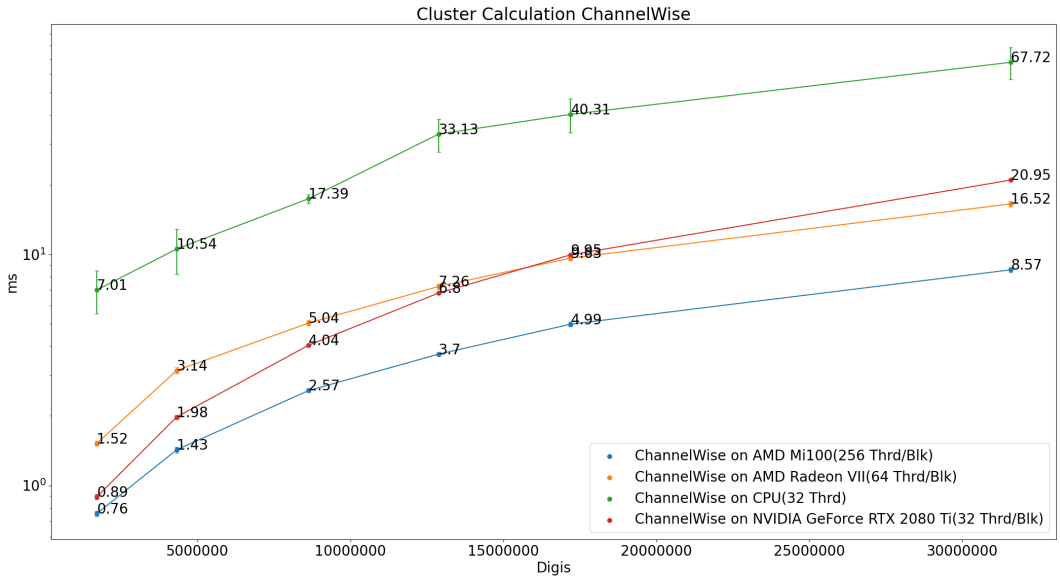

Abbildung 18: Ergebnisse von unterschiedlicher Hardware beim Berechnen der Cluster und Kanalorientiertem Vorgehen

```
void calculateClustersDigiWise(Digi *digis, DigiConnector *connector,
int iModule, int threadId, unsigned int nDigis){

  for(auto currIter = threadId; currIter < nDigis; currIter += blockSize){
    if(!connector[currIter].hasPrevious()){
      if(!connector[currIter].hasNext()){
        //if Cluster has 1 element
        createClusterFromConnectors1(iModule,digis,currIter);
      } else if(!digiConnector[connector[currIter].next()].hasNext()){
        //if Cluster has 2 elements
        createClusterFromConnectors2(iModule,digis,connector,currIter);
      } else {
        //if Cluster has N elements
        createClusterFromConnectorsN(iModule,digis,connector,currIter);
      }
    }
  }
}
```

Codebeispiel 6: Clusterberechnung mit digiorientiertem Vorgehen

Den Erwartungen entsprechend sehen die Ergebnisse der Optimierung aus (Abbildung 19). Ähnlich der Ergebnisse des Clusterfinders schneiden die Grafikkarten besser ab als die im Vergleich enthaltene CPU. Es ist aber auch hier zu erkennen, dass die CPU mit größeren Schwankungen zu kämpfen hat als die GPUs.

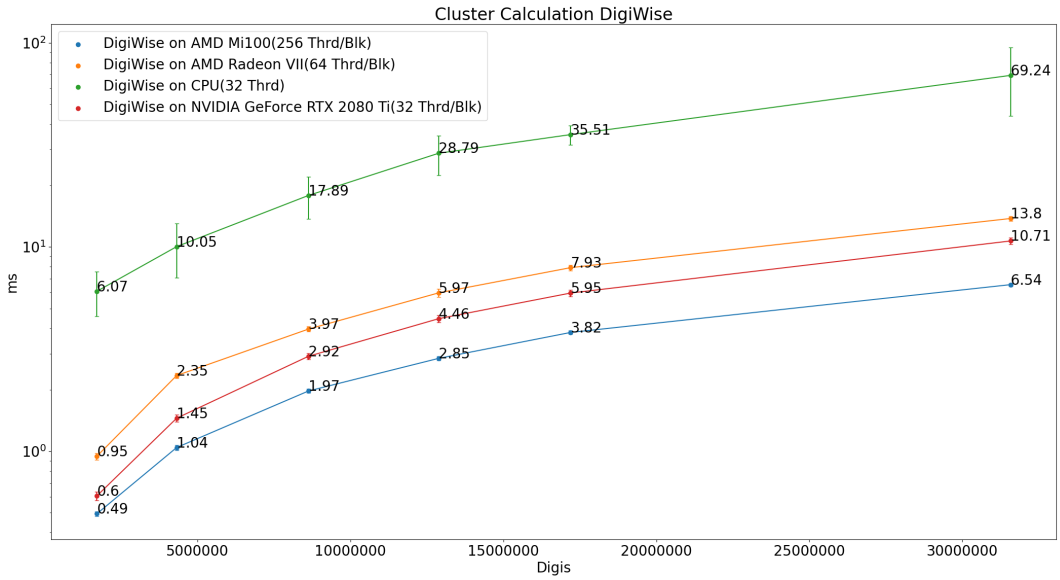

Abbildung 19: Ergebnisse von unterschiedlicher Hardware beim Berechnen der Cluster und Digiorientiertem Vorgehen

3.4 Sortieren der Cluster

Bevor die Cluster zu Hits zusammengefasst werden, ist es notwendig, diese noch einmal anhand der Zeit zu sortieren. Dieses Sortieren kann ebenfalls, wie das Sortieren der Digis, auf der Grafikkarte unter Zuhilfenahme des in Abschnitt 3.1 beschriebenen Algorithmus erfolgen. Daher ist eine neue Optimierung an dieser Stelle nicht erforderlich.

3.5 Finden der Hits

Das Finden der Hits ist der letzte Schritt der lokalen Rekonstruktion. Hierbei werden die berechneten Cluster von Vor- und Rückseite der Detektorplatten (siehe 2.3) zu Hits zusammengefasst. Der hier verwendete Algorithmus ist bereits parallelisiert und bietet daher schon eine ausreichend gute Performance für die lokale Rekonstruktion. Der folgende Abschnitt befasst sich also lediglich mit kleinen Optimierungen des Hitfinders und verlagert unter anderem Berechnungen in den Clusterfinder, die im Hitfinder stattfinden.

3.5.1 Ausgangsalgorithmus des Hitfinders

Der Hitfinder, welcher im Anschluss an das Sortieren der Cluster ausgeführt wird, verknüpft Cluster der Vor- und Rückseite eines Moduls. Dabei nutzt er zum Einen die beim Berechnen der Cluster ermittelte mittlere Zeit, in welcher die einzelnen Digis des Cluster gemessen wurden und des Weiteren die Standardabweichung σ des Betrachteten und die maximal gemessene Standardabweichung auf Vor- und Rückseite.
Im Anschluss an die Überprüfung der zeitlichen Überschneidung der Cluster folgt die Überprüfung der lokalen Überschneidung. Überschneiden sich zwei Cluster sowohl zeitlich als auch lokal, so werden diese zu einem Hit zusammengefasst und gespeichert. Da hier alle Cluster der Vorderseite mit allen Clustern der Rückseite verglichen werden (bis auf diejenigen, welche durch vorherige Iterationen ausgeschlossen werden können. Siehe Iterator Prinzip aus 3.2.3), kommt es bei einer iterativen Durchführung hier zu einer quadratischen Laufzeit. Durch vorangegangene Optimierungen wurde dieser Abschnitt bereits parallelisiert, indem jeweils ein Thread einen Cluster der Vorderseite des Detektors bearbeitet (siehe Codebeispiel 7).

```
for (int iClusterF = xpu::thread_idx::x();
iClusterF < nClustersF; iClusterF += xpu::block_dim::x()) {
        ClusterIndex clsIdxF   = clusterIdxF[iClusterF];
    ClusterData clsDataF = clusterDataF[clsIdxF.fIdx];

    for (int iClusterB = startB; iClusterB < nClustersB; iClusterB++) {
      ClusterIndexclsIdxB   = clusterIdxB[iClusterB];
      ClusterData clsDataB = clusterDataB[clsIdxB.fIdx];
      calculateIntersection();
```

Codebeispiel 7: Pseudocode des Hitfinders

3.5.2 Optimierungen des Hitfinders

Wie zuvor beschrieben werden zur Berechnung der zeitlichen Überschneidungen die zeitlichen maximalen Fehlerwerte aller Cluster benötigt. Der originale Code führt diese Berechnung im Hitfinder durch, indem er, wie in Codebeispiel 8 gezeigt, einmal über alle Cluster iteriert und die Werte nach und nach mit dem zuletzt gefundenen Maximum vergleicht. Dazu muss auf alle Cluster einmal zugegriffen und somit die jeweiligen Werte aus dem Speicher gelesen werden.
Da im Clusterfinder die Fehlerwerte von jedem Cluster durch die dortige Berechnung der Werte bereits im Cache zur Verfügung stehen, liegt es nahe, die Calculation dorthin auszulagern. Dazu werden zwei neue Variablen pro Modul vorgestellt, welche den maximalen Fehler der Vor- und Rückseite beinhalten. Jedes Mal, wenn durch die Clusterberechnung (siehe 3.3) ein neuer Cluster ermittelt wurde, werden nun die maximalen Fehlerwerte überprüft und gegebenenfalls erneuert. Codebeispiel 9 zeigt exemplarisch das Vorgehen der Optimierung. "atomic_cas" ist dabei eine Funktion, die entsprechend einer zu überprüfenden Bedingung einen Wert threadsicher und atomar ersetzt.
In Abbildung 21 und Abbildung 20 werden die unterschiedlichen Zeiten der hier beschriebenen Optimierung dargestellt. Der Vergleich bezieht sich hier nicht auf die originale CPU-Variante, sondern auf die bereits teilweise optimierte. Daher ist an dieser Stelle nur ein kleiner Performance-Anstieg von knapp 10ms auf den beiden AMD-Karten zu sehen. CPU und Nvidia-GPU verlieren dabei etwas an Geschwindigkeit, bei welcher sich der Verlust von knapp 5ms aber noch im Bereich der Standardabweichung aufhält. Hier ist zu erwähnen, dass durch das Auslagern die Berechnung der Cluster etwas langsamer wird. Der Tradeoff zahlt sich aber dennoch aus, wenn auch nur im kleinen Rahmen.

```
float maxTerrF = 0.f;
for (int i = 0; i < nClustersF; i++) {
  maxTerrF = xpu::max(clusterDataF[i].fTimeError, maxTerrF);
}

float maxTerrB = 0.f;
for (int i = 0; i < nClustersB; i++) {
  maxTerrB = xpu::max(clusterDataB[i].fTimeError, maxTerrB);
}
```

Codebeispiel 8: Berechnung des maximalen Fehlers eines Clusters auf Modulebene

```
void saveMaxError(float errorValue, int iModule) const {
  float *maxError = isBackside(iModule) ? maxErrorBack: maxErrorFront;
  float old{};
  do {
    old = *maxError;
    if (old >= errorValue){ break; }
  } while (!xpu::atomic_cas(maxError,*maxError,max(errorValue,*maxError)));
}
```

Codebeispiel 9: Atomares Speichern des maximalen Fehlers durch CompareAndSwap

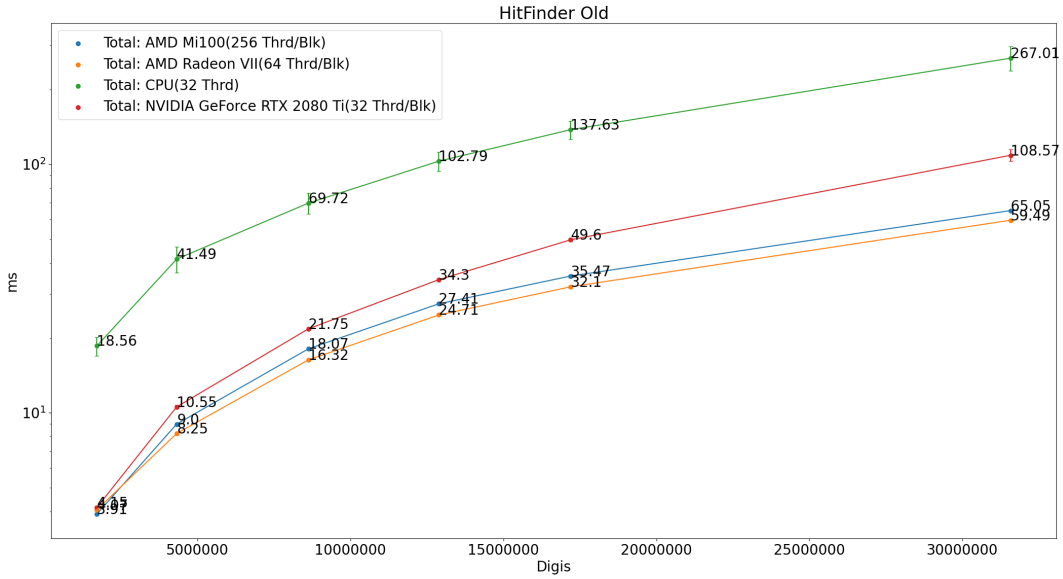

Abbildung 20: Finden der Hits mit Error-Berechung im HitFinder

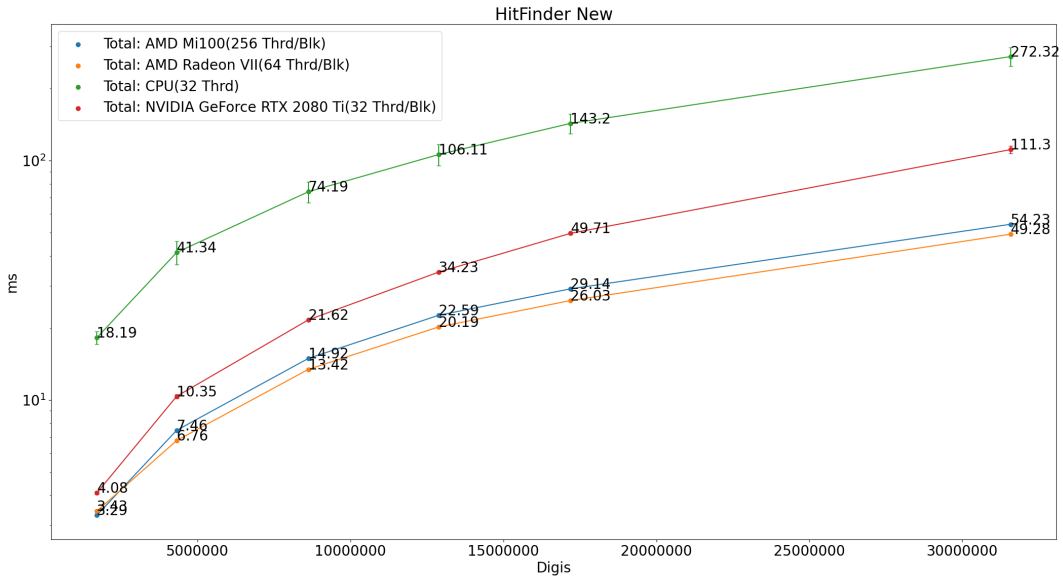

Abbildung 21: Finden der Hits mit Error-Berechung im ClusterFinder

4 Ergebnisse

In den vorherigen Kapiteln wurden die Algorithmen der lokalen Rekonstruktion des Silicone Tracking Systems optimiert und auf Grafikkarten übertragen. Dabei trägt jeder einzelne Teil zum Gesamtbild bei. Doch die Optimierungen kommen dabei nicht ohne Kompromisse aus.
Der erste Kompromiss ist die bereits in 3.2.2 beschriebene Cluster- und somit auch Hitabweichung des Resultats. Da es sich dabei aber nur um einen kleinen Unterschied und um die Definitionsfrage eines Clusters handelt, kann diese Änderung, falls überhaupt benötigt, noch im Nachhinein in den Code eingebaut werden.
Der zweite Kompromiss ist das Kopieren der Daten vom System auf die Grafikkarte. Um daher einen fairen Vergleich zum ursprünglichen Algorithmus zu haben, müssen auch diese Werte in der Berechnung mit in Betracht gezogen werden. Dabei ist nicht zu vergessen, dass die CPU keinen zusätzlichen Kopieraufwand der Daten in Anspruch nimmt. Der zusätzliche Aufwand wird in Abbildung 22 und Abbildung 23 dargestellt. Hierbei beschreibt Abbildung 22 den Kopiervorgang der Digis vom Host (CPU/RAM) zum Device (GPU-Speicher) und Abbildung 23 den Kopiervorgang der berechneten Hits vom Device zum Host. CPU-Zeiten sind

hier nicht aufgeführt, da hier keine Daten kopiert werden müssen.
Der Aufwand des Kopierens hält sich dabei stark in Grenzen und verändert das Gesamtergebnis der Optimierung schlussendlich nur marginal. Das war zudem erwartbar, da die Schnittstellen (PCIe3 x16) innerhalb des genutzten Servers mit einer Geschwindigkeit von bis zu 16GByte/s arbeiten.

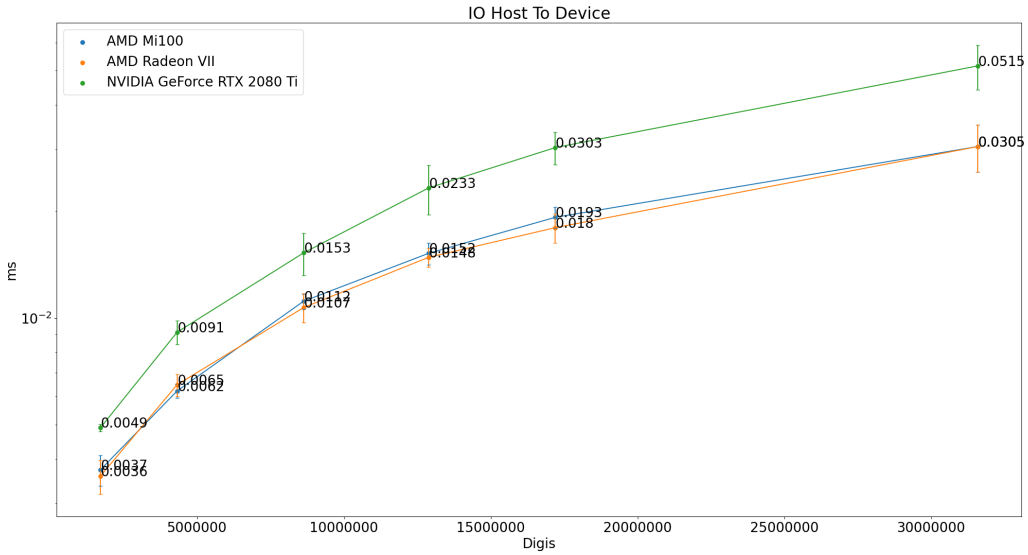

Abbildung 22: Zeitverbrauch des Kopiervorgangs der eingegebenen Digis vom Host zum Device

Unter Einbezug des Kopiervorgangs vor und nach der lokalen Rekonstruktion, dem Sortieren, Offset berechnen, Cluster finden, Cluster berechnen, Cluster Sortieren und Hits finden, ist das Ergebnis nach der Optimierung bis zu 21 Mal (Abbildung 24 logarithmisch, Abbildung 25 linear) schneller als der ursprüngliche Algorithmus, welcher auf der CPU ausgeführt wurde.
Dabei ist zu berücksichtigen, dass der im Vergleich stehende Algorithmus der CPU auch bereits unter der Nutzung der OpenMP Funktionen parallelisiert war. Der Vergleich zu einem komplett linear ausgeführten Code auf einer Recheneinheit würde im Vergleich zu dem hier gezeigten Prozess noch weit schlechter abschneiden. Selbst die Ausführung des optimierten Algorithmus unter Verwendung der CPU ist im Vergleich zum orignalen Algorithmus mehr als 3 mal so schnell. Somit ist nicht nur gezeigt, dass eine Auslagerung auf eine GPU für einen Performanceanstieg sorgt, sondern auch der Algorithmus unter gleichen Bedingungen optimaler für eine parallele Abarbeitung sorgt

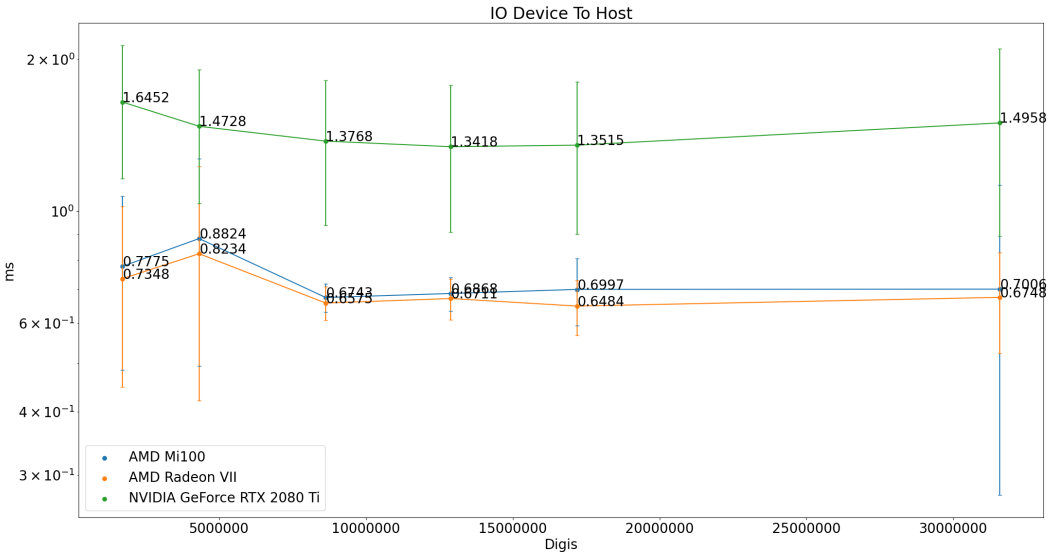

Abbildung 23: Zeitverbrauch des Kopiervorgangs der eingegebenen Digis vom Device zum Host

als der Bisherige.
Limitiert wird der auf der neue Algorithmus maßgeblich durch die Bandbreite der Devices. Auch wenn die Mi100 maßgeblich schneller in der gesamte lokalen Rekonstruktion sein sollte, ist hier nur ein kleiner Geschwindigkeitsunterschied nachweisbar. Dies ist auf die vielen Lese- und Schreiboperationen des neuen Algorithmus zurückzuführen und die damit einhergehende Ausnutzung der Bandbreite.
Somit lässt sich abschließend sagen, dass eine Erhöhung der Speicherbandbreite der GPUs zu einem weiteren Geschwindigkeitszuwachs des Algorithmus führen würde.

Algorithm	**HardwareType**	**Time**	**Percentage**
Original	CPU	1749.9ms	100,0 %
Optimized	CPU	510.96ms	29,2 %
	2080TI	164.53ms	9,4 %
	Radeon VII	110.41ms	6,3 %
	Mi100	84.46ms	4,8 %

Tabelle 2: Geschwindigkeiten bei 30.000.000 Digis im Direktverleich

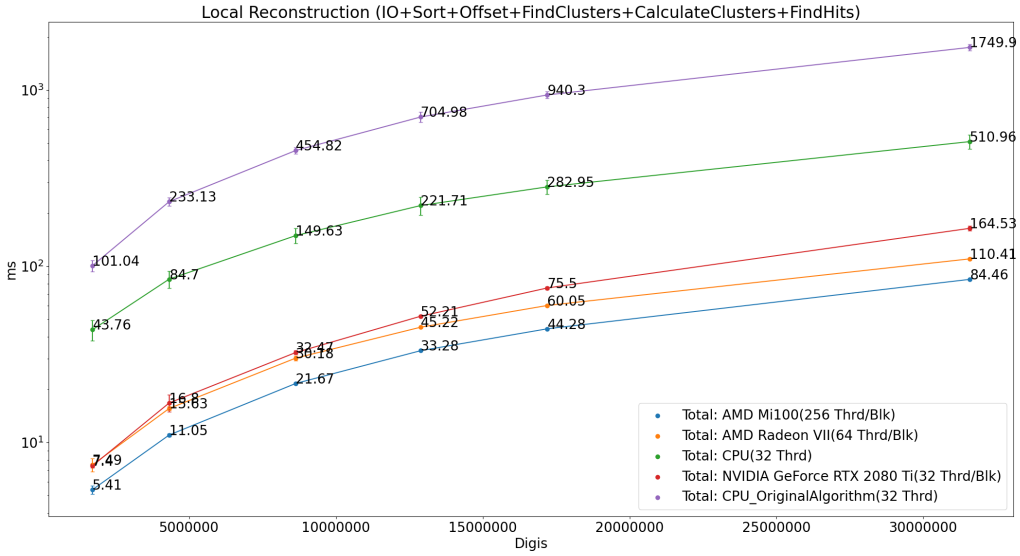

Abbildung 24: Vergleich der Ergebnisse von der optimierten lokalen Rekonstruktion (mit kanalorientiertem Clusterfinder und digiorientierter Clusterberechnung) zum Ausgangsalgorithmus auf unterschiedlichen Devices(Logarithmische Auftragung)

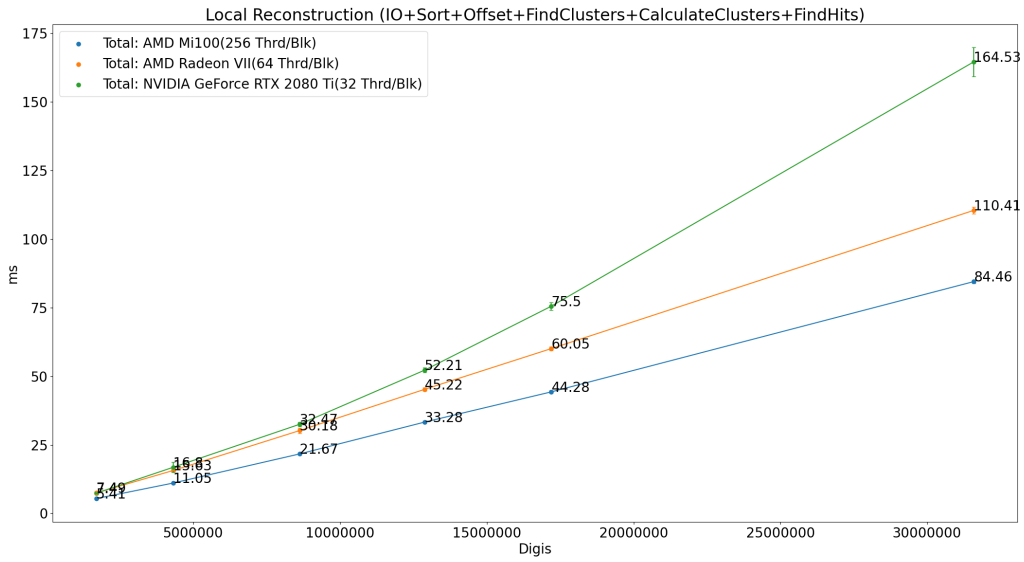

Abbildung 25: Vergleich der Ergebnisse von der optimierten lokalen Rekonstruktion (mit kanalorientiertem Clusterfinder und digiorientierter Clusterberechnung) zum Ausgangsalgorithmus auf unterschiedlichen GPUs(Lineare Auftragung)

5 Zusammenfassung

Angefangen bei der Sortierung bewirkte jeder der einzelnen Optimierungen der lokalen Rekonstruktion eine Verbesserung der Performance. In Summe gesehen lässt sich abschließend sagen, dass sich eine Parallelisierung und Auslagerung des Codes auf Grafikkarten rentiert hat. Der Geschwindigkeitszuwachs, beschrieben in Abschnitt 4, ist der Ertrag der einzelnen Optimierungen.
Durch die Optimierung des Sortierens werden in Zukunft nicht nur die Zeiten des STS, sondern auch die aller anderen Detektoren schrumpfen, welche die Funktionalitäten nutzen. Hervorgehoben sei an dieser Stelle noch einmal das XPU-Framework, das die Grundlage für diese Optimierung bot.
Das Finden und Berechnen der Cluster bewirkte abschließend den größten Performanceanstieg der gesamten lokalen Rekonstruktion. Lineare Abarbeitungsschritte wurden parallelisiert und der gesamte Arbeitsaufwand möglichst ausgeglichen auf die einzelnen Recheneinheiten der GPU verteilt.
Dadurch läuft die lokale Rekonstruktion nicht nur auf einer GPU, sondern auch im Vergleich zum bereits durch OpenMP optimierten Algorithmus schneller auf einer CPU. Je nach Verfügbarkeit der Ressourcen kann der Algorithmus durch das XPU-Framework einfach auf CPU oder GPU ausgeführt werden, obwohl gesagt sei, dass sich eine Ausführung auf einer GPU gerade bei großen Datenmengen rentieren würde, da es auch hier noch einen Performanceanstieg gibt.
Allgemein konnte ein Speedup von ca. 21 mit der schnellsten GPU-Variante erlangt werden (siehe Abbildung 24). Potential für weitere Verbesserungen könnte der Hitfinder noch bieten, indem die im Hintergrund liegende Mathematik und die dort zugrundeliegenden Algorithmen überdacht und optimiert werden.
Den aktuellen Code zum Hitfinder findet man unter folgendem Repository `https://github.com/fweig/CbmRootGPU.git` (Stand 25. Februar 2022).

6 Referenzen

Literatur

[1] Facility for Antiproton and Ion Research. Das compressed baryonic matter experiment bei fair. https://fair-center.eu/fileadmin/fair/experiments/CBM/documents/CBM_flyer_ToT.pdf. accesed on 2021-10-22.

[2] P. Senger and V. Friese. Cbm report 2012-01, nuclear matter physics at sis-100, the cbm collaboration. 2012.

[3] Hanna Malygina, Joachim Stroth, and Peter Senger. Hit reconstruction for the silicon tracking system of the cbm experiment. Technical report, Collaboration FAIR: CBM, 2018.

[4] J.M. Heuser, W.F.J. Müller, V. Pugatch, P. Senger, C.J. Schmidt, C. Sturm, and U. Frankenfeld. Gsi. technical desgin report for the cbm. http://repository.gsi.de/record/54798/files/GSI-Report-2013-4.pdf, 2013. accessed on 2019-03-05.

[5] John Adams Institute. Status of the international accelerator facility fair. https://www.adams-institute.ac.uk/news/status-international-accelerator-facility-fair. accessed on 2022-02-02.

[6] J De Cuveland, V Lindenstruth, CBM collaboration, et al. A first-level event selector for the cbm experiment at fair. In *Journal of physics: Conference series*, volume 331, page 022006. IOP Publishing, 2011.

[7] GSI. Gsi cbmroot. https://redmine.cbm.gsi.de/projects/cbmroot, 2018. accesseed on 2021-10-22.

[8] GSI. Gsi fairroot. https://cbmroot.gsi.de/, 2018. accessed on 2021-10-22.

[9] Intel. What is a gpu? https://www.intel.de/content/www/de/de/products/docs/processors/what-is-a-gpu.html. accessed on 2022-01-15.

[10] Nvidia. Thread hierachy in cuda programming. http://cuda-programming.blogspot.com/2012/12/thread-hierarchy-in-cuda-programming.html. accessed on 2019-03-05.

[11] Felix Weiglhofer. Xpu-framework. https://git.cbm.gsi.de/fweig/xpu. accessed on 2022-01-06.

[12] Jkrieger. Image mergesort. https://de.wikipedia.org/wiki/Mergesort#/media/Datei:Mergesort_example.png. accessed on 2021-10-22.

[13] Oded Green, Saher Odeh, and Yitzhak Birk. Merge path-a visually intuitive approach to parallel merging. *arXiv preprint arXiv:1406.2628*, 2014.

[14] Oded Green, Robert McColl, and David A Bader. Gpu merge path: a gpu merging algorithm. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 331–340, 2012.

[15] Nvidia Reseach Sean Baxter. Merge. https://moderngpu.github.io/merge.html. accessed on 2021-10-22.

[16] Volker Friese. A cluster-finding algorithm for free-streaming data. In *EPJ Web of Conferences*, volume 214, page 01008. EDP Sciences, 2019.

[17] CBM. cbmroot. https://git.cbm.gsi.de/computing/cbmroot. accessed on 2022-01-06.

7 Danksagung

Abschließend bedanke ich mich bei allen Personen, die mich während dieser Arbeit unterstützt und zum Abschluss beigetragen haben.
Mein Dank gilt Herrn Prof. Dr. Lindenstruth und dem FIAS dafür, dass ich in ihrem Institut diese Masterarbeit ablegen durfte.
Des Weiteren möchte ich mich bei Volker Friese und allen Mitarbeitern der GSI bedanken.
Besonders hervorzuheben ist an dieser Stelle Herr Dr. PD. Andreas Redelbach, der für die Betreuung und somit den Erfolg dieser Arbeit Sorge getragen hat.
Außerdem bedanke ich mich ganz besonders bei Felix Weiglhofer, der bei Problemen in jedem Bereich Hilfestellungen und Lösungen bieten konnte und durch seine Arbeit die Grundlage für die hier gezeigten Optimierungen geboten hat.
Mein Dank gilt auch allen weiteren Mitarbeitern des Lehrstuhls von Herrn Prof. Dr. Lindenstruth für das kontinuierliche Feedback und die gestellten und verwalteten Ressourcen.
Zum Schluss gilt mein Dank meiner Familie und Freunden, die mich sowohl moralisch als auch bei der Korrektur dieser Arbeit unterstützt haben.