Fachhochschule Aachen
Campus Jülich

Fachbereich: Medizintechnik und Technomathematik
Studiengang: Technomathematik

Beschleunigung eines Spurfindealgorithmus für den Straw Tube Tracker des $\overline{\text{P}}$ANDA-Detektors durch Parallelisierung mit CUDA C

Masterarbeit von Jette Schumann

Jülich, den 30. September 2015

Eigenständigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Ort und Datum	Unterschrift

Diese Arbeit wurde betreut von:

1. Prüfer: Prof. Dr. Andreas Terstegge
2. Prüfer: Dr. Tobias Stockmanns

Die vorliegende Masterarbeit wurde am Institut für Kernphysik der Forschungszentrum Jülich GmbH erstellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Das bisherige Bild der inneren Struktur von Hadronen (wie z. B. Protonen) wirft noch Fragen auf. Die Eigenschaften der Elementarteilchen, aus denen die Hadronen aufgebaut sind, und der Kräfte, die zwischen diesen Elementarteilchen wirken, können zum Teil nicht detailliert beschrieben werden. Experimente im Bereich der Mittel- und Hochenergiephysik sollen Aufschluss über die Natur der Elementarteilchen bringen. Für derartige Experimente sind komplexe Beschleunigeranlagen und Detektorsysteme erforderlich. Geladene Teilchen werden auf Energien von mehreren GeV bis TeV beschleunigt und anschließend aufeinander oder auf stationäre Target-Materialien geschossen. Durch den hohen Impuls der beschleunigten Teilchen gehen aus den Zusammenstößen eine Vielzahl von Sekundärteilchen hervor, die oft sehr kurzlebig sind. Zur Erfassung dieser Teilchen wird ein Detektorsystem benötigt, das eine präzise Messung über einen großen Raumwinkelbereich ermöglicht. Moderne Detektoren erlauben eine genaue Messung der Flugbahnen der Teilchen, ihrer Energien und Impulse sowie eine Identifikation der entstandenen Teilchenarten.

Mit dem $\overline{\text{P}}$ANDA-Experiment sollen Fragen aus dem Bereich der Kern- und Teilchenphysik geklärt werden, die auf das vollständige Verständnis der Quantenchromodynamik abzielen. An $\overline{\text{P}}$ANDA werden Zusammenstöße von beschleunigten Antiprotonen mit stationären Protonen untersucht. Es wird derzeit als ein Teil der Beschleunigeranlage FAIR in Darmstadt gebaut. Eine Besonderheit des $\overline{\text{P}}$ANDA-Detektors ist, dass kein Hardware-Trigger verwendet wird, der physikalisch interessante Ereignisse erkennt und nur für diese eine permanente Speicherung der anfallenden Daten veranlasst. Aufgrund der speziellen Eigenschaften der zu messenden physikalischen Vorgänge wurde ein anderer Ansatz gewählt. Alle Detektor-Komponenten messen kontinuierlich und die anfallende Datenmenge wird erst anschließend gefiltert, wodurch besondere Ansprüche an die auswertende Software gestellt werden.

An $\overline{\text{P}}$ANDA kommt es mit einer Ereignisrate von bis zu 2×10^7 /s zu Teilchenzusammenstößen und eine große Anzahl von Sekundärteilchen entsteht. Das resultiert in einer Datenmenge von bis zu 200 GB/s. Von dieser können nur 100-200 MB/s in den Massenspeicher geschrieben werden. Aus diesem Grund können nur physikalisch interessante Ereignisse zur späteren Untersuchung permanent gespeichert werden. Dies erfordert eine Rekonstruktion der Ereignisse aus den Detektor-Signalen in Echtzeit. Es wird Software benötigt, die die Daten möglichst schnell verarbeitet und somit auch eine schnelle Filterung ermöglicht. Die Rekonstruktion von Ereignissen ist ein mehrstufiger Prozess,

bestehend aus mehreren Verfahren und Algorithmen. Eine Stufe dessen ist die Spurfindung (engl. *Trackfinding*). Die Aufgabe eines Spurfinders liegt darin, eine große Menge von Detektor-Signalen bezüglich der gemessenen Flugbahnen zu gruppieren. Eine resultierende Gruppe enthält dann nur die Signale, die zu einer eindeutigen Teilchenflugbahn gehören. In der Bachelorarbeit „Entwicklung eines schnellen Algorithmus zur Suche von Teilchenspuren im Straw Tube Tracker des $\overline{\text{P}}$ANDA-Detektors“ [1] wurde ein solches Verfahren entwickelt. Das Verfahren wurde im Rahmen der vorliegenden Arbeit weiterentwickelt und durch Parallelisierung unter Verwendung einer Grafikkarte beschleunigt, um die erforderliche schnelle Rekonstruktion von Ereignissen zu ermöglichen. Zur Realisierung dessen wurden rechenzeitintensive Teile des Verfahrens mit Hilfe von CUDA C auf einer NVIDIA-Grafikkarte implementiert. Es wurde untersucht, ob der Einsatz einer Grafikkarte eine effiziente und günstige Variante zur Steigerung der Performance darstellt.

Im 2. Kapitel dieser Arbeit werden das FAIR-Beschleunigerzentrum und das $\overline{\text{P}}$ANDA-Experiment vorgestellt. Dabei wird der grundlegende Aufbau des $\overline{\text{P}}$ANDA-Detektors beschrieben und gesondert auf den zentralen spurgebenden Detektor, den *Straw Tube Tracker* (STT), eingegangen.

In Kapitel 3 wird das Spurfinde-Verfahren für den STT vorgestellt. Das zur Simulation benutzte Framework PandaRoot wird vorab eingeführt. Es wird auf die Funktionsweise des Verfahrens und die Abweichungen zur anfänglichen Version eingegangen. Mit einer Analyse des Laufzeitverhaltens wird die Auswahl der zu parallelisierenden Teile gestützt.

Anschließend wird in Kapitel 4 die Grafikkartenprogrammierung mit CUDA C eingeführt. CUDA C ermöglicht durch Erweiterungen der Programmiersprache C parallelen Code auf einer Grafikkarte auszuführen. Das Programmier- und Hardwaremodell von CUDA C werden vorgestellt.

In Kapitel 5 werden die parallelisierten Teile des Spurfinders detailliert dargelegt. Die Programmierung auf einer Grafikkarte bringt bestimmte Anforderungen z. B. an die Datenstrukturen mit sich, welche geschildert werden. Es gibt zwei grundlegende Ansätze für die Parallelisierung des Spurfinders. Das Verfahren an sich kann parallelisiert werden und es kann parallel für verschiedene Eingabedaten ausgeführt werden. Diese beiden Ansätze werden ausführlich beschrieben und es wird erklärt, wie paralleler Code in PandaRoot eingebunden werden kann.

Es folgt eine Analyse des parallelen Codes in Kapitel 6. Mit Hilfe eines speziellen Profilers wird der parallele Code auf bestimmte Performance-Metriken untersucht. Es wird überprüft inwieweit der Spurfinder effizient auf einer Grafikkarte ausgeführt werden kann.

Im letzten Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und mögliche zukünftige Erweiterungen und Verbesserungen des Verfahrens umrissen.

2 Physikalischer Hintergrund

Bei dem zu beschleunigenden Verfahren handelt es sich um einen Spurfinder, der die gemessenen Signale des *Straw Tube Trackers* (STT) zu den ursprünglichen Flugbahnen der Teilchen gruppieren soll. Der STT ist als spurgebender Detektor Teil einer sehr komplexen Detektoranlage, des $\overline{\text{P}}$ANDA-Detektors. Dieser dient der Erfassung der Daten für das $\overline{\text{P}}$ANDA-Experiment, welches neben anderen Experimenten an einer Beschleunigeranlage namens FAIR verwirklicht werden wird. In den nächsten Abschnitten werden FAIR, $\overline{\text{P}}$ANDA und der STT kurz eingeführt.

2.1 FAIR – Facility for Antiproton and Ion Research

FAIR ist eine internationale Forschungseinrichtung, die derzeit in Darmstadt neben dem Gelände der *Gesellschaft für Schwerionenforschung* (GSI) entsteht. Es handelt sich um eine Beschleunigeranlage, an welcher mit Antiprotonen- und Ionenstrahlen experimentiert werden wird. Sie bietet eine komplexe Infrastruktur, sodass unterschiedliche physikalische Programme parallel betrieben werden können. Das Ziel der geplanten Experimente ist es, grundlegende Fragen zur Entstehung des Universums und der Struktur von Materie zu beantworten. Dabei wird ein Fokus auf folgende Forschungsgebiete gelegt: Atom-, Plasma-, Antiprotonen-, Kernstruktur- und Kernmateriephysik. Viele der bisher noch ungeklärte Fragen dieser Gebiete sollen mit einer Vielzahl von Experimenten an FAIR aufgeschlüsselt werden, die sich in vier Gruppen einteilen lassen - die vier *Säulen* von FAIR [2]:

APPA Die Abkürzung steht für *Atomic, Plasma Physics and Applications*. Mit Experimenten in diesen Bereichen soll u. a. das Plasma, der sogenannte vierte Aggregatzustand, bei niedrigen Temperaturen und hohen Drücken erforscht werden, wie es z. B. im Inneren von großen Planeten vorkommt. Außerdem wird bei Experimenten mit schweren Ionen die kosmische Strahlung untersucht, welche bei bemannten Weltraummissionen eine große Rolle spielt.

CBM Das *Compressed Baryonic Matter*-Experiment dient zur Erforschung komprimierter Kernmaterie mit einer hohen energetischen Dichte und wird Bedingungen des Frühstadiums des Universums nachstellen.

NUSTAR Experimente im Bereich *Nuclear Structure, Astrophysics and Reactions* sollen dazu beitragen die Entstehung von Elementen, die schwerer als Eisen sind, zu erforschen. Zu diesem Zweck werden seltene Isotope erzeugt und ihr Aufbau untersucht. Es wird die Erforschung schwerer Elemente und die Aufklärung astrophysikalischer Phänomene angestrebt.

$\bar{\text{P}}$ANDA Das *Antiproton Annihilation at Darmstadt*-Experiment hat u. a. das Ziel zu erklären, welche Eigenschaften die starke Kraft hat, die Quarks in Hadronen zusammenhält. Im Abschnitt 2.2 wird genauer auf das physikalische Programm und den zugehörigen Detektor eingegangen.

Neben der Grundlagenforschung dient FAIR auch der angewandten Forschung. z. B. können durch die Möglichkeit der Erzeugung hochenergetischer Ionenstrahlen neue Wege zur Kernfusion als eine zukünftige Energiequelle erforscht werden. Außerdem werden die Experimente ein Voranschreiten im Bereich der Raumfahrt unterstützen. Der Einfluss kosmischer Strahlung auf Astronauten und technische Komponenten in Satelliten und Raumschiffen wurde bisher nicht ausgiebig erforscht - FAIR verspricht neue Erkenntnisse. [3]

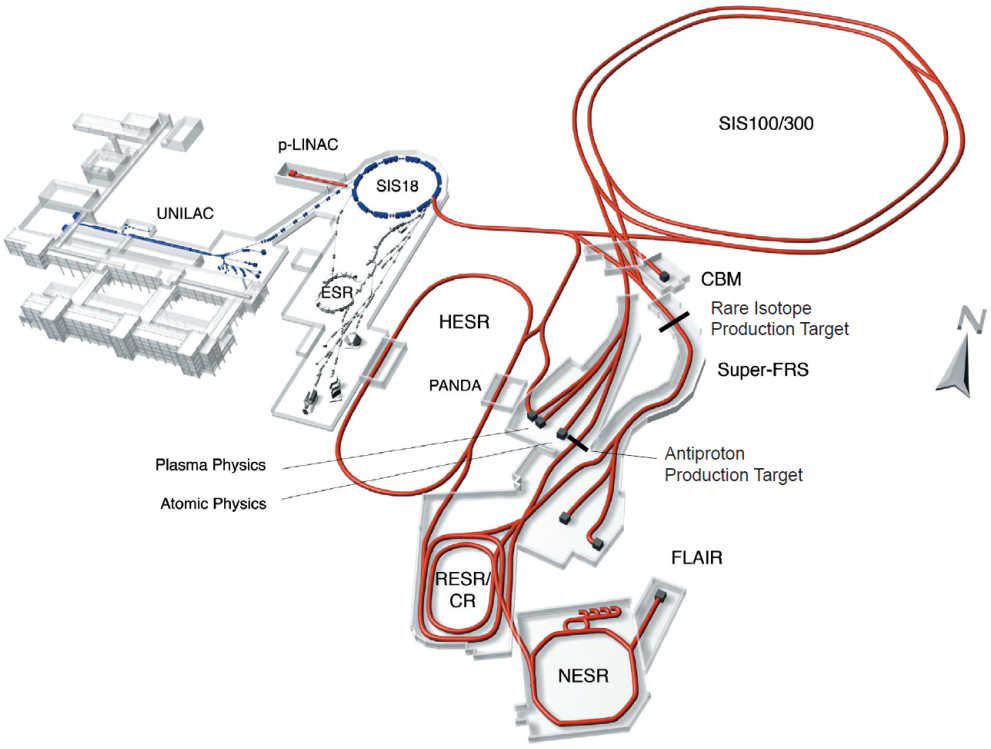


Abbildung 2.1: Schematische Ansicht von FAIR (bereits existierende Komponenten in blau, geplante Komponenten in rot). [4, S. 9]

Abbildung 2.1 zeigt eine schematische Darstellung der Beschleunigeranlage von FAIR. Diese schließt bereits bestehende Komponenten der GSI ein, welche

als Vorbeschleuniger dienen (SIS18). Das Herzstück der Beschleunigeranlage bildet ein Doppelring (SIS100/300) mit einem Umfang von 1100 m, welcher im Anschluss Protonen auf bis zu 29 GeV beschleunigen wird. Es schließt sich ein komplexes System aus Komponenten zur Strahlführung, Target-Systemen und Speicherringen an, mit dem eine große Vielfalt von Teilchenstrahlen erzeugt und den verschiedenen Experimenten zugeführt werden kann. Zu den Speicherringen zählen der *Collector Ring* (CR), der *Recuperated Experimental Storage Ring* (RESR), der *New Experimental Storage Ring* (NESR) und der *High-Energy Storage Ring* (HESR). Diese Speicherringe sind wiederum mit Experimentstationen und Kühl- und Targetsystemen ausgestattet. [4, S. 9-12]

Das $\bar{P}$ANDA-Experiment wird am HESR eingerichtet werden. Die dafür benötigten Antiprotonen kommen in der Natur für gewöhnlich nicht vor und müssen über ein mehrstufiges System beginnend mit Protonen erzeugt werden. Die Protonen werden über p-LINAC, SIS18 und SIS100 auf bis zu 29 GeV beschleunigt und auf ein Wolfram-Target geschossen. Aus Kernreaktionen im Wolfram-Target gehen u. a. die gewünschten Antiprotonen hervor, welche im CR gesammelt und anschließend in den HESR eingespeist werden. Der HESR ermöglicht eine Beschleunigung der Antiprotonen auf bis zu 14.5 GeV und kann diese dem $\bar{P}$ANDA-Detektor zuführen, welcher direkt im Ring platziert ist.

2.2 $\bar{P}$ANDA – Antiproton Annihilation at Darmstadt

$\bar{P}$ANDA ist als eine bedeutsame Komponente von FAIR geplant und wird mit Antiprotonen, die über den HESR zur Verfügung gestellt werden, experimentieren. Dabei werden Antiproton-Proton-Annihilationen und die Reaktionen von Antiprotonen mit schweren Kernen untersucht. Eine Annihilation ist der Prozess der Paarvernichtung bei dem ein Fundamentalteilchen mit seinem korrespondierenden Antiteilchen zusammenstößt. Da Impuls und Energie der sich auslöschenden Teilchen erhalten bleiben, gehen aus diesem Zusammenstoß andere Teilchen hervor, deren Eigenschaften es zu untersuchen gilt.

2.2.1 Physikalisches Programm

Das Experiment soll Aufschluss über Fragen aus dem Bereich der Hadronen- und Kernphysik liefern. Hadronen sind Teilchen, die der *starken Wechselwirkung* unterliegen. Diese Wechselwirkung wird auch als *starke Kraft* bezeichnet und wird durch die Theorie der *Quantenchromodynamik* (QCD) beschrieben. Bekannte Vertreter der Hadronen sind die Nukleonen (Protonen und Neutronen) aus welchen Atomkerne aufgebaut sind. Alle Hadronen wiederum bestehen aus Quarks und werden durch die starke Kraft zusammengehalten. Als Austauschteilchen für die starke Kraft dienen sogenannte Gluonen. Gluonen

interagieren nicht nur mit Quarks sondern auch mit anderen Gluonen (Selbstwechselwirkung). Die QCD beschreibt die Wechselwirkung zwischen Quarks nur für kleine Distanzen gut. Wachsen die Abstände der Quarks auf die Größe des Nukleons, kommt es zu dem Effekt, dass die starke Kraft mit anwachsender Distanz zwischen den Quarks größer wird. Sie ist so stark, dass sich allein aus der Feldenergie zwischen den Quarks neue Quark-Antiquark-Paare bilden, welche neue Hadronen formen. Aus diesem Grund können Quarks nicht als freie Teilchen untersucht werden. Dieses Phänomen wird als *Confinement* bezeichnet und ist in der QCD bisher nicht vollständig verstanden worden. Quarks tauchen nur in Hadronen in einem Gebilde aus drei Quarks (Baryonen) oder einem Quark-Antiquark-Paar (Mesonen) auf. Andere Konstellationen mit z. B. 4 oder 6 Quarks sind theoretisch nicht ausgeschlossen, wurden aber bisher noch nicht experimentell entdeckt. Eine weitere Besonderheit ist, dass Hadronen viel schwerer sind als die Summe der Masse ihrer Bestandteile. Es wird angenommen, dass dies auf die Eigenschaften der starken Kraft zurückzuführen ist. Durch Experimente mit hochenergetischen Antiprotonenstrahlen soll die Entstehung der hadronischen Masse und die Natur der starken Wechselwirkung erfasst werden. [5, S. 3 ff.]

Die geplanten Experimente an $\overline{\text{P}}$ANDA lassen sich in folgende Forschungsbereiche einteilen [6, S. 4 f.], [5, S. 9 ff.]:

Nukleonenstruktur Der genaue innere Aufbau der Bestandteile der Atomkerne soll erforscht werden. Viele Detailaspekte dessen müssen noch untersucht werden.

Hadronen Spektroskopie Die Grund- und Anregungszustände von Hadronen sollen präzise vermessen werden, um ein besseres Verständnis der QCD zu erlangen. Dabei wird auch nach exotischen Zuständen gesucht. Es existieren z. B. *Glueballs*, die nur aus Gluonen bestehen, und *Hybride*, bestehend aus einem Quark-Antiquark-Paar und angeregten Gluonen. Die Eigenschaften dieser sind auf die starke Wechselwirkung zurückzuführen, welche erforscht werden soll.

Hadronen in Materie Hadronen verhalten sich in gebundenen Zuständen anders als freie Teilchen. Insbesondere haben sie in gebundenen Zuständen eine geringere Masse. Der Ursprung und die Eigenschaften der hadronischen Masse soll erforscht werden.

Hyperkerne Hyperkerne sind Systeme in welchen die Quarks der Nukleonen durch *Strange-Quarks* ersetzt werden. Auf diese Weise wird ein weiterer Freiheitsgrad, die *Strangeness* eingeführt. Diese Erweiterung führt zu neuartigen Elementen im Periodensystem, deren Eigenschaften untersucht werden sollen.

2.2.2 Experimentaufbau

Zur Realisierung des Experiments werden die Antiprotonen aus der Reaktion eines primären Protonenstrahls aus dem SIS100 mit einem Target erzeugt. Die Produktionsrate der Antiprotonen wird ca. $2 \times 10^7\,\mathrm{s}^{-1}$ betragen. Die Antiprotonen werden im CR gespeichert bis eine Menge von 5×10^5 produziert wurde. Danach werden sie in den HESR eingespeist. Dieser Speicherring ist 574 m lang und liefert einen Antiprotonenstrahl von hoher Qualität und Intensität für das $\bar{P}$ANDA-Experiment. Der Strahlimpuls kann zwischen 1.5 und 15 GeV/c reguliert werden. In der finalen Version von FAIR gibt es zwei Betriebsformen des HESRs. Im *high-luminosity*-Modus wird eine maximale Luminosität von $L = 2 \times 10^{32}\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ bei einer Impulsauflösung von $\frac{\Delta p}{p} \leq 10^{-4}$ erreicht. Die Luminosität beschreibt die Anzahl von Teilchenbegegnungen innerhalb einer bestimmten Zeit auf einer Fläche. Je höher die Luminosität, desto größer ist die Ereignisrate. Im *high- resolution*-Modus wird eine bessere Impulsauflösung von $\frac{\Delta p}{p} \leq 5 \times 10^{-5}$ bei einer verringerten Luminosität von $L = 10^{31}\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ erreicht. [7, S. 2]

Der Strahl wird im Inneren des $\bar{P}$ANDA-Detektors in einem Vakuum mit Target-Material kollidieren, das über ein Target-System zugeführt wird. Die Target-Systeme müssen verschiedene Anforderungen erfüllen. Zum einen sollten sie möglichst wenig Raum in der Interaktions-Zone einnehmen und dürfen sich nicht negativ auf das Vakuum auswirken. Die Targets sollten homogen sein und räumlich präzise eingeschränkt werden können. Hinzu kommt, dass für viele Experimente eine Dichte von mindestens $4 \times 10^{15}\,\mathrm{Atome/cm}^2$ erforderlich ist. Es ist nicht möglich allen Anforderungen mit einem einzigen Target-System gerecht zu werden. Daher stehen verschiedene Target-Systeme zur Verfügung, die flexibel ausgetauscht werden können. Zu den wichtigsten Target-Systemen in $\bar{P}$ANDA zählen das *Pellet Target*, das *Cluster Jet Target* und *Nuclear Targets*. Das Pellet Target führt dem Antiprotonenstrahl über eine Düse von oben gefrorene Tröpfchen von Wasserstoff zu. Das Cluster Jet Target wandelt über eine Düse Wasserstoffgas in einen Cluster-Strahl. Cluster sind Gebilde aus einzelnen Wasserstoff-Molekülen, die durch wechselwirkende Kräfte zusammengehalten werden. Um die Reaktionen von Antiprotonen mit schweren Kernen zu untersuchen, sollen Nuclear Targets in Form vom Draht- oder Folien-Targets eines bestimmten festen Materials genutzt werden. [5, S. 43 ff.]

2.2.3 Der $\bar{P}$ANDA-Detektor

Über den HESR wird ein hochenergetischer Antiprotonenstrahl zur Verfügung gestellt, der auf das stationäre Target im Zentrum des Detektors gerichtet wird. Durch den hohen Impuls der Antiprotonen gehen aus den Teilchenzusammenstößen Sekundärteilchen hervor, die sich sowohl in Richtung des Antiprotonenstrahls als auch um den Stoßpunkt herum ausbreiten. Um den relevanten

Raumwinkelbereich abzudecken, ist ein komplexes Detektorsystem nötig, welches in Abbildung 2.2 zu sehen ist. Das Detektorsystem setzt sich aus zwei Magnetspektrometern zusammen, aus dem *Target Spectrometer* und dem *Forward Spectrometer*. Bewegt sich ein geladenes Teilchen durch ein Magnetfeld, kommt es zur Ablenkung des Teilchens durch die Wirkung der Lorentz-Kraft. Die Lorentz-Kraft ist von der Ladung und Geschwindigkeit des Teilchens und der Stärke des Magnetfeldes abhängig. Durch die Ablenkung von Teilchen in den Magnetsprektrometern lassen sich daher Rückschlüsse auf die Impulse der Teilchen ziehen. Der $\overline{\text{P}}$ANDA-Detektor besteht aus Sub-Detektoren, die entweder auf die Spurfindung, die Teilchenidentifikation oder die Energiebestimmung spezialisiert sind und sich gegenseitig ergänzen. Nur durch das Zusammenspiel von Detektor-Komponenten aus allen drei Bereichen ist eine vollständige Messung der physikalischen Ereignisse möglich.

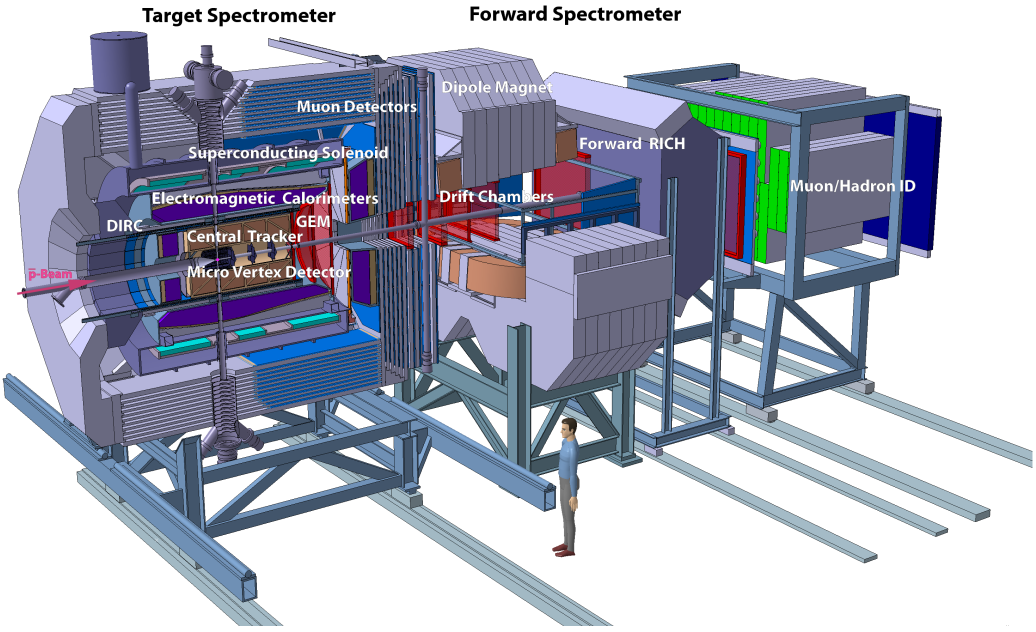


Abbildung 2.2: Schematische Darstellung des $\overline{\text{P}}$ANDA-Detektors. [8, S. 14]

Target Spectrometer

Das Target Spectrometer befindet sich direkt um dem Stoßpunkt. Es dient zur Messung von Teilchen nahe der Kollisionsregion deren Flugbahnen einen großen Polarwinkel besitzen. Die Messung erfolgt mit Hilfe eines Solenoid-Magnetfeldes der Stärke 2 T. Die zugehörigen Detektoren sind zwiebelschalenartig um den Stoßpunkt angeordnet. Für die Teilchenzusammenstöße werden Targets von oben senkrecht auf den Antiprotonenstrahl geführt. Die dafür benötigte Apparatur durchdringt alle Schichten des Spektrometers.

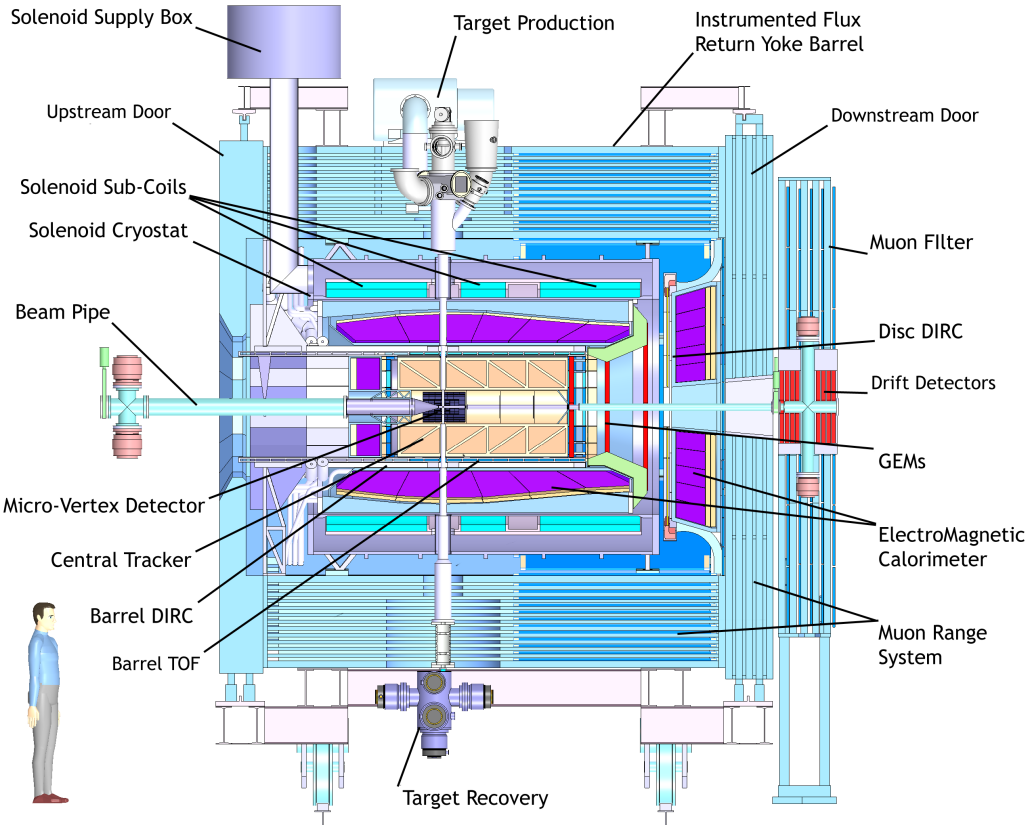


Abbildung 2.3: Schematische Darstellung des Target Spectrometers des $\bar{P}$ANDA-Detektors. [6, S. 13]

In Abbildung 2.3 ist ein vereinfachter Querschnitt durch das Target Spectrometer zu sehen. Es ist in drei Bereiche strukturiert, die verschiedene Winkelbereiche abdecken: ein Fass (22° bis 140°), eine End-Kappe in Vorwärtsrichtung (bis hinunter zu 5°/10° in vertikaler/horizontaler Ebene) und die End-Kappe in Rückwärtsrichtung (145° bis 170°). Zu den spurgebenden Detektoren im Target Spectrometer zählen der *Micro Vertex Detector* (MVD), der *Straw Tube Tracker* (STT) und der *Gas Electron Multiplier Detector* (GEM). Der MVD erlaubt es, Spuren von geladenen Teilchen nahe dem Wechselwirkungspunkt zu messen. Der STT wird auch als *Central Tracker* bezeichnet und wird im Abschnitt 2.3 ausführlicher beschrieben. Der GEM befindet sich weiter vorne in Richtung des Strahls und kann Flugbahnen erfassen, die einen kleinen Winkel von maximal 22° zur Strahlachse haben und vom STT nicht wahrgenommen wurden. Zur Teilchenidentifikation werden ein *Detection of internally reflection light*- (DIRC), Myonen- und *Time of flight*-Detektor eingesetzt. Der DIRC-Detektor nutzt den Cherenkov-Effekt zur Bestimmung der Teilchart. Seine Komponenten sind im Fass (Barrel DIRC) und in der End-Kappe in Vorwärtsrichtung (Disc DIRC) zu finden. Der TOF-Detektor (Barrel TOF)

dagegen erlaubt die Teilchenidentifikation anhand einer präzisen Zeitmessung. Elektromagnetische Kalorimeter tragen zur Energiebestimmung elektromagnetisch wechselwirkender Teilchen (Elektronen, Positronen und Gamma-Teilchen) bei. [6, S. 12 ff.]

Forward Spectrometer

Mit dem Forward Spectrometer werden Teilchen erfasst, die sich mit einem vertikalen bzw. horizontalen Winkel von bis zu $\pm 5°$ bzw. $\pm 10°$ bezogen auf die Richtung des Antiprotonenstrahls vom Stoßpunkt entfernen. Zur Messung wird ein $2\,\mathrm{T\,m}$ Dipol-Magnet genutzt. Als spurgebender Detektor dient der *Forward Tracker*. Dessen Komponenten werden in Vorwärtsrichtung an verschiedenen Punkten platziert und bestehen aus Straw Tubes, wie sie auch im STT zu finden sind. Zur Teilchenidentifikation dient der *Ring Imaging Cherenkov*-Detektor (RICH), der sich wie der DIRC-Detektor den Cherenkov-Effekt zu Nutze macht. Wie im Target Spectrometer sollen auch in diesem Magnetspektrometer TOF- und Myonen-Detektoren und ein elektromagnetische Kalorimeter zum Einsatz kommen. [6, S. 16 ff.]

Datenerfassung

Eine Besonderheit von $\overline{\mathrm{P}}$ANDA ist, dass kein Hardware-Trigger zur Selektion und Reduktion der anfallenden Daten genutzt wird. Vergleichbare Experimente setzen ein mehrstufiges System ein, dessen erste Stufe solch ein Hardware-Trigger ist. Ein derartiger Trigger kann mit Hilfe vereinzelter, schneller Sub-Detektoren entscheiden, welche Daten physikalisch interessant sind. Die restlichen Detektor-Komponenten werden daraufhin angewiesen, die Daten entsprechend zu filtern. Die gesammelten Daten der akzeptierten Ereignisse aller Detektor-Komponenten werden anschließend in die nächste Trigger-Ebene zur weiteren Filterung oder direkt in den vorgesehenen Speicher verschoben und ausgewertet. Die Puffer-Kapazitäten dieser Ebenen sind daher meist für die Zeit zur Datenerfassung ausschlaggebend. Der Einsatz eines Hardware-Triggers hat zur Folge, dass nur ein Bruchteil der gesamten anfallenden Datenmenge ausgelesen wird. Aufgrund der Ähnlichkeit der Signale physikalisch interessanter Ereignisse mit den Untergrund-Ereignissen, kann für $\overline{\mathrm{P}}$ANDA kein Hardware-Trigger eingesetzt werden. Signale zu vieler relevanter Ereignisse würden nicht ausgelesen werden. Es wurde ein anderes Konzept entwickelt, dass besser auf die hohen Datenraten ausgelegt ist. Jeder Detektor arbeitet autonom, misst kontinuierlich und vorverarbeitet die Signale. Das Herausfiltern physikalisch interessanter Ereignisse erfolgt auf der Auslese-Ebene und erfordert die vollständige Rekonstruktion eines Ereignisses. Dies ermöglicht es, neuartige Trigger-Algorithmen zu erproben und macht das Filtern sehr flexibel. Allerdings fällt eine sehr hohe Datenmenge an. Bei einer Ereignisrate von bis zu $2 \times 10^7\,\mathrm{Events/s}$ ist mit einer Datenmenge von $200\,\mathrm{GB/s}$ zu rechnen. Von

diesen können jedoch nur 100-200 MB/s in den vorgesehenen Massenspeicher geschrieben werden. Daher besteht ein dringender Bedarf an schneller Software, die die Ereignisse nach bestimmten Kriterien möglichst schnell filtert. [8, S. 24]

2.3 STT – Straw Tube Tracker

2.3.1 Das Tracking-System von $\overline{\text{P}}$ANDA

Das Tracking-System von $\overline{\text{P}}$ANDA hat die Aufgabe, möglichst genau die Flugbahnen aller Sekundärteilchen, die aus dem primären Stoß und sekundären Zerfallsvertices hervorgehen, zu bestimmen und die zugehörigen Impulse zu ermitteln. Das magnetische Feld des Solenoiden im Target Spectrometer wirkt parallel zur Strahlachse und hat zur Folge, dass die geladenen Teilchen durch die Lorentz-Kraft transversal kreisförmig abgelenkt werden und im Idealfall eine helixförmige Flugbahn annehmen. Der Krümmungsradius der Flugbahn gibt Aufschluss über den Impuls des Teilchens. Anhand der STT-Signale kann schon eine Vielzahl der Teilchenspuren rekonstruiert werden. Um aber die maximale Anzahl von Teilchenspuren zu rekonstruieren, müssen Informationen mehrerer Sub-Detektoren eingeholt und in der richtigen Reihenfolge zusammengeführt werden. Nur so ist es möglich, die Flugbahnen vom Produktionsort des Teilchens bis zum Verlassen des Detektors vollständig wiederzugeben. Für die Rekonstruktion der zentralen Spuren bedeutet dies, dass die Signale des MVDs, des STTs und des GEMs vereint werden müssen. Flugbahnen mit einem sehr kleinen Polarwinkel verlassen das Solenoid-Magnetfeld zu früh, um korrekt gemessen werden können. Für deren Rekonstruktion werden Detektor-Komponenten des Forward Spectrometers hinzugezogen. [6, S. 19]

Bei der Spur-Rekonstruktion ist zu beachten, dass die Teilchen mit den Detektor-Materialien wechselwirken und es zu Vielfachstreuung und Energieverlust kommt. Das führt zur Veränderung der Flugbahnen der Teilchen. Die durch die Lorentz-Kraft hervorgerufene Krümmung der Spur nimmt zu. Spiralförmige Flugbahnen mit immer kleiner werdenden Radien sind die Folge.

2.3.2 Aufbau und Eigenschaften

Der Straw Tube Tracker ist der zentrale spurgebende Detektor für geladene Teilchen. Er befindet sich im Target Spectrometer des $\overline{\text{P}}$ANDA-Detektors und umschließt den MVD (siehe Abbildung 2.3). Die Aufgabe des STTs liegt darin, spiralförmige Flugbahnen geladener Teilchen mit einer hohen örtlichen Auflösung zu messen. Mit Hilfe dieser Messung kann der Impuls des Teilchens und der spezifischen Energieverlust als Beitrag zur Teilchenidentifikation bestimmt werden. Der Impuls der nachzuweisenden geladenen Teilchen liegt zwischen einigen 100 MeV/c und 8 GeV/c. Der STT besteht aus 4 636 einzelnen

Straw Tubes (Driftröhrchen), welche in einer dichten Packung in der Form eines Zylinders um den Stoßpunkt herum angeordnet sind. Der innere Radius der Zylinders beträgt 150 mm und der äußere Radius 420 mm. Die Gesamtlänge ist 1650 mm, wobei sich das Target auf einer Höhe von 550 mm befindet, sodass 1100 mm in Richtung der Strahls abgedeckt werden. Aus Gründen der Wartung und Zugänglichkeit ist der Detektor in zwei zylindrische Halbschalen unterteilt, welche in Abbildung 2.4 zu sehen sind.

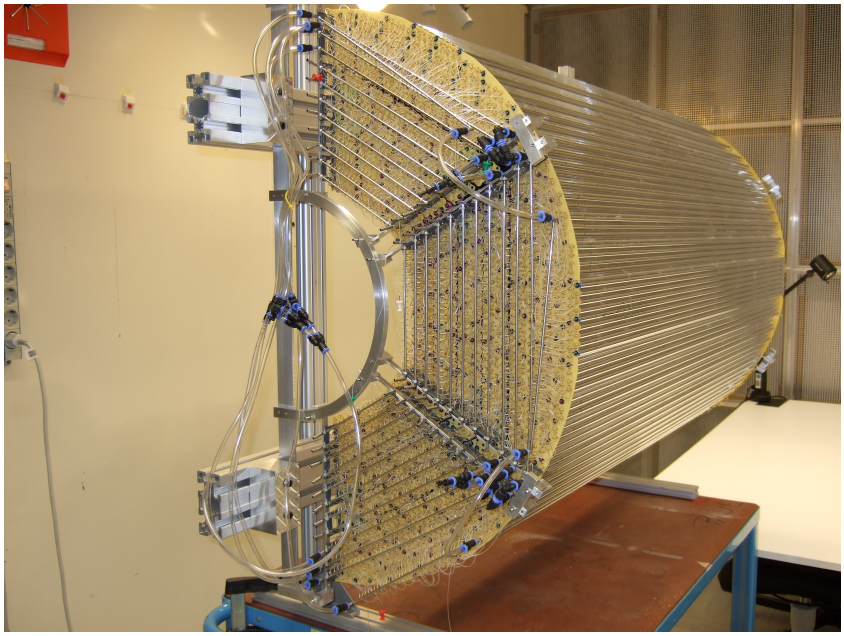


Abbildung 2.4: Halbschale eines STT-Prototypen des $\overline{\text{P}}$ANDA-Detektors. [9]

Durch die Wirkung der Lorentz-Kraft ist eine helixförmige Flugbahn der Teilchen zu erwarten. Um diese zu rekonstruieren, werden die Straws teils parallel und teils etwas gedreht zur Strahlachse angeordnet. Mit den Signalen der parallelen Straws können die Kreisparamter der Helix in der x-y-Projektion bestimmt werden. Mit Hilfe der gedrehten Straws kann zusätzlich eine z-Information gemessen werden, mit welcher die Steigung der Helix berechnet wird. Die Straw Tubes werden in 27 ebenen Schichten in einer hexagonalen Form um den MVD angeordnet. Sie werden über einen leichten Rahmen zusammengehalten und fixiert. Im Inneren befinden sich 8 Schichten paralleler Straws. Die 11 äußeren Reihen verlaufen ebenfalls parallel zur Strahlachse, wobei die 7 abschließenden Reihen fortlaufend weniger Straw Tubes enthalten und zur zylindrischen Form beitragen. Zwischen diesen parallelen Straws befinden sich 4 Doppel-Schichten gedrehter Röhrchen. Die Reihen sind paarweise entgegengesetzt mit einem Winkel von $\pm 2.9°$ relativ zur Strahlachse ausgerichtet. Die Anordnung der Schichten kann mit Abbildung 2.5 nachvollzogen werden. [6, S. 22 ff.]

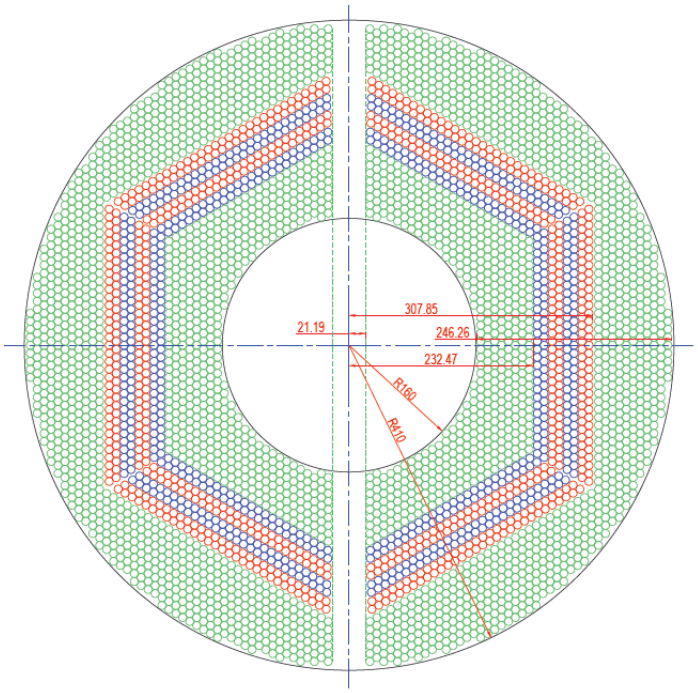


Abbildung 2.5: Anordnung der zur Strahlachse parallelen (grün) und gedrehten Schichten (blau/rot) von Straw Tubes im Querschnitt. [6, S. 27]

2.3.3 Die Straw Tubes

Die Straw Tubes besitzen einen Durchmesser von 10 mm und sind 1500 mm lang. Einige der Tubes sind kürzer, um das Volumen lückenlos ausfüllen zu können. Die Straw Tubes sind mit Auslese- und Montage-Komponenten in Abbildung 2.6 zu sehen. Straw Tubes bestehen aus einer Mylar-Folie der Dicke 27 µm, die von innen und außen mit Aluminium beschichtet ist. Die Straws sind mit einem Gas-Gemisch (Ar/CO_2) gefüllt und werden durch einen Überdruck von 1 bar stabilisiert. Die innere Schicht der Röhrchen ist leitend und fungiert als Kathode. Entlang der Zylinderachse befindet sich ein Draht als Anode. Zwischen dem Draht und der Hülle wirkt ein elektrisches Feld. Passiert ein geladenes Teilchen den gasgefüllten Innenraum der Straws, kommt es zur Ionisation des Gases. Es entstehen positiv geladene Ionen und Elektronen. Die positiv geladenen Ionen driften zum Rand der Tubes. Die Elektronen bewegen sich dagegen in die Mitte zum Draht und lösen ein Signal aus, das am Ende des Drahtes in Form eines Ladungspulses gemessen wird. Der Draht unterliegt einer Spannung im kV-Bereich und hat einen Durchmesser von wenigen µm. Dadurch ist das elektrische Feld in der Nähe des Drahtes so stark, dass es weitere Gas-Ionisationen hervorruft. In Abhängigkeit von der Höhe der Spannung und der Eigenschaften des Gases kann eine Verstärkung um das 10^4- bis 10^5-

fache des ursprünglichen Signals erreicht werden, was ein Auslesen des Signals möglich macht. Der spezifische Energieverlust eines geladenen Teilchens wird aus der Anzahl der ionisierten Elektronen pro Spur-Länge des Straw-Signals abgeleitet. Dieser Wert kann zur Bestimmung der Teilchenart genutzt werden.

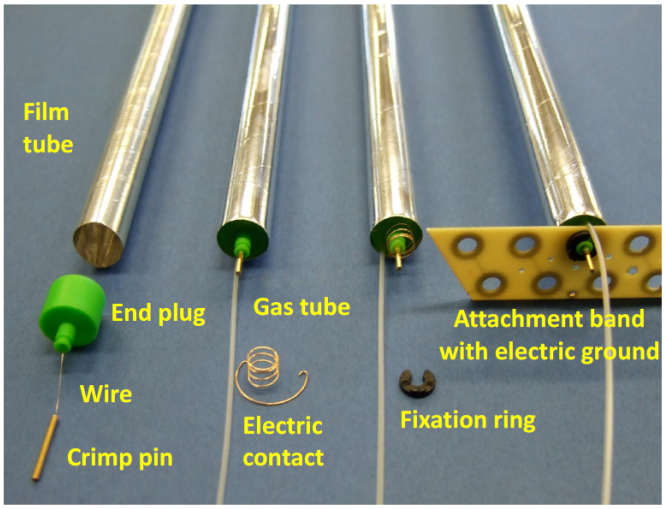


Abbildung 2.6: Komponenten der Straw Tubes. [6, S. 23]

Um herauszufinden, an welcher Stelle das Teilchen die Röhrchen passiert hat, wird die Drift-Zeit des ersten am Draht eintreffenden Elektrons gemessen. Damit können sogenannte *Isochrone* bestimmt werden, die alle Punkte beinhalten, von denen die gleiche Drift-Zeit ausgeht. Sie werden durch einen Zylinder mit dem Draht als Achse beschrieben. Zur Bestimmung der Isochrone wird ein Bezugszeitpunkt, die t_0-Zeit, benötigt. Die Flugzeiten der Teilchen sind im Vergleich zu den Drift-Zeiten vernachlässigbar. Das bedeutet, dass für alle Signale eines physikalischen Ereignisses von der gleichen t_0-Zeit ausgegangen werden kann. Die t_0-Zeit kann aus den Signalen zeitgebender Detektoren oder mit aufwendigeren Verfahren auch ausschließlich aus den STT-Signalen ermittelt werden.

Die Beziehung zwischen Drift-Zeit und Isochronen ist u. a vom Gas, elektrischen und magnetischen Feld abhängig. Es erfolgte eine Kalibrierung mit Hilfe von Referenz-Flugbahnen, von denen Drift-Zeit und genauer Verlauf bekannt waren. Um die Flugbahnen im STT zu rekonstruieren, müssen die Signale einer Folge von mehreren Straws zusammengeführt und der am besten passende Spur-Fit ermittelt werden. Um eine Information in z-Richtung zu erhalten, müssen Signale gedrehter Straws hinzugezogen werden. [6, S. 22 f.]

3 CPU-Version des Spurfinders

Dieser Arbeit liegt ein Spurfindealgorithmus zu Grunde, der im Rahmen der Bachelorarbeit „Entwicklung eines schnellen Algorithmus zur Suche von Teilchenspuren im Straw Tube Tracker des $\overline{\text{P}}$ANDA-Detektors“ [1] entwickelt wurde. Dabei handelt es sich um einen Spurfinder, der Signale des STTs bezüglich der Flugbahnen gruppiert. Dieser Spurfinder ist in das Simulations-Framework *PandaRoot* integriert, welches vorab in Abschnitt 3.1 eingeführt wird. Das Verfahren zur Spurfindung wurde weiterentwickelt und optimiert, sodass nur noch Kern-Bestandteile des ursprünglichen Verfahrens erhalten geblieben sind. Die neue Version des Spurfinders wird in Abschnitt 3.2 vorgestellt. Anschließend wird auf die zur Zeitmessung eingesetzten Mittel und das Laufzeitverhalten der einzelnen Verfahrensschritte eingegangen (Abschnitt 3.3).

3.1 PandaRoot

Die Anlage von FAIR und der $\overline{\text{P}}$ANDA-Detektor befinden sich derzeit noch in der Entwicklung. Um die Detektoren zu verbessern und geeignete Verfahren zur Auswertung der Daten zu erarbeiten, werden die physikalischen Vorgänge mit Hilfe von Software simuliert. Die Simulationen werden auch im Betrieb des Experimentes weiter genutzt um z. B. interessante Reaktionen auf ihr Verhalten im Detektor zu untersuchen. Für die Experimente an $\overline{\text{P}}$ANDA erfolgt dies mit dem Framework *PandaRoot* [10]. Dieses dient speziell der Simulation und Analyse der $\overline{\text{P}}$ANDA-Experimente an FAIR und baut auf dem Framework FairRoot [11] auf. FairRoot wurde im Rahmen des FAIR-Projektes entwickelt, um ein einheitliches Programmiergerüst für die Experimente von FAIR zu schaffen. FairRoot nutzt ROOT [12] als Basis-Framework.

3.1.1 ROOT

ROOT ist ein objektorientiertes Datenanalyse-Framework, das zur Auswertung und für Simulationen von Experimenten im Bereich der Hochenergiephysik eingesetzt wird. Es wurde am CERN (Europäische Organisation für Kernforschung) entwickelt und in C++ implementiert. Die Steuerung von ROOT erfolgt über eine modifizierte Version von C++, die mit einem speziellen Interpreter namens CINT interpretiert werden kann. Dies ermöglicht ein schnelles und direktes Eingreifen in den Simulations- und Analyseprozess, da die Zeit

zum Kompilieren und Linken reduziert wird. Die Prozesse in ROOT können interaktiv über ein Command Line Interface oder in Form von Makros gesteuert werden. Für die Simulation und Analyse von Experimenten wird in der Regel der Batchbetrieb mit Makros vorgezogen. Zur schnellen Verarbeitung großer Datenmengen wurden eigene Formate zur Eingabe, Ausgabe und Speicherung von Daten entwickelt. ROOT bietet außerdem Werkzeuge zum Erstellen von Histogrammen, zur Ausgleichsrechnung, zur Visualisierung mit 2D/3D-Grafiken u. v. m. [13]

3.1.2 Simulationen mit PandaRoot

„Um das Verhalten eines Systems zu simulieren, können analytische oder stochastische Mittel eingesetzt werden. Erstere führen zu komplexen Bewegungs-Gleichungen, die gelöst werden müssen. Dies findet z. B. bei der Berechnung des Bahnverlaufs eines geladenen Teilchens im magnetischen Feld Anwendung. Um die Wechselwirkung von Teilchen mit Materie zu simulieren, wird der stochastische Ansatz gewählt. Es gibt eine große Anzahl möglicher physikalischer Vorgänge, die ablaufen können. Aus diesem Grund wird so ein Prozess mit Pseudozufallszahlen simuliert. Ein derartiges Vorgehen zählt zu den Monte-Carlo-Methoden. ROOT stellt eine Schnittstelle (Virtual Monte Carlo) zur Verfügung, mit der Monte-Carlo-Programme wie Geant3, Geant4 und Fluka genutzt werden können. Die Monte-Carlo-Simulation erzeugt u. a. für jedes Teilchen einen Weg durch die Materie/Detektorkomponenten und passt die Eigenschaften des Teilchens (z. B. Energie) an. Dabei werden Schritt für Schritt probabilistisch Wechselwirkungen und physikalische Vorgänge miteinbezogen. Nach dieser Stufe schließt sich die Simulation der Auslese-Elektronik an, die sogenannte Digitalisierung.“ [1, S. 16 f.]

PandaRoot wurde so entworfen, dass es sowohl mit simulierten Daten als auch später mit realen Messwerten arbeiten kann. Es erlaubt die Simulation von Events und den entsprechenden Detektor-Signalen. Unter einem Event wird ein Ereignis des $\overline{\text{P}}$ANDA-Experimentes verstanden. Dazu zählt das Wechselwirken eines Teilchens des Strahls mit einem Teilchen des Targets und die daraus resultierenden Vorgänge und Sekundärteilchen. Es sind Verfahren in das Framework eingebunden, die aus den simulierten Detektor-Signalen die Teilchenflugbahnen rekonstruieren, die Teilchen identifizieren und analysieren. Die Simulations- und Rekonstruktionsschritte lassen sich wie folgt einteilen [10]:

Generieren von Events Mit Hilfe von *Event-Generatoren* werden physikalische Vorgänge definiert bzw. simuliert. Sie erzeugen Daten mit Informationen über die entstandenen Teilchen (Energie, Impuls, ...). Um den verschiedenen physikalischen Aspekten des Experiments gerecht zu werden, wurden eine Vielzahl von Event-Generatoren implementiert. Zu den wichtigsten zählen folgende [14, S. 4]:

- Mit Hilfe des *Box-Generators* können Teilchen mit Gleichverteilungen innerhalb eines definierten Bereiches (z. B. Winkel- oder Impulsbereich) erzeugt werden.
- *EvtGen* ist ein Generator mit dem komplexe Zerfallsketten definiert und simuliert werden können.
- Der *DPM-Generator* erlaubt Simulationen von Antiproton-Proton-Kollisionen unter Einsatz des *dualen Partonmodells*. Es können Untergrund-Ereignisse erzeugt und die Auslastung der Detektoren und Teilchenraten untersucht werden.

Transport durch den Detektor Die zuvor simulierten Daten dienen als Eingabe für das sogenannte *Transport-Modell*. Dieses überträgt die Informationen über die Teilchen auf die Detektorgeometrie und benutzt eines der Monte-Carlo-Programme. Die Flugbahnen der Teilchen werden berechnet und es wird ermittelt, an welchen Punkten die Bestandteile der einzelnen Detektorkomponenten passiert werden. Dabei werden Wechselwirkungen wie z. B. mit dem magnetischen Feldern oder den Detektormaterialien und auch Teilchenzerfälle berücksichtigt. Dieser Schritt setzt eine detaillierte Definition des gesamten $\overline{\text{P}}$ANDA-Detektors voraus. Die dafür benötigten Klassen sind bereits implementiert. Die vom Monte-Carlo-Programm erzeugten Daten werden in einer ROOT-Datei abgespeichert.

Digitalisierung Dies ist der letzte Schritt der Event-Simulation. Aus den berechneten Monte-Carlo-Punkten werden die Signale erstellt, die von den Detektoren wirklich ausgelöst werden würden. Signalverzögerungen und Messungenauigkeiten werden dabei einbezogen. Zudem ist es möglich, dass manche Punkte, die berechnet worden sind, gar nicht in ein Signal umgewandelt werden, da z. B. ein Schwellenwert nicht erreicht wurde. Durch die Digitalisierung werden die simulierten physikalischen Daten in Messwerte umgewandelt, die der Detektor bei einem realen Experiment liefern würde.

Rekonstruktion Das Ziel der Rekonstruktion ist es, aus den digitalisierten Daten möglichst alle Informationen über die ursprünglichen physikalischen Events zu gewinnen. Dazu müssen die Detekorinformationen zunächst in physikalische Größen, wie z. B. 3D-Punkte, Energie und Zeit, überführt werden. Anschließend werden die Signale verschiedenartiger Detektoren zusammengeführt. Zur Rekonstruktion der Flugbahnen wird die riesige Menge an Signalen (Hits) aufgetrennt und zu den potenziellen Flugbahnen (Tracks) gruppiert. Die Gruppen enthalten dann Hits, die von der gleichen Teilchenspur stammen könnten (Track-Kandidaten). Diesen Prozess der Gruppierung bezeichnet man als *Trackfinding*. Zur genauen Beschreibung der Flugbahnen muss die Bahn gefunden werden, die

alle Hits am besten approximiert. Dafür werden Verfahren zum *Trackfitting* benötigt. Diese berechnen die genauen Parameter der gefundenen Spuren. Es wird zwischen lokalen und globalen Tracks unterschieden. Die lokalen Tracks beschreiben nur einen Teil der gesamten Flugbahn eines Teilchens innerhalb eines Sub-Detektors des $\overline{\text{P}}$ANDA-Detektors. Die Informationen der Sub-Detektoren müssen zu einem globalen Track zusammengeführt werden. Es schließt sich die Teilchenidentifikation an, bei der den Flugbahnen eine Teilchenart zugewiesen wird. Auf diese Weise kann das Event vollständig rekonstruiert werden.

Analyse Mit PandaRoot und zusätzlichen Software-Paketen können die rekonstruierten Teilchen und Events bezüglich bestimmter Eigenschaften genauer untersucht werden. Es wird ermittelt, aus welchen primären Teilchenzusammenstößen die rekonstruierten Flugbahnen der gemessenen Sekundär-Teilchen hervorgegangen sind.

In Abbildung 3.1 ist die Abfolge der Schritte dargestellt. Die Aufteilung dieser Schritte in verschiedene Makros ist eine steuerbare Option. Es ist auch möglich alle Schritte in einem Makro auszuführen und nur das Ergebnis zu speichern. Werden jedoch die Daten für die Simulation, Digitalisierung, usw. nach dem entsprechenden Schritt gespeichert, erlaubt dies eine Kapselung der Daten. Die Daten können z. B. zum Testen alternativer Verfahren verwendet werden. Im Regelfall werden die Daten und Makros wie in Abbildung 3.1 separiert.

3.1.3 Qualitätskontrolle

Um die Qualität der zur Rekonstruktion eingesetzten Verfahren zu untersuchen, werden die rekonstruierten Spuren mit den simulierten Flugbahnen verglichen, die in den Monte-Carlo-Daten vermerkt sind. Zu diesem Zweck erhält jedes Datenobjekt, das die in Abbildung 3.1 dargestellten Schritte durchläuft, einen oder mehrere *FairLinks*, die eindeutig auf die Daten zeigen, die zur Erzeugung des Datenobjektes verwendet wurden.

„Ein FairLink enthält Informationen über einen *Zweig* und die *Position* in diesem Zweig. Über die Angabe des Zweiges in Form einer Branch-ID kann die Datenstruktur ermittelt werden, die einem Objekt (in der Simulation/Rekonstruktion) zu Grunde liegt. Die verschiedenen Objekte werden in Form von `TClonesArrays` abgespeichert. Mit der Position wird der genaue Eintrag in dieser Datenstruktur identifiziert. [...] Zum Beispiel werden aus einem vom Monte-Carlo-Programm berechneten MCTrack, der den physikalischen Track repräsentiert, MCPoints erzeugt. Dies sind die Punkte, an denen die MCTracks die verschiedenen Detektorkomponenten passieren. Basierend auf einem MC-Point werden Detektorsignale simuliert und digitalisiert, aus welchen wiederum Hits erzeugt werden.“[1, S. 89]

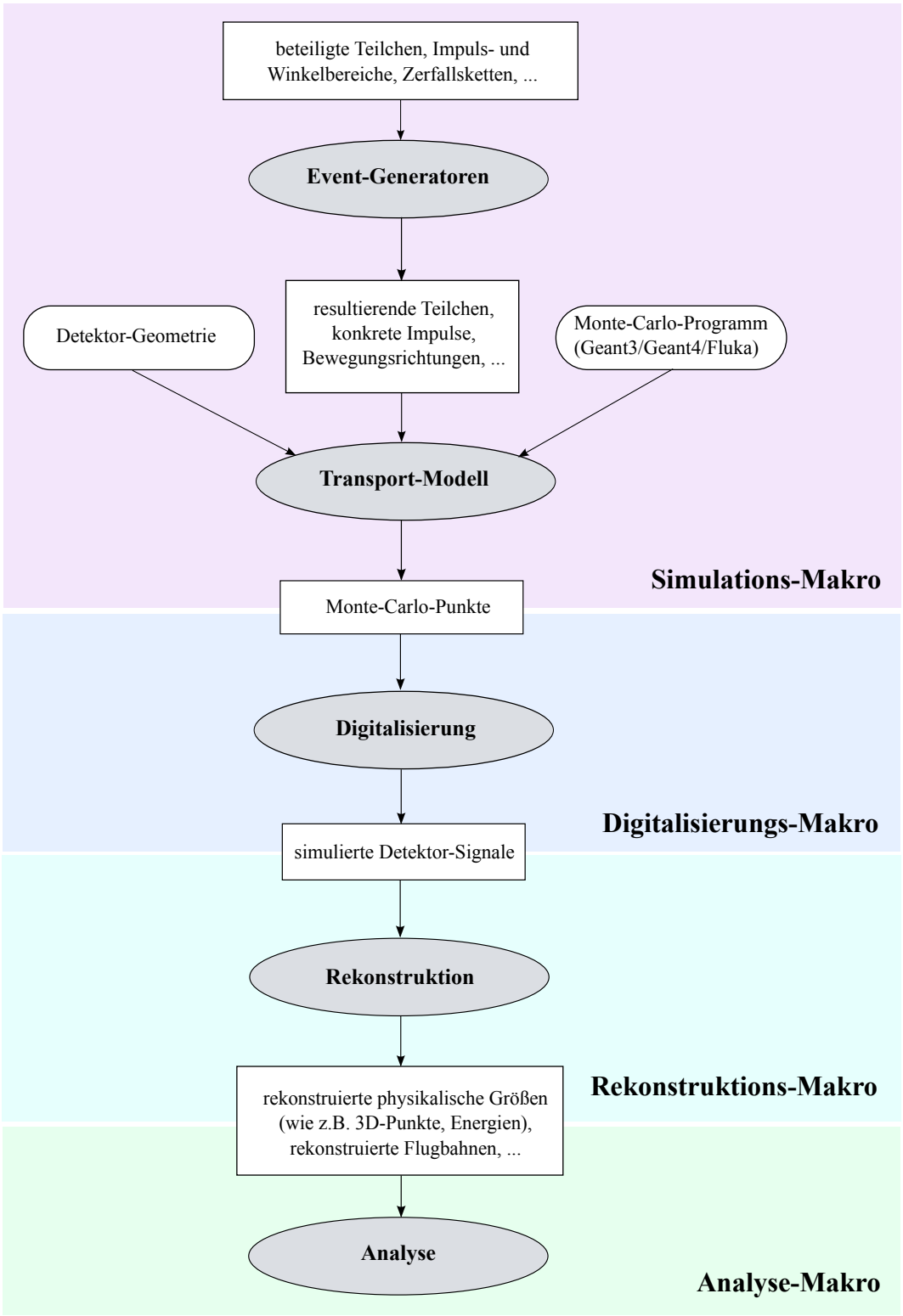


Abbildung 3.1: Darstellung der üblichen Abfolge von Makros zur Simulation, Rekonstruktion und Analyse innerhalb des PandaRoot-Frameworks. Das Spurfinde-Verfahren wird im Rekonstruktions-Makro angestoßen.

Werden die Hits z. B. vom Spurfinder gruppiert, so enthält diese Gruppe von Hits FairLinks auf die ursprünglichen Hits. Diese verweisen wiederum auf die ursprünglichen MCPoints und über diese kann der zugrundeliegende MCTrack ermittelt werden.

3.1.4 Visualisierung

Zur Überprüfung der Module und der Güte der Trackfinding- und Trackfitting-Verfahren werden die Ergebnisse mit *Eve* dargestellt. „Eve steht für *Event Visualization Environment* und ist ein Framework von ROOT, mit dem die Events dargestellt werden können. Es bietet u. a. Klassen zur Darstellung von Baumstrukturen, von Listen und der verschiedenen Geometrien für die Detektoren. Die Flugbahnen der Teilchen können in einer 3D-Ansicht aus verschiedenen Blickwinkeln betrachtet werden. Für die 3D-Ansicht werden automatisch 2D-Projektionen erzeugt. Mit Eve können die Monte-Carlo-Punkte, Hits und Teilchenspuren für die verschiedenen Detektoren dargestellt werden.“ [1, S. 20]

Abbildung 3.2 zeigt beispielhaft die Darstellung eines Events innerhalb des STTs mit Eve. Die dargestellten Informationen wurden auf die Teilchenflugbahnen reduziert, die Signale im STT ausgelöst haben. Zu sehen ist ein Querschnitt des STTs mit dem primären Teilchenzusammenstoß in der Mitte (blau), den Teilchenflugbahnen (grau), den Monte-Carlo-Punkten (pink) und den Straw Tubes, die einen Hit ausgelöst haben (hell hervorgehoben).

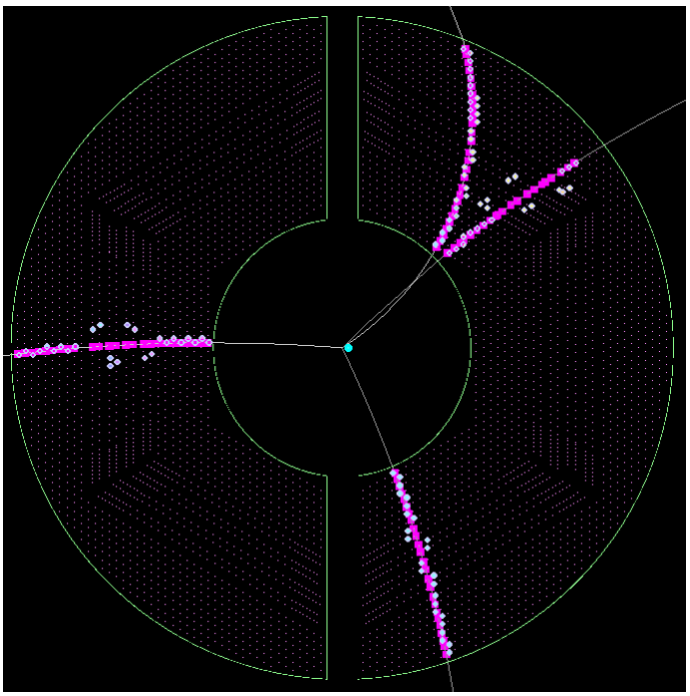


Abbildung 3.2: Visualisierung eines Events im STT mit Eve.

3.2 Das Spurfinde-Verfahren

Bei dem implementierten Verfahren handelt es sich um einen Spurfinder, der die STT-Hits eines Events bezüglich potenzieller Flugbahnen gruppiert. Als Eingabe dienen alle STT-Signale eines simulierten Events. Das Verfahren erstellt Gruppen von STT-Hits, die nur Hits enthalten, die einer einzigen Teilchenflugbahn zuzuordnen sind. Der Schritt des Trackfindings ist ein wichtige Stufe bei der Rekonstruktion der Ereignisse. Die riesige Menge von Hits wird aufgespalten und es werden erste Ansatzpunkte für das Rekonstruieren der Flugbahnen geschaffen.

Zur Realisierung des Spurfinders wurden zwei grundlegende Verfahren entwickelt: der *TrackletGenerator* und der *HitCorrector*. Mit dem TrackletGenerator werden erste Tracklets generiert und diese anschließend kombiniert. Tracklets sind Bruchstücke von Tracks in Form einer ersten Gruppierung ihrer STT-Hits. Ein Track kann durch das Verfahren in mehrere Tracklets zerlegt werden. Daher werden die Tracklets anschließend kombiniert, um den vollständigen Track zu erhalten. Der HitCorrector dient zur Berechnung einer genauen Postion des Hits einer Tube durch Einbeziehen der Isochron-Information.

Es gibt zwei Ausführungsformen des Spurfinders. Stehen Isochron-Informationen zur Verfügung, wird zuerst der HitCorrector ausgeführt und der TrackletGenerator kann anschließend mit genaueren Hit-Informationen rechnen. Gibt es keine Isochron-Informationen entfällt dieser Vorteil.

Bei dem TrackletGenerator handelt es sich um eine Weiterentwicklung des Spurfinders der Arbeit „Entwicklung eines schnellen Algorithmus zur Suche von Teilchenspuren im Straw Tube Tracker des $\overline{\text{P}}$ANDA-Detektors“ [1]. Vor allem das Kombinieren der Tracklets wurde stark abgewandelt. Der HitCorrector ist eine neue Erweiterung zur Verbesserung des Spurfinde-Verfahrens. Im Rahmen dieser Arbeit wurden Teile des TrackletGenerators parallelisiert, weshalb nur dieser Teil in den folgenden Abschnitten ausführlich beschrieben wird. Eine Parallelisierung des HitCorrectors wurde bisher nicht umgesetzt.

3.2.1 Der TrackletGenerator

Die Ablauf des TrackletGenerators lässt sich in drei wesentliche Schritte einteilen. Als erstes werden primäre Tracklets mit einem Verfahren generiert, das an einen *zellulären Automaten* angelehnt ist. Mit Hilfe von Nachbarschaftsbeziehungen werden erste Gruppen von Hits (primäre Tracklets) ermittelt, die zu dem gleichen Track gehören. An Kreuzungsbereichen von Tracks treten Ambiguitäten auf. Abbildung 3.3 zeigt ein Beispiel für eindeutige Hits, entstehende Ambiguitäten und die resultierenden primären Tracklets. Hits im Kreuzungsbereich können den Tracks nicht eindeutig zugeordnet werden. Sie werden allerdings dazu genutzt, um Informationen über mögliche Kombinationen von Tracklets zu erhalten. Auch hier wird eine Art zellulärer Automat verwen-

det. Die möglichen Kombinationen werden mit einem Kreis-Fit bewertet und akzeptiert oder verworfen. Anschließend werden Hits, die in keinem Tracklet und in keiner Kombination von Tracklets enthalten sind, über Abstandsberechnung der besten Gruppierung zugeordnet (wenn möglich). Es folgt eine genaue Beschreibung dieser drei Stufen.

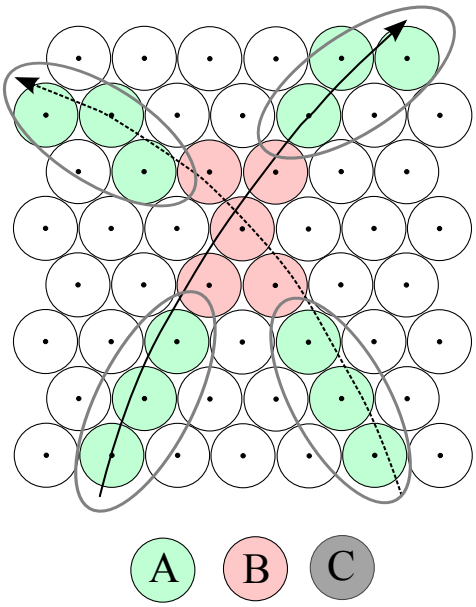


Abbildung 3.3: Skizzierung des grundlegenden Idee des TrackletGenerators. Zu sehen ist ein Querschnitt durch eine Menge von Straw Tubes. Zwei sich in der x-y-Projektion kreuzende Tracks passieren den STT und lösen Signale in den entsprechenden Straw Tubes aus. A: Hits, die eindeutig zugeordnet werden können; B: Hits mit Ambiguitäten; C: resultierende Tracklets, die anschließend kombiniert werden.

1. Tracklet-Generieren

Das Generieren der ersten Teilstücke eines Tracks (der primären Tracklets) orientiert sich an einem zellulären Automaten. Ein zellulärer Automat dient zur abstrakten Modellierung dynamischer Systeme, die sich aus einer Menge von Zellen zusammensetzen. Jeden Zelle besitzt einen diskreten Zustand und eine bestimmte Anzahl angrenzender Nachbarn. Es werden Übergangsregeln für die möglichen Zustände definiert. Dabei ergibt sich der neue Zustand einer betrachteten Zelle ausschließlich aus ihrem eigenem aktuellen Zustand und den Zuständen der Nachbarn. Die Änderung der Zustände erfolgt simultan, d. h. sie erfolgt gleichzeitig für alle Zellen. Weiterführende Informationen zu zellulären Automaten und diesem Verfahrensschritt sind in [1] zu finden.

Jede Straw Tube fungiert als eine **Zelle**. Jede Zelle hat zwischen 2 und 22 benachbarte Zellen. Für die parallel verlaufenden Tubes liegt die Anzahl zwischen 2 und 6, für die gedrehten wächst sie bis auf 22 an. Die Zellen besitzen einen initialen eindeutigen **Zustand**, bei dem es sich um die Tube-ID handelt. Die Tube-ID ist eine fortlaufende Nummer, die sich aus der Durchnummerierung der Straw Tubes des STTs von innen nach außen ergibt. Der Zustand einer Zelle ist stellvertretend für eine Track-ID, die den Track identifiziert zu dem die Zelle gehört. Vor Ausführung des zellulären Automaten ist die Track-ID für alle involvierten Zellen verschieden. Nach der Beendigung des Verfahrens wurde für bestimmte Gruppen von Zellen die gleiche Track-ID bzw. der gleiche Zustand generiert. Das bedeutet, dass sie zum gleichen physikalischen Track gehören. Nur Tubes, die einen Hit signalisiert haben, können zu einem Track gehören. Sie werden als „aktive Zellen“ bezeichnet. Nur diese aktiven Zellen besitzen einen Zustand und werden in die Aktualisierungsvorgänge des zellulären Automaten einbezogen.

Bei der **Anpassung der Zustände** werden die betrachteten Zellen noch weiter gefiltert. Es werden nur aktive Zellen einbezogen, die ein oder zwei aktive Nachbarn haben (eindeutige Zellen). Bei zwei aktiven Nachbarn ist davon auszugehen, dass die betrachtete Zelle ein Verbindungsstück innerhalb eines Tracks darstellt. Hat sie nur einen aktiven Nachbarn, handelt es sich um einen Anfangs-/Endpunkt eines Tracklets. Hat eine Zelle keinen aktiven Nachbarn, kann nichts gruppiert werden. Aktive Zellen mit drei und mehr aktiven Nachbarn (mehrdeutige Zellen) können dem richtigen Tracklet nicht eindeutig zugeordnet werden. Sie liegen z. B. in Kreuzungs- oder Verzweigungsbereichen von Tracks, die eine hohe Nachbarschaftsdichte aktiver Zellen aufweisen. Durch den Ausschluss dieser Ambiguitäten werden nur eindeutig zusammengehörige Zellen gruppiert. Die **Übergangsregel** für die gefilterten eindeutigen Zellen lautet:

> „Hat eine Zelle einen oder zwei aktive Nachbarn, ermittele das Minimum aus den Track-IDs *gleichartiger* Nachbarn und der eigenen. Setze die Track-ID der Zelle auf dieses Minimum.“ [1, S. 27]

Gleichartige Nachbarzellen sind diejenigen, die auch nur ein oder zwei aktive Nachbarn besitzen. Auf diese Weise nehmen die Zustände der Zellen, die eindeutig gruppiert werden können, nach und nach die minimale Track-ID an. Der zelluläre Automat stoppt, wenn sich in zwei aufeinanderfolgenden Iterationsschritte kein Zustand mehr ändert.

Die Funktionsweise des zellulären Automaten für zwei sich kreuzende Tracks kann mit Abbildung 3.4 nachvollzogen werden. Es ist ein modellhafter Ausschnitt des STTs dargestellt. Die zu messenden Teilchen durchqueren die Tubes von unten beginnend. Der Tube-Nummerierung entsprechend wurden die

Start-Zustände so vergeben, dass sie von unten nach oben anwachsen. Der zelluläre Automat liefert nur eine vollständige Track-Gruppe von Zellen, wenn der Track keine Ambiguitäten wie Kreuzungen oder Verzweigungen aufweist. Da diese Eigenschaft nur ein Bruchteil der zu erwartenden Tracks erfüllt, schließen sich weitere Schritte im Sinne der Spurfindung an.

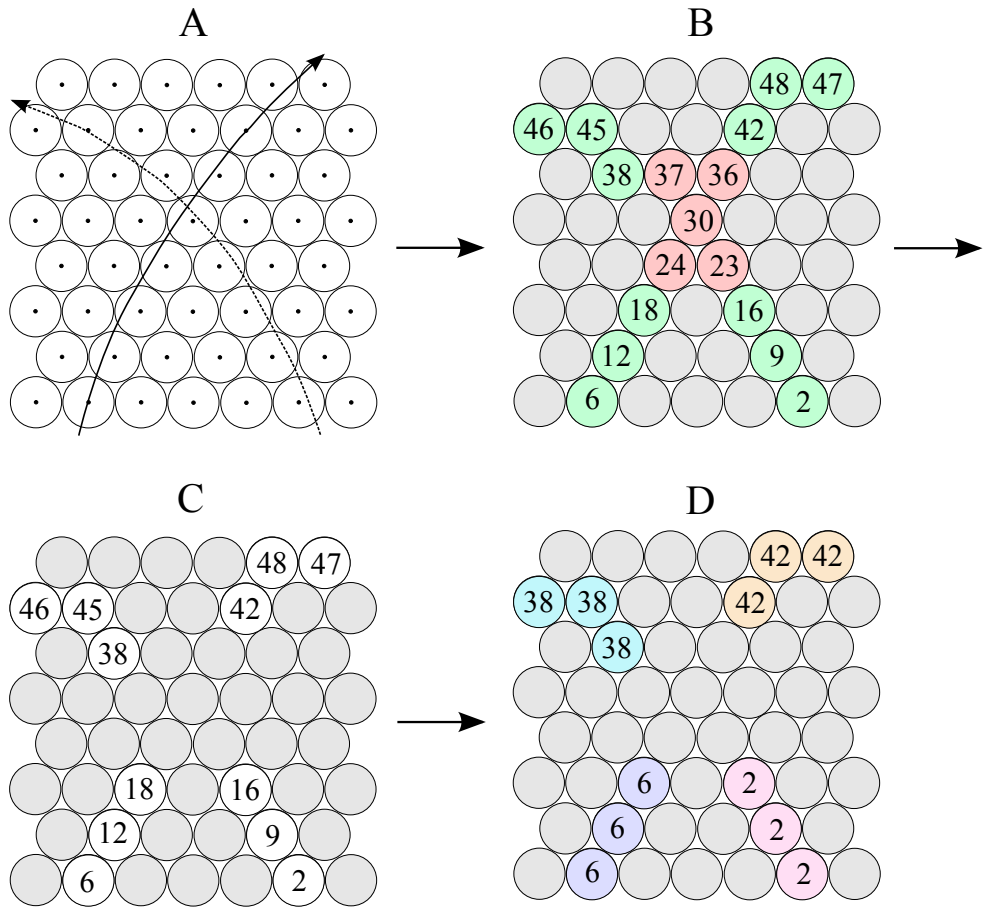


Abbildung 3.4: Funktionsweise des zellulären Automaten für eindeutige Zellen. A: zwei sich kreuzende Tracks im Querschnitt des STTs, B: nur aktive Zellen werden betrachtet, Unterscheidung in eindeutige (grün) und mehrdeutige Zellen (rot), C: Ausblenden von Ambiguitäten, D: die Anwendung des zellulären Automaten ergibt vier Tracklets; zugrundeliegende Grafik aus [1, S. 28].

2. Tracklet-Kombinieren

Hinterlassen Tracks Hits in Tubes, die nicht eindeutig einem Tracklet zugewiesen werden können, sind diese Tracks unvollständig. Zellen mit mehr als zwei aktiven Nachbarn wurden beim Generieren der Zustände ausgeblendet und keinem Tracklet zugeordnet. Dies hinterlässt Lücken und führt dazu, dass

ein Track in mehrere Tracklets zerfällt. Das Ziel des Kombinierens ist es, diese Tracklets zusammenzufügen, sodass ein vollständiger Track entsteht.

Zu diesem Zweck werden folgende drei Schritte ausgeführt:

1. Generieren von Zuständen für mehrdeutige Zellen (Multi-Zustände) mit Hilfe eines zellulären Automaten. Die Zustände speichern Informationen über Kombinationsmöglichkeiten der Tracklets.

2. Berechnen aller Kombinationsmöglichkeiten von Tracklets mit einem rekursiven Verfahren.

3. Herausfiltern der physikalisch sinnvollen Kombinationen durch Approximation einer Kreisbahn.

Die ersten beiden Schritte unterscheiden sich deutlich von der in der Bachelorarbeit [1] entwickelten Version. Die dritte Stufe ist erhalten geblieben.

In dieser Stufe des zellulären Automaten werden alle **Zellen** mit mehr als zwei aktiven Nachbarn involviert. Es ist nicht möglich diesen Zellen eine eindeutige Track-ID zuzuweisen, da sie zu mehreren Tracklets gehören können. Daher werden alle Track-IDs der potenziellen Tracklets, zu denen die mehrdeutigen Zellen gehören könnten, in einem **Multi-Zustand** gespeichert. Beim Generieren der Multi-Zustände sind nur aktive Zellen involviert, die mehr als zwei aktive Nachbarn besitzen oder an diese angrenzen. Das heißt, es werden auch die Anfangs-/End-Zellen eines Tracklets einbezogen, die an Bereiche mit dichten Nachbarschaften aktiver Zellen grenzen. Eine Änderung des Zustandes erfolgt allerdings nur für mehrdeutige Zellen. Der Multi-Zustand dieser Zellen stellt die Möglichkeiten von Track-IDs der Tracklets dar, welchen sie zugeordnet werden könnten. Diese Menge an Möglichkeiten ist zu Beginn leer und füllt sich mit wiederholter Anwendung der **Übergangsregel**:

> Hat eine Zelle mehr als zwei aktive Nachbarn, ersetze den eigenen Multi-Zustand durch eine Kopie aller Track-IDs der Zustände der angrenzenden aktiven Nachbarn.

Die Regel wird so oft angewendet bis sich kein Multi-Zustand mehr ändert. Nach Abschluss der Iterationen besitzen benachbarte mehrdeutige Zellen die gleiche Anzahl von Einträgen in ihren Multi-Zuständen. Wichtig ist hierbei, dass nicht nur gleichartige Nachbarn bei der Anpassung der Zustände betrachtet werden, sondern auch eindeutige aktive Zellen. Die eindeutigen Zellen sind die Startpunkte für die ersten Kopier-Vorgänge der Zustände. In Abbildung 3.5 sind die Iterationsschritte für mehrdeutige Zellen im Kreuzungsbereich von zwei Tracks dargestellt. Nach der Anpassung der Zustände für eindeutige und mehrdeutige Zellen liegen Tracklets und Informationen zur Kombination dieser Tracklets vor.

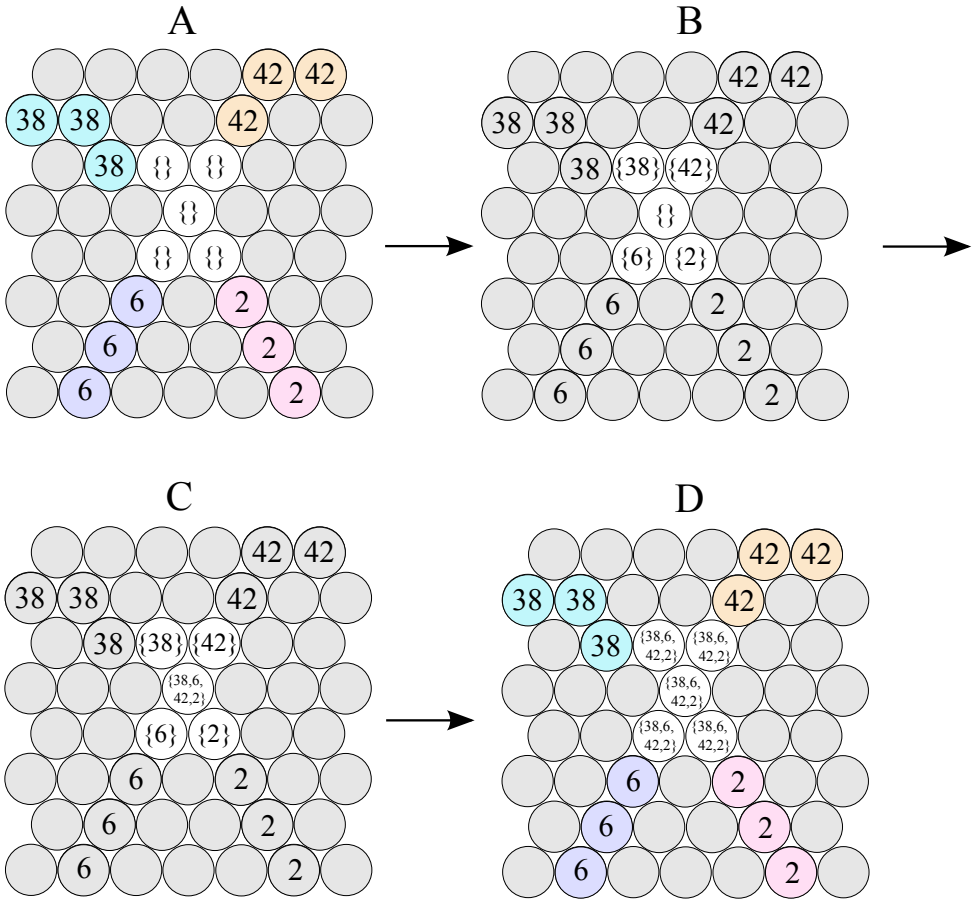


Abbildung 3.5: Funktionsweise des zellulären Automaten für mehrdeutige Zellen. A: Ergebnis des zellulären Automaten für eindeutige Zellen mit leerer Menge als Startzustand, B: Zustände eindeutiger Zellen werden kopiert, C: Zustände mehrdeutiger Zellen werden kopiert, D: endgültige Zustände.

Mit Hilfe der Multi-Zustände können die an mehrdeutige Zellen angrenzende Tracklets kombiniert werden. Für die primären Tracklets wird überprüft, ob sie an eine mehrdeutige Zelle grenzen. Der Zustand dieser angrenzenden Zelle wird ausgelesen und es werden **Kombinationen** mit den Tracklets gebildet, deren Track-ID in der Zustandsmenge enthalten ist. Für das Tracklet mit der Track-ID 6 in Abbildung 3.5 (Stufe D) werden z. B. Kombinationen mit den Tracklets 38, 42 und 2 gebildet. Alle möglichen Kombinationen werden vorerst in Form einer Menge von Track-IDs gespeichert. Ob diese Kombinationen physikalisch sinnvoll sind, wird erst in einem anschließenden Schritt überprüft.

Zum Kombinieren der Tracklets wird ein rekursives Verfahren eingesetzt. Wird für ein Tracklet bzw. eine Kombination von Tracklets eine mehrdeutige Zelle als Nachbar gefunden, wird dessen Zustand ausgelesen. Es werden alle möglichen Kombinationspaare aufgestellt. Für jedes gebildete Paar beginnt

der Kombinationsprozess erneut. Um den Kombinationsaufwand so gering wie möglich zu halten, werden folgende Einschränkungen getroffen:

- Es wird nur mit den Tracklets gestartet, die in der ersten inneren Reihe des STTs beginnen.
- Grenzt keine mehrdeutige Zelle an das Tracklet, ist das Kombinieren beendet.
- Hinzuzufügende Track-IDs dürfen in der bisherigen Kombination noch nicht enthalten sein.
- Rückläufige Kombinationen werden vermieden, d. h. der Weg durch einen Bereich mit mehrdeutigen Zellen wird im Kombinationsprozess nur einmalig durchquert.

Für das Beispiel in Abbildung 3.5 bedeutet das Folgendes: Wurde Tracklet 6 mit Tracklet 38 kombiniert, wird für Tracklet 38 nach einem weiteren mehrdeutigen Nachbar gesucht. Für weitere Kombinationsmöglichkeiten wird der zwischen Tracklet 6 und 38 liegende Kreuzungsbereich nicht erneut durchlaufen/überprüft. Durch das Verhindern rückläufiger Kombinationen werden auf diese Weise viele nicht sinnvolle Kombinationen ausgeschlossen. Auch das Entstehen von potenziellen Schleifen bei der Suche nach Kombinationen wird dadurch unterdrückt.

Wurden alle möglichen Kombinationen von Track-IDs gebildet, wird ihre Tauglichkeit untersucht. Es erfolgt eine **Filterung der Kombinationen**. Dazu werden die Spuren zu den in Kombinationen gruppierten Hits berechnet. Die potentiellen Flugbahnen werden durch eine Kreisbahn approximiert, da in der x-y-Projektion aufgrund der wirkenden Lorentz-Kraft kreisförmige Spuren zu erwarten sind. Für jede Kombination von Tracklets wird die bestmögliche Kreisbahn zu den Hit-Positionen, die in der Kombination enthalten sind, berechnet. Wird der TrackletGenerator ohne den HitCorrector ausgeführt, handelt es sich bei den Hit-Positionen um die Mittelpunkte der Straw Tubes. Diese Annahme ist vereinfacht, denn in den meisten Fällen passieren die Teilchen die Straw Tubes nicht direkt in der Mitte durch den Draht.

Anschließend werden die berechneten Kreisbahnen bewertet. Liegt zwischen der Kreisbahn und einer Hit-Position ein Abstand vor, der größer als der Radius einer Straw Tube ist, dann wird die Kombination verworfen. Ein Abstand von dieser Größe ist physikalisch nicht möglich, da dann ein Signal in einer anderen Straw Tube ausgelöst worden sein müsste. Ist der Abstand zu groß, liegen die gruppierten Signale nicht auf einer Kreisbahn oder der Fit wurde aufgrund der ungenauen Hit-Positionen ungenügend berechnet. In beiden Fällen wird die Kombination verworfen.

Zur Approximation der Kreisbahnen wird der *Riemann-Fit* verwendet. Der Riemann-Fit überführt das quadratische Problem der Kreisapproximation im zweidimensionalen Raum in ein lineares Problem im Dreidimensionalen. Das Verfahren bietet eine schnelle explizite Lösung für das Problem und ist ein Bestandteil des PandaRoot-Frameworks. Genauere Informationen zu diesem Verfahren sind in [1, S. 30 ff.] zu finden.

Nach dem Abschluss dieser Stufe des Spurfinders liegen im Regelfall größere Tracklets als nach der ersten Stufe vor. Tracklets, die nicht kombiniert werden konnten, werden ebenfalls als Track-Kandidaten angesehen (zumal diese auch schon vollständige Tracks darstellen können). Sie müssen mindestens drei Hits umfassen, sodass eine Kreisbahn berechnet werden kann.

3. Zuordnung von Hits

Beim Generieren und Kombinieren der Tracklets wurden mehrdeutige Zellen von der Zuordnung ausgeschlossen, da diese nicht eindeutig vorgenommen werden konnte. Mit Hilfe des Riemann-Fits wurden Informationen über den Verlauf der Flugbahnen berechnet. Dies ermöglicht es über eine Abstandsberechnung bisher nicht eingruppierte Hits einem Tracklet zuzuordnen. Für jeden verbleibenden Hit wird der Abstand zu allen berechneten Kreisbahnen der Tracklets und ihrer Kombinationen in der gleichen Detektorhälfte berechnet. Es erfolgt eine Zuordnung zu dem Tracklet bzw. der Kombination, die den kleinsten euklidischen Abstand zum Hit besitzt. Dabei muss der Abstand kleiner gleich dem Radius einer Straw Tubes sein, da die Zuordnung sonst nicht sinnvoll ist. Als Hit-Position wird im Normalfall der Mittelpunkt der Straw Tube gewählt. Beim Einsatz des HitCorrectors werden die korrigierten Positionen verwendet. Zu den verbleibenden Hits zählen auch diejenigen von Tracklets mit weniger als drei Hits.

Hits gedrehter Straw Tubes

Eine gesonderte Rolle in diesem Verfahren nehmen die Signale gedrehter Straw Tubes ein. Das gesamte Verfahren arbeitet in einer 2D-Projektion des STTs, das heißt es wird ein Querschnitt durch den STT betrachtet. Die zellulären Automaten verwenden ein zweidimensionales Gitter. An dieser Stelle können die gedrehten Straw Tubes wie die parallen behandelt werden. Sie haben allerdings mehr Nachbarn, weshalb sie oft zu den mehrdeutigen Zellen zählen.

Zur Berechnung der Flugbahnen wird eine x-y-Projektion in Form einer Kreisbahn betrachtet. Es erfolgt eine Abstandsberechnung zu den Mittelpunkten der Straw Tubes. Bisher fehlt eine Projektion der Signale der gedrehten Straw Tubes in die x-y-Ebene. Für parallele Straw Tubes kann zur Vereinfachung die Position des Drahtes für diesen Zweck verwendet werden. Sie ist in z-Richtung überall gleich. Dies ist bei den gedrehten Straw Tubes nicht der

Fall. Demzufolge können die Hits gedrehter Straw Tubes bei der Berechnung des Riemann-Fits nicht einbezogen werden. Das bedeutet, dass ein Tracklet möglicherweise mehr als 3 Hits umfassen muss, damit eine Kreisbahn berechnet werden kann. Besteht ein Tracklet nur aus Hits von gedrehten Straw Tubes, kann eine Kombination mit diesem Tracklet nicht bewertet werden. Die Hits des Tracklets haben keinen Einfluss auf die Berechnung der Kreisbahn. Außerdem ist es nicht möglich verbleibende Hits von gedrehten Straw Tubes bereits berechneten Flugbahnnen zuzuordnen, da eine Abstandsberechnung nicht möglich ist.

3.2.2 Der HitCorrector

Der HitCorrector ist ein Verfahren, das zur Ausführung des TrackletGenerators nicht zwingend nötig ist. Das Verfahren wird an dieser Stelle nur kurz erklärt, da es im Rahmen dieser Arbeit nicht parallelisiert wurde.

Mit dem HitCorrector kann die Berechnung der Kreisbahnen mit dem Riemann-Fit deutlich verbessert werden. Unter Einbeziehung der Isochron-Radien der Hits wird nach den x-y-Positionen auf den Isochronen gesucht, an denen das Teilchen die Straw Tube passiert hat. So werden anstelle der Mittelpunkte die vom HitCorrector berechneten Hit-Positionen durch einen Kreis approximiert.

Durch den Einsatz des HitCorrectors können zu den vom TrackletGenerator erzeugten Tracklets Kreisbahnen berechnet werden, die den projizierten Flugbahnen sehr ähnlich sind. Dies ist für das Zusammenführen von Informationen mehrerer Detektoren hilfreich. Den approximierten Flugbahnen können über Abstandsberechnungen Hits anderer Detektoren zugeordnet werden. Dies ist ein wichtiger Schritt für das globale Trackfinding. Auf das lokale Trackfinding in Kombination mit dem TrackletGenerator im STT hat der HitCorrector nur einen Einfluss, wenn der Riemann-Fit und Abstandsrechnungen involviert sind. So könnten andere Kombinationsmöglichkeiten zugelassen werden und die Zuordnung verbleibender Hits ein abgewandeltes Ergebnis erzielen.

Für viele Tracks lässt sich der Verlauf anhand der Isochron-Radien benachbarter Hits erschließen. Der HitCorrector benötigt daher Informationen über die Nachbarschaftsbeziehungen der Tubes, die Hits eines Events und zugehörige Isochron-Radien. Derzeit wird auf die von PandaRoot simulierten Isochrone zugriffen. Eine Berechnung der Isochron-Radien ist noch umzusetzen. Auch hier wird wieder die x-y-Projektion betrachtet. Für parallele Straw Tubes handelt es sich daher um Kreise um den Mittelpunkt der Straw Tubes. Gedrehte Straw Tubes werden nicht in das Verfahren einbezogen, da deren Isochrone einen gedrehten Zylinder darstellen. Es fehlt noch eine geeignete Projektion in die x-y-Ebene. Abbildung 3.6 zeigt einen Track mit den zugehörigen Isochronen.

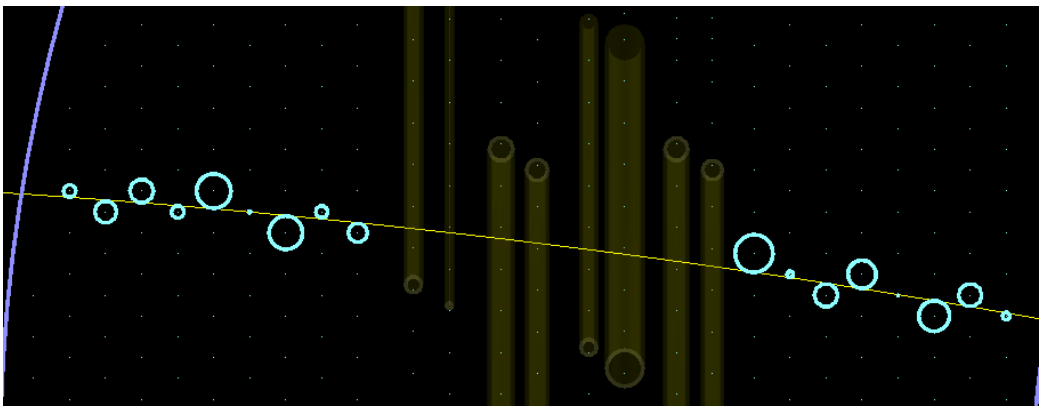


Abbildung 3.6: Isochrone paralleler (links und rechts) und gedrehter Straw Tubes (Mitte).

Für die Funktionsweise des Verfahrens wird vereinfachend angenommen, dass die Flugbahnen zwischen den Isochronen von Hits benachbarter Straw Tubes linear verlaufen. Außerdem wird angenommen, dass aufeinanderfolgende Linien zwischen den Isochronen annähernd in die gleiche Richtung verlaufen. Die Grundidee des Verfahrens kann mit Abbildung 3.7 nachvollzogen werden. Angenommen ein Hit α hat zwei Hit-Nachbarn β und γ, dann:

1. Berechne die inneren und äußeren Tangenten zwischen den Isochronen von α und β und von α und γ (B).

2. Finde für α, β und α, γ jeweils die zwei Tangenten, die sich am ähnlichsten sind (C).

3. Der korrigierte Punkt für α ist der Schnittpunkt der gemittelten Tangenten am Isochron-Kreis von α (D).

Der HitCorrector lässt sich in zwei wesentliche Stufen gliedern:

Initialisierung Im Rahmen der Initialisierung wird nach Konstellationen von Isochron-Radien benachbarter Signale gesucht, die den Verlauf der Flugbahn eindeutig erschließen lassen. Nach Beendigung dieser Stufe liegen für einige Hits eindeutige korrigierte Positionen vor.

Adaption Für die restlichen Hits wird in der zweiten Stufe über eine Adaption ausgehend von den eindeutigen Werten eine Entscheidung getroffen. Anhand der Lage und Größe der Isochrone wird versucht, mit Hilfe der initialisierten eindeutigen Hits korrigierte Positionen für die anderen Hits zu bestimmen.

Da der Berechnungsaufwand mit der Anzahl der Hit-Nachbarn steigt, werden nur Hits mit zwei bis vier Hit-Nachbarn in das Verfahren einbezogen.

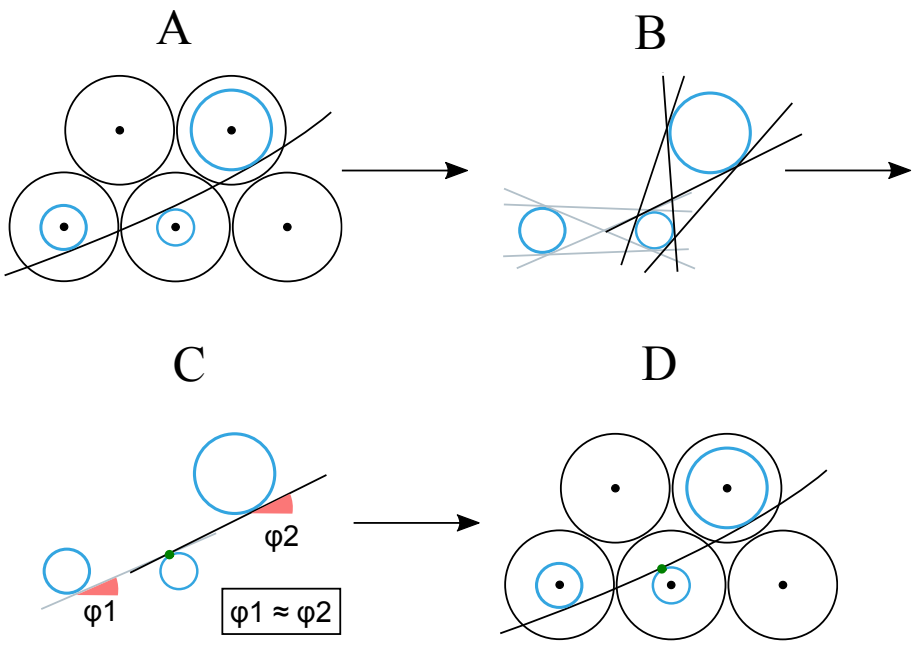


Abbildung 3.7: Berechnung korrigierter Positionen mit dem HitCorrector. A: Flugbahn eines Teilchens mit entsprechenden Isochronen, B: benachbarte Isochronen mit inneren und äußeren Tangenten, C: Tangenten, die sich am ähnlichsten sind und zugehöriger Schnittpunkt , D: der Schnittpunkt ist die korrigierte Position.

Bei der Berechnung der Tangenten kann es zwei Paare ähnlicher Tangenten geben. Abbildung 3.8 zeigt zwei Beispiele solcher Konstellationen. Für Beispiel A kann keine Entscheidung darüber getroffen werden, welche Tangente und somit welche Position die richtige ist. Für Beispiel B wird über eine Mittelung eine eindeutige Position bestimmt. Es werden noch andere Anordnung von Isochronen untersucht, die an dieser Stelle nicht beschrieben werden. Nach der Initialisierung wird versucht für die Nachbarn der eindeutigen Hits eine korrigierte Position zu berechnen. War dies erfolgreich, so kann die Position für den Nachbarn des Nachbarn untersucht werden usw. Für Beispiel A müsste für den linken oder rechten Nachbar eine eindeutige Position berechnet worden sein, damit eine Entscheidung getroffen werden kann. Liegt z. B. die korrigierte Position für den linken Nachbar auf der unteren Seite der Isochrone, so wird dies für die mittlere und im nächsten Schritt dann auch für die rechte Tube angenommen. Gibt es also in einer Menge von benachbarten Hits nach der Initialisierung eine eindeutig korrigierte Position, kann mit Hilfe dieser Information bei Mehrdeutigkeiten eine Entscheidung für den Rest der Hits dieser Menge getroffen werden. Dabei müssen viele Sonderfälle bezüglich der Lage und Größe benachbarter Isochrone abgehandelt werden. Abbildung 3.9 zeigt die korrigierte Hit-Positionen.

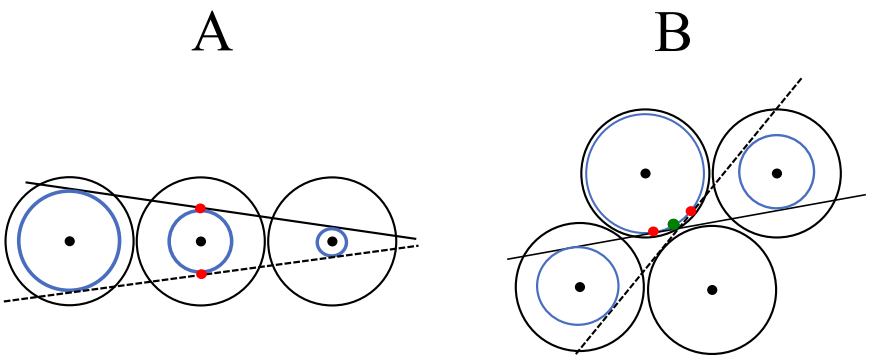


Abbildung 3.8: Nicht eindeutige Konstellationen von Isochronen. A: Der Track könnte ober- und unterhalb der Isochrone verlaufen, B: Bis zu einer bestimmten Differenz der Steigungen der Tangenten wird ein gemittelter Wert gewählt, um eine eindeutige Position berechnen zu können.

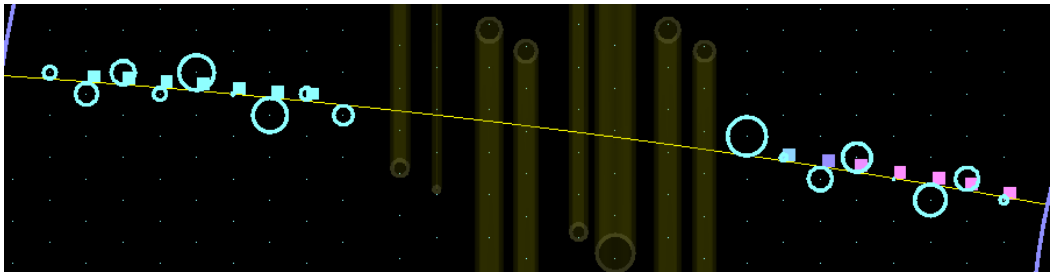


Abbildung 3.9: Mit dem HitCorrector berechnete Hit-Positionen. Die korrigierten Positionen werden durch die linke untere Ecke der Quadrate dargestellt. Für gedrehte Straw Tubes ist die Berechnung noch nicht möglich.

3.2.3 Implementierung des Verfahrens

Integration in PandaRoot

Um das Spurfinde-Verfahren in PandaRoot zu integrieren, wurde eine *Task* implementiert. Eine Task definiert verschiedene Funktionen mit denen das Einlesen der Daten, die Ausführung des Verfahrens und das Wegschreiben der Ergebnisse für eine bestimmte Anzahl aufeinanderfolgender Events gesteuert werden kann. Die Koordinierung der Funktionsaufrufe wird von der Laufzeitumgebung von ROOT übernommen. Es können Funktionen definiert werden, die jeweils für jedes Event oder einmalig für alle Events zum Anfang oder Ende ausgeführt werden sollen.

Ein Objekt einer Task wird in einem Rekonstruktions-Makro erzeugt und der Laufzeit-Umgebung als eine auszuführende Aufgabe bekannt gemacht.

Ein- und Ausgabewerte

Als Eingabewerte für den Spurfinder dient eine Menge von STT-Hits, bei denen es sich um Signale von den Straw Tubes eines Events handelt. Diese Hits werden mit PandaRoot simuliert, über eine ROOT-Datei zur Verfügung gestellt und in der Task für jedes Event verarbeitet. Die eventweise Bearbeitung der Daten ist eine vereinfachte Vorgehensweise. Im realen Experiment-Ablauf gibt es zu Beginn keine Gruppierung in Events, da kontinuierlich Messdaten ausgelesen werden. Diese Information ist so direkt nur im Rahmen der Simulation verfügbar. Soll das Verfahren im realen Experiment eingesetzt werden, muss es mit einem kontinuierlichen Datenstrom arbeiten können. Das anschließende Auftrennen dieses Datenstroms in Gruppen von Signalen, die zu einem physikalischen Event gehören, muss noch implementiert werden. Daher wird mit den durch die Simulation bekannten Gruppen von Hits (pro Event) gerechnet.

Die Ergebnisse des Spurfinders werden in Datenstrukturen wie Tracks (`PndTracks`) oder Track-Kandidaten (`PndTrackCands`) überführt. Track-Kandidaten stellen eine Gruppierung von Hits dar, während Tracks zusätzliche Informationen über die Flugbahn wie z. B. den Impuls enthalten.

Erstellte Klassen

Ein Überblick über die implementierten Klassen ist in Abbildung 3.10 zu sehen. Bei der Implementierung des Verfahrens wurden Programmierrichtlinien von PandaRoot eingehalten. Die Klassennamen des PandaRoot-Frameworks beginnen mit dem Präfix „Pnd“, Attribute starten mit „f“ und Methodennamen mit einem Großbuchstaben. `typedefs` und `structs` enden mit „_t“. Bei einem `TClonesArray`[1] handelt es sich um eine Realisierung eines Containers von ROOT, die eigenständig und effizient Speicherplatz verwaltet.

Es wurde eine Task namens `PndSttCellTrackFinderTask` implementiert, die einmalig ein Objekt der Klasse `PndSttCellTrackFinder` erzeugt. Die Task liest für jedes Event STT-Hits aus einer ROOT-Datei ein und gibt sie an den Spurfinder weiter. Der Spurfinder liest zu Beginn die Informationen über die Lage und Eigenschaften der Straw Tubes des STTs ein. Diese Informationen sind für alle Events konstant und werden daher auch nur einmal generiert. Für jedes Event wird das gleiche Spurfinder-Objekt benutzt, weshalb eine Funktion zum Zurücksetzen der berechneten Werte angeboten wird.

Für das Anwenden des TrackletGenerators und HitCorrectors muss eine Instanz der Klassen `PndSttCellTrackletGenerator` bzw. `PndSttHitCorrector` erzeugt werden. Die Klassen `PndSttCellTrackFinderData` dient zum Verteilen der Nachbarschaftsbeziehungen und ähnlichen Daten an den TrackletGenerator und HitCorrector. Diese Daten werden einmalig vor der Ausführung der Verfahren erzeugt und an sie weiter gereicht.

[1] ROOT-Klassen beginnen mit `T`.

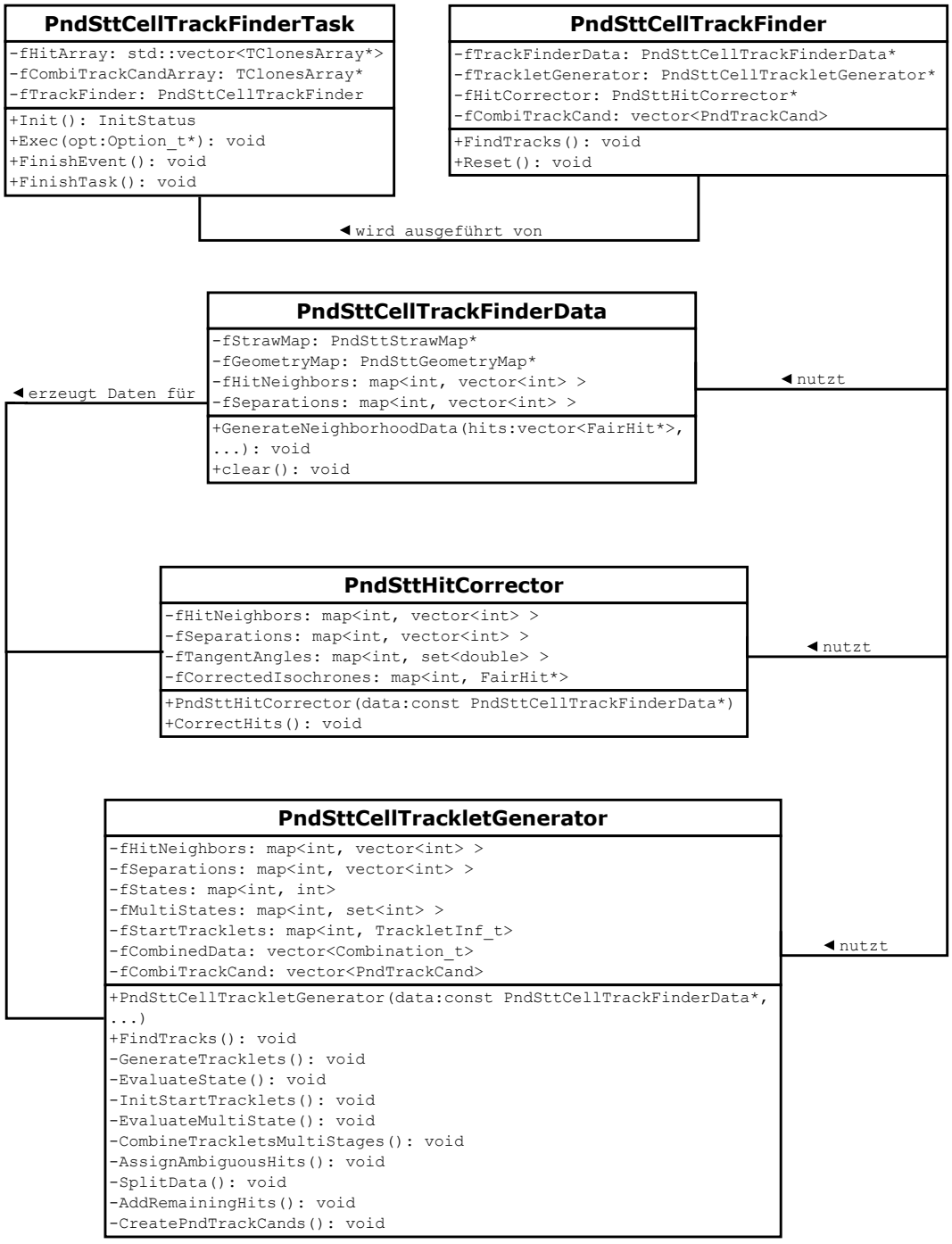


Abbildung 3.10: Vereinfachtes Klassendiagramm der in PandaRoot integrierten Klassen zur Realisierung eines Spurfinders für den STT. Nur die wichtigsten Attribute, Methoden und Parameter sind dargestellt.

In der Task kann über die Funktion `FindTracks()` des TrackFinder-Objektes das Trackfinding-Verfahren angestoßen werden. Für den Tracklet-Generator steht eine gleichnamige Funktion zur Verfügung. Der TrackletGenerator benötigt eine Instanz des Daten-Objektes, um auf die Datenstrukturen `fHitNeighbors` und `fSeparations` zuzugreifen. Es wurden Maps und Vektoren der Standard Template Library verwendet, da die Anzahl der Hits und somit auch Hit-Nachbarn für jedes Event unterschiedlich ist. Die dynamischen Datenstrukturen erlauben eine einfache Handhabung der Daten. Eine Auflistung der wichtigsten Datenstrukturen, die der TrackletGenerator benutzt, ist in Tablle 3.1 zu finden.

Datenstruktur	**Beschreibung**
`fHitNeighbors`	Speichert zu jeder Tube-ID eines Hits die Tube-IDs der aktiven Hit-Nachbarn
`fSeparations`	Gruppierung der Einträge aus `fHitNeighbors` entsprechend der Anzahl aktiver Nachbarn
`fStates`	Speichert zur Tube-ID einer eindeutigen Zelle den Zustand
`fMultiStates`	Speichert zur Tube-ID einer mehrdeutigen Zelle die Zustände
`fStartTracklets`	Track-IDs der als erstes generierten Tracklets mit zugehöriger Tracklet-Information
`fCombinedData`	Speicherung der akzeptierten Kombinationen von Start-Tracklets
`fCombiTrackCand`	Ergebnis des TrackletGenerators
`TrackletInf_t`	Struktur zur Speicherung von Tracklet-Informationen, wie z. B. Anzahl enthaltener paralleler/gedrehter Tubes und die gefittete Kreisbahn
`Combination_t`	Struktur zur Speicherung der Track-IDs von kombinierten Tracklets und der angepassten Tracklet-Information

Tabelle 3.1: Übersicht über die wichtigste Datenstrukturen des `PndSttCell-TrackletGenerators`.

Innerhalb der `FindTracks()`-Methode des TrackletGenerators werden Funktionen aufgerufen, die die bereits beschriebenen Stufen der Spurfindung realisieren. Folgende Funktionen stehen zur Verfügung:

`GenerateTracklets()` Führt den zellulären Automaten für die eindeutigen und mehrdeutigen Zellen aus. Dazu werden die folgenden drei Methoden nacheinander aufgerufen.

`EvaluateState()` Implementierung des zellulären Automaten für die eindeutigen Zellen (ein bis zwei aktiven Nachbarn).

`InitStartTracklets()` Berechnet für die zuvor generierten Tracklets Informationen, die zum Kombinieren benötigt werden. Z. B. die Anzahl von Signalen gedrehter und paralleler Straw Tubes. Erst bei mindestens drei Hits paralleler Straw Tubes kann ein Riemann-Fit zum Bewerten der Kombinationen berechnet werden.

`EvaluateMultiState()` Implementierung des zellulären Automaten für die mehrdeutigen Zellen (mehr als zwei aktive Nachbarn).

`CombineTrackletsMultiStages()` Rekursives Verfahren, dass alle Kombinationsmöglichkeiten berechnet. Nachdem alle Möglichkeiten aufgestellt wurden, werden die zugehörigen Tracklet-Informationen berechnet und die physikalisch sinnvollen Kombinationen herausgefiltert.

`AssignAmbiguousHits()` Hier erfolgt eine gesonderte Behandlung mehrdeutiger Zellen, deren Zustand nur eine oder zwei Track-IDs enthält. Bei nur einer Track-ID, kann die Zelle sofort dem Tracklet mit der entsprechenden Tube-ID zugeordnet werden. Bei zwei Track-IDs erfolgt die Zuordnung erst nach einer Abstandsüberprüfung des Hits zur Kreisbahn.

`SplitData()` Es werden nicht-kombinierte Tracklets und Tracklets mit weniger als drei Hits herausgefiltert. Es wird versucht die Hits dieser Tracklets mit der folgenden Funktion an andere Tracklets/Kombinationen zuzuordnen.

`AddRemainingHits()` Über Abstandberechnungen wird versucht die verbleibenden Hits in die bisher berechneten Tracklets/Kombinationen einzugruppieren. Dabei wird die naheliegendste Kreisbahn ausgewählt. Eine Zuordnung erfolgt nur, wenn der Abstand kleiner ist als der Radius einer Straw Tube.

`CreatePndTrackCands()` Die Gruppen der Straw Tubes, gespeichert in Tracklets und Kombinationen von Tracklets, werden in Track-Kandidaten überführt.

3.2.4 Ergebnisse

Im Fokus der Spurfindung stehen Tracks, die sich vom Inneren des STTs nach außen bewegen. Manche Tracks sind so stark gekrümmt, dass sie im STT kreisen und eine dichte Menge von Hits hinterlassen. Der STT stößt für solche Fälle an die Grenzen seiner geometrischen Auflösung. Es ist sehr schwierig derartig stark gekrümmte Flugbahnen zu rekonstruieren, weshalb die Rekonstruktion dieser vernachlässigt werden kann.

Die Ergebnisse des Spurfinders wurden für 1 000 Events untersucht, die mit dem Event-Generator EvtGen[2] simuliert wurden. Mit Hilfe der Task `PndTrackingQualityAnalysisNewLinks` wurden die rekonstruierten Tracks der Events mit den simulierten verglichen. Das Ergebnis ist in Tabelle 3.2 dargestellt. Es wurden nur Tracks auf ihre Rekonstruktionsqualität untersucht, die mindestens 5 Hits besitzen. Mindestens drei Hits sind erfolderlich, um eine Kreisbahn zu fitten. Durch das Einbeziehen zwei weiterer Hits, kann die Qualität des Fits verbessert werden. Außerdem kann der HitCorrector zum Einsatz kommen. Bei einem Tracklet von 5 Hits kann er drei korrigierte Positionen liefern. Besteht ein Tracklet nur aus drei Hits, kann nur der mittlere korrigiert werden. Zur Analyse wurden die simulierten Flugbahnen (Monte-Carlo-Tracks) mit den rekonstruierten verglichen (mit Hilfe von FairLinks). Bei der Qualität der Rekonstruktion wird in reine und unreine Tracks unterscheiden. *Rein* bedeutet, dass der Track-Kandidat nur Hits enthält, die von dem gleichen simulierten Monte-Carlo-Track (MC-Track) ausgehen. Ist er *unrein*, enthält der Track-Kandidat Hits von denen mindestens 70 % von einem MC-Track stammen. Bei *Ghosts* handelt es sich um rekonstruierte Tracks, die fälschlicherweise Hits von mehreren MC-Tracks enthalten und keine unreinen Tracks sind.

MC-Tracks mit mindestens 5 Hits	3880
Davon rein und vollständig gefunden	28.63 %
Davon rein, aber unvollständig gefunden	60.85 %
In unreinen Track-Kandidaten gefunden	3.25 %
Gefunden	92.73 %
Nicht gefunden	7.27 %
Entstandene Ghosts	423

Tabelle 3.2: Ergebnisse der Suche von MC-Tracks in den vom TrackletGenerator erzeugten Track-Kandidaten.

Die Ergebnisse sind nicht mehr mit denen der Bachelorarbeit [1] zu vergleichen. Neben dem Spurfinder wurde die Simulation, die Digitalisierung und Analyse weiterentwickelt, was zu anderen Gegebenheiten führt. Bei der Bewertung der Ergebnisse muss beachtet werden, dass die Signale gedrehter Straw

[2]unter Verwendung des *Decay-Files* `psi2s_jpsi2pi_1k.evt`

Tubes nur bedingt in das Verfahren involviert sind. Außerdem werden mehrfache Signale von der gleichen Straw Tube nicht weiter betrachtet, da die Daten überschrieben werden. Es wird ein beliebiger Hit ausgewählt, sodass die anderen gar nicht zugeordnet werden können.

3.3 Laufzeitverhalten

Ziel der Laufzeitanalyse des Spurfinders ist es, die rechenzeitintensivsten Funktionen ausfindig zu machen. Darauf basierend kann entschieden werden, auf welche Funktionen ein Fokus bei der Parallelisierung gelegt werden sollte. Im Folgenden wird beschrieben, wie die Laufzeiten erfasst worden sind und die Ergebnisse werden analysiert.

3.3.1 Zeitmessung

Um die Ausführungszeiten der einzelnen Funktionen zu messen, wurden von ROOT zur Zeitmessung bereitgestellte Klassen verwendet. Der Einsatz eines Profilers wie *gprof* erwies sich als schwierig, da PandaRoot nicht als Stand-Alone-Programm konzipiert wurde und viele Bibliotheken dynamisch geladen werden. ROOT bietet zur Zeitmessung die Klasse `TStopWatch` an. In dieser Klasse stehen Funktionen wie `Start()`, `Stop()` und `Continue()` an. Mit Hilfe von Zeitstempeln werden die verstrichenen Sekunden ermittelt. Zur Analyse der Laufzeiten müsste vor und nach jedem relevanten Funktionsaufruf `Start()` bzw. `Stop()` aufgerufen und die Zeit gespeichert werden. Die Klasse bietet keine Möglichkeit zur internen Speicherung mehrerer Zeitstempel w. z. B. über einen Stack. Daher wurde auf den Gebrauch dieser Klasse verzichtet und stattdessen direkt Zeitstempel an den gewünschten Stellen erzeugt und gespeichert. Für das Generieren der Zeitstempel wurde die ROOT-Klasse `TTimeStamps` benutzt. Zur Ermittlung der verstrichenen Zeiten wurden erst nach Abschluss aller Zeitmessungen die Differenzen berechnet und in eine Datei geschrieben. So wurde der Overhead zur Zeitmessung möglichst gering gehalten.

3.3.2 Analyse der einzelnen Funktionen

Laufzeiten

Zur Analyse des Laufzeitverhaltens der einzelnen Funktionen der Klasse `PndSttCellTrackletGenerator` wurden vor und nach dem Aufruf der Funktionen Zeitstempel gespeichert. Der Spurfinder wurde mit 1 000 Events getestet. Für die relevanten Funktionen wurde eine mittlere absolute Ausführungszeit und der mittlere relative Anteil der Laufzeit der Funktionen an `FindTracks()` pro Event berechnet.

Die durchschnittliche Ausführungszeit der Funktion `FindTracks()` (also des gesamten Spufindeverfahrens) beträgt auf dem eingesetzten PC[3] ungefähr 7.7 ms/Event. In Abbildung 3.11 ist zu sehen, dass die Funktionen `EvaluateMultiState()`, `EvaluateState()` und `CreatePndTrackCands()` die durchschnittlich längste Ausführungszeit in Anspruch nehmen.

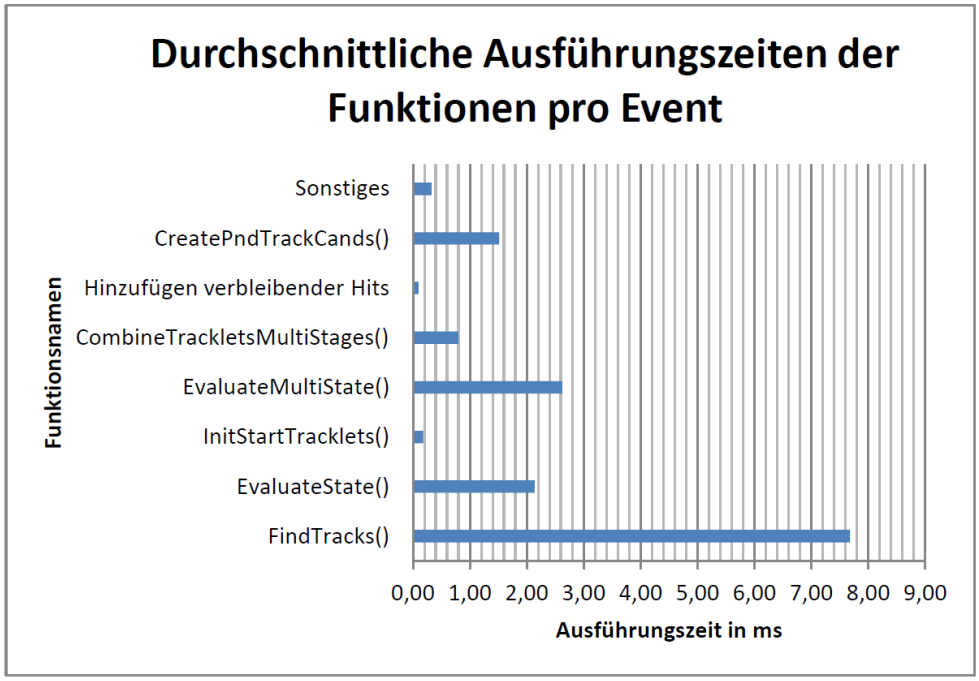


Abbildung 3.11: Durchschnittliche Ausführungszeiten der Funktionen des TrackletGenerators pro Event gemittelt für 1 000 Events.

Die Ausführungszeiten der Funktionen `AssignAmbiguousHits()`, `SplitData()` und `AddRemainingHits()` wurden aufgrund der geringen Laufzeiten zu „Hinzufügen verbleibender Hits“ zusammengefasst. Unter „Sonstiges“ fallen das Initialisieren von `fStates` und Operationen zwischen den Funktionsaufrufen w. z. B. Prüfen des Ausgabe-Levels, Wegschreiben oder Aktualisieren von Daten.

Das Diagramm in Abbildung 3.12 zeigt den durchschnittlichen relativen Anteil der Ausführungszeit der Funktionen an der Ausführung Funktion `FindTracks()`. Hier ist zu sehen, dass die Anteile für die gleichen Funktionen am höchsten sind. Der Anteil von `EvaluateMultiState()` erscheint im Vergleich mit den Daten in Abbildung 3.11 gering. Das bedeutet, dass es einige Events mit extremen Werten gibt, die die durchschnittliche Ausführungszeit deutlich steigern (siehe Abbildung 3.13). `EvaluateState()` benötigt im

[3]OptiPlex 7010 mit QuadCore i7-3770

Schnitt den größten Anteil der gesamten Ausführungszeit des Verfahrens, gefolgt von `CreatePndTrackCands()`.

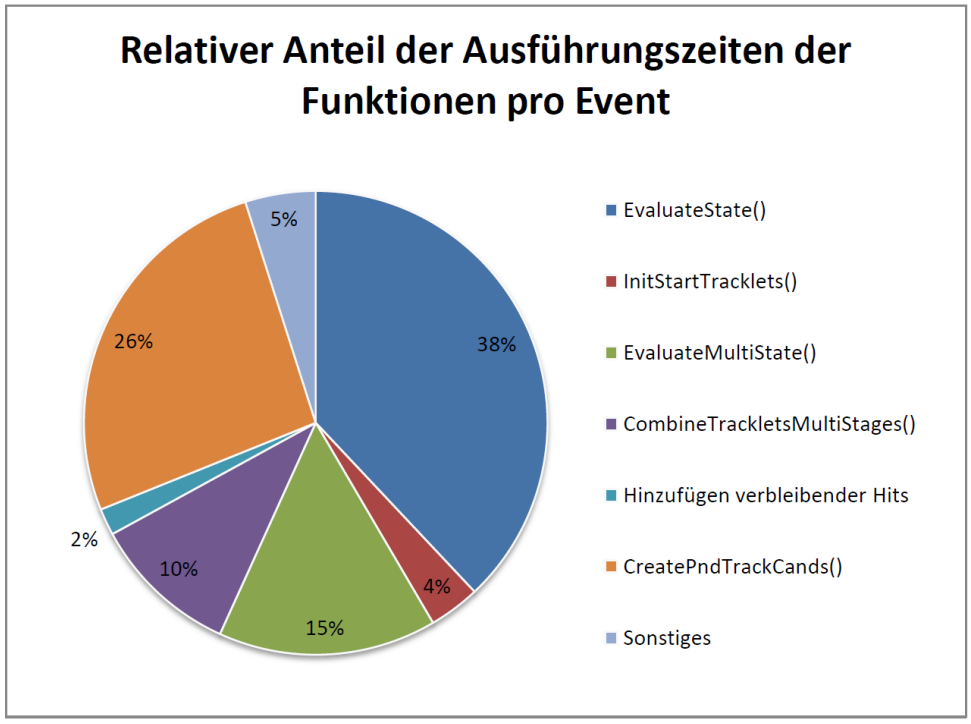


Abbildung 3.12: Durchschnittlicher relativer Anteil der Laufzeiten der Funktionen an `FindTracks()` gemittelt für 1 000 Events.

Ausreißer

Für `EvaluateMultiState()` beträgt die maximale gemessene Ausführungszeit 590.12 ms. Das entsprechende Event ist in Abbildung 3.13 zu sehen. Die Ursache für die lange Ausführungsdauer sind stark gekrümmte Tracks, die eine dichte Nachbarschaft von Hits erzeugen. Es gibt 445 mehrdeutige Zellen für die im Schnitt jeweils 35 Track-IDs als endgültiger Zustand generiert werden. Das resultiert in 694 Kombinationsmöglichkeiten, von denen 389 akzeptiert werden. Die Ausführungszeit von `CreatePndTrackCands()` ist für dieses Event mit 31.62 ms auch maximal, da entsprechend viele Track-Kandidaten erzeugt und gespeichert werden.

Ein Maximalbeispiel für `EvaluateState()` ist in Abbildung 3.14 zu sehen. Der Grund für die lange Ausführungszeit ist der lange Track in Kreisform. Dieser Track hinterlässt nur wenige mehrdeutige Hits, wodurch die Zustände von langen Ketten eindeutiger Hits angepasst werden müssen. Die Funktion benötigt zur Anpassung aller Zustände 7.60 ms.

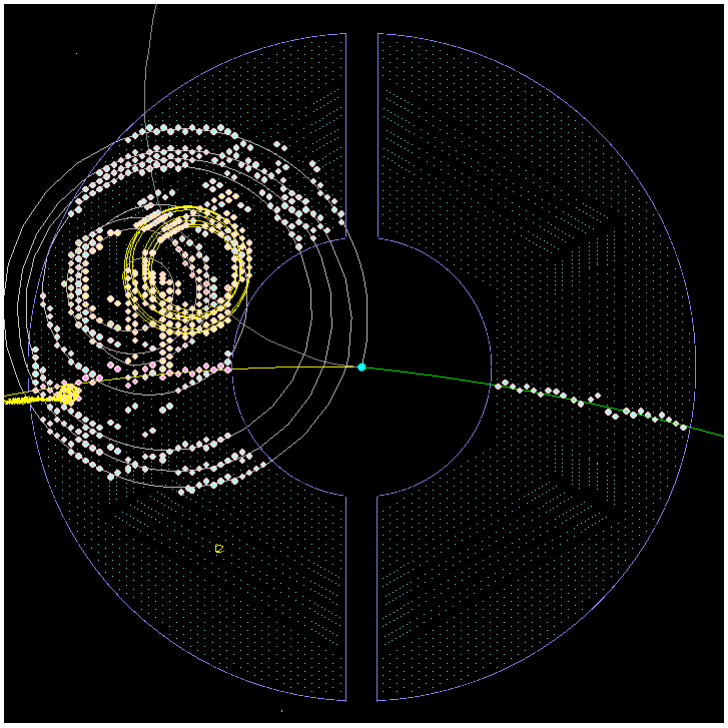


Abbildung 3.13: Event mit maximaler Ausführungszeit für `EvaluateMultiState()`.

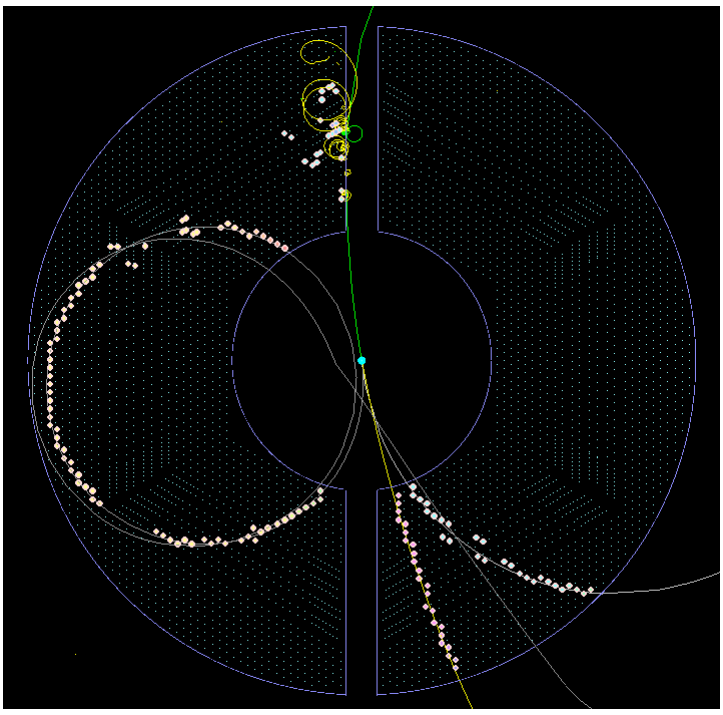


Abbildung 3.14: Event mit maximaler Ausführungszeit für `EvaluateState()`.

Parallelisierung

Die rechenzeitintensivsten Funktionen sind `EvaluateMultiState()`, `EvaluateState()` und `CreatePndTrackCands()`. Die Funktionen zum Generieren der Zustände bieten eine gute Voraussetzung für eine Parallelisierung. Es handelt sich dabei um die Implementierung eines zellulären Automaten, dessen Übergangsregel parallel ausgeführt/angewendet werden kann. `CreatePndTrackCands()` ist der abschließende Schritt des Verfahrens in dem die Ergebnisse in die gewünschten Datenstrukturen überführt und gespeichert werden. Die Ausführungszeit dieser Funktion ist später im Experimentbetrieb vernachlässigbar, da sie nur benötigt wird, um Datenstrukturen zu erzeugen, die im Rahmen der Simulation benötigt werden. Beim Einsatz im realen Experiment sollen die Datenstrukturen des Spurfinders, soweit dies möglich ist, unverändert weiterverwendet werden. Daher ist eine Umformatierung nicht nötig. Aus diesem Grund wird als erstes eine Parallelisierung der Verfahrensschritte `EvaluateState()` und `EvaluateMultiState()` umgesetzt, welche in den folgenden Kapitel beschrieben wird.

4 Programmierung auf einer Grafikkarte mit CUDA C

Zur Beschleunigung des Spurfinders wurden Teile des Verfahrens auf einer Graphics Processing Unit (GPU) implementiert, sodass sie dort parallel von mehreren Threads ausgeführt werden können. Es wurde eine GeForce GTX 750 Ti von NIVIDIA eingesetzt, bei der es sich um eine CUDA-fähige GPU handelt. CUDA bietet u. a. ein Programmiermodell an, mit dem die C/C++-Programmierung auf NVIDIA-GPUs durch syntaktische Erweiterungen der Sprachen leicht zugänglich gemacht wird. Zur Parallelisierung des Spurfinders wurde kein Gebrauch von spezifischen C++-Erweiterungen gemacht. Die GPU-Programmierung erfolgte daher mit CUDA C. CUDA C definiert einige zusätzliche Schlüsselwörter, sodass C-ähnlicher Code auf der GPU ausgeführt werden kann. Im Folgenden werden GPUs, die Programmierung mit CUDA C und die Besonderheiten der CUDA-Architektur genauer vorgestellt.

4.1 GPU – Graphics Processing Unit

Die Weiterentwicklung der *Central Processing Units* (CPUs) stößt an ihre physikalischen und wirtschaftlichen Grenzen. Die Anzahl der Transistoren pro Chip kann z. B. aufgrund der Wärmeentwicklung nicht beliebig weiter erhöht werden. Der Einsatz mehrerer CPUs in Form von Multicore-Chips schafft Abhilfe. Ein anderer Ansatz ist die Nutzung von *Graphics Processing Units* (GPUs), die eine sehr hohe Rechenleistung mit sich bringen. Das Design von GPUs ist darauf ausgelegt, Prozesse massiv parallel auszuführen. Ursprünglich wurden sie zur Verarbeitung von Daten zur Darstellung der Computergrafik entwickelt (wie z. B. zum 3D-Rendering). GPUs werden darüber hinaus für wissenschaftliche, analytische oder technische Anwendungen eingesetzt. Dies bezeichnet man als *GPGPU computing* (General Purpose Computation on Graphics Processing Unit). Das Design der GPUs unterscheidet sich deutlich von dem der CPUs. CPU-Chips besitzen große Cache-Speicher, wenige komplexe Rechenwerke (ALUs) und ein Steuerwerk, das die Befehlsausführung und Besonderheiten wie Branch Prediction umsetzt. GPUs dagegen besitzen nur kleine Caches, aber eine große Anzahl von einfachen ALUs, die parallele Operationen ausführen können. Heutige Systeme besitzen sowohl eine CPU als auch eine GPU, welche beide für die Bearbeitung unterschiedlicher Auf-

gabentypen optimiert sind (heterogene Systeme). Die vielen ALUs der GPU ermöglichen eine weitaus höhere Rechenleistung. Diese kann allerdings erst ab einer bestimmten Problemgröße effektiv genutzt werden, da eine Kommunikation zwischen CPU und GPU erforderlich ist, die meist über den vergleichsweise langsamen PCI-Express-Bus erfolgt. Zur Berechnung benötigte Daten müssen vom Hauptspeicher in den Speicher der GPU übertragen werden. Dieser Vorgang kostet Zeit. Wird die GPU jedoch effektiv zur Problemlösung eingesetzt, kann die Rechengeschwindigkeit um einige Größenordnungen im Vergleich zur CPU gesteigert werden. [15, S. 1 ff.]

4.2 CUDA C

Mit CUDA wurde 2006 von NVIDIA eine Architektur eingeführt, die es ermöglicht die parallele Rechenleistung der NVIDIA GPUs mit Hilfe eines neuartigen Programmier-Modells zu nutzen. Parallele Abläufe können mit High-Level-Programmiersprachen wie C, C++, Fortran u. v. m. definiert werden. CUDA bietet Erweiterungen in Form von Schlüsselwörtern für diese Sprachen, um die Ausführung von parallelem Code auf der GPU umzusetzen. Für C/C++-Code stellt CUDA C die Erweiterungen mit einigen C++-Erweiterungen zur Verfügung. Mit Hilfe der Erweiterungen können drei grundlegende Schlüsselfunktionen zur Parallelisierung genutzt werden: eine hierarchische Einteilung in Threads, das Nutzen vom Shared Memory und die Synchronisation der Prozesse. Diese ermöglichen eine Parallelität auf Daten- und Thread-Ebene. [16, S. 4 ff.] Zur Programmierung mit CUDA C werden folgende Dinge benötigt [17, S. 14 ff.]:

- eine CUDA-fähige Grafikkarte,
- einen Gerätetreiber von NVIDIA,
- das CUDA Development Toolkit und
- der Standard C/C++-Compiler.

Zu den wichtigsten CUDA-fähigen Grafikkarten von NVIDIA zählen Produkte der Linien *GeForce*, *Tesla* und *Quadro*. Neben diesen gibt es noch CUDA-GPUs für Multidisplay-Grafiklösungen (NVS) und für mobile Geräte (Tegra). Die Quadro-GPUs werden im Grafik-Bereich eingesetzt. Sie bieten die Leistung und Hardware zur Bewältigung anspruchsvoller Grafikaufgaben und zur professionellen 2D- und 3D-Visualisierung. Die Tesla-Grafikprozessoren sind auf die Beschleunigung komplexer Datenanalysen und wissenschaftlicher Berechnungen ausgelegt. Das GeForce-Modell deckt den PC-Gaming-Bereich ab. Für jede Produktlinie gibt es sowohl preiswerte Einsteiger-Grafikkarten als auch professionelle High-End-Lösungen. [18]

Mit der *Compute Capability* einer NVIDIA-GPU, werden die speziellen Eigenschaften der Grafikkarten zusammengefasst. Dazu zählen Informationen wie die Speicherkapazitäten, aber auch programmierspezifische Möglichkeiten. Die Compute Capability wird durch eine Versionsnummer repräsentiert (5.3 ist derzeit die höchste Version).

CUDA erlaubt einen Mix von CPU- und GPU-Code. Dies bezeichnet man als *heterogenes Programmieren*. Die Vorgehensweise ist in Abbildung 4.1 veranschaulicht. Ein Programm startet auf dem *Host* (der CPU). Zur Berechnung benötigte Daten werden auf das *Device* (die GPU) kopiert, die Ausführung einer Funktion auf dem Device wird initiiert (Kernel-Launch) und nach Abschluss der Berechnung wird das Ergebnis zurück auf den Host kopiert. Für den Host und das Device werden separate Speicher verwaltet.

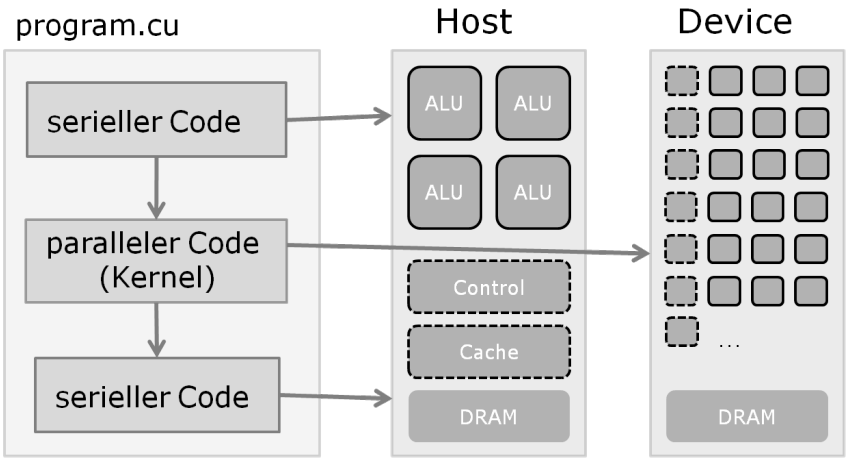


Abbildung 4.1: Heterogenes Programmiermodell von CUDA. Host- und Device-Komponenten sind nur skizzenhaft dargestellt.

Zum Kompilieren des Device-Codes wird `nvcc` eingesetzt. Dieser Compiler unterstützt für den Device-Code nicht alle C++-Erweiterungen, wie z. B. die C++ Standard Library. Es ist möglich, objektorientiert mit bestimmten Einschränkungen auf dem Device zu programmieren [16, S. 163]. Von den C++-Erweiterungen wurde bei der Implementierung des Spurfinders kein Gebrauch gemacht. Daher wird nur CUDA C genauer vorgestellt.

4.3 Das CUDA-Programmiermodell

Die folgenden Abschnitte beschreiben, wie Code parallel von mehreren Threads auf der GPU ausgeführt werden kann und wie diese Threads hierarchisch gruppiert werden können. Zudem wird erläutert, wie die Daten an die Threads aufgeteilt werden können und es wird eingeführt, wie verschiedene Funktionen auf der GPU zeitgleich ausgeführt werden können.

4.3.1 Kernel

Die Funktionen, die parallele Berechnungen auf dem Device ausführen, werden als *Kernel* bezeichnet. Ein Kernel wird N-mal parallel von N verschiedenen CUDA-Threads ausgeführt. Kernel sind durch das Schlüsselwort `__global__` gekennzeichnet. Dieser Qualifier gibt an, dass die Funktion vom Host-Code aufgerufen werden kann und auf dem Device ausgeführt wird. Der Funktionsaufruf wird um `<<<...>>>` erweitert. Innerhalb dieser Klammer erfolgt die Konfiguration der Ausführung auf dem Device, auf welche im nächsten Unterabschnitt genauer eingegangen wird. Es wird festgelegt, wie viele Threads in welcher Gruppierung den Kernel ausführen.

Im Programm-Beispiel 4.1 wird ein Kernel namens `add` mit einem Block von N Threads gestartet (Zeile 18). Auf dem Device werden die Vektoren `a` und `b` der Länge N addiert und im Vektor `c` gespeichert (Zeile 1-4). Dabei berechnen die Threads parallel jeweils nur eine Komponente von `c`. An den entsprechenden Index kann innerhalb der Kernel über eine Built-In-Variable `threadIdx.x` von CUDA gelangt werden. Dabei handelt es sich um den Thread-Index im Bereich 1 bis N, der den Threads eindeutig zugewiesen wird. Um in Kernel auf die Vektoren `a` und `b` zugreifen zu können, müssen diese vor dem Aufruf des Kernels vom Host auf das Device kopiert werden. [16, S. 9]

Listing 4.1: Vorgehensweise beim heterogenen Programmieren mit CUDA C.

```
1  __global__ void add(int *a int *b, int *c){
2    int i=threadIdx.x;
3    c[i]=a[i]+b[i];
4  }
5
6  int main(void){
7
8    int *a,*b,*c;
9    ...
10   int *dev_a, *dev_b, *dev_c;
11   cudaMalloc((void**) &dev_a,size);
12   cudaMalloc((void**) &dev_b,size);
13   cudaMalloc((void**) &dev_c,size);
14
15   cudaMemcpy(dev_a, &a, size, cudaMemcpyHostToDevice);
16   cudaMemcpy(dev_b, &b, size, cudaMemcpyHostToDevice);
17
18   add<<<1,N>>>(dev_a, dev_b, dev_c);
19
20   cudaMemcpy(&c, dev_c, size, cudaMemcpyDeviceToHost);
21   ...
22   cudaFree(dev_a);
23   cudaFree(dev_b);
24   cudaFree(dev_c);
25 }
```

Host und Device haben separate Speicher, die verwaltet werden müssen. Kernel arbeiten auf dem Device-Speicher. Für jede Datenstruktur auf die innerhalb eines Kernels zugegriffen werden soll, muss der Speicher explizit alloziert werden (Zeile 11-13). Dies muss auch für den Ergebnisvektor `c` erfolgen. Zum Allozieren von linearen Speicher wird `cudaMalloc` genutzt. Diese Funktion alloziert Speicher der übergebenen Größe auf dem Device, auf den über einen entsprechenden Pointer zugegriffen werden kann. Freigegeben wird der Speichert mit `cudaFree`. [16, S. 21 f.]

Der richtige Umgang mit Host- und Device-Pointern liegt in der Verantwortung des Programmierers. Pointer, die mit `cudaMalloc` alloziert worden sind, zeigen auf Speicher im Device. Dementsprechend können diese Pointer auf der Host-Seite nicht zum Lesen und Schreiben von dem allozierten Speicher genutzt werden. Die Device-Pointer können lediglich an das Device weitergegeben und für Kopier-Vorgänge genutzt werden. [17, S. 26]

Mit der Funktion `cudaMemcpy` wird der Inhalt der Vektoren `a` und `b` in das Device Memory kopiert. Nach dem Kernel-Aufruf wird das Ergebnis aus dem Device Memory zum Host in den Vektor `c` kopiert.

Mit Hilfe von *Dynamic Parallelism* ist es möglich, auf dem Device innerhalb eines Kernels einen weiteren Kernel zu starten. Das verringert den Transfer zwischen Host und Device, da kein Umweg über die Kommunikation mit dem Host erfolgt und ermöglicht die verschachtelte Ausführung mehrerer Kernel. Dynamic Parallelism ist eine Erweiterung des Programmier-Modells, die ab Compute Capability 3.5 genutzt werden kann. [16, S. 132 ff.]

4.3.2 Indexing

Mit dem Programmier-Modell von CUDA kann das zu bearbeitende Problem in grobe, unabhängige Teil-Probleme zerlegt werden und die Lösung dieser wiederum feiner aufgeteilt werden. Dabei erfolgt eine parallele Ausführung auf zwei Ebenen.

Die unabhängigen Teil-Probleme werden parallel in Blöcken von Threads bearbeitet und die Threads eines Blockes arbeiten ebenfalls parallel. Die Blöcke können in einem Grid in bis zu drei Dimensionen angeordnet werden. Auch die Anordnung der Threads innerhalb der Blöcke kann sich auf bis zu drei Dimensionen erstrecken. Dieses abstrakte Modell erlaubt je nach Struktur der an die Threads verteilten Daten einen einfachen Zugriff auf Einträge in Vektoren (1D), Matrizen (2D) oder Volumen (3D).

Die Anzahl der Threads und Blöcke wird beim Aufruf eines Kernels wie folgt definiert: `<<< Anzahl der Blöcke in dem Grid, Anzahl der Threads pro Block >>>`. Alle Threads eines Grids führen den gleichen Kernel aus. Über die Built-In-Variablen `gridDim`, `blockDim`, `blockIdx` und `threadIdx` kann innerhalb des Kernels auf die Dimension des Grids/der Blöcke und auf die Anzahl der Blöcke/Threads in einer bestimmten Dimension

zugegriffen werden. `blockIdx` und `threadIdx` besitzen daher eine x-,y- und z-Komponente. [16, S. 10 f.]

Für die Dimension 2 wäre die Struktur eines Kernel-Calls z. B. `<<< <2,3>, <3,4> >>>`. In dem Grid werden jeweils 2 Blöcke in x- und 3 in y-Richtung angeordnet. Innerhalb eines Blockes werden 3 Threads in x- und 4 in y-Richtung erstellt. Abbildung 4.2 zeigt die entsprechende Durchnummerierung der Thread- und Block-Indizes.

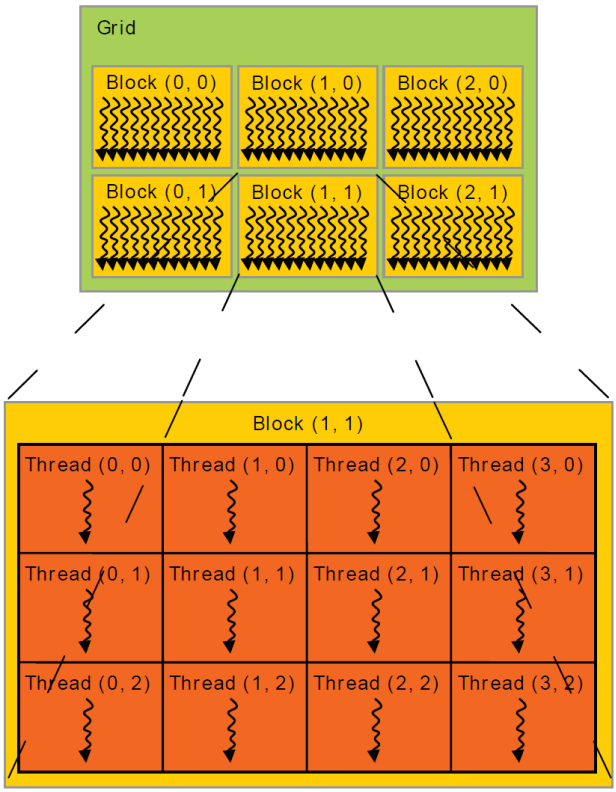


Abbildung 4.2: Hierarchie der CUDA-Threads. [16, S. 11]

Innerhalb eines Kernels wird mit Hilfe der oben genannten Built-In-Variablen eine Thread-ID berechnet, die einen Thread innerhalb eines Blockes eindeutig identifiziert. Über diese Thread-ID kann dann der entsprechende Teil an Daten aus der gesamten Datenmenge aller Threads ausgewählt werden. Auf diese Weise kann die Arbeit auf die Threads verteilt werden. Tabelle 4.1 zeigt die Berechnung der Thread-IDs anhand der Built-In-Variablen.

Die Blöcke müssen unabhängig voneinander ausgeführt werden können, da sie in einer beliebigen Reihenfolge auf unterschiedlichen Rechenkernen ausgeführt werden können. Threads eines Blockes können auf gemeinsame Daten effizient über *Shared Memory* zugreifen (siehe Abschnitt 4.5). Um die Ausführung der Threads und Speicherzugriffe zu synchronisieren, kann innerhalb des Kernels die Funktion `__syncthreads()` aufgerufen werden. Diese

Funktion stellt eine Barriere für die Threads eines Blockes dar. Erst wenn alle Threads `__syncthreads()` aufgerufen haben, wird der darauffolgende Code ausgeführt.

D	Thread-Index	Eindeutige Thread-ID
1	(`threadIdx.x`)	`threadIdx.x`
2	(`threadIdx.x`, `threadIdx.y`)	`threadIdx.x +threadIdx.y*blockDim.x`
3	(`threadIdx.x`, `threadIdx.y`, `threadIdx.z`)	`threadIdx.x+threadIdx.y*blockDim.x +threadIdx.z*blockDim.x*blockDim.y`

Tabelle 4.1: Berechnung der Thread-IDs pro Block für verschiedene Dimensionen.

4.3.3 Asynchrone parallele Ausführung

CUDA-Funktionen können synchron oder asynchron in Bezug auf den Host-Prozess ausgeführt werden. Bei asynchronen Funktionen wird die Kontrolle direkt nach dem Funktionsaufruf an den Host-Thread übergeben unabhängig davon, ob die Ausführung der Funktion auf dem Device beendet ist. Kernel-Calls erfolgen asynchron. Dies ermöglicht eine parallele Ausführung von Host- und Device-Code. Beim Aufruf synchroner Funktionen erhält der Host erst wieder die Kontrolle, wenn die Aufgaben vollständig bearbeitet worden sind, z. B. bei `cudaMemcpy`. [16, S. 31 ff.]

In Abhängigkeit von der eingesetzten Grafikkarte können folgende Prozesse zeitgleich ausgeführt werden:

- Der Datentransfer zwischen Host und Device Memory und die Ausführung eines Kernels,
- die Ausführung verschiedener Kernel und
- verschiedene Datentransfers.

Um diese Eigenschaften nutzen zu können, müssen die Kopier- bzw. Rechenvorgänge asynchron ausgeführt werden.

Das asynchrone Kopieren von Daten kann mit `cudaMemcpyAsync()` angestoßen werden. Die zu kopierenden Daten müssen im *Page-Locked* bzw. *Pinned Memory* abgelegt werden. Das bedeutet, dass die Daten durch die virtuelle Speicherverwaltung nicht ausgelagert werden können. Dies ist für das asynchrone Kopieren zwingend erforderlich, da die Daten nicht verändert/ausgelagert werden dürfen, bevor das Kopieren abgeschlossen ist. Pinned Memory sollte

mit Bedacht in begrenzten Maßen verwendet werden, da der zur Seitenauslagerung verfügbare Speicher reduziert wird, was die Performance des Systems auf der Host-Seite einschränken kann. Mit den Funktionen `cudaMallocHost()` und `cudaFreeHost()` kann Pinned Host-Memory alloziert und freigegeben werden. [16, S. 29 ff.]

Die Nebenläufigkeit der Vorgänge wird mit Hilfe von *Streams* realisiert. Ein Stream ist eine Folge von Befehlen, die in der gegebenen Reihenfolge ausgeführt werden. Verschiedene Streams können ihre Befehle unabhängig von einander oder auch parallel ausführen. Streams können mit `cudaStreamCreate()` erzeugt werden. Beim Aufruf von CUDA-Funktionen kann dann der entsprechende Stream angegeben werden. Wird kein Stream angegeben, wird der *Default Stream* benutzt. Die Funktionen werden der Reihe nach ausgeführt. Befehle von verschiedenen Streams werden nicht nebenläufig ausgeführt, sobald ein Befehl im Default Stream ausgeführt wird. Außerdem kommt es zur impliziten Synchronisation, wenn Pinned Host Memory oder Device Memory alloziert/-gesetzt wird.

Programmbeispiel 4.2 zeigt die grundlegende Vorgehensweise bei der Programmierung mit Streams in CUDA.

Listing 4.2: Programmbeispiel zur Verwendung von Streams.

```
1  cudaStream_t streams[NUM_STREAMS];
2  for(int i=0; i<NUM_STREAMS; ++i){
3    cudaStreamCreate(&streams[i]);
4  }
5  int* pinned_memory_ptr;
6  cudaMallocHost(&pinned_memory_ptr, size);
7  ...
8
9  int size_per_stream=size/NUM_STREAMS;
10 for(int i=0; i<NUM_STREAMS; ++i){
11   cudaMemcpyAsync(dev_inp_ptr+i*size_per_stream,
         pinned_memory_ptr+i*size_per_stream, size_per_stream,
         cudaMemcpyHostToDevice, streams[i]);
12
13   kernel<<<NUM_BLOCKS, NUM_THREADS_PER_BLOCK, 0, streams[i]>>>(
         dev_inp_ptr+i*size_per_stream, dev_out_ptr+i*
         size_per_stream, size_per_stream);
14
15   cudaMemcpyAsync(pinned_memory_ptr+i*size_per_stream,
         dev_out_ptr+i*size_per_stream, size_per_stream,
         cudaMemcpyDeviceToHost, streams[i]);
16 }
17
18 for(int i=0; i<NUM_STREAMS; ++i){
19   cudaStreamDestroy(streams[i]);
20 }
21 cudaFreeHost(pinned_memory_ptr);
```

Es werden NUM_STREAMS Streams erstellt, an die das Kopieren der Daten im Pinned Memory aufgeteilt wird (Zeile 1-4). Jeder Stream kopiert einen gleich großen Anteil der Datenmenge und führt Berechnungen im Kernel auf dem entsprechenden Bruchteil der Daten aus. In diesem Beispiel werden die Ergebnisse in den gleichen Bereich des Pinned Memories geschrieben. Beim Aufruf der Kopiervorgänge (Zeile 11 und 15) und der Kernel (Zeile 13) wird in jeder Iteration ein anderer Stream angegeben. In der Start-Konfiguration des Kernels wird der Stream an vierter Stelle angegeben. Mit dem dritten Parameter kann die Anzahl an Bytes definiert werden, die im Shared Memory pro Block dynamisch alloziert werden sollen (Default ist 0). Werden die Streams so zugewiesen, kopiert jeder Stream die Daten zum Device, führt den Kernel aus und kopiert das Ergebnis zurück. Inwieweit die Streams parallel ausgeführt werden können, ist von den Eigenschaften des Devices und von Zuordnung der Befehle an die Streams abhängig. [16, S. 32 ff.]

Für Grafikkarten, die keine parallelen Kopiervorgänge unterstützen, kann mit dem Code in Listing 4.2 keine nebenläufige Ausführung der Streams erreicht werden. Es ist nur eine Copy Engine vorhanden, was dazu führt, dass die Streams nicht parallel ausgeführt werden können. Der Kopiervorgang vom Host zum Device wird Stream `i+1` erst dann zugewiesen, nach dem Stream `i` das Kopieren vom Device zum Host zugeordnet wurde. Dementsprechend muss Stream `i+1` auf Stream `i` warten. Die Zuordnung des Zurück-Kopierens in einer separaten, späteren Schleife, wie in Listing 4.3, schafft Abhilfe. Stream `i+1` kann Daten zum Device kopieren, während Stream `i` den Kernel ausführt. Sobald die Daten von den letzten Streams zum Device kopiert worden sind, können die Ergebnisse der ersten Streams auf den Host kopiert werden.

Listing 4.3: Streams werden in geänderter Reihenfolge gefüllt, um eine Überlappung zu erreichen.

```
1 for(int i=0; i<NUM_STREAMS; ++i){
2   cudaMemcpyAsync(dev_inp_ptr+i*size_per_stream,
        pinned_memory_ptr+i*size_per_stream, size_per_stream,
        cudaMemcpyHostToDevice, streams[i]);
3
4   kernel<<<NUM_BLOCKS, NUM_THREADS_PER_BLOCK, 0, streams[i]>>>(
        dev_inp_ptr+i*size_per_stream, dev_out_ptr+i*
        size_per_stream, size_per_stream);
5 }
6 for(int i=0; i<NUM_STREAMS; ++i){
7   cudaMemcpyAsync(pinned_memory_ptr+i*size_per_stream,
        dev_out_ptr+i*size_per_stream, size_per_stream,
        cudaMemcpyDeviceToHost, streams[i]);
8 }
```

4.4 Hardwaremodell

Das Herz der NVIDIA GPUs sind die *Streaming Multiprocessors*. Wird ein Kernel aufgerufen, werden die Blöcke an die verfügbaren Multiprozessoren verteilt und diese kümmern sich um die Ausführung. Die Multiprozessoren erstellen und verwalten die zugewiesenen Threads. Zudem sind sie für das Scheduling zuständig und führen Hunderte von Threads gleichzeitig aus. Zur Bewältigung dieser Aufgaben wird die *Single-Instruction, Multiple-Thread*-Architektur (SIMT) eingesetzt. Ein Multiprozessor bearbeitet Threads in Gruppen von 32 parallelen Threads, den *Warps*. Die Threads der zugewiesenen Blöcke werden nummeriert und in Warps unterteilt. Die Warps werden dann von *Warp-Schedulern* ausgeführt. Die Aufteilung der Threads in Warps ist jedes mal die gleiche. Die Threads besitzen eine Thread-ID und werden aufsteigend in Einheiten von 32 Threads beginnend bei 0 gruppiert. Dieses Modell ist hoch skalierbar. Je mehr Multiprozessoren zur Verfügung stehen, desto schneller können die Threads der Blöcke ausgeführt werden. [16, S. 68 f.]

Für einen Warp wird eine gemeinsame Instruktion gleichzeitig ausgeführt (daher der Name SIMT-Architektur). Haben alle Threads eines Warps die gleiche Abfolge von Befehlen, dann ist die Ausführung voll effizient. Alle Threads sind *aktiv*. Weicht die Befehls-Abfolge eines Threads ab (z. B. durch if-/else-Branches) wird dieser Thread *inaktiv*. Der Warp wird dann seriell für alle möglichen Pfade ausgeführt, bis sich wieder alle Threads auf dem gleichen Ausführungs-Pfad befinden. Ein Thread kann außerdem inaktiv sein, wenn er die Berechnungen schon früher beendet hat als die anderen oder wenn es einer der letzten Thread eines Blockes ist, dessen Thread-Anzahl kein Vielfaches von 32 ist. Für die korrekte Programmierung ist eine Kenntnis der Warps und ihrer Verarbeitung nicht nötig. Allerdings lässt sich durch die Beachtung dieser Eigenschaften die Performance wesentlich verbessern. 32 Threads sollten im besten Fall die gleiche Instruktions-Abfolge bearbeiten. Aufgrund der SIMT-Architektur wird die Thread-Anzahl üblicherweise so gewählt, dass sie ein Vielfaches von 32 ist. [16, S. 69]

Für jeden Warp befindet sich der Ausführungs-Kontext auf dem Chip bis alle Threads abgearbeitet worden sind. Daher verursacht ein Kontext-Wechsel durch den Warp-Scheduler keine Kosten. Der Warp-Scheduler wählt den Warp aus, der Threads besitzt, die bereit zur Ausführung sind. Jeder Multiprozessor stellt Register und Shared Memory zur Verfügung, die sich die Threads eines Blockes teilen. Die Größe dieser Speicher ist dafür ausschlaggebend, wie viele Blöcke und Warps vom Multiprozessor ausgeführt werden können. Außerdem gibt es eine maximale Anzahl von Blöcken und Warps pro Multiprozessor. Diese Eigenschaften hängen von der *Compute Capability* der Grafikkarte ab. [16, S. 70]

Aufgrund dieser Architektur und aus Erfahrungswerten ergeben sich für die Anzahl der Blöcke und Threads pro Kernel folgende Richtlinien [19, S. 46 f.]:

- Die Anzahl der Threads pro Block sollte ein Vielfaches von 32 (Warp-Größe) sein.
- Zwischen 128 und 256 Threads pro Block ist ein guter Startwert zur Untersuchung der Performance in Abhängigkeit von der Anzahl der Threads.
- Es ist besser, mehrere kleine Blöcke pro Multiprozessor zu starten anstatt einen großen. Das kann vor allem bei Kernel, die `__syncthreads()` benutzen, von Vorteil sein.
- Pro Multiprozessor sollten mehrere Blöcke zur Ausführung zur Verfügung stehen, sodass bei `__synchthreads()` eines Blockes ein anderer ausgeführt werden kann.

4.5 Device Memory

Die von einem Kernel angestoßenen Threads können auf verschiedene Speichertypen zugreifen, die sich in Bezug auf Größe, Zugriffszeit, Verfügbarkeit der Daten und Sichtbarkeit für andere Threads unterscheiden. In den nächsten Abschnitten werden die genauen Eigenschaften der Speicherarten beschrieben.

4.5.1 Speicherhierarchie

Jeder Thread besitzt einen privaten Speicher in Form von *Local Memory* und *Registern*. Threads eines Blockes können auf das gleiche *Shared Memory* zugreifen. Dies ermöglicht eine Kooperation zwischen den Threads eines Blockes. Alle Threads haben unabhängig vom Block Zugriff auf das *Global Memory*. Die Zugriffsmöglichkeiten der Threads auf die unterschiedlichen Speicherarten ist in Abbildung 4.3 zu sehen. Es gibt noch zwei weitere Read-Only-Speichertypen: *Constant* und *Texture Memory*. Auf beide Speicher wird global zugegriffen. Sie sind so optimiert, dass sie bei bestimmten Einsatzmöglichkeiten eine bessere Performance als das einfache Global Memory bieten. Texture Memory wird z. B. im Grafikbereich eingesetzt. [16, S. 12 ff.]

Abbildung 4.4 skizziert in welchen Speicherarten Daten abgelegt werden, wenn sie vom Host zum Device kopiert werden und wie die Multiprozessoren auf die Daten zugreifen.

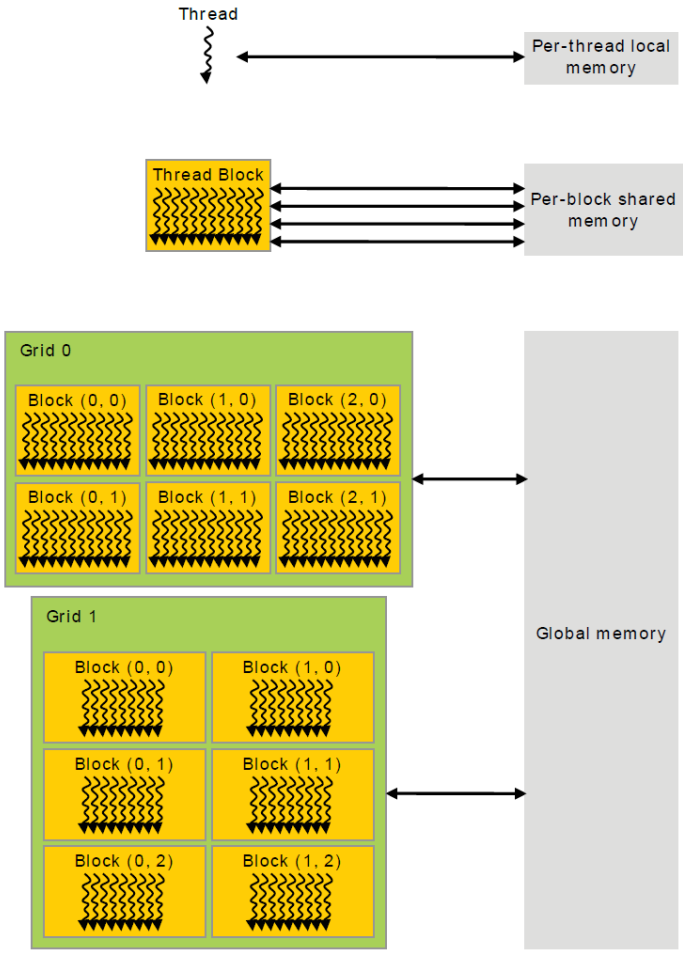


Abbildung 4.3: Zugriff durch die Threads auf die verschiedenen Speicherarten. [16, S. 13]

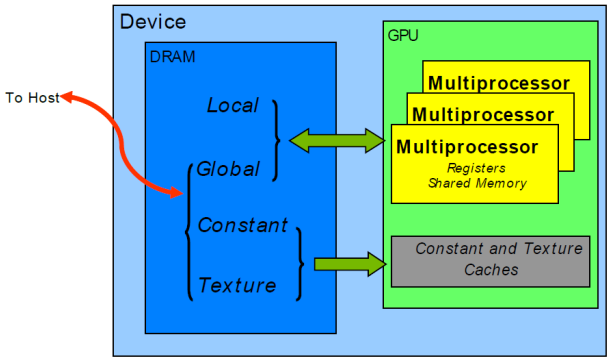


Abbildung 4.4: Anordnung der verschiedenen Speicherarten auf dem Device. [19, S. 27]

Über Qualifier wird bestimmt, wo die Daten auf der Grafikkarte gespeichert werden. Tabelle 4.2 stellt eine Übersicht der verschiedenen Qualifier in CUDA und die Eigenschaften des zugeordneten Speichers dar. Automatische Variablen werden im Register abgelegt, es sei denn es ist nicht mehr genug Speicherplatz vorhanden oder es handelt sich um Arrays. Auf Speichertypen, die sich auf dem Chip befinden, können die Threads schneller zugreifen. Daher sollten Daten, die mehrere Threads eines Blockes benötigen im Shared Memory und nicht im Global Memory abgelegt werden. Benötigen Threads verschiedener Blöcke Zugriff auf gleiche Daten, können diese nur über das Global oder Constant Memory bekannt gemacht werden. [16, S. 89 ff.]

Qualifier	Memory	Zugriff	Lage	Lebenszeit
`int var`	Register	1 Thread	On-Chip	Thread
`int var[10]`	Local	1 Thread	Off-Chip	Thread
`__shared__`	Shared	Alle Threads eines Blockes	On-Chip	Block
`__device__`	Global	Alle Threads, Host	Off-Chip	Anwendung
`__constant__`	Constant	Alle Threads, Host	Off-Chip	Anwendung

Tabelle 4.2: Qualifier von CUDA, die festlegen in welchem Speicher die Daten auf dem Device abgelegt werden.

4.5.2 Speicherzugriffe

Greifen Threads eines Warps auf Daten im Device Memory zu, kann es sein, dass in Abhängigkeit von der Verteilung der Daten im Adress-Raum mehrfache Speicherzugriffe nötig sind. Die Auswirkung der Verteilung der Daten auf den Durchsatz ist dabei von den verschiedenen Speicherarten abhängig.

Global Memory befindet sich nicht auf dem Chip, Speicherzugriffe werden allerdings im L2 gecached. Der Zugriff der Threads eines Warps auf globalen Speicher kann zu einer oder mehreren Transaktionen im Speicher führen. Abhängig von Größe der zu lesenden Daten und der Verteilung der Speicheradressen der Threads können die Lesevorgänge zusammengefasst werden (*Coalescing*). Je mehr Transaktionen (pro benötigte Daten) erforderlich sind, desto mehr unnötige Daten werden gelesen und der Durchsatz wird reduziert. Für Compute Capability 5 werden Zugriffe auf den globalen Speicher im L2-Cache zwischengespeichert. Aus diesem werden für einen Warp immer Cache-Lines von 32 Byte gelesen. Lesen die Threads eines Warps jeweils 4 Byte, so werden im besten Fall auch nur 128 Byte gelesen. Sind die benötigten Daten

ungünstig im Speicher verteilt, kann es sein, dass pro Warp eine Cache-Line gelesen wird (also insgesamt 1 024 Byte). Um ein zusammenhängendes Lesen zu ermöglichen, sollten Daten, auf die parallel von mehreren Threads zugegriffen werden, zusammenhängend ohne Lücken gespeichert werden. Das bedeutet, dass die Daten nicht logisch pro Thread gruppiert werden sollten, sondern entsprechend der parallelen Zugriffe. Greifen Threads auf das i-te Element eines 2D-Arrays zu, sollte dies über folgende Adressierung erfolgen:

```
offset+i*NUM_THREADS+threadIdx.x
```

Der schnell anwachsende Index ist der Thread-Index und nicht, wie im logischen Kontext i (`offset+threadIdx.x*size+i`). Die gleiche Information für verschiedene Threads wird zusammenhängend gespeichert. Der Speicherzugriff kann optimal umgesetzt werden, wenn die Anzahl der Threads und die Größe des Arrays ein Vielfaches der Warp-Größe ist. Trifft dies nicht auf die Größe des Arrays zu, kann durch Padding auf das nächst größere Vielfache der Warp-Größe der Zugriff auf den Speicher beschleunigt werden. [16, S. 78 ff., 192 ff.]

Local Memory befindet sich, wie auch das Global Memory, nicht auf dem Chip. Daher ist der Zugriff auf lokalen Speicher genau so teuer wie auf globalen. Local Memory wird so verwaltet, dass für aufeinanderfolgende Thread-IDs aufeinanderfolgende Wörter von 32 Bit gelesen werden. Solange Threads eines Warps von der gleichen relativen Adresse lesen, kann der lokale Speicher vollständig zusammenhängend gelesen werden. [16, S. 80 f.]

Shared Memory befindet sich auf dem Chip und zeichnet sich durch eine höhere Bandbreite und eine geringe Latenzzeit im Vergleich zum Local oder Global Memory aus. Shared Memory ist in gleich große Speichereinheiten, sogenannte Bänke, unterteilt, auf die parallel zugegriffen werden kann. Fallen Speicherzugriffe auf die gleiche Bank, kommt es zum Bank-Konflikt und die Zugriffe werden serialisiert. Für Compute Capability 5 besitzt das Shared Memory 32 Bänke, die so organisiert sind, dass aufeinanderfolgende 32-Bit-Wörter in aufeinanderfolgenden Bänken zu finden sind. Der Zugriff durch die Threads eines Warps erzeugt keinen Bank-Konflikt, wenn sie auf eine beliebige Adresse innerhalb des gleichen 32-Bit-Wortes zugreifen. In diesem Fall wird beim lesenden Zugriff das Wort an die Threads per Broadcast übertragen und beim schreibenden Zugriff, schreibt nur ein einziger Thread an jede Adresse (welcher das ist, ist nicht definiert). [16, S. 199]

4.6 Terminologie

Host Der Host ist die CPU, von der Kopiervorgänge von und zum Device und Kernel-Launches angestoßen werden. (Eigentlich ist die CPU ein Bestandteil des Hosts. Hier wird es als Synonym verwendet.)

Device Das Device ist die GPU für parallele Berechnungen auf der eine große Anzahl von Threads parallel ausgeführt werden kann.

Streaming Multiprocessor Ein Streaming Multiprocessor (SM) hat Zugriff auf eine bestimmte Anzahl von Recheneinheiten und Speicher (je nach Compute Capability). Er ist für das Verwalten, das Erstellen, das Scheduling und die Ausführung der Warps zuständig.

Kernel Ein Kernel ist eine Funktion, die parallel von mehreren Threads auf dem Device für jeden Thread des Grids ausgeführt wird.

Grid Ein Grid ist eine Menge von Threads, die einen Kernel ausführen. Es kann bis zu drei Dimensionen haben und ist in Blöcke von Threads unterteilt.

Block Ein Block ist eine Menge von Threads, die von einem Streaming Multiprocessor ausgeführt werden. Threads eines Blockes haben Zugriff auf Shared Memory und können explizit synchronisiert werden.

Thread Ein Thread ist ein Prozess, der den Device-Code parallel ausführt.

Warp Ein Warp ist eine Gruppe von 32 Threads, für die ein Streaming Multiprozessor parallel die gleiche Instruktion ausführt.

Compute Capability Die Compute Capability ist eine Versionsnummer, die die Eigenschaften einer NVIDIA GPU zusammenfasst.

5 GPU-Version des Spurfinders

In diesem Kapitel wird beschrieben, wie Teile der CPU-Version des Spurfinders mit Hilfe einer NVIDIA-Grafikkarte parallelisiert wurden. Der Einsatz einer Grafikkarte wird motiviert und die Vorgehensweise bei der Programmierung mit CUDA C wird erklärt. Das Nutzen einer Grafikkarte bringt bestimmte Einschränkungen mit sich. Die Eigenschaften der eingesetzten Grafikkarte werden untersucht und im Hinblick auf den Spurfinder bewertet. Die Parallelisierung des Spurfinders kann auf zwei Ebenen erfolgen. Das Trackfinding-Verfahren an sich kann parallelisiert werden und es kann wiederum parallel für verschiedene Events ausgeführt werden. Beide Aspekte und ihre Umsetzung werden ausführlich behandelt. Anschließend wird auf die Integration des parallelen Codes in PandaRoot eingegangen. In diesem Kapitel wird nur die performanteste Version des parallelen Codes vorgestellt. Alternativen hierzu und ihre Bewertung folgen in Kapitel 6.

5.1 Motivation und Vorgehensweise

Damit der Spurfinder entsprechend der im $\overline{\text{P}}$ANDA-Experiment anfallenden Eventrate von bis zu 2×10^7 Events/s online Ergebnisse liefern kann, ist eine hohe Rechenleistung notwendig. Diese Rechenleistung wird vor Ort, integriert in die Datenaufnahme und Steuerung des Experimentes benötigt. Der Einsatz von Grafikkarten zu diesem Zweck stellt eine günstige Alternative dar.

Als Test zur Parallelisierung des Spurfinders auf GPUs wird eine GeForce GTX 750 Ti eingesetzt. Dabei handelt es sich um ein leistungsfähiges Einsteiger-Modell, das eine Performance-Steigerung bei geringem Stromverbrauch ermöglicht. Die derzeitigen Kosten für diese Grafikkarte betragen zwischen 100 und 200 Euro [20]. Die Gesamtleistung des Systems kann bei Bedarf zudem durch den Einsatz mehrerer Grafikkarten skaliert werden.

Zur Programmierung auf der Grafikkarte wird CUDA C eingesetzt. Es werden spezifische Eigenschaften der GeForce GTX 750 Ti genutzt. Diese werden parametrisch verwendet, sodass der Code auch auf andere Modelle übertragbar ist. Um die Entwicklung und das Debugging zu erleichtern, wird vorerst außerhalb des Frameworks PandaRoot parallelisiert. Dazu werden die notwendigen Daten aus dem Framework extrahiert und in Dateien geschrieben. Die Dateien werden eingelesen, an das Device übertragen und verarbeitet. Der CUDA-Code

wird zu Beginn komplett unabhängig von PandaRoot entwickelt. Nach Fertigstellung des parallelen Codes wird dieser (soweit es möglich ist) in PandaRoot integriert.

5.2 Gegebenheiten der verwendeten GPU

5.2.1 Einschränkungen durch die GPU

Das CUDA-Toolkit bietet eine Menge von Bibliotheken, die speziell auf die NVIDIA-CUDA-Grafikkarten ausgelegt sind. Mit CUDA C/C++ ist es jedoch nicht möglich die C++ Standard Template Library (STL) zu verwenden. CUDA liefert mit *Thrust* eine parallele C++ Template Library, die an die STL angelehnt ist. Zur Zeit unterstützt Thrust nur Datenstrukturen in Form von Vektoren. Diese erleichtern das Speicher-Management und den Datentransfer zwischen Host und Device. Entsprechend der STL sind Algorithmen wie Sortieren, Transformationen und Reduktionen auf die Vektoren anwendbar. Es ist nicht möglich, die Thrust-Algorithmen direkt innerhalb eines Kernels zu benutzen. Die entsprechenden Funktionen können nur vom Host aufgerufen werden. [21]

Dies schränkt den Gebrauch für das Spurfinde-Verfahren stark ein, da z. B. die Summenbildung beim Generieren der Zustände nicht auf dem Device angestoßen werden kann. Die CPU-Version des Spurfinders nutzt neben Vektoren weitere dynamische Datenstrukturen wie Maps und Sets der STL. Diese Datenstrukturen können auf dem Device nicht benutzt werden. Die Daten müssen linearisiert und auf dem Device muss Speicherplatz einer gegebenen Größe alloziert werden.

Unter Berücksichtigung bestimmter Einschränkungen ist es möglich auf dem Device objektorientiert mit C++ zu programmieren. PandaRoot ist ein objektorientiertes Framework. Aufgrund der hohen Komplexität des Frameworks und der vom Spurfinder genutzten Klassen erscheint es sinnvoll, das objektorientierte Design an einer geeigneten Stelle zu verlassen und auf die Nutzung von Objekten auf dem Device zu verzichten. Das bedeutet, dass alle nötigen Datenobjekte/-strukturen und Beziehungen zwischen den Daten in eine für die GPU-Programmierung geeignete Form überführt (meist durch Linearisierung) und die Algorithmen an die spezifischen Fähigkeiten der GPU angepasst werden müssen.

5.2.2 Eigenschaften der GeForce GTX 750 Ti

Die genaue Eigenschaften der verwendeten Grafikkarten sind in Tabelle 5.1 aufgelistet. Die Eigenschaften wurden mit Hilfe von CUDA-Funktionen ausgelesen und der Beschreibung der Architektur für Compute Capability 5 entnommen. Beim Aufruf eines Kernels auf dieser GPU werden die Blöcke an 5

Streaming Multiprozessoren (SM) aufgeteilt, die jeweils 128 Rechenkerne benutzen können. Insgesamt können sich im Idealfall 10 240 Threads auf dem Device befinden (5 SM×64 Warps×32 Threads). Das ist die maximale Anzahl an Threads für die die Ressourcen auf dem Chip zur Verfügung gestellt werden können. Die Anzahl der tatsächlich ansässigen Threads ist durch die Speicherkapazitäten pro Thread/Block/SM begrenzt. Die Anzahl der davon parallel ausgeführten Threads weicht wiederum ab, da pro SM über den Warp-Scheduler nur ein Teil der ansässigen Warps ausgeführt wird. Beim Aufruf eines Kernels, der mehr Speicher pro Thread/Block/SM in Anspruch nimmt als zur Verfügung gestellt wird, schlägt der Kernel-Launch fehl.

Eigenschaft	**Wert**
Compute Capability	5.0
Anzahl von Multiprozessoren	5
CUDA Cores pro Multiprozessor	128
Warps pro Multiprozessor	64
Warp Scheduler pro Multiprozessor	4
Größe der Warps	32
Blöcke pro Multiprozessor	32
Shared Memory pro Multiprozessor	64 KB
Threads pro Multiprozessor	2 048
Local Memory pro Thread	512 KB
Global Memory	2 147 155 968 Byte
Constant Memory	65 536 Byte
Shared Memory pro Block	49 152 Byte
Max. Anzahl von Threads pro Block	1 024
Max. Größe der Dimensionen eines Blocks	1 024 × 1 024 × 64
Max. Größe der Dimensionen des Grids	2 147 483 647 × 65 535 × 65 535
Nebenläufigkeit eines Datentransfers und Kernels	möglich
Nebenläufigkeit verschiedener Kernel	möglich
Nebenläufigkeit verschiedener Datentransfers	nicht möglich

Tabelle 5.1: Eigenschaften der NVIDIA-Grafikkarte GeForce GTX 750 Ti .

In der Tabelle sind außerdem die maximalen Größen der Dimensionen der Blöcke und des Grids zu sehen. Die maximale Anzahl von 1 024 Threads pro Block muss entsprechend auf die Dimensionen aufgeteilt werden. Eine gültige Block-Dimension wäre z. B. 16 × 8 × 8.

Die verwendete Grafikkarte verfügt nur über eine Copy-Engine. Das bedeutet, dass Kopier-Vorgänge nicht parallel ausgeführt werden.

5.3 Parallelisierung auf Algorithmus-Ebene

Zur Beschleunigung des Spurfinders werden die rechenzeitintensivsten Funktionen `EvaluateState()` und `EvaluateMultiState()` parallelisiert. Diese Funktionen verwenden aufgrund der variierenden Anzahl von STT-Hits pro Event Datenstrukturen in Form von Maps, Sets und Vektoren. Diese Datenstrukturen werden linearisiert, sodass sie auf das Device übertragen und dort verarbeitet werden können. Für die oben genannten Funktionen werden entsprechende Kernel implementiert. Zudem wird eine weitere Kernel-Funktion entworfen, die aus den STT-Hits die nötigen Informationen über Hit-Nachbarn extrahiert. Die Kernel werden so implementiert, dass die Berechnungen parallel für die Hits eines Events erfolgen.

5.3.1 Konzeption des parallelen Codes

Die grundlegende Idee ist, dass ein Thread die Daten für einen Hit bzw. eine Zelle des zellulären Automaten berechnet. Für ein Event wird ein Block mit der entsprechenden Anzahl von Threads gestartet. Es wurden folgende drei Kernel implementiert:

`ExtractData()` Diese Funktion erhält als Input die Nachbarschaften der Straw Tubes und die STT-Hits. Für jeden Hit werden parallel die Hit-Nachbarn bestimmt und der Start-Zustand initialisiert. Jeder Thread durchläuft für den jeweiligen Hit die benachbarten Tubes und prüft, ob es sich dabei um einen Hit handelt.

`EvaluateStates()` Mit Hilfe der Hit-Nachbarn werden die Zustände der eindeutigen Hits (weniger als drei Hit-Nachbarn) generiert. Jeder Thread eines eindeutigen Hits adaptiert so lange das Minimum, bis sich kein Zustand innerhalb des Blocks ändert. Die Ausführung des Kernels ist erst beendet, wenn alle endgültigen Zustände ermittelt wurden.

`EvaluateMultiStates()` Mit Hilfe der Hit-Nachbarn werden die Zustände der mehrdeutigen Hits (mehr als zwei Hit-Nachbarn) generiert. Auch dieser Kernel wird erst beendet, wenn die endgültigen Zustände berechnet wurden.

Abbildung 5.1 zeigt den wesentlichen Programmablauf und die Kommunikation zwischen Host und Device. Durch das Kopieren der Daten auf das Device (und zurück) ein Zeit-Overhead. Wird der Spurfinder mehrfach hintereinander ausgeführt, sollten Daten, die für jedes Event gleich sind, nur einmal übertragen werden. So wird der Overhead gering gehalten. Daher wird eine Unterscheidung zwischen *statischen* und *dynamischen* Daten getroffen. Bei den statischen Daten handelt es sich um die Eigenschaften der Straw Tubes. Dazu zählen die Anzahl der Straw Tubes und die Anzahl ihrer Nachbarn. Diese

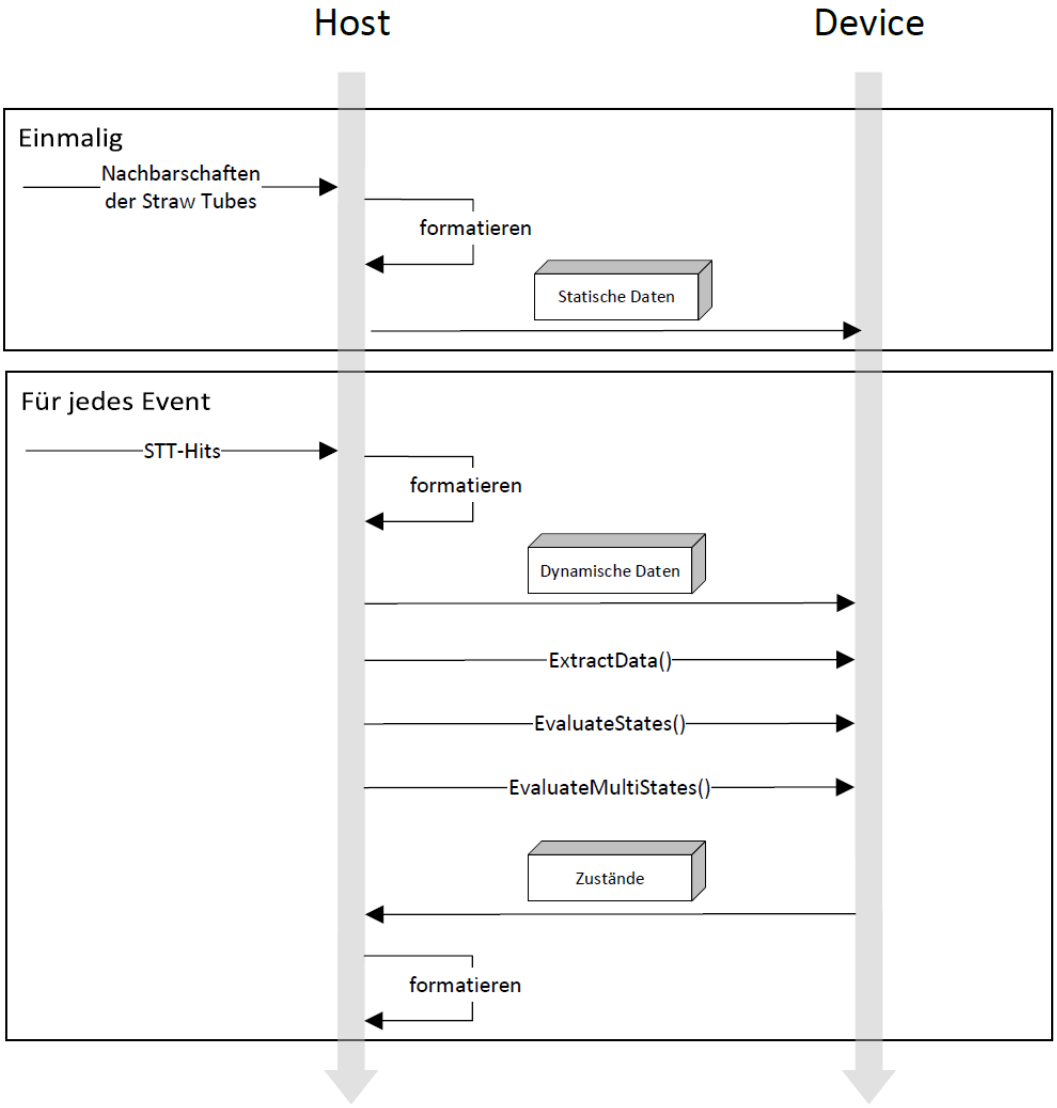


Abbildung 5.1: Übersicht des Kontroll- und Datenflusses zwischen Host und Device zum Generieren der Zustände.

Informationen müssen nur einmal vor der Ausführung des Spurfinders zum Device kopiert werden. Die STT-Hits sind dagegen immer unterschiedlich und müssen pro Event an die Grafikkarte übertragen werden – sie sind dynamisch.

5.3.2 Überführung der dynamischen Datenstrukturen in lineare Arrays

Sämtliche dynamische Datenstrukturen wie Maps und Sets wurden aufgebrochen und in Form von Arrays gespeichert. Dabei muss eine Balance zwischen dem verwendeten Speicherplatz und dem Mehraufwand zur Nachbildung der

Beziehungen, die in den ursprünglichen Daten gespeichert werden, gefunden werden. Die variierende Anzahl der Hits pro Event erschwert die Aufgabe, da auf dem Device Speicher einer definierten Größe alloziert werden muss.

Nachbarschaften der Tubes

Für jede Tube muss bekannt sein, welche und wie viele Nachbarn sie hat. Über ein Objekt der PandaRoot-Klasse `PndSttGeometryMap` wurden 4 542 Tubes und ihre Nachbarn ausgelesen. Diese Informationen müssen vom Host in einem geeigneten Format gespeichert und dann zum Device übertragen werden.

Die Anzahl der Nachbarn pro Tube ist sehr unterschiedlich. Gedrehte Tubes und Tubes, die an diese angrenzen, besitzen zwischen 3 und 22 Nachbarn. Dies trifft auf 2 102 Tubes zu. Die restlichen 2 440 Tubes haben nur 2 bis 6 Nachbarn. Die Informationen werden in einem 1D-Array (`tubeNeighborings`) gespeichert, für welches definiert sein muss, wo sich die Informationen für eine Tube befinden und wie viele Einträge (also Nachbarn) vorhanden sind. Die Struktur des Arrays ist in Abbildung 5.2 dargestellt.

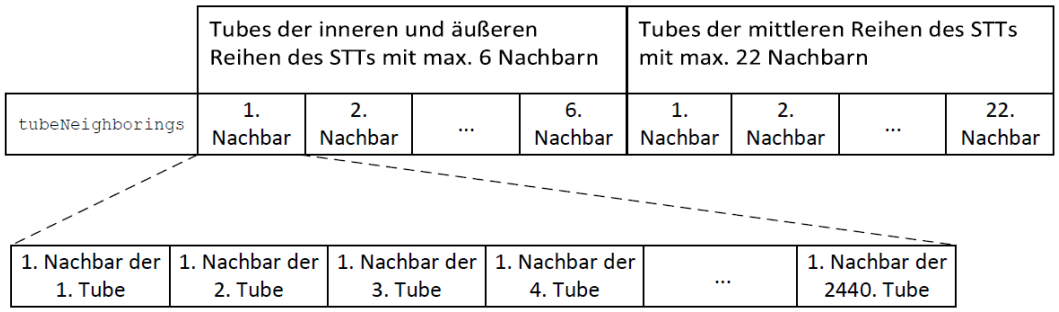


Abbildung 5.2: Speichern der Nachbarschaften aller Tubes in einem Array.

Um `tubeNeighborings` eine gleichmäßige Struktur zu geben, wird die Anzahl der Nachbarn auf 6 (für Tubes mit 2-6 Nachbarn) oder 22 (für Tubes mit 3-22 Nachbarn) festgelegt. Falls weniger Nachbarn existieren, wird dies durch den Eintrag einer 0 signalisiert, da die gültigen Tube-IDs erst bei 1 beginnen. Damit ergeben sich insgesamt 60 884 Einträge ($6 * 2\,440 + 22 * 2\,102$). Durch die Unterscheidung in „Tubes mit wenigen Nachbarn“ und „Tubes mit vielen Nachbarn“ wird Speicherplatz eingespart. Für die Tubes mit maximal 6 Nachbarn werden statt 52 680 Einträgen ($2\,440 * 22$) nur 14 640 ($2\,440 * 6$) Einträge gespeichert.

In `tubeNeighborings` werden als erstes alle Einträge für Tubes mit potentiellen 6 Nachbarn und danach die Nachbarn der Tubes mit maximal 22 Nachbarn gespeichert. Anhand der Tube-ID kann entschieden werden, ob auf den hinteren oder vorderen Teil des Arrays zugegriffen werden muss. Die Tube-IDs beginnen im innersten Ring des STTs mit 1 und werden nach außen größer. Die Nachbarschafts-Informationen für die mittleren Reihen werden am Ende

von `tubeNeighborings` gespeichert, da sich dort die gedrehten Straw Tubes mit vielen potenziellen Nachbarn befinden. Für die äußeren Reihen stehen die Daten in der Mitte des Arrays. Auf diese Weise kann die Information über die Nachbarschaften in einem gemeinsamen Array gespeichert werden.

Die Nachbarn werden nicht logisch pro Straw Tube gruppiert, sondern im Sinne des Zugriffs durch die Threads. Threads werden gleichzeitig auf den i-ten Nachbarn von verschiedenen Tubes zugreifen. Aus diesem Grund werden alle 1. Nachbarn zusammenhängend gespeichert, dann alle 2. u. s. w.

Listing 5.1 zeigt die Berechnung eines absoluten Indexes für `tubeNeighborings` anhand der Tube-ID. In der Implementierung wurden sämtliche Angaben über die Anzahl von Tubes und ihrer Nachbarn mit Präprozessor-Makros definiert, sodass sie leicht angepasst werden können. Mit den Makros in Zeile 6 und 7 wird der Bereich der Straw Tubes definiert, die gedreht sind oder an gedrehte Tubes angrenzen (also viele Nachbarn haben).

Listing 5.1: Zugriff auf die linearisierten Nachbarschaftsbeziehungen.

```
1  //Macros fuer moegliche Anpassung der Anzahl von Tubes/Nachbarn
2  #define NUM_STRAWS 4542
3  #define MAX_SKEWED_NEIGHBORS 22
4  #define MAX_UNSKEWED_NEIGHBORS 6
5  #define START_TUBE_ID_SKEWED 855
6  #define END_TUBE_ID_SKEWED 2956
7  #define NUM_SKEWED_STRAWS (END_TUBE_ID_SKEWED-
       START_TUBE_ID_SKEWED+1)
8  #define NUM_UNSKEWED_STRAWS (NUM_STRAWS-NUM_SKEWED_STRAWS)
9
10 //Array zur Speicherung der Nachbarschaften
11 int tubeNeighborings[NUM_SKEWED_STRAWS*MAX_SKEWED_NEIGHBORS+
       NUM_UNSKEWED_STRAWS*MAX_UNSKEWED_NEIGHBORS]={ 0 };
12
13 //Berechnung Indices fuer den i-ten Nachbarn der Straw Tube
       Tube-ID
14 if(tubeID<START_TUBE_ID_SKEWED){
15   //Tube der inneren Reihen --> liegt am Anfang von
         tubeNeighborings
16   index=(i*NUM_UNSKEWED_STRAWS)+(tubeID-1);
17 } else if(tubeID>END_TUBE_ID_SKEWED){
18   //Tube der aeusseren Reihen --> liegt in der Mitte von
         tubeNeighborings
19     index=(i*NUM_UNSKEWED_STRAWS)+(tubeID-1-(END_TUBE_ID_SKEWED
           -START_TUBE_ID_SKEWED+1));
20 } else{
21   //Tube der inneren Reihen --> liegt am Ende von
         tubeNeighborings
22   index=(i*NUM_SKEWED_STRAWS)+(tubeID-START_TUBE_ID_SKEWED+(
         NUM_UNSKEWED_STRAWS*MAX_UNSKEWED_NEIGHBORS));
23 }
```

Hits

Die Hits werden in einem Array gespeichert. Sie werden vom Host in das entsprechende Format überführt und zum Device kopiert. Es wird davon ausgegangen, dass es keine mehrfachen Hits pro Straw Tube gibt. In der CPU-Version werden die Daten einfach überschrieben. Für die parallele Implementierung werden mehrfache Hits pro Tube ausgeschlossen. Sie werden bis auf einen Hit verworfen.

Die Hits werden über den Eintrag der Tube-IDs der entsprechenden Straw Tubes in `sttHits` gespeichert. Gemäß der Strukturierung in `tubeNeighborings` werden erst Hits mit maximal 6 möglichen Nachbarn gespeichert und anschließend die Hits mit mehr Nachbarn. Hintergrund dessen ist, dass Hits von Warps durch 32 parallele Threads verarbeitet werden für die gemeinsam auf den Speicher zugegriffen wird. Liegen die Daten für diese Hits hintereinander im Speicher, kann der Durchsatz verbessert werden (siehe Abschnitt 4.5.2). Da die Nachbarschaften gleich strukturiert sind, ist eine gleichartige Sortierung der Hits von Vorteil. Die initiale Größe des Arrays beträgt 4 542 (Anzahl der Tubes im STT). Pro Event treten deutlich weniger Hits auf und auch nur diese werden an das Device übertragen.

Hit-Nachbarn

Auf dem Device werden für jeden Hit mittels Zugriff auf die Nachbarschaften in `tubeNeighborings` die Hit-Nachbarn ermittelt. Die Hit-Nachbarn werden im Array `hitNeighbors` gespeichert. Dabei wird nicht die Tube-ID eines Nachbarn gespeichert, sondern der Index der Tube in `sttHits`. Das hat den Vorteil, dass beim Zugriff auf linearisierte Datenstrukturen der Tubes direkt über diesen Index zugegriffen werden kann. Bei Verwendung von Tube-IDs müsste stets ein Shift von `-1` erfolgen, da die Tube-IDs bei 1 beginnen und die Indices bei 0. Die Nachbarn der Hits werden in der gleichen Reihenfolge gespeichert wie in `sttHits`. Wie in `tubeNeighborings` ist die Anzahl der Nachbarn der ersten eingetragenen Hits maximal 6, für die restlichen maximal 22. Um beim Zugriff auf `hitNeighbors` den richtigen Index zu ermitteln, werden die Anzahl der Hits mit 6 möglichen Nachbarn und die Anzahl der restlichen Hits benötigt. Die Berechnung des Indexes innerhalb von `hitNeighbors` ergibt sich nicht aus der Tube-ID sondern aus dem zugehörigen Index in `sttHits`.

Zustände

Die Zustände der eindeutigen und mehrdeutigen Zellen werden auf dem Device ermittelt und in den Arrays `states` und `multiStates` gespeichert. In `states` wird zu einem Hit direkt die zugehörige Tube-ID vermerkt. In `multiStates` dagegen, werden anstelle der Zustände die Indizes (in Bezug

auf `sttHits`) der angrenzenden eindeutigen Hits gespeichert. Die Motivation zur Speicherung der Indizes wird in Abschnitt 5.3.5 erläutert.

In beiden Arrays ist für jeden Hit ein Eintrag für den Zustand/die Zustände vorgesehen. Das bedeutet, dass mehr Speicher als eigentlich benötigt alloziert wird. Anhand der Anzahl Hit-Nachbarn könnte entschieden werden, ob ein eindeutiger Zustand (maximal 2 Hit-Nachbarn) oder Multi-Zustand (mehr als 2 Hit-Nachbarn) generiert wird. Würden die Daten aufgeteilt werden, müssten noch zusätzliche Informationen über diese Verteilung gespeichert werden. Durch das Verteilen der Daten geht die ursprüngliche bekannte Anordnung verloren und die Indizes müssten neu berechnet bzw. zusätzlich gespeichert werden. Dadurch würde ein zusätzlicher Computing-Overhead entstehen, der durch das Nutzen von mehr Speicher umgangen wird. Um die Indizierung beizubehalten, wird für jeden Hit ein eindeutiger und ein Multi-Zustand vorgesehen. Für einen Hit an Position `i` in `sttHits` wird die Zustands-Information als `i`-ter Eintrag in `states` bzw. `multiStates` gespeichert.

Für die Speicherung der Zustände eindeutiger Zellen muss jeweils nur für einen einzigen Eintrag Speicherplatz angefordert werden. Die Arrays `sttHits` und `states` haben die gleiche Größe. Die Größe von `multiStates` ist dagegen flexibel und schwierig festzulegen. Die maximale Anzahl der Multi-Zustände für einen Hit wird erst während der Laufzeit des Programms festgelegt. Die Anzahl ist davon abhängig, wie viele eindeutige Zellen an eine zusammenhängende Menge von mehrdeutigen Zellen angrenzen. Diese flexible Anzahl stellt ein Problem dar, da auf dem Device für die Multi-Zustände Speicherplatz einer definierten Größe alloziert werden muss. Um eine obere Grenze festzulegen, wurden die Multi-Zustände von 1 000 Events bezüglich der maximalen Anzahl von Einträgen untersucht. Das Ergebnis ist in Tabelle 5.2 zu sehen.

Anzahl der Einträge in den Multi-Zuständen	0-5	6-10	11-15	16-20	≥21
Anteil der Events	82.4 %	16 %	1.2 %	0.3 %	0.1 %

Tabelle 5.2: Anteil der Events bezüglich einer maximalen Anzahl von Einträgen in den Multi-Zuständen.

Darauf basierend wurde die maximale Anzahl der Einträge in `multiStates` pro Hit auf 20 festgelegt. Damit können die Multi-Zustände für 99.9 % der simulierten Events ordnungsgemäß gespeichert werden. Für nur eines der 1 000 Events wurden mehr als 20 Einträge in den Multi-Zuständen generiert. Für dieses Event wird das Generieren der Multi-Zustände vorzeitig beendet, wenn die maximale Anzahl erreicht ist. Dabei handelt es sich um ein Event mit stark kreisenden Flugbahnen, deren Rekonstruktion mit dem Spurfinder ohnehin nur begrenzt möglich ist.

`multiStates` besitzt 20 mal so viele Einträge wie `sttHits`. Die Zustände werden Thread-orientiert gespeichert. Zuerst wird der 1. Eintrag im Multi-Zustand aller Hits gespeichert, dann der 2. u. s. w.

5.3.3 Extrahieren der Hit-Nachbarn

Das Ermitteln der Hit-Nachbarn wird im Kernel `ExtractData()` implementiert. Für jeden STT-Hit wird die Anzahl der Nachbarn bestimmt, die ebenfalls einen Hit signalisiert haben. Dies ermöglicht eine Unterscheidung zwischen eindeutigen und mehrdeutigen Zellen und bildet die Grundlage für das Generieren der Zustände. Die Informationen werden für jeden Hit parallel von verschiedenen Threads ermittelt. Ein Thread mit dem Index `i` berechnet die Hit-Nachbarn für die an Position `sttHit[i]` gespeicherte Tube-ID. Dementsprechend ist die maximale Anzahl aktiver Threads gleich der Anzahl der Einträge in `sttHits`. Ein Thread benötigt Zugriff auf

- **`sttHits`**, um die Tube-ID eines Hits zu ermitteln,
- **`tubeNeighborings`**, um die Nachbarn für die Tube eines Hits auszulesen,
- **`numUnskewedHits`** und **`numSkewedHits`**, um den Index in `tubeNeighborings` und `hitNeighbors` zu berechnen, und
- **`hitIndices`**, um leichter überprüfen zu können, ob es sich bei den Nachbarn um Hits handelt.

Bei `hitIndices` handelt es sich um zusätzliche Informationen, die übertragen werden, um das Verfahren zu beschleunigen. `hitIndices` speichert an der Position `tubeID` den Index des zugehörigen Hits in `sttHits`. Das Array wird mit `-1` initialisiert, was signalisiert, dass für die Tube kein Hit verfügbar ist. Für jeden Hit werden die Nachbarn ausgelesen. In einer Schleife über die 6 oder 22 möglichen Nachbarn werden die zugehörigen Einträge in `hitIndices` geprüft. Um `0`-Einträge in `tubeNeighborings` gleich behandeln zu können, wird in `hitIndices` ein zusätzlicher Wert an Position `0` gespeichert. Die Indizes sämtlicher Nachbar-Tubes können wie folgt ausgelesen werden:

```
hitIndices[tubeNeighborings[Index i-ter Nachbar]]
```

Jeder Thread speichert die Indizes der Hit-Nachbarn und die Anzahl der Hit-Nachbarn und ermittelt die Start-Zustände für die Hits/Zellen. Durch das Initialisieren der Start-Zustände an dieser Stelle erübrigt sich die Ausführung eines weiteren Kernels. Die Daten werden an die Position in den Arrays geschrieben, die sich pro Block eindeutig aus dem Thread-Index ergibt. Auf diese Weise werden folgende Arrays parallel von verschiedenen Threads gefüllt:

- **`hitNeighbors`** enthält die Hit-Nachbarn für Hit-Tubes mit maximal 6 möglichen Nachbarn, gefolgt von Einträgen für Hit-Tubes mit maximal 22 Nachbarn (siehe Abschnitt 5.3.2).

- **`numHitNeighbors`** enthält an Position `i` die Anzahl der Hit-Nachbarn für die Tube in `sttHits[i]`. Mit Hilfe dieser Information kann beim Generieren der Zustände leicht überprüft werden, ob es sich um eine eindeutige oder mehrdeutige Zelle handelt.

- **`states`** enthält für eindeutige Hits (maximal 2 Hit-Nachbarn) die Tube-ID und für mehrdeutige Hits die Zahl `4543` (Anzahl der Straw Tubes im STT + 1). Dies verschafft einen Vorteil beim Generieren der Zustände, der in Abschnitt 5.3.4 beschrieben wird.

Nachdem der Kernel ausgeführt wurde, müssen keine Daten vom Device zum Host kopiert werden. Die Hit-Nachbarn müssen nicht auf der Host-Seite bekannt sein. Auf diese Information wird nur innerhalb der anderen Kernel zugegriffen. Die Zustände in `states` werden nur initialisiert. Auch diese Information wird auf der Host-Seite nicht benötigt. Daher entfällt das Kopieren und es werden lediglich die Device-Pointer an die Kernel weitergegeben, die die Zustände generieren.

5.3.4 Generieren der Zustände

Der Kernel `EvaluateStates()` implementiert das Generieren der Zustände für eindeutige Zellen. Es sind nur die Threads aktiv, deren Thread-ID kleiner ist als die Anzahl der eindeutigen Hits. Jeder Thread durchläuft die zugehörigen Hit-Nachbarn und ermittelt aus seinem Zustand und den Zuständen der Hit-Nachbarn das Minimum. Dabei werden unter Einbeziehung der Werte aus `numHitNeighbors` auch nur die vorhanden Hit-Nachbarn überprüft. Anschließend setzt jeder Thread den Zustand der entsprechenden Zellen auf das so ermittelte Minimum. In der CPU-Version wird eine Unterscheidung zwischen eindeutigen und mehrdeutigen Nachbarn gemacht. Nur die Zustände eindeutiger Nachbarn werden in die Evaluation der Zustände einbezogen. Um eine derartige Sonderbehandlung zu umgehen, werden die Zustände mehrdeutiger Nachbarn in der GPU-Version mit `4543` initialisiert. Da das Minimum adaptiert wird und die größte gültige Tube-ID `4542` ist, haben die Zustände der mehrdeutigen Hit-Nachbarn keinen Einfluss auf das Ergebnis. Sie können wie eindeutige Hit-Nachbarn behandelt werden und es ist keine Verzweigung nötig.

Die neu ermittelten Zustände werden ohne Synchronisations-Maßnahmen direkt in `states` geschrieben. Durch die parallele Ausführung der Threads werden sie daher nicht mehr synchron (wie in der CPU-Version) geändert.

Das bedeutet, dass für eine Zelle bereits aktualisierte Zustände der Nachbar-Zellen gelesen werden können. Ist dies der Fall, wird für die Zelle der finale Zustand möglicherweise mit weniger Iterationsschritten generiert als in der CPU-Version.

Die Zustände werden solange geändert, bis keine Änderungen mehr an `states` durch die verschiedenen Threads vorgenommen werden. Zur Überprüfung der Abbruchbedingung werden nicht, wie in der CPU-Version, die Zustände summiert und die Summen aufeinanderfolgender Iterationen verglichen. Stattdessen erhöht jeder Thread einen lokalen Zähler um `1`, falls der Zustand geändert wurde, andernfalls behält dieser den initialen Wert `0`. Die Zähler aller Threads werden aufsummiert. Ist die Summe `0`, ist das Generieren der Zustände abgeschlossen. Das zugehörige Struktogramm ist in Abbildung 5.3 zu sehen.

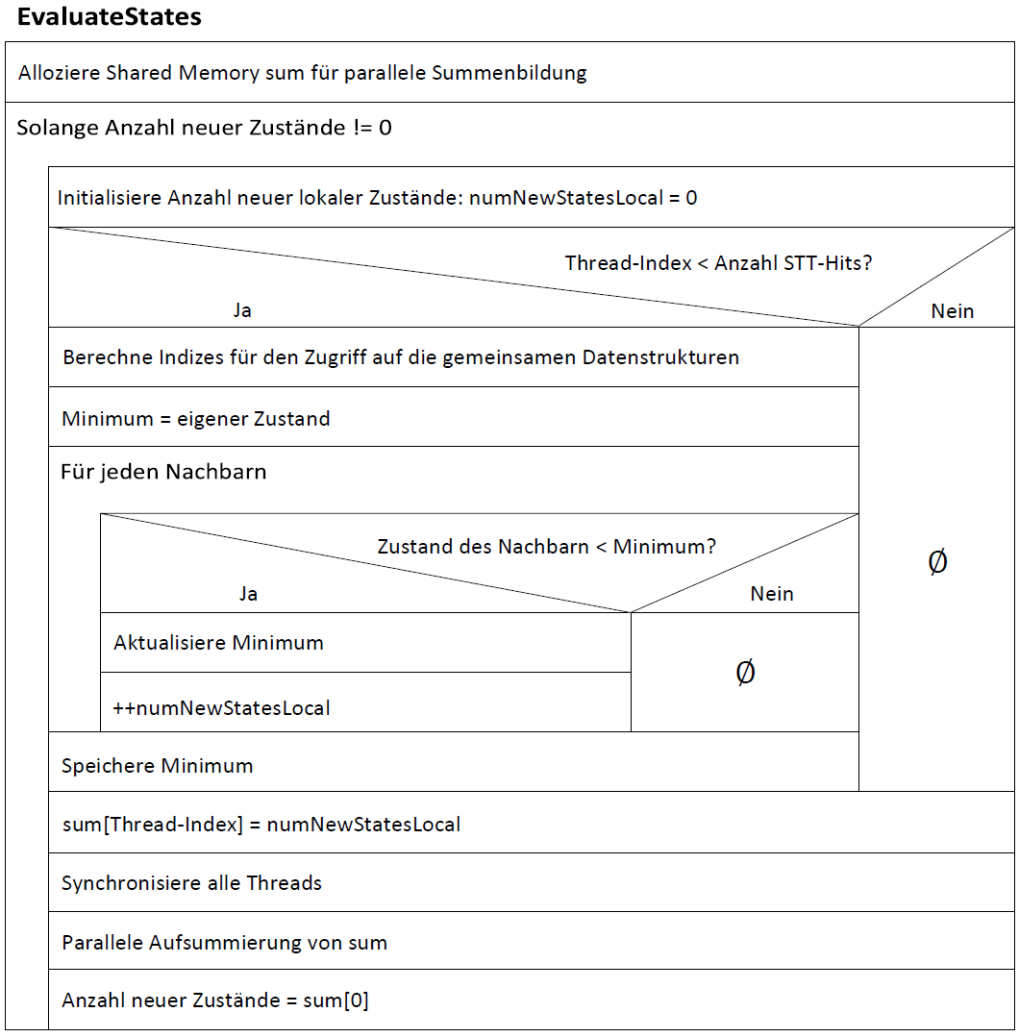


Abbildung 5.3: Struktogramm des Kernels `EvaluateStates()`.

Ein Thread muss Zugriff auf die Zähler der anderen Threads haben, um die Abbruchbedingung zu überprüfen. Daher werden die Zähler in einem Array (`sum`) im Shared Memory abgelegt. Jeder Thread schreibt den Wert seines Zählers an eine jeweils andere Position. Bevor aus dem Shared Memory gelesen wird, müssen die Threads synchronisiert werden. So wird sichergestellt, dass jeder Thread den endgültigen Wert des Zählers in `sum` abgelegt hat. Die Summe der Werte in `sum` berechnet nicht jeder Thread für sich allein. Die Berechnung wird parallel vorgenommen und aufgeteilt. Haben `N` Threads ihre Zähler in `sum` abgelegt, so berechnen die ersten `N/2` Threads die Summe aus `sum[Thread-Index]` und `sum[Thread-Index + N/2]`. Der Eintrag an Stelle `sum[Thread-Index]` wird durch die Summe ersetzt. Die Anzahl der aktiven Threads halbiert sich dann von Schritt zu Schritt, bis in `sum[0]` die endgültige Summe steht. Voraussetzung dafür ist, dass der Kernel mit einer Thread-Anzahl gestartet wurde, die einer Potenz von zwei entspricht. Mit Listing 5.2 kann das Verfahren nachvollzogen werden.

Listing 5.2: Nutzen von Shared Memory zur Summenbildung und Überprüfung der Abbruchbedingung.

```
1  //Shared Memory fuer Ueberpruefung der Abbruchbedingung
2  __shared__ int sum[MAX_THREADS_PER_BLOCK];
3  int numNewStatesFinal=1;
4  int numNewStatesLocal;
5
6  while(numNewStatesFinal!=0){
7    numNewStatesLocal=0;
8    //bestimme (und setze) neuen Zustand, setze bei Veraenderung
         numNewStatesLocal
9    ...
10
11   sum[threadIdx.x]=numNewStatesLocal;
12   //mache Counter fuer alle Threads sichtbar
13   __syncthreads();
14
15   //paralleles Aufsummieren
16   int offset=MAX_THREADS_PER_BLOCK/2;
17   while(offset!=0){
18     if(threadIdx.x<offset){
19       sum[threadIdx.x]+=sum[threadIdx.x+offset];
20     }
21     __syncthreads();
22     offset/=2;
23     }
24
25   numNewStatesFinal=sum[0];
26   //mache Summe fuer alle Threads sichtbar
27   __syncthreads();
28 }
```

5.3.5 Generieren der Multi-Zustände

Der Kernel `EvaluateMultiStates()` generiert die Zustände für mehrdeutige Zellen. Es handelt sich um die zweite Stufe des zellulären Automaten. Da die Berechnung nur für mehrdeutige Hits vorgenommen wird, sind auch hier weniger Threads (im Vergleich zur Gesamtanzahl der Hits) aktiv. Für jeden Hit mit mehr als zwei aktiven Nachbarn werden die Hit-Nachbarn aus `hitNeighbors` durchlaufen. Bei dem neuen Multi-Zustand handelt es sich um eine Kopie aller Zustände der angrenzenden Nachbarn.

Die CPU-Version wurde so implementiert, dass diese Stufe erst nach dem Generieren der Zustände für die eindeutigen Hits ausgeführt werden kann. Die Multi-Zustände sind von den endgültigen Zuständen (Track-IDs) der eindeutigen Nachbar-Zellen abhängig. Wird anstelle der Track-IDs ein Verweis auf die zugehörige Tube in Form der Tube-ID gespeichert, können beide Stufen unabhängig voneinander ausgeführt werden. Die Tube-ID ist ein fester Wert, der nicht vom zellulären Automaten verändert wird. Aus diesem Grund ist es möglich, das Generieren der Zustände für eindeutige und mehrdeutige Hits parallel auszuführen. Für die in den Multi-Zuständen gespeicherten Tube-IDs muss nach Beendigung beider Funktionen lediglich der zugehörige Zustand ausgelesen werden.

Anstelle der Tube-IDs wird in `multiStates` der Hit-Index der entsprechenden Tube in `sttHits` vermerkt. Das hat den Vorteil, dass ein Zugriff auf die Daten in `sttHits` aus dem Kernel heraus komplett entfällt. Die Hit-Indizes können aus `hitNeighbors` gelesen werden. Die Multi-Zustände werden mit `-1` initialisiert. Damit wird signalisiert, dass es keinen weitere Eintrag gibt. Um diesen Wert vor der Ausführung des Kernels auf dem Device zu setzen, kann `cudaMemset` benutzt werden. Alternativ können die Werte auf dem Host initialisiert und zum Device kopiert werden.

Die Multi-Zustände werden in der CPU-Version wie folgt generiert. Die Zustände eindeutiger Nachbarn werden aus `states` und die Multi-Zustände mehrdeutiger aus `multiStates`gelesen und in einem neuen temporären Multi-Zustand gespeichert. Wurden alle Nachbarn durchlaufen, wird der alte Multi-Zustand durch den neuen ersetzt. Eine Orientierung an diesem CPU-Code liefert keine akzeptable parallele Implementierung, da die Threads parallel auf die Multi-Zustände zugreifen. Werden die Einträge in den Multi-Zuständen ersetzt, würden ohne zeitintensive Synchronisationsvorgänge inkonsistente Multi-Zustände entstehen. Daher konstruieren die Threads in der GPU-Version keinen neuen Multi-Zustand sondern ergänzen weitere Einträge. Das Struktogramm des Kernels ist in Abbildung 5.4 dargestellt. Es wird nur ein neuer Eintrag in den Multi-Zustand eingefügt, wenn der Wert noch nicht enthalten ist und es weniger als 20 Einträge gibt. Ist dies der Fall, wird der neue Wert an die nächste freie Position (mit dem Wert `-1`) eingetragen. Beim

Ergänzen der Multi-Zustände mehrdeutiger Nachbarn werden nur gültige Einträge übernommen ($\neq -1$).

Durch das Ergänzen ist kein Synchronisieren nötig. Entweder lesen andere Threads schon den neuen erweiterten Multi-Zustand oder den alten. Das Ändern der Zustände erfolgt nicht synchron und kann nach weniger Schleifen-Durchläufen (im Vergleich zur CPU-Version) das endgültige Ergebnis liefern. Es wird eine maximale Anzahl von 20 Einträgen pro Multi-Zustand festgelegt (siehe Abschnitt 5.3.2). Ist diese Anzahl erreicht, wird kein weiterer Eintrag hinzugefügt und der zuständige Thread hat den endgültigen Multi-Zustand generiert. Das Generieren der Multi-Zustände für Events, für die mehr als 20 Einträge möglich sind, wird folglich vorzeitig beendet. Das Überprüfen der Abbruchbedingung erfolgt wie in Abschnitt 5.3.4 beschrieben. Die Threads haben einen Zähler, der erhöht wird, falls ein neuer Eintrag im Multi-Zustand ergänzt wurde. Statt der Zustände werden die Zähler aller Threads durch Nutzen von Shared Memory parallel aufsummiert.

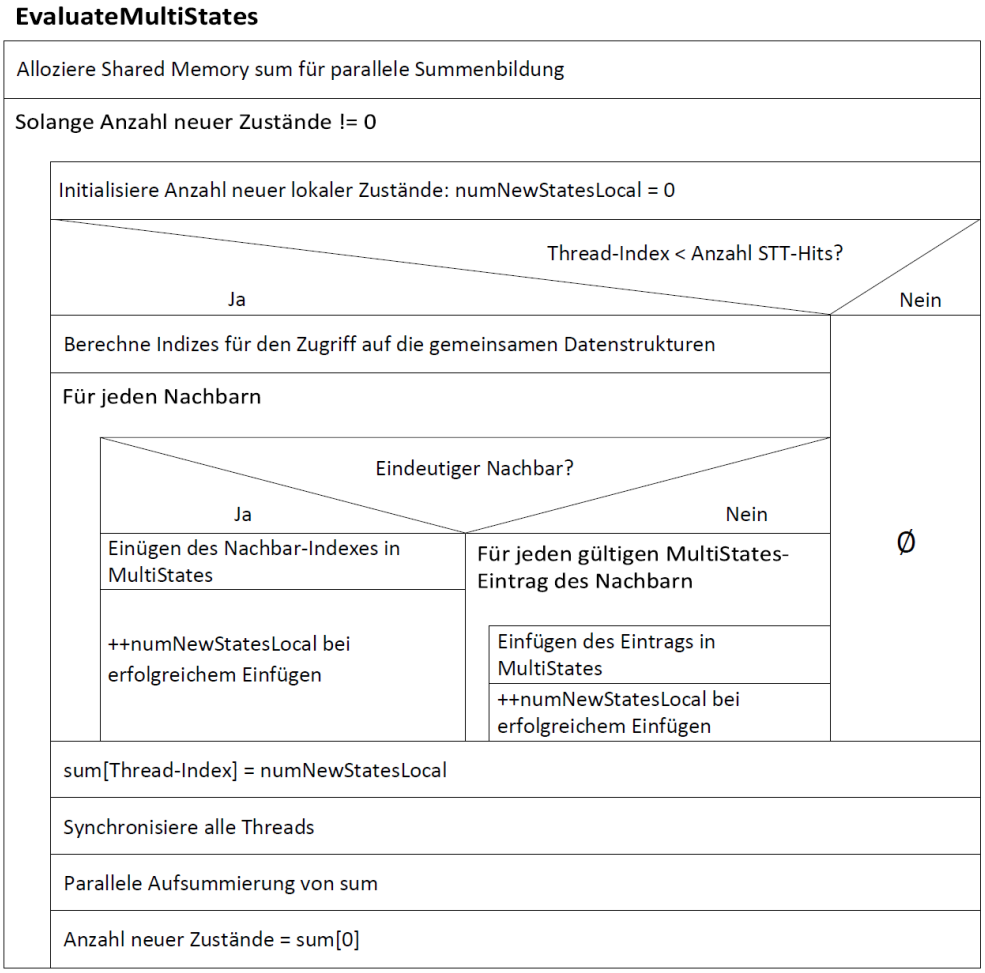


Abbildung 5.4: Struktogramm des Kernels `EvaluateMultiStates()`.

5.3.6 Ausführung und Startkonfiguration der Kernel

Da Abhängigkeiten der Kernel `EvaluateStates()` und `EvaluateMultiStates()` aufgehoben wurden, könnten die Funktionen parallel in zwei verschiedenen Streams ausgeführt werden. Für eine Nebenläufigkeit dieser Streams müssten die Eingabedaten jedem Stream zur Verfügung gestellt werden, da die Streams unabhängig voneinander ausgeführt werden (siehe Abschnitt 4.3.3). Das bedeutet, dass auch `ExtractData()` in beiden Streams ausgeführt werden muss und die STT-Hits zweimal übertragen werden müssen. Wäre der Rechenaufwand der zellulären Automaten im Vergleich zum Kopieraufwand und zur Ausführungszeit von `ExtractData()` sehr hoch, wäre dieser Ansatz vorzuziehen. Da dies aber nicht der Fall ist (siehe Abschnitt 6.2.3), werden die Daten nur einmal synchron kopiert und die Kernel `ExtractData()`, `EvaluateStates()` und `EvaluateMultiStates()` hintereinander im Default-Stream ausgeführt.

Die drei Kernel werden nacheinander für die STT-Hits eines Events ausgeführt. Für ein Event wird ein Block mit der entsprechenden Anzahl an Threads gestartet. Die Anzahl beträgt mindestens 128 und muss aufgrund der Summenbildung eine Potenz von Zwei sein. Dadurch wird eine Einschränkung getroffen, da maximal 1 024 Threads pro Block gestartet werden können. Events mit mehr Hits werden vernachlässigt. Diese Vorgehensweise ist vertretbar, da bei derartig vielen Hits die Nachbarschaften aktiver Straw Tubes sehr dicht sind (vor allem im Bereich der gedrehten Tubes). Je mehr Hit-Nachbarn die Tubes haben, umso schwieriger ist es eindeutige primäre Tracklets zu generieren und das Verfahren stößt an seine Grenzen. Bei mehr als 1 024 Hits müssten die Threads die Daten für mehr als einen Hit berechnen. Nachdem die Threads die Berechnungen für die ersten 1 024 Hits vorgenommen haben, starten sie in einer zweiten Iteration die Berechnungen für die nächsten 1 024 Hits erneut. Die Anzahl aktiver Threads wäre im zweiten Durchlauf sehr gering. Eine alternative Berechnung über die Block-Grenzen hinaus, kann nicht umgesetzt werden. Die Blöcke müssen unabhängig voneinander ausgeführt werden. Sie generieren aber Zustände, die voneinander abhängig sind.

Um zu verifizieren, dass eine maximale Anzahl von 1 024 Hits (Threads) ausreichend ist, wurde für 1 000 Events die Anzahl der STT-Hits untersucht. Die Werte sind in Tabelle 5.3 zu sehen. Für einige Events sind mehr als 1 024 Hits zu verarbeiten. Nachdem mehrfache Hits pro Tube auf nur einen reduziert worden sind, ist die maximale aufgetretene Anzahl auf 580, so dass die Einschränkung auf 1 024 Hits keines der Events betroffen hätte.

Alle Kernel werden mit der gleichen Anzahl an Threads gestartet. Für das Generieren der Zustände sind eigentlich weniger Threads nötig, da in `EvaluateStates()` bzw. `EvaluateMultiStates()` nur Threads für eindeutige bzw. mehrdeutige Hits aktiv sind. Durchschnittlich gibt es pro Event ungefähr 70 eindeutige Hits. Demzufolge würde der Block ohnehin mit mehr

Threads (128) gestartet werden. Außerdem ist für die korrekte Funktionsweise der Summenbildung eine Thread-Anzahl nötig, bei der es sich um eine Potenz von Zwei handelt.

	Durchschnitt	**Minimum**	**Maximum**
Gesamt	110	15	1 270
Ohne mehrfache Hits pro Tube	95	15	580
Nur eindeutige Hits ohne mehrfache Hits pro Tube	67	11	164

Tabelle 5.3: Anzahl der STT-Hits pro Event; gemessen mit den gleichen 1 000 Events wie im Abschnitt 3.3.

5.3.7 Übersicht verwendeter Datenstrukturen

Eigenschaften der Datenstrukturen # – Anzahl, st – Straw Tubes, sth – Tubes mit Hit, st_x – Tubes mit x Nachbarn		
Name **Größe**	**Indexing**	**Besonderheiten**
tubeNeighborings #st_6*6+ #st_22*22	[i*shift+tubeIndex] ≙ i-ter Nachbar	• statische Daten • erst st_6, dann st_22 • kein Nachbar ≙ 0
sttHits #sth	[i] ≙ i-ter Hit	• erst sth_6, dann sth_22 • nur ein Hit pro st
hitIndices #st+1	[tubeID] ≙ Index des Hits in sttHits	• Default: -1 • Eintrag für tubeID=0
hitNeighbors #sth_6*6+ #sth_22*22	[i*shift+hitIndex] ≙ i-ter Hit-Nachbar	• erst sth_6, dann sth_22 • kein Hit-Nachbar ≙ 0 • nur auf Device verfügbar
numHitNeighbors #sth	[i] ≙ #Hit-Nachbarn des i-ten Hits	• entscheidet ob eindeutige/ mehrdeutige Zelle
states #sth	[i] ≙ Zustand des i-ten Hits	• Initialisierung für eindeutige Hits mit Tube-ID • Initialisierung für mehrdeutige Hits mit #st+1
multiStates #sth*20	[i*shift+hitIndex] ≙ i-ter Eintrag im Multi-Zustand eines Hits	• speichert Hit-Indizes statt Tube-IDs/Zustände • 20 Einträge pro Hit • Default: -1

Tabelle 5.4: Eigenschaften der verwendeten Datenstrukturen.

5.3.8 Abweichungen von der CPU-Version

Neben den Änderungen der Datenstrukturen wurden auch Teile des Verfahrens aufgrund der Gegebenheiten der GPU und im Sinne der Optimierung abgewandelt. Es folgt eine Auflistung der wichtigsten Abweichungen von der CPU-Version:

- Als Eingabe wird ein Array mit STT-Hits erwartet, das keine mehrfachen Hits pro Tube enthält. Die Hits müssen bezüglich ihrer maximalen Anzahl von benachbarten Tubes sortiert sein. Im Array sind als erstes Einträge für Hits von Tubes der inneren und äußeren Reihen des STTs enthalten, gefolgt von Hits, die von gedrehten Tubes oder deren Nachbarn signalisiert wurden.

- Beim Generieren der Zustände erfolgt die Änderung der Zustände nicht mehr synchron, da die Threads vor dem Schreiben der neuen Zustände nicht synchronisiert werden. Das bedeutet, dass Threads schon aktualisierte Zustände ihrer Nachbarn lesen können, die in der CPU-Version erst im nächsten Iterationsschritt zur Verfügung stehen würden. Daher ist es möglich, dass die GPU-Version weniger Iterationen bis zu den endgültigen Zuständen durchläuft.

- Um zu Überprüfen, ob sich die Zustände in aufeinanderfolgenden Iterationen nicht mehr ändern, wird die Anzahl der Änderungen untersucht (nicht die Summe der Zustände).

- Die Multi-Zustände können unabhängig von den endgültigen Zuständen der eindeutigen Zellen generiert werden.

- Die Anzahl der Einträge in die Multi-Zustände der Hits ist auf 20 begrenzt, da auf dem Device kein Set benutzt werden kann, das dynamisch erweitert wird. Für Events deren mehrdeutige Hits eigentlich größere Multi-Zustände besitzen, wird die Anpassung der Zustände beim Erreichen von 20 Einträgen abgebrochen. Das bedeutet, dass nicht die Information für alle angrenzenden Tracklets gespeichert wird. Das kann zur Folge haben, dass in den anschließenden Schritten weniger oder auch mehr Kombinationen überprüft werden. Mehr Kombinationsmöglichkeiten kommen zu Stande, wenn aufgrund fehlender Einträge rückläufige Wege nicht erkannt werden.

- Die Multi-Zustände und Hit-Nachbarn werden über Indizes der entsprechenden Hits gespeichert.

- Um mit Hilfe der Multi-Zustände herauszufinden, welche primären Tracklets kombiniert werden könnten, muss zusätzlich der Zustand des Eintrags ausgewertet werden.

5.4 Parallelisierung auf Event-Ebene

Bisher wurde nur auf die Parallelisierung einzelner Funktionen des Spurfinders eingegangen. Es ist zudem möglich, die parallelisierte Version zusätzlich parallel für mehrere Events auf der Grafikkarte auszuführen. Dazu wird ein Kernel mit `N` Blöcken von Threads gestartet, die parallel die Berechnungen für `N` Events vornehmen. Mit Hilfe von Streams kann eine weitere Ebene der Parallelität erzeugt werden. Sie erlauben die überlappende Ausführung von Datentransfers und Kernel und die Nebenläufigkeit mehrerer Kernel. Die Parallelisierung auf Event-Ebene führt zu einer besseren Ausnutzung der GPU-Ressourcen (siehe Abschnitt 6.3) und ist auch im Hinblick auf eine Multi-Device-Architektur besonders wichtig, wie sie an $\overline{\text{P}}$ANDA vorgesehen ist.

5.4.1 Programmablauf

Auf der Host-Seite ist definiert, wie viele Events (`NUM_EVENTS`) mit wie vielen STT-Hits bearbeitet werden sollen. Es werden einmalig Speicherplätze für alle nötigen Datenstrukturen auf dem Device alloziert. Das heißt, für jede Datenstruktur gibt es nur ein Array auf dem Device, dass die Informationen für alle Events speichert.

Die grundlegende Idee ist, dass Streams die Kopier- und Rechenvorgänge für eine Vielzahl von Events vornehmen. Durch das geschickte Füllen der Streams werden diese wiederum parallel mit anderen Streams ausgeführt. Jeder Stream kopiert/berechnet die Daten für eine bestimme Menge von Events. Während ein Stream diese Aufgaben für ein ihm zugewiesenes Datenpaket bearbeitet, können andere Streams Kopier- und Rechenvorgänge für weitere verschiedene Datenpakete vornehmen. Im Code kann die Größe der Gruppe von Events bzw. die Anzahl der Blöcke über das Präprozessor-Makro `NUM_BLOCKS` gesetzt werden. Die Anzahl der zu erstellenden Streams ergibt sich aus `NUM_EVENTS/ NUM_BLOCKS`. Der Einfachheit halber wird davon ausgegangen, dass bei der Division kein Rest entsteht.

Die Kopiervorgänge und Kernel werden asynchron angestoßen und einem Stream zugeordnet. Jeder Stream kopiert einen Bruchteil der STT-Hits aller Events zum Device, führt die Berechnungen auf diesem Bruchteil der Daten aus und füllt den entsprechenden Teil des Ergebnis-Arrays durch Kopieren vom Device zum Host. Der Zugriff der Streams auf die gemeinsamen Datenstrukturen ergibt sich aus dem Stream-Index, aus `NUM_BLOCKS` und der Anzahl der Hits für diese Menge von Events. Mit diesen Werten kann der entsprechende Adressbereich innerhalb des gemeinsamen Arrays berechnet werden für den der Stream zuständig ist. Listing 5.3 zeigt wie die Streams erzeugt und befüllt werden. Die daraus resultierende parallele Ausführung der Streams ist in Abbildung 5.5 zu sehen. Die Überlappung der Streams hängt von der Block-Anzahl, die pro Stream bearbeitet wird, ab und wird in Kapitel 6 diskutiert.

Listing 5.3: Erstellen und Füllen der Streams zur parallelen Bearbeitung von Gruppen von Events.

```
1   //Erstellen der Streams
2   cudaStream_t eventStream[NUM_EVENTS/NUM_BLOCKS];
3   for(int i=0; i<(NUM_EVENTS/NUM_BLOCKS); ++i){
4     cudaStreamCreate(&eventStream[i]);
5   }
6
7   //Schleife ueber alle Streams
8   for(int i=0; i<(NUM_EVENTS/NUM_BLOCKS); ++i){
9     //Berechne pointer_shiftx und num_to_copyx
10  ...
11  //Kopiere asynchron zum Device
12    cudaMemcpyAsync(dev_multiStates+pointer_shift1,
          multiStates_pinned+pointer_shift1, num_to_copy1,
          cudaMemcpyHostToDevice, eventStream[i]);
13    cudaMemcpyAsync(dev_sttHits+pointer_shift2, sttHits_pinned+
          pointer_shift2, num_to_copy2, cudaMemcpyHostToDevice,
          eventStream[i]);
14    cudaMemcpyAsync(dev_hitIndices+pointer_shift3,
          hitIndices_pinned+pointer_shift3, num_to_copy3,
          cudaMemcpyHostToDevice, eventStream[i]);
15
16    //Kernel-Launches
17    ExtractData<<<NUM_BLOCKS,NUM_THREADS_PER_BLOCK, 0,
          eventStream[i]>>>(...);
18    EvaluateStates<<<NUM_BLOCKS,NUM_THREADS_PER_BLOCK,0,
          eventStream[i]>>>(...);
19    EvaluateMultistates<<<NUM_BLOCKS,NUM_THREADS_PER_BLOCK,0,
          eventStream[i]>>>(...);
20   }
21
22  for(int i=0; i<(NUM_EVENTS/NUM_BLOCKS); ++i){
23  //Berechne pointer_shiftx und num_to_copyx
24  ...
25    //Kopiere asynchron zum Host
26    cudaMemcpyAsync(states_pinned+pointer_shift4, dev_states+
          pointer_shift4, num_to_copy4, cudaMemcpyDeviceToHost,
          eventStream[i]);
27    cudaMemcpyAsync(multiStates_pinned+pointer_shift5,
          dev_multiStates+pointer_shift5, num_to_copy5,
          cudaMemcpyDeviceToHost,eventStream[i]);
28  }
29
30  //Synchronisiere Streams und gebe sie frei
31  for(int i=0; i<(NUM_EVENTS/NUM_BLOCKS); ++i){
32    cudaStreamSynchronize(eventStream[i]);
33    cudaStreamDestroy(eventStream[i]);
34  }
```

Zum asynchronen Kopieren müssen die betroffenen Daten auf der Host-Seite im Pinned Memory liegen (siehe Abschnitt 4.3.3). Die Input-Werte in `sttHits` und `hitIndices` werden vom Host in den Pinned Memory kopiert und dann aus diesem an das Device übertragen. Die Initialisierung der Multi-Zustände erfolgt auf der Host-Seite. Anschließend werden die Daten in den Pinned Memory geschrieben und zum Device kopiert. Bevor das Kopieren der Daten aus dem Pinned Memory abgeschlossen ist, werden die Kernel aufgerufen. Da die verwendete GPU nicht das parallele Kopieren vom und zum Host unterstützt werden die Ergebnisse in einer separaten Schleife zum Host kopiert. Auch hier muss Pinned Memory verwendet werden. Nachdem alle Aufgaben an die Streams verteilt worden sind, werden sie synchronisiert und freigegeben.

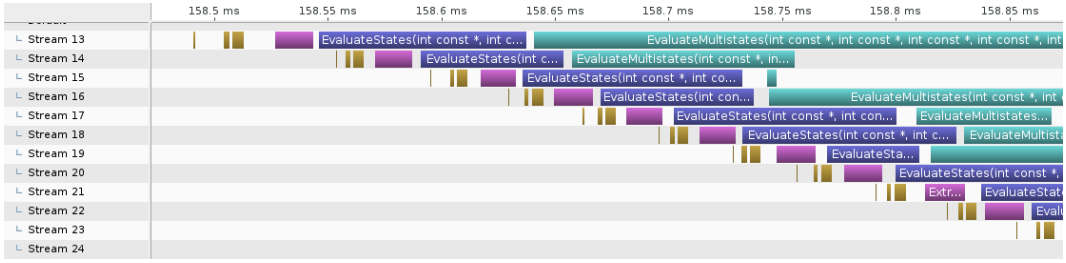


Abbildung 5.5: Ausschnitt aus dem NVIDIA Visual Profiler. Nachdem die Daten für einen Stream kopiert wurden (gelb), startet der darauffolgende Stream. Die Kernel (lila, blau und grün) können parallel ausgeführt werden.

5.4.2 Zusätzlich benötigte Datenstrukturen

Um innerhalb der Kernel den Zugriff auf die Daten einer bestimmten Menge von Events zu koordinieren, müssen zusätzliche Informationen an das Device übertragen werden. Mit Hilfe dieser wird der Adressbereich berechnet, für den ein Kernel zuständig ist. Es muss bekannt sein, auf die wievielte Gruppierung von Events zugegriffen wird. Dafür wird dem k-ten Stream `k*NUM_BLOCKS` übergeben. Dies stellt den Offset eines Streams innerhalb der gemeinsamen Datenstruktur dar. Das zu bearbeitende Event ergibt sich aus dem Block-Index. Um pro Kernel die Daten für die Events an die Blöcke aufteilen zu können, muss zusätzlich bekannt sein, wie viele Hits pro Event vorhanden sind. Diese Anzahl muss feiner in „Anzahl der Hits mit maximal 6 Nachbarn“ und „Anzahl der Hits mit maximal 22 Nachbarn“ aufgeteilt werden, da die Daten z. B. `hitNeighbors` mit einem unterschiedlichen Versatz gespeichert werden.

Beim Einlesen der STT-Hits vom Host werden die zusätzlichen Informationen gespeichert. Es werden zwei Arrays zum Speichern der Anzahl von Hits

mit wenig Nachbarn (`indices_6`) und mit vielen Nachbarn (`indices_22`) angelegt. Diese enthalten an der Stelle `i+1` die bis zum Event `i` aufsummierten Anzahlen. Die Arrays enthalten `NUM_EVENTS+1` Einträge und werden bei `i=0` mit `0` initialisiert. Daraus ergibt sich für ein Event eines Blocks des Streams `k` die in Tabelle 5.5 dargestellten Indizies. Abbildung 5.6 zeigt ein zugehöriges Beispiel für zwei Streams, die jeweils einen Kernel mit zwei Blöcken ausführen.

Stream-Offset	`k*NUM_BLOCKS`
Event-Offset	`k*NUM_BLOCKS+blockIdx.x`
Anzahl der Hits mit wenig Nachbarn	`indices_6[eventOffset+1]` `-indices_6[eventOffset]`
Anzahl der Hits mit vielen Nachbarn	`indices_22[eventOffset+1]` `-indices_22[eventOffset]`
globaler Offset für gemeinsam genutzte Datenstrukturen ohne Shift	`indices_6[eventOffset]` `+indices_22[eventOffset]`
globaler Offset für gemeinsam genutzte Datenstrukturen mit Shift	`indices_6[eventOffset]*6` `+indices_22[eventOffset]*22`

Tabelle 5.5: Berechnung der Adress-Offsets innerhalb der Kernel, die verschiedenen Streams zugeordnet werden und mehrere Blöcke ausführen.

Globale Offsets ohne Shift werden für Datenstrukturen benötigt, deren Einträge unabhängig von der Anzahl von Hits mit vielen oder wenig Nachbarn gespeichert werden, wie z. B. `states`. Für Einträge in z. B. `hitNeighbors` besteht eine Abhängigkeit zur Anzahl der Nachbarn, sodass ein anderer Index berechnet werden muss. Die Arrays `indices_6` und `indices_22` werden vor dem Befüllen der Streams aufgrund der geringen Datenmenge mit `cudaMemcpy` an das Device übertragen (synchrones Kopieren). Bei den Kernel-Aufrufen werden die entsprechenden Device-Pointer übergeben.

5.4.3 Startkonfiguration

Jeder Kernel wird mit `NUM_BLOCKS` Blöcken und der maximalen Anzahl von 1 024 Threads gestartet. Eine geringere Thread-Anzahl könnte ausreichend sein. Sie ist von der maximalen Anzahl von Hits der `NUM_BLOCKS` Events abhängig und müsste in einer Schleife über die entsprechenden Hit-Anzahlen ermittelt werden. Aufgrund des dabei entstehenden Overheads wird darauf verzichtet und die Kernel werden immer mit der maximalen Anzahl von Threads pro Block gestartet.

Die maximale Anzahl der Blöcke, die mit einem Kernel gestartet werden können, ist für die eingesetzte Grafikkarte 2 147 483 647. Diese Anzahl kann allerdings nicht erreicht werden, da es nicht möglich ist, die benötigten Daten

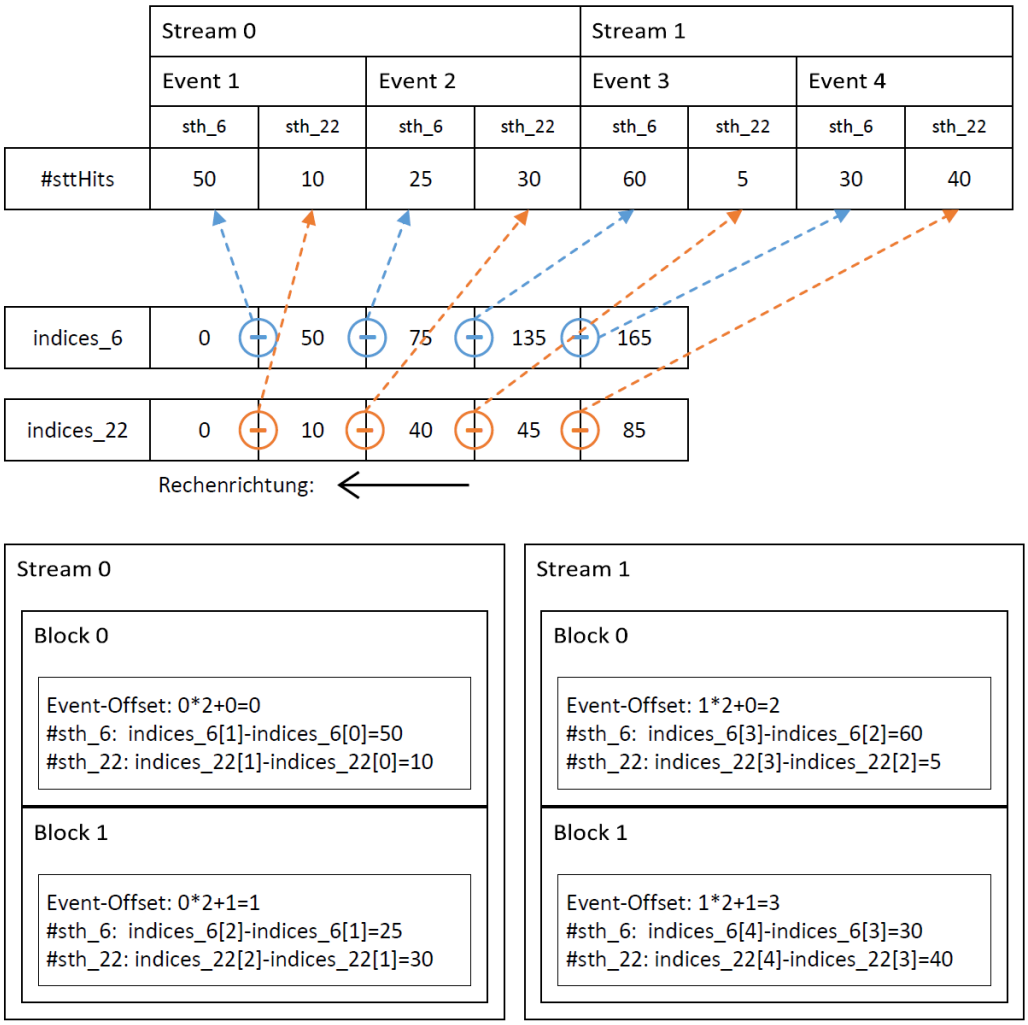


Abbildung 5.6: Beispiel für die Berechnung der Indizes für nebenläufige Streams, die jeweils die Daten für zwei Events (in zwei Blöcken) berechnen.

für so viele Events auf der Grafikkarte abzulegen. Durch den globalen Speicher ist die maximale Anzahl auf ungefähr 60 000 Events beschränkt.

Bei dem Start der Kernel kann entweder eine große Block-Anzahl und somit weniger Streams gewählt werden oder umgekehrt. Die Analyse (siehe Kapitel 6) ergibt, dass es besser ist, größere Datenpakete zu übertragen und die Streams mit der Bearbeitung von Hunderten bis Tausenden Events zu beschäftigen. Als eine gute Startkonfiguration hat sich eine Block-Anzahl von 3 000 bewährt.

5.5 Einbindung in PandaRoot

Bisher wurde nur die auf Algorithmus-Ebene parallelisierte Version des Spurfinders in PandaRoot eingebunden. Für den Spurfinder wurde als CPU-Version eine Task implementiert, die die Daten nacheinander für verschiedene Events berechnet. Für die Beschleunigung der Berechnungen werden zum Generieren der Zustände die entsprechenden Kernel mit einem Block gestartet.

Da bereits CUDA-Code anderer Klassen in PandaRoot integriert wurde, wurde das Einbinden der parallelisierten Funktionen analog dazu vorgenommen. Der Device-Code ist in der separaten Datei `trackletGenerator.cu` verfügbar. In dieser Datei sind zusätzliche Wrapper-Funktionen implementiert, welche innerhalb des CPU-Codes des Spurfinders aufgerufen werden können. Die Wrapper-Funktionen führen sämtliche CUDA-Befehle aus. Sie allozieren Speicher auf dem Device, kopieren Daten und starten die Kernel. Sie sind als `extern` deklariert, d. h. sie können innerhalb mehrerer Quellcode-Dateien genutzt werden, sind aber nur einmalig in `trackletGenerator.cu` definiert.

Der Build-Prozess von PandaRoot wird mit Hilfe von CMake [22] realisiert. Die CUDA-Bibliotheken werden in der `CMakeLists.txt`-Datei des Spurfinders geladen und der Source-Code in `trackletGenerator.cu` bekannt gemacht.

Über eine boolesche Variable in der `PndSttCellTrackFinderTask`-Klasse kann entschieden werden, ob das Device benutzt werden soll oder nicht. Soll der parallele Code ausgeführt werden, wird diese Information bis zum `PndSttCellTrackletGenerator`-Objekt durchgereicht. Um die Nachbarschaften der Straw Tubes nur einmalig zu kopieren, erfolgt dies in der `Init`-Funktion der Task. Die Nachbarschaften werden aus der `PndSttGeometryMap` gelesen und in ein Array umformatiert. Es wird eine Wrapper-Funktion aus `trackletGenerator.cu` aufgerufen, die den nötigen Speicherplatz auf dem Device alloziert und das Array kopiert. Der resultierende Device-Pointer wird bis zum `PndSttCellTrackletGenerator`-Objekt weitergegeben. Auf diese Weise wird jedes mal auf das gleiche Array zugegriffen. Innerhalb des TrackletGenerators müssen die STT-Hits des Events in die vorgesehene Struktur gebracht und an die Wrapper-Funktion übergeben werden. Die Wrapper-Funktion kopiert die Daten zum Device und startet die drei Kernel `ExtractData()`, `EvaluateStates()` und `EvaluateMultiStates()` für ein Event. Die berechneten Zustände werden zum Host kopiert und formatiert. Die Ergebnisse werden in den Attributen `fStates` und `fMultiStates` gespeichert, damit für die anschließenden Schritte die noch nicht parallelisierten Funktionen aufgerufen werden können. Nachdem der Spurfinder für alle Events ausgeführt wurde, wird in der Task der Speicherplatz für die Nachbarschaften der Tubes freigegeben.

Die Umsetzung der Parallelisierung auf Event-Ebene innerhalb von PandaRoot erfordert eine grundlegende Umstrukturierung des Codes, die sehr komplex ist und in einem separaten Software-Projekt vorgenommen werden soll. Statt der event-basierten Bearbeitung innerhalb der Task, müssten zeitbasierte Datenpakete von STT-Hits verarbeitet werden. Da eine derartige Verarbeitung der Daten durch den Spurfinder bisher noch nicht umgesetzt ist, konnte die Bearbeitung von Daten mehrerer Events in Streams noch nicht integriert werden.

6 Analyse der GPU-Version

In diesem Kapitel werden die auf Algorithmus- und Event-Ebene parallelisierten Versionen des Spurfinders mit Hilfe eines Profilers von NVIDIA untersucht und die Optimierung wird mit einer Performanceanalyse belegt. Der eingesetzte Profiler wird kurz vorgestellt und die damit untersuchten Performance-Metriken werden dargelegt. Anschließend wird der Device-Code ausführlich analysiert. Am Ende des Kapitels werden Möglichkeiten zur weiteren Optimierung beschrieben.

6.1 NVIDIA Visual Profiler und Performance-Metriken

Mit dem NVIDIA Visual Profiler (NVVP) [23] kann die Performance von CUDA-Anwendungen über eine grafische Oberfläche dargestellt und analysiert werden. Der Profiler entdeckt Performance-Engpässe und schlägt Möglichkeiten zur Beseitigung oder Verringerung dieser Engpässe vor.

Im *timeline view* werden entlang eines Zeitstrahls CPU- und GPU-Aktivitäten dargestellt, die wiederum in feinere Aktivitäten untergliedert sind. In Abbildung 6.1 ist ein Beispiel zu sehen. In Reihen, die der Gruppe *Thread* zugeordnet sind, sind die Aktivitäten eines CPU-Threads zu sehen, der Funktionen aus dem CUDA driver oder CUDA runtime API aufruft. Auch der Profiling-Overhead wird angezeigt. Unter dem Namen des Devices sind die Aktivitäten von Kopiervorgängen, der Kernel und der Streams zu finden. [23, S. 4 ff.]

Im *analysis view* hat der Benutzer u. a die Möglichkeit die Anwendung in Bezug auf Datenübertragung, Nebenläufigkeit, Auslastung und Kernel-Performance zu untersuchen. Es wurden folgende Metriken untersucht[23, S. 74 ff.],[19, S. 17 ff.]:

Bandbreite Die Bandbreite gibt an, mit welcher Rate Daten übertragen werden (auch Datentransferrate).

Speicher-Effizienz Im Rahmen der Analyse der genutzten Bandbreite wird für die einzelnen Speicher (Shared und Global Memory) das Verhältnis des abgerufenen Speichers zum eigentlich benötigten Speicher untersucht.

Auslastung Die Auslastung beschreibt das Verhältnis der Anzahl der aktiven Warps pro Multiprozessor zur maximalen Anzahl der möglichen aktiven Warps.

Effizienz der Warp-Ausführung Mit dieser Effizienz wird das Verhältnis der durchschnittlich aktiven Threads pro Warp zur maximalen Anzahl der aktiven Threads pro Warp beschrieben.

Nutzung der Rechenleistung Es wird der Anteil der Rechenkapazität berechnet, der durch die Ausführung der Kernel genutzt wird.

Nebenläufigkeit Auf einigen Grafikkarten können mit Hilfe von Streams Kernel und Datentransfers überlappend ausgeführt werden und auch mehrere Kernel parallel bearbeitet werden. Der Wert der Nebenläufigkeit gibt an, ob diese spezielle Eigenschaft im Device-Code genutzt wird oder nicht.

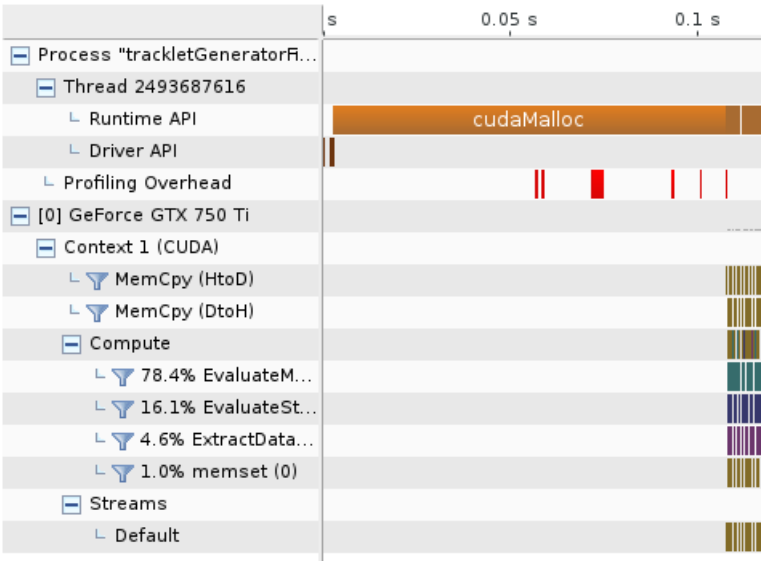


Abbildung 6.1: *Timeline view* des NVIDIA Visual Profilers.

Die Bandbreite ist einer der wichtigsten Faktoren, die Einfluss auf die Performance haben. Es sollte so oft wie möglich auf schnellen Speicher (z. B. Shared Memory) zugegriffen werden, während der Zugriff auf langsamen Speicher (z. B. Global Memory) vermieden werden sollte. Die GeForce GTX 750 Ti kann eine Speicherbandbreite von bis zu 86.4 GB/s erreichen. Zudem sollte auf die Anordnung der Daten im Speicher geachtet werden, sodass der Durchsatz beim Lesen/Schreiben durch die Threads eines Warps möglichst hoch ist (Coalescing). Auch der Datentransfer zwischen Host und Device sollte optimiert werden. Die Bandbreite zwischen Host und Device ist deutliche geringer als die zwischen dem Device und Device-Speicher. Für die verwendete

GPU ist sie maximal 8 Gbit/s. Aufgrund des Overheads eines Transfers, sollten statt mehrerer kleiner ein großes Datenpaket übertragen werden. Durch Nutzen von Pinned Memory zur Datenübertragung kann die höchste Bandbreite zwischen Host und Device erreicht werden. Pinned Memory erlaubt zudem die Nebenläufigkeit von Datentransfer und Kernel-Ausführung. [19, S. 18 ff.]

Die Auslastung ist ein Maß dafür, wie gut die Hardware ausgenutzt wird. Aufeinanderfolgende Instruktionen der Threads werden sequenziell bearbeitet. Falls sich die Ausführung eines Warps verzögert, werden andere Warps ausgeführt, um Latenzzeiten zu verbergen und die Hardware weiter zu beschäftigen. Eine höhere Auslastung ist nicht immer mit einer besseren Performance verbunden. Es gibt eine Grenze ab der die Performance durch eine erhöhte Auslastung nicht mehr verbessert werden kann. Ein Kernel mit einer geringeren Auslastung hat i. A. mehr Register-Speicher zur Verfügung als ein Kernel mit höherer Auslastung. Das kann dazu führen, dass weniger Daten ausgelagert werden müssen, was sich positiv auf die Performance auswirkt. Ab einer Auslastung von 50 % wird in den meisten Fällen durch eine weitere Erhöhung dieser die Performance nicht entsprechend verbessert. Eine zu geringe Auslastung sorgt allerdings für einen Performance-Verlust, da Speicher-Latenzen nicht mehr verborgen werden können. [19, S. 43 ff.]

Die Multiprozessoren der GPU sollten so gut wie möglich ausgelastet sein. Die Aufgaben zur Bewältigung der Arbeit sollten gut auf die Multiprozessoren aufgeteilt werden können, sodass keine Warte-/Ruhezeiten entstehen. Die Wahl der Anzahl der Threads und Blöcke ist dabei ein wichtiger Faktor. Die Richtlinien zur Startkonfiguration wurden bereits in Abschnitt 4.4 beschrieben.

6.2 Parallelisierung auf Algorithmus-Ebene

Im Folgenden wird der Spurfinder als Stand-Alone-Version (Abschnitte 6.2.1 und 6.2.3) und als PandaRoot-integrierte Version (Abschnitt 6.2.2) analysiert. Dabei wird in den ersten beiden Abschnitten die finale Version des Spurfinders untersucht. Abschnitt 6.2.3 analysiert die verschiedenen Entwicklungsstufen, die bis zur finalen Version durchlaufen worden sind. Für die Stand-Alone-Versionen wurden die Laufzeitmessungen und weitere Analysen mit dem NVVP durchgeführt. Der dabei entstandene Profiling-Overhead wurde vernachlässigt. Innerhalb von PandaRoot wurden Zeitstempel zur Messung der Ausführungszeiten verwendet. Zum direkten Vergleich mit der CPU-Version wurden die Berechnungen für die gleichen 1 000 Events ausgeführt, die zur Bestimmung der CPU-Laufzeiten in Abschnitt 3.3 genutzt wurden.

6.2.1 Ergebnisse des Profilings

Die Analyse für 1 000 nacheinander bearbeiteter Events hat Folgendes ergeben:

- Die zwischen Host und Device (und umgekehrt) zu übertragenen Datenpakete sind zu klein, um die mögliche volle Bandbreite zu nutzen.
- Alle Kernel lesen mit einer geringen Effizienz aus dem Global Memory (24.4 %).
- Die meisten Kernel schreiben mit einer geringen Effizienz in das Global Memory (31.5 %).
- Die Auslastung beträgt im Durchschnitt nur 6.3 % für alle Kernel.
- Die Warps aller Kernel werden mit einer mittleren Effizienz ausgeführt (46.8 %).
- Die Ausführungszeit der Kernel umfasst 54.1 % der gesamten Ausführungszeit.
- Es werden keine Kopier- und Rechenvorgänge parallel ausgeführt.
- Es werden keine Kernel parallel ausgeführt.

Die Daten werden für jedes Event zum und vom Device kopiert. Dabei ist die Menge der übertragenen Daten zu gering, um die mögliche Bandbreite zu nutzen. Zugriffe innerhalb der Kernel auf das Global Memory müssen optimiert werden. Die Ausführung eines Kernels pro Aufgabe und Event ist nicht effizient. Auch die Warps werden nicht effizient ausgeführt. Das ist darauf zurückzuführen, dass die Anzahl aktiver Threads in `EvaluateStates` und `EvaluateMultiStates` deutlich geringer ist, als die Anzahl der Threads mit der die Kernel gestartet worden sind.

Die Berechnungen innerhalb der Kernel sind nicht komplex genug, um das Device auszulasten. Die Größe des Grids (also ein Block pro Event) ist zu klein um Latenzzeiten bei Berechnungen und Speicherzugriffen zu verbergen. Bei Synchronisations-Maßnahmen innerhalb eines Blockes kann nicht zur Ausführung eines anderen Blockes gewechselt werden. Zudem ist nur ein Multiprozessor aktiv, da nur ein Block verteilt werden kann. Die parallelisierte Version liefert bei einer Bearbeitung von 1 000 aufeinanderfolgenden Events eine durchschnittliche Laufzeit von 0.575 ms pro Event (inkl. Allozieren und Kopieren). Somit konnte die Laufzeit für das Generieren der Zustände um den Faktor 10 verbessert werden. Allerdings werden die Ressourcen der GPU durch diese parallelisierte Version nicht effizient ausgeschöpft. Dies kann mit der parallelen Verarbeitung von Daten für mehrere Events erreicht werden (siehe Abschnitt 6.3).

Abbildung 6.2 zeigt die Device-Aktivität für den parallelisierten Code. Ein vergrößerter Ausschnitt ist in Abbildung 6.3 zu sehen. Das Device führt die Kopiervorgänge und die Berechnungen für jeweils ein Event nacheinander aus.

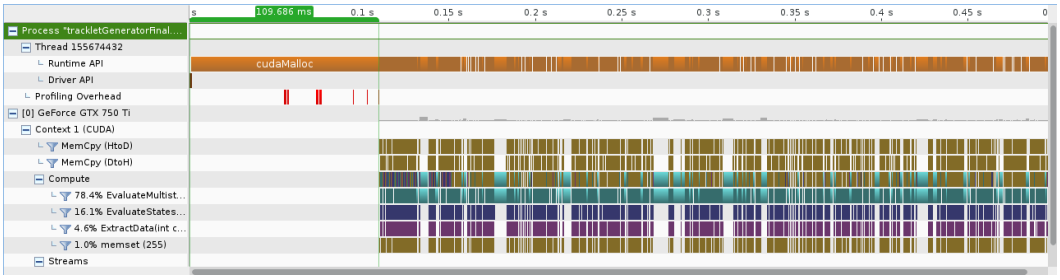


Abbildung 6.2: Visualisierung der Ausführung von 1 000 aufeinanderfolgenden Events mit dem NVVP.

Es gibt keine Überlappungen, da keine Streams genutzt werden. Die meisten Kernel werden nur sehr kurz ausgeführt. Bei längeren Rechenzeiten ruht die Copy-Engine des Devices. Mit `memset` werden die Einträge in `multiStates` initialisiert. Für die Initialisierung der Laufzeit-Umgebung von CUDA benötigt der CPU-Thread 0.1 s. Wird demzufolge die Anzahl der zu bearbeitenden Events erhöht, kann eine bessere durchschnittliche Laufzeit erreicht werden. Diese Maßnahme reicht trotzdem nicht aus, um das Device voll auszulasten. Eine deutliche Performance-Steigerung ist somit allein mit einer Parallelisierung auf Algorithmus-Ebene nicht zu erreichen.

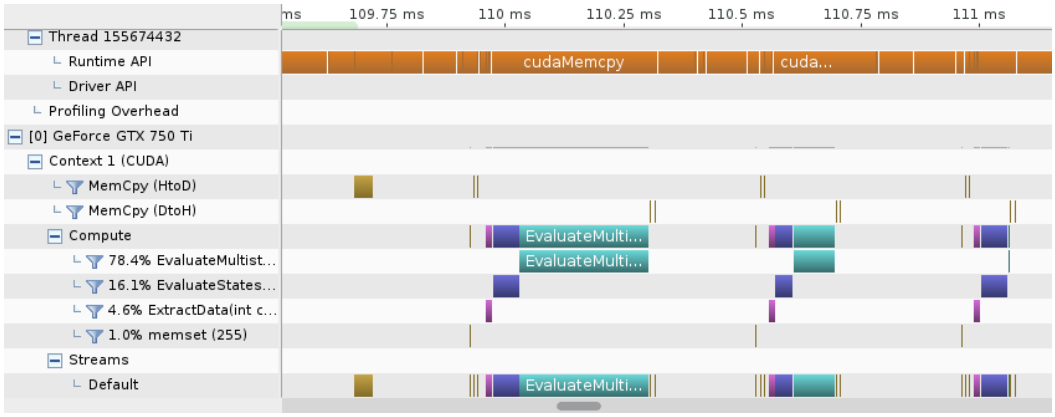


Abbildung 6.3: Vergrößerter Ausschnitt zu Abbildung 6.2.

6.2.2 Zeitmessung innerhalb von PandaRoot

Der Device-Code wurde in PandaRoot eingebunden und wird mit der entsprechenden Task jeweils für ein Event ausgeführt. Analog zur Analyse der CPU-Version wurde die Laufzeit der einzelnen Funktionen über Zeitstempel erfasst. Abbildung 6.4 zeigt die durchschnittlichen Laufzeiten der parallelisierten Version im Vergleich mit der CPU-Version für 1 000 Events.

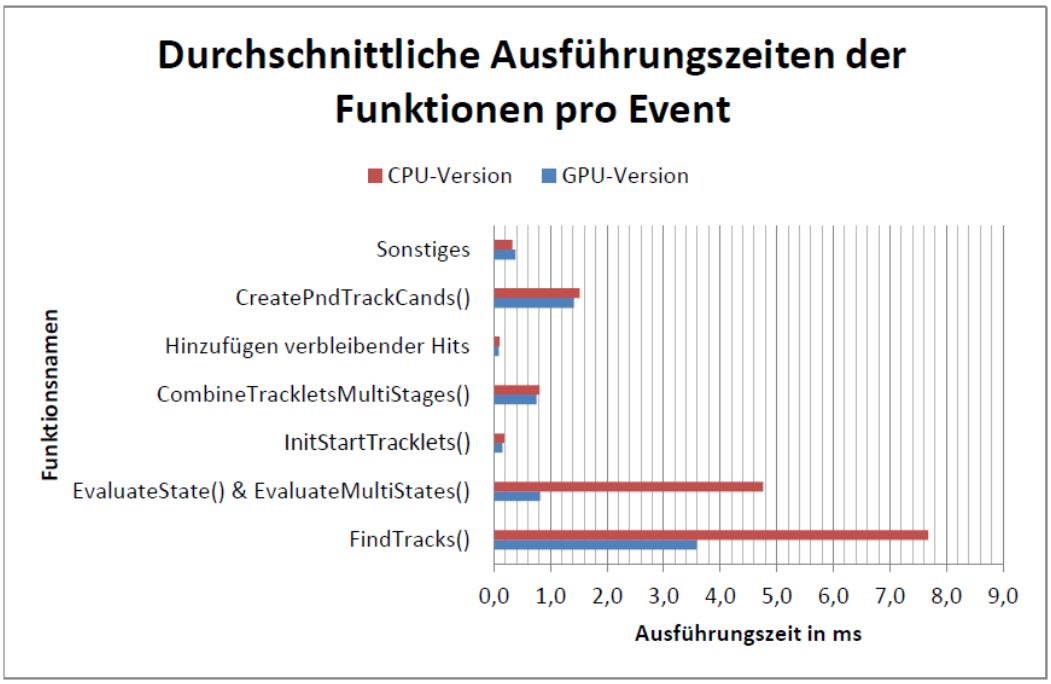


Abbildung 6.4: Durchschnittliche Ausführungszeiten der Funktionen des TrackletGenerators der CPU- und GPU-Version für 1 000 Events.

Für den parallelen Code wurden die Zeitstempel vor und nach dem Aufruf der Wrapper-Funktionen erfasst. Demnach wurde die Gesamtzeit für das Allozieren, das Kopieren, die Ausführung der drei Kernel und das Freigeben der Ressourcen erfasst. Die gemessene Zeit ist bei dem Eintrag „EvaluateStates() & EvaluateMultiStates()“ zu finden.

Die gesamte Ausführungszeit des Spurfinders konnte auf die Hälfte (3.58 ms) reduziert werden. Da die Funktionen der CPU-Version zum Generieren der Zustände 62 % der Laufzeit ausmachen, muss die Ausführungszeit aufgrund der restlichen Anteile der nicht parallelisierten Funktionen mindestens 2.9 ms betragen. Die Laufzeit zum Generieren der Zustände hat sich auf ein Fünftel der ursprünglichen Zeit verringert. Es wurde eine maximale Laufzeit des parallelen Codes von 9.51 ms gemessen. Dies ist auf die hohe Anzahl von mehrdeutigen Hits (fast 400) für das verarbeitete Event zurückzuführen. Für die CPU-Version liegt die maximale Laufzeit für `EvaluateMultiStates()` dagegen bei 584.88 ms. Dabei handelt es sich um ein anderes Event, für das viele Einträge in den Multi-Zuständen generiert wurden.

Durch die Benutzung der parallelen Funktionen haben sich die Laufzeiten der restlichen unveränderten CPU-Funktionen nicht wesentlich geändert. Durch die Beschränkung der Einträge in `multiStates` wird das Generieren der Multi-Zustände für Events beim Erreichen der maximalen Anzahl von 20

Einträgen vorzeitig beendet. Durch das Generieren weniger Multi-Zustände können Kombinationsmöglichkeiten entfallen, wodurch die Ausführungszeit von `CombineTrackletsMultiStages()`, `CreatePndTrackCands()` und beim Hinzufügen verbleibender Hits verringert werden kann. Die geringen Laufzeit-Unterschiede der `InitStartTracklets()`-Funktionen sind nicht auf die Verarbeitung von anderen Daten zurückzuführen, da die CPU-Laufzeiten in Abhängigkeit von z. B. Speicherzugriffen variieren können. Unter „Sonstiges“ fällt bei der GPU-Version zusätzlich das Vor- und Nachbereiten der Datenstrukturen.

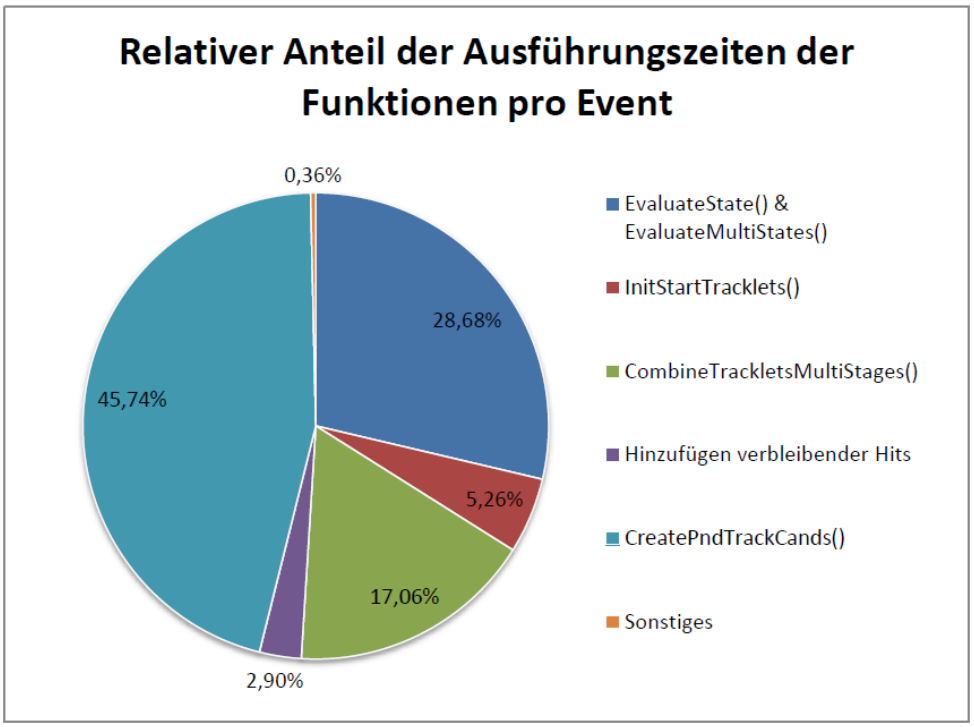


Abbildung 6.5: Durchschnittlicher relativer Anteil der Laufzeiten der Funktionen an `FindTracks()`.

Durch die geringen Ausführungszeiten der parallelisierten Funktionen verschieben sich die relativen Anteile pro Event entsprechend der Abbildung 6.5. Als nächstes sollten die Funktionen `CombineTrackletsMultiStages()` und `InitStartTracklets()` parallelisiert werden. Die Funktion `CreatePndTrackCands()` dient zum Umformatieren der Daten im Rahmen des Frameworks, was im späteren Experimentbetrieb nicht erforderlich ist.

Die Analyse-Task hat ergeben, dass sich an der Qualität des Spurfinders nichts geändert hat. Das bedeutet, dass die Annahme von maximal 20 möglichen Multi-Zuständen vertretbar ist. Lediglich die Anzahl der Ghosts ist um 2 % gestiegen. Das ist darauf zurückzuführen, dass einige rückläufige Kombinationswege durch das Fehlen von Multi-Zuständen nicht erkannt werden können.

6.2.3 Entwicklungsstufen

Bis zur endgültigen Version des Programms wurden verschiedene Möglichkeiten in Bezug auf das Verfahren, den Programmablauf und den Speicherzugriff erprobt. Dabei wurden folgende Entwicklungsstufen durchlaufen:

1. In der ersten Version des Spurfinders wird die Thrust-Bibliothek zur Aufsummierung der Zustände genutzt. Da die Bibliotheksfunktionen nicht innerhalb eines Kernels genutzt werden können, wird nach einer Iteration zum Generieren der Zustände die Kontrolle zurück an den Host gegeben. Mit Hilfe der Funktion `reduce()` von Thrust werden die Zustände auf dem Device aufsummiert und das Ergebnis wird dem Host übermittelt. Die Überprüfung der Abbruchbedingung erfolgt auf dem Host. Für ein Event werden die einfachen und die Multi-Zustände in einem einzigen Kernel generiert. Der Kernel muss für ein Event so oft aufgerufen werden, bis sich kein Zustand mehr ändert. Das Initialisieren der Zustände eindeutiger Hits erfolgt in einem zusätzlichen Kernel.

2. Aufgrund der hohen Rechenanteile der Thrust-Funktionen in der vorherigen Version wird auf die Nutzung von Thrust verzichtet. Die Summenbildung wurde eigenständig implementiert und erfolgt im Kernel mit Hilfe von Shared Memory. Nach einer Iteration werden die Zustände aufsummiert und dann zum Host kopiert. Die `while`-Schleife zur Überprüfung der Abbruchbedingung befindet sich noch auf dem Host.

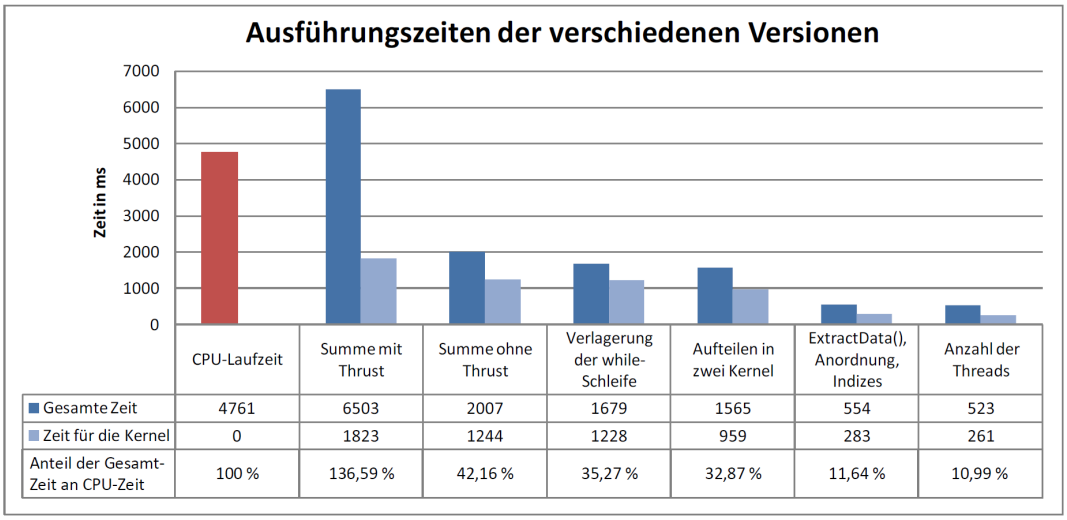
3. Die `while`-Schleife wird in den Kernel verlagert. Pro Event wird nur ein Kernel zum Generieren der Zustände aufgerufen. Dieser ist so lange aktiv, bis alle endgültigen Zustände berechnet worden sind.

4. Das Generieren der einfachen und Multi-Zustände wird auf zwei Kernel verteilt. Ein Kernel generiert die Zustände eindeutiger Hits, der andere die Zustände mehrdeutiger Hits.

5. `ExtractData()` erhält als zusätzlichen Eingabewert das Array `hitIndices` und übernimmt das Initialisieren der Zustände eindeutiger Hits. Die Anordnung der Einträge in `tubeNeighborings`, `sttHits` und `multiStates` wurde geändert. Statt einer logischen Gruppierung der Daten werden die gleiche Informationen für verschiedene Threads zusammenhängend gespeichert. Anstelle der Tube-IDs werden in `hitNeighbors` und `multiStates` die entsprechenden Indizes aus `sttHits` gespeichert.

6. Die Anzahl der Threads mit der ein Kernel gestartet wird ist von der Anzahl der STT-Hits abhängig. Statt der maximalen Anzahl von 1 024 Threads wird ausgehend von der Anzahl der STT-Hits die nächstgrößere Potenz von Zwei bestimmt.

Mit diesen Optimierungen war es möglich, die Laufzeit des Verfahrens deutlich zu verringern. Das Diagramm in Abbildung 6.6 zeigt die Ausführungszeiten der verschiedenen Versionen für insgesamt 1 000 Events. Die gemessenen Laufzeiten können in Abhängigkeit von der Ausführung der parallelen Threads durch die Hardware etwas variieren. Die Größenordnung bzw. das Verhältnis sollte dabei erhalten bleiben.



	CPU-Laufzeit	Summe mit Thrust	Summe ohne Thrust	Verlagerung der while-Schleife	Aufteilen in zwei Kernel	ExtractData(), Anordnung, Indizes	Anzahl der Threads
Gesamte Zeit	4761	6503	2007	1679	1565	554	523
Zeit für die Kernel	0	1823	1244	1228	959	283	261
Anteil der Gesamt-Zeit an CPU-Zeit	100 %	136,59 %	42,16 %	35,27 %	32,87 %	11,64 %	10,99 %

Abbildung 6.6: Benötigte Zeit zum Generieren der Zustände (inkl. Allozieren und Kopieren) der verschiedenen Versionen für insgesamt 1 000 Events.

Aus den gemessenen Zeiten geht hervor, dass das Nutzen der Thrust-Bibliothek für das Problem nicht geeignet ist und einen sehr großen Overhead erzeugt, sodass sogar mehr Rechenzeit als in der CPU-Version benötigt wird. Die Verlagerung der `while`-Schleife in den Kernel führt zu weniger Kernel-Aufrufen und einer geringeren Laufzeit. Auch die Implementierung des Generierens der Zustände für eindeutige und mehrdeutige in jeweils einem Kernel ist von Vorteil. Durch das zusätzliche Array `hitIndices` konnte die Ausführungszeit von `ExtractData()` von 521 ms auf 12 ms reduziert werden. Durch die geänderte Anordnung der Daten im Speicher und das Speichern von Indizes verringert sich die Laufzeit von `EvaluateStates()` um fast 30 %, die Laufzeit von `EvaluateMultiStates()` um fast 40 %. Mit der Berechnung der nächstgrößeren Thread-Anzahl konnte die Laufzeit nicht mehr wesentlich verkürzt werden.

6.3 Parallelisierung auf Event-Ebene

Analog zu Abschnitt 6.2 wurde der Device-Code mit dem NVVP und den gleichen Eingabe-Daten analysiert. Bei einem Input von mehr als 1 000 Events wurden diese wiederholt eingelesen und in einem großen Datenblock gespeichert.

6.3.1 Ergebnisse des Profilings

Mit einer Block-Anzahl von 3 000 hat die Analyse für die Berechnung der Daten für 60 000 Events Folgendes ergeben:

- Die zwischen Host und Device (und umgekehrt) übertragenen Datenpakete sind groß genug, um die volle Bandbreite zu nutzen.
- Die Speicher-Effizienz für den Zugriff auf Global Memory hat sich nur geringfügig verbessert.
- Nur für den Kernel `ExtractData()` wird eine geringe Auslastung von 10.4 % erreicht.
- Die Effizienz der Warp-Ausführung hat sich nur geringfügig verbessert.
- Die Ausführungszeit der Kernel umfasst 71.5 % der gesamten Ausführungszeit.
- Kopier- und Rechenvorgänge und mehrere Kernel werden parallel ausgeführt.

Die Bandbreite zwischen Host und Device wir nun effizient genutzt. Bis auf `ExtractData()` sind die Kernel komplex genug und erreichen eine Auslastung von fast 100 %. Die Speicher- und Warp-Effizienz haben sich nur geringfügig geändert, da an den Speicherzugriffen und dem Branching nichts geändert wurde. Bei einer Ausführung von 60 000 Events in 3 000 Blöcken (also 20 Streams) wurde für das Generieren der Zustände (inkl. Allozieren und Kopieren) eine durchschnittliche Laufzeit von 0.045 ms pro Event auf dem Device erfasst. Die Laufzeit der auf Algorithmus-Ebene parallelisierten Version konnte um mehr als das 10-fache beschleunigt werden. Damit wurde die Laufzeit zum Generieren der Zustände der CPU-Version insgesamt um den Faktor 100 beschleunigt. Eine weitere Performance-Steigerung sollte möglich sein, wenn noch mehr Events mit einer erhöhten Anzahl von Streams bearbeitet werden. Allerdings ist der Speicher die limitierende Größe (siehe Abschnitt 6.3.2).

Abbildung 6.7 zeigt die Aktivitäten für die Bearbeitung der gesamten 60 000 Events. Für das Anlegen des Pinned Memory für alle Events werden ca. 0.25 s benötigt. In der darauffolgenden Lücke von 0.22 s wird das Pinned Memory

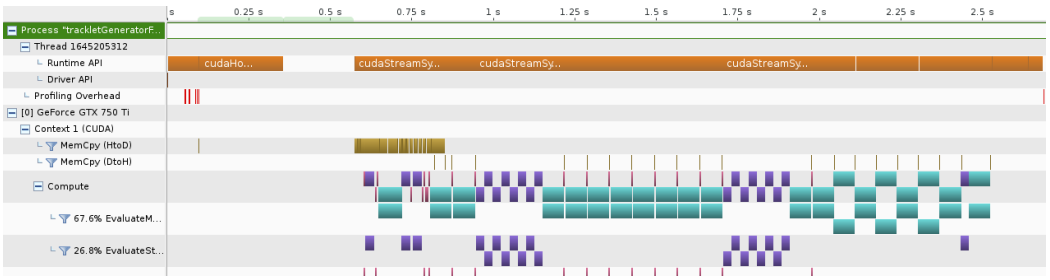


Abbildung 6.7: Visualisierung der Ausführung von 60 000 Events mit einer Anzahl von 3 000 Blöcken.

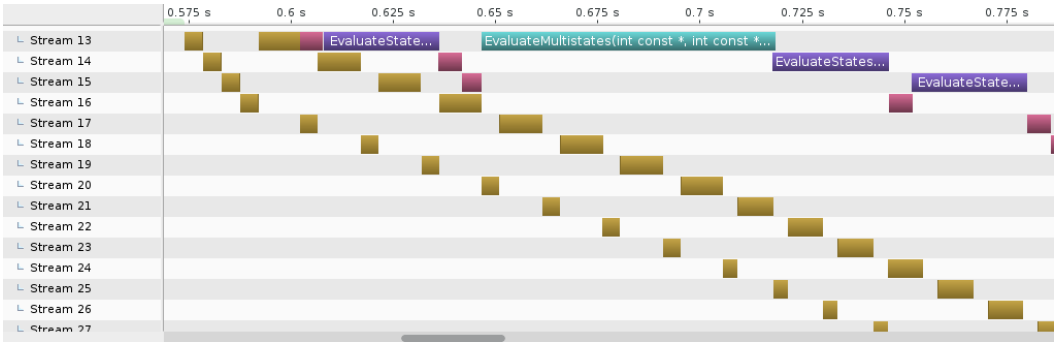


Abbildung 6.8: Vergrößerter Ausschnitt zur Darstellung der Streams aus Abbildung 6.7.

vom Host mit den Eingabe-Daten gefüllt. Diese werden anschließend nacheinander fast ohne Unterbrechungen zum Device kopiert. Das Kopieren der Ergebnisse zum Host erfolgt verzögert, da die Kernel noch aktiv sind. Bei der Initialisierung größerer Datenmengen hat `memset` zu Verzögerungen geführt. Aus diesem Grund werden die `multiStates` auf dem Host initialisiert und zum Device kopiert. Die mehrzeilige „Compute“-Reihe deutet auf eine überlappende Ausführung der Kernel hin. Die Kernel rechnen nun deutlich länger.

Die überlappende Ausführung der Vorgänge ist in Abbildung 6.8 zu sehen. Sobald ein Kopiervorgang (gelb) beendet ist, wird ein neuer angestoßen. Auch das Ende und der Anfang aufeinanderfolgender Kernel wird nebenläufig ausgeführt.

6.3.2 Beschränkung durch den verfügbaren Speicher

Die Ausführung der Kernel für eine große Menge von Events ist vor allem durch das verfügbare Global Memory auf dem Device beschränkt. Es stehen 2 GB zur Verfügung, auf die im CUDA-Kontext zugegriffen werden kann. Für eine Menge von `#E` Events lässt sich die Anzahl der Integer-Werte, für die Speicherplatz alloziert werden muss, wie folgt berechnen (mit `#sth_6` $\widehat{=}$ Anzahl der Hits mit wenig Nachbarn, `#sth_20` $\widehat{=}$ Anzahl der Hits mit vielen Nachbarn):

```
          sttHits  #sth_6 + #sth_20 +
       hitIndices  (4542 + 1) * #E +
    hitNeighbors  #sth_6 * 6 + #sth_20 * 20 +
numHitNeighbors  #sth_6 + #sth_20 +
          states  #sth_6 + #sth_20 +
     multiStates  (#sth_6 + #sth_20) * 20 +
       indices_6  #E + 1 +
      indices_22  #E + 1 +
                  = 29 * #sth_6 + 43 *#sth_20 + 4545 * #E + 2
```

Für 1 000 Events wurden die Werte `#sth_6`=55 636 und `#sth_20`=39 559 ermittelt. Damit ergibt sich eine Gesamtanzahl von 7 859 483 Integer-Werten für ein Event. Mit einer Speichergröße von 4 Byte pro Wert, wird für 1 000 Events Speicherplatz von fast 32 MB benötigt. Wird davon ausgegangen, dass das Device ausschließlich für die Ausführung des Spurfinders genutzt wird, sollten Daten für maximal 60 000 Events an das Device übertragen werden, da auch noch Speicher für die Laufzeit-Umgebung und z. B. für die Streams erforderlich ist.

Es werden ungefähr 27 MB an Pinned Memory für 1 000 Events benötigt. Da auf der Host-Seite von einem Arbeitsspeicher mit mehreren GB ausgegangen werden kann, ist das Pinned Memory derzeit keine limitierende Größe.

6.3.3 Block-Anzahl vs. Stream-Anzahl

Bei einer definierten Menge von eingehenden Events können die Daten entweder durch eine geringe Anzahl von Kernel-Ausführungen mit einer hohen Block-Anzahl bearbeitet werden oder es werden viele Streams genutzt, die in einem Kernel weniger Events bearbeiten. Die Wahl hat Einfluss auf die Größe der kopierten Datenpakete und die benötigte Rechenzeit der einzelnen Kernel.

Es wurde eine Parameterstudie durchgeführt. Diese hat ergeben, dass für eine steigende Anzahl von Blöcken Folgendes gilt:

- Der Durchsatz beim Kopieren zwischen Hosts und Device (und umgekehrt) steigt, da größere Datenpakete kopiert werden.
- Es werden weniger Streams erstellt.
- Es werden weniger Kernel aufgerufen.
- Die Zeit in der Kernel nebenläufig ausgeführt werden, nimmt ab.
- Die reale Ausführungszeit der Kernel nimmt ab.
- Die gesamte Ausführungszeit auf dem Device nimmt ab einer bestimmten Größe geringfügig zu.

– Es entstehen längere Ruhezeiten der Copy-Engine nachdem alle Daten zum Device übertragen worden sind. Es muss auf die Terminierung der Kernel gewartet werden, bevor die Ergebnisse zum Host kopiert werden können.

Nachfolgend wird die Performance des Codes bei einer sehr geringen und einer hohen Anzahl von Blöcken analysiert. Werden die 60 000 Events mit einer kleinen Block-Anzahl von 100 ausgeführt, werden insgesamt 1 800 Kernel von 300 Streams aufgerufen. Die reale Ausführungszeit der Kernel ist insgesamt 3.99 s. Durch die stark überlappende Ausführung (siehe Abbildung 6.9) beträgt die gesamte Laufzeit nur 2.70 s. Die Copy-Engine arbeitet fast ununterbrochen. Bis zu einer Block-Anzahl von ungefähr 3 000 bleibt die Gesamtlaufzeit fast konstant. Für höhere Anzahlen wurde ein leichter Anstieg beobachtet. Bei einer sehr großen Block-Anzahl von 20 000 werden nur drei Streams erstellt, die jeweils drei Kernel aufrufen. Die Laufzeit der Kernel beträgt 1.92 s, die Gesamtlaufzeit 2.90 s. Die Kernel werden für nur 0.3 % der Ausführungszeit (der Kernel) parallel ausgeführt. Die Copy-Engine ist nur zu Beginn stark ausgelastet (siehe Abbildung 6.10).

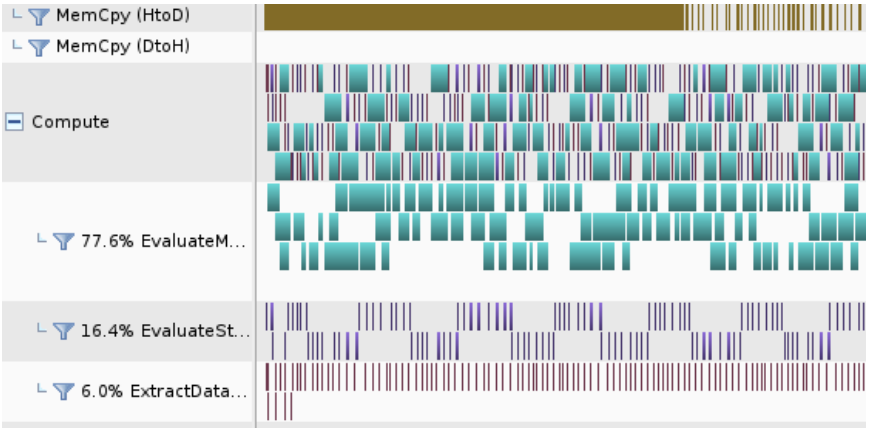


Abbildung 6.9: Aktivitäten auf dem Device bei einer Block-Anzahl von 100.

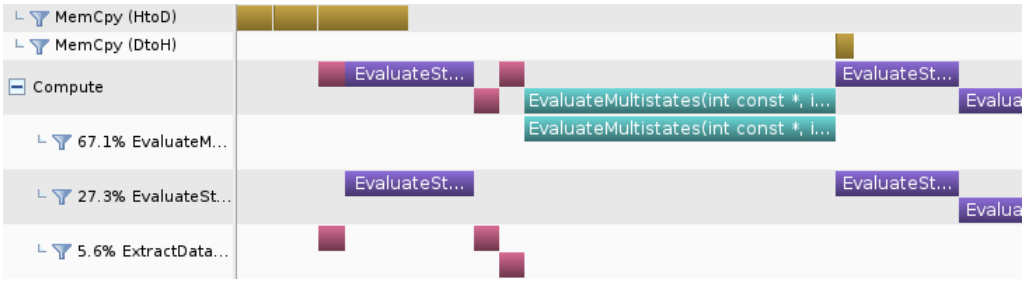


Abbildung 6.10: Aktivitäten auf dem Device bei einer Block-Anzahl von 20 000.

Die Analyse der Parameter hat gezeigt, dass eine Anzahl von 3 000 Blöcken eine gute Startkonfiguration darstellt. Die Datenpakete sind groß genug, um die volle Bandbreite zu nutzen. Die Kernel sind komplex genug, um die Multiprozessoren gut auszulasten. Nachdem alle Eingabedaten zum Device kopiert worden sind, entstehen Lücken in denen die Copy-Engine nicht aktiv ist. Diese können genutzt werden, um neue Daten an das Device zu übertragen. Um den im Global Memory verfügbaren Speicherplatz nicht zu überschreiten, muss auf der Host-Seite eine entsprechende Speicher-Verwaltung vorgenommen werden (siehe Abschnitt 6.5).

6.4 Zusammenfassung

Die Analyse des Device-Codes bezüglich der in Abschnitt 6.1 beschrieben Metriken hat ergeben, dass die Parallelisierung des Spurfinders auf Algorithmus-Ebene allein nicht geeignet ist, um die eingesetzte Grafikkarte effizient zu nutzen. Kopier- und Rechenvorgänge eines Events werden erst angestoßen, wenn die Ergebnisse für das vorherige Event zum Device kopiert worden sind. Zwar können die Speicherzugriffe noch optimiert werden, jedoch ist die Datenmenge pro Event zu gering, um effizient Kopieren zu können und die Multiprozessoren mit Kernel-Ausführungen auszulasten.

Mit der Parallelisierung auf Event-Ebene ist es mit Hilfe von Streams möglich, Daten für Tausende von Events überlappend zu kopieren und verarbeiten zu lassen. Zwischen Host und Device werden Daten gemeinsam für sehr viele Events übertragen, sodass ein effizientes Kopieren erfolgt. Durch das Starten eines Grids mit mehreren Blöcken, können die Blöcke an verschiedene Multiprozessoren verteilt werden und sorgen für eine gute Auslastung. Die bereits parallelisierten Funktionen müssten ungefähr noch um das 1 000-fache beschleunigt werden, damit die Daten entsprechend der Eventrate verarbeitet werden können. Beim Einsatz einer Multi-Device-Architektur mit etwa 1 000 Grafikkarten würde das Generieren der Zustände (bei einer linearen Skalierung) schnell genug erfolgen.

Die derzeitige Implementierung auf Event-Ebene lässt eine parallele Ausführung der Funktionen für 60 000 Events zu. Die begrenzende Komponente ist dabei das verfügbare Global Memory der eingesetzten Grafikkarte. Als Alternative könnte eine Grafikkarte mit höheren Speicherkapazitäten genutzt werden. Dabei muss allerdings auch genügend Arbeitsspeicher auf der Host-Seite zur Verfügung stehen, um das asynchrone Kopieren über Pinned Memory zu ermöglichen.

Tabelle 6.1 zeigt die zum Generieren der Zustände benötigten Laufzeiten bei der Parallelisierung auf Algorithmus- und Event-Ebene. Es konnte eine deutliche Beschleunigung der Funktionen des Spurfinders mit Hilfe einer GPU

erreicht werden. Der Einsatz von GPUs im Rahmen der Online-Verarbeitung von Daten an $\overline{\text{P}}$ANDA erweist sich daher als praktikabel.

Spurfinder-Version	Laufzeit zum Generieren der Zustände	Anteil an der CPU-Zeit
CPU	4.76 ms	100 %
GPU auf Algorithmus-Ebene	0.575 ms	12.08 %
GPU auf Event-Ebene	0.045 ms	0.94 %

Tabelle 6.1: Übersicht über die Optimierung der Laufzeiten der parallelisierten Funktionen.

6.5 Weitere Optimierungsansätze

Innerhalb aller Kernel muss der langsame Zugriff auf Global Memory verringert werden. Wird auf Daten mehrfach zugegriffen, sollten sie im Shared Memory zwischengespeichert werden. Dabei muss beachtet werden, dass das Zwischenspeichern der Nachbarschaften oder der Multi-Zustände viel Speicherplatz erfordert und Shared Memory eine rare Ressource ist.

Es hat sich gezeigt, dass keine volle Auslastung durch den Kernel `ExtractData()` gegeben ist. Die Berechnungen könnten in `EvaluateStates()` aufgenommen werden, um den Overhead eines Kernel-Launches zu reduzieren. Um den Datentransfer zwischen Host und Device zu verringern, können die Multi-Zustände auf dem Device innerhalb eines Kernels initialisiert werden.

Trotz der Parallelisierung auf Event-Ebene ist es nicht möglich, die Daten für Hunderttausende von Events parallel zu verarbeiten. Der begrenzende Faktor ist der verfügbare globale Speicher auf dem Device. Natürlich ist es möglich, den gesamten Prozess von vorne anzustoßen, sobald die Ergebnisse für alle 60 000 Events vorliegen. Mit dieser Vorgehensweise wird die Kapazität des Devices nicht voll ausgeschöpft. Zudem kann die durchschnittliche Berechnungszeit für die Events nicht ausreichend reduziert werden.

Um Grenzen des verfügbaren Speichers zu umgehen, ist auf der Seite des Hosts eine komplexe Speicherverwaltung nötig. Sobald die Ergebnisse für eine bestimmte Menge von Events auf dem Host zur Verfügung stehen, müssen die Speicher mit den Daten für neue Events gefüllt werden. Dies betrifft das Pinned und Global Memory. Der Speicher darf dabei nicht freigegeben und neu alloziert werden. Diese Funktionen sind synchron und würden der parallelen Ausführung der Streams entgegenwirken. Der verfügbare Speicher müsste wiederverwendet und mit neuen Daten überschrieben werden. Beim Wiederverwenden des Speichers muss darauf geachtet werden, dass die Anzahl der Hits variiert. Es müssen ausreichend große Arrays zu Verfügung stehen.

Der Host-Code kann parallel zum Device-Code ausgeführt werden. Diese Funktion wurde bisher nur bedingt genutzt. Während die Kernel noch aktiv sind können die Daten der vorhergehenden beendeten Kernel überschrieben werden. Durch das verzögerte Kopieren der Ergebnisse vom Device zum Host entstehen Ruhezeiten der Copy-Engine. Diese können genutzt werden, um die neuen Daten zum Device zu kopieren.

7 Zusammenfassung und Ausblick

Es wurden Teile eines Spurfinde-Verfahrens parallelisiert, das Signale des Straw Tube Trackers (STT), des zentralen spurgebenden Detektors des $\overline{\text{P}}$ANDA-Detektors, zu den gemessenen Flugbahnen gruppiert. Zur Parallelisierung wurde eine CUDA-fähige Grafikkarte (GPU) von NVIDIA eingesetzt. CUDA ist eine Architektur, mit der Programmierern die Entwicklung von Software auf GPUs über ein spezielles Programmier-Modell leicht zugänglich gemacht wird. Zur Programmierung auf der GPU wurde CUDA C eingesetzt, das die Programmiersprache C um einige zusätzliche Schlüsselwörter erweitert.

Das Laufzeitverhalten der verschiedenen Schritte/Funktionen des Spurfinders wurde untersucht. Anschließend wurden mit CUDA C die rechenzeitintensivsten Teile parallelisiert. Dabei handelt es sich um Verfahrensschritte in denen eine Art zellulärer Automat eingesetzt wird. Straw Tubes, die einen Hit signalisiert haben, fungieren als Zellen. Sie besitzen einen Start-Zustand, der in vielen Iterationsschritten im Sinne der Spurfindung angepasst wird.

Es wurde heterogener Code entwickelt. Das bedeutet, dass ein Mix von Code vorliegt, der auf der Central Processing Unit (CPU) oder der GPU ausgeführt wird. Das Programm startet auf der CPU. Daten, die für die parallele Berechnungen benötigt werden, werden zur GPU kopiert, dort parallel durch viele Threads verarbeitet und das Ergebnis wird anschließend zurückkopiert.

Beim Generieren der Zustände verwendet der Spurfinder vor allem dynamische Datenstrukturen der Standard Template Library. Die Programmierung auf einer GPU bringt Einschränkungen mit sich, sodass diese Datenstrukturen im Code, der auf der GPU ausgeführt werden soll, nicht genutzt werden können. Sämtliche Sets, Maps und Vektoren und ihre Beziehungen zueinander wurden daher in eindimensionale Arrays überführt. Die Arrays wurden dabei so strukturiert, dass auf die Daten effizient durch parallele Threads zugegriffen werden kann.

Mit CUDA C kann der Programmierer für eine Funktion definieren, von wie vielen Threads der gleiche Code parallel ausgeführt und wie diese Threads zu Kommunikationszwecken gruppiert werden sollen. Darauf basierend gibt es bei der Parallelisierung des Spurfinders zwei grundlegende Ansätze: die Parallelisierung auf Algorithmus-Ebene und die Parallelisierung auf Event-Ebene. Bei der Parallelisierung auf Algorithmus-Ebene wird nacheinander für jedes Event eine Gruppe von Threads gestartet, die die Daten parallel verarbeiten. Ein Thread führt die Berechnungen für einen ihm zugewiesenen Hit durch (zeitgleich mit den anderen Threads). Auf diese Weise werden die Rechen-

vorgänge des Spurfinders parallelisiert. Zudem ist es möglich den parallelisierten Code für mehrere Events nebenläufig auszuführen. Mit der Parallelisierung auf Event-Ebene wird der parallelisierte Spurfinder für mehrere Events zeitgleich auf der GPU ausgeführt. Dafür werden mehrere Gruppen von Threads gestartet, die zeitgleich den gleichen Code für verschiedene Events ausführen.

Mit der Parallelisierung auf Algorithmus-Ebene konnte die ursprüngliche Ausführungszeit der Funktionen um den Faktor 10 beschleunigt werden. Durch die Parallelisierung auf Event-Ebene war es möglich, den Beschleunigungsfaktor auf 100 zu erhöhen. Der Einsatz einer Grafikkarte zur Beschleunigung des Spurfinders stellt somit eine gute Möglichkeit dar, wenn auf der GPU Berechnungen für sehr viele Events parallel vorgenommen werden.

Die Rechenleistung einer GPU kann erst ab einer bestimmten Problemgröße effizient genutzt werden. Die Analyse des parallelisierten Codes hat ergeben, dass dies mit der Parallelisierung auf Algorithmus-Ebene allein nicht erreicht werden kann. Bei Speicherzugriffen und Rechenvorgängen auftretende Latenzzeiten können nicht verborgen werden. Trotzdem war es damit möglich, die Laufzeit des gesamten Spurfinde-Verfahrens innerhalb von PandaRoot auf die Hälfte zu reduzieren. Tests haben ergeben, dass Änderungen am Spurfinde-Verfahren, die aufgrund der Gegebenheiten der GPU vorgenommen wurden, kaum Auswirkungen auf die Qualität der gefundenen Spuren haben.

Bei der Parallelisierung auf Event-Ebene wird die GPU durch die hohe Anzahl der Ereignisse, für die die Daten parallel verarbeitet werden, effizient genutzt. Dabei ist die Anzahl der Events durch den auf der GPU verfügbaren Speicher begrenzt. Die parallele Berechnung für eine Vielzahl von Events wurde noch nicht in PandaRoot integriert, da die Struktur der derzeitigen Implementierung des Spurfinders eine sequenzielle, vollständige Berechnung der Ergebnisse pro Event vorsieht.

Bevor der parallelisierte Spurfinder im realen Experiment eingesetzt wird, sollten die verbleibenden Verfahrensschritte ebenfalls auf eine GPU überführt werden. Die Ausführungszeit des gesamten Verfahrens ist bisher auf die Laufzeit der nicht parallelisierten Funktionen begrenzt. Bisher wurden nur die Funktionen parallelisiert, die ursprünglich 60 % der Laufzeit des Spurfinders ausgemacht haben. Zudem muss statt der eventweisen Bearbeitung mit einem kontinuierlichen Datenstrom gerechnet werden können. Um der hohen Eventrate des $\overline{\text{P}}$ANDA-Experimentes gerecht zu werden, kann eine GPU mit einer höheren Rechen- oder Speicherkapazität eingesetzt werden. Außerdem kann die Leistung mehrerer GPUs genutzt werden, um mehr Daten in der gleichen Zeit zu verarbeiten.

Literatur

[1] Jette Schumann. »Entwicklung eines schnellen Algorithmus zur Suche von Teilchenspuren im Straw Tube Tracker des $\overline{\text{P}}$ANDA-Detektors«. Bachelorarbeit. Fachhochschule Aachen, Campus Jülich, Aug. 2013.

[2] *Facility for Antiproton and Ion Research. Anlage zur Forschung mit Antiprotonen und Ionen*. Darmstadt, 2013. URL: http://www.fair-center.de/fileadmin/fair/publications_FAIR/FAIR_Flyer2013_04_de.pdf (besucht am 21. 07. 2015).

[3] Nina Hall, Hrsg. *FAIR - Facility for Antiproton and Ion Research. An international science centre in Europe for studying the building blocks of matter and the evolution of the Universe*. Darmstadt, 2013. URL: http://www.fair-center.de/fileadmin/fair/publications_FAIR/FAIR_Broschuere_autumn2013_V3_72dpi.pdf (besucht am 21. 07. 2015).

[4] H. H. (Editor in Chief) Gutbrod u. a., Hrsg. *FAIR - Baseline Technical Report*. Bd. 1. GSI Darmstadt, Sep. 2006. ISBN: 3981129806. URL: http://www.fair-center.de/fileadmin/fair/publications_FAIR/FAIR_BTR_1.pdf.

[5] The PANDA Collaboration. »Technical Progress Report for: PANDA (AnitProton Annihilation at Darmstadt) Strong Interaction Studies with Antiprotons«. In: (2005). URL: http://www-panda.gsi.de/archive/public/panda_tpr.pdf (besucht am 22. 07. 2015).

[6] PANDA Collaboration u. a. »Technical design report for the PANDA (AntiProton Annihilations at Darmstadt) Straw Tube Tracker«. English. In: *The European Physical Journal A* 49.2 (2013), S. 1–104. ISSN: 1434-6001. DOI: 10.1140/epja/i2013-13025-8. URL: http://dx.doi.org/10.1140/epja/i2013-13025-8 (besucht am 22. 07. 2015).

[7] Diego Bettoni, Hrsg. *The PANDA Experiment at FAIR*. Ithaca, NY: SLAC econf, 2007. URL: http://arxiv.org/abs/0710.5664 (besucht am 21. 07. 2015).

[8] PANDA Collaboration u. a. »Physics Performance Report for PANDA: Strong Interaction Studies with Antiprotons«. In: (März 2009). arXiv: 0903.3905 [hep-ex]. URL: http://arxiv.org/abs/0903.3905 (besucht am 22. 07. 2015).

[9] Artur Cebulla (Forschungszentrum Jülich GmbH IKP).

[10] Stefano Spataro und the PANDA Collaboration. »The PandaRoot framework for simulation, reconstruction and analysis«. In: *Journal of Physics: Conference Series* 331.3 (2011), S. 032031. DOI: 10.1088/1742-6596/331/3/032031. URL: http://stacks.iop.org/1742-6596/331/i=3/a=032031 (besucht am 22. 07. 2015).

[11] M. Al-Turany u. a. »The FairRoot framework«. In: *Journal of Physics: Conference Series* 396.2 (2012), S. 022001. DOI: doi:10.1088/1742-6596/396/2/022001. URL: http://stacks.iop.org/1742-6596/396/i=2/a=022001 (besucht am 20. 09. 2015).

[12] Rene Brun und Fons Rademakers. »ROOT — An Object Oriented Data Analysis Framework«. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 389.1 (1997), S. 81–86.

[13] »ROOT Data Analysis Framework, User's Guide«. In: (2014). URL: https://root.cern.ch/drupal/content/root-users-guide-600 (besucht am 03. 08. 2015).

[14] S Spataro. »Simulation and event reconstruction inside the PandaRoot framework«. In: *J. Phys.: Conf. Ser.* 119.3 (2008), S. 032035. DOI: 10.1088/1742-6596/119/3/032035. URL: http://dx.doi.org/10.1088/1742-6596/119/3/032035.

[15] Gerassimos Barlas. *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann, 2014. ISBN: 9780124171404.

[16] NVIDIA Corporation. *CUDA C Programming Guide*. Version 7. 2015. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (besucht am 26. 07. 2015).

[17] Jason Sanders und Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN: 0131387685, 9780131387683.

[18] NVIDIA Corporation. *NVIDIA Produkte und Technologien*. 2015. URL: http://www.nvidia.de/page/products.html (besucht am 07. 09. 2015).

[19] NVIDIA Corporation. *CUDA C Best Practices Guide*. Version 7. 2015. URL: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (besucht am 26. 07. 2015).

[20] NVIDIA Corporation. *GeForce GTX 750 Ti Grafikkarte*. 2015. URL: http://www.nvidia.de/object/geforce-gtx-750-ti-de.html (besucht am 07. 09. 2015).

[21] NVIDIA Corporation. *THRUST QUICK START GUIDE*. Version 7. 2015. URL: http://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf (besucht am 07. 09. 2015).

[22] Inc. Kitware. *CMake 3.3.2 Documentation*. 2015. URL: http://cmake.org/cmake/help/v3.3/ (besucht am 20. 09. 2015).

[23] NVIDIA Corporation. *Profiler User's Guide*. Version 7.5. Sep. 2015. URL: http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf (besucht am 15. 09. 2015).