technische universität
dortmund

Fakultät für Elektrotechnik und
Informationstechnik
Lehrstuhl für Hochfrequenztechnik
Prof. Dr.-Ing. Peter M. Krummrich

Bachelorarbeit

Speicheroptimierte Simulation der Impulspropagation in Mehrmodenfasern auf GPUs

Calvin Timmer

Erstprüfer:
Prof. Dr.-Ing. Peter M. Krummrich

Zweitprüfer:
M.Sc. Marius Brehler

Datum:
07.09.2015

Lehrstuhl für Hochfrequenztechnik

Inhaltsverzeichnis

1. Einleitung

Die optische Übertragungstechnik bildet heutzutage die Grundlage eines Großteils der digitalen Datenübertragung. Die massive Steigerung der Leistungsfähigkeit von optischen Übertragungssystemen hat in den letzten Jahrzehnten das immense Wachstum des weltweiten Datenverkehrs ermöglicht. Inzwischen erreichen die bestehenden Technologien jedoch immer mehr die theoretische Kapazitätsgrenze des wichtigsten Übertragungsmediums, der Glasfaser. Bei dem erwarteten Wachstum des weltweiten Datenverkehrs wird ein Erreichen dieser Grenze innerhalb von ein oder zwei Jahrzehnten angenommen [1, 2].

Die Steigerung der Kapazität von Glasfasern ist durch verschiedene technologische Vorschritte gelungen. In den frühen Jahren ermöglichte vor allem ein Fortschritt bezüglich der Materialien und Herstellungsverfahren längere und effizientere Übertragungsstrecken. Durch die Erfindung der optischen Verstärker wurde die mögliche Übertragungsdistanz weiter gesteigert. Doch erst die Verwendung mehrerer Wellenlängen des Lichts innerhalb einer Glasfaser hat zu dem immensen Anstieg der Übertragungskapazität geführt, der den massenhaften Einsatz von Glasfaser bedingt hat [1]. Bei dem sogenannten Wellenlängenmultiplexbetrieb (Wavelength Division Multiplexing, WDM) werden mehrere unabhängige Datenströme parallel in einer Faser übertragen, indem diese verschiedenen Wellenlängen des Lichts aufgeprägt werden. Am Empfänger können die einzelnen Datenströme durch Filter wieder voneinander getrennt werden. Die wirtschaftlich nutzbaren Bandbreiten sind jedoch aufgrund der Dämpfungscharakteristik der Fasern und der verfügbaren Sende- und Empfangskomponenten begrenzt und heutzutage beinahe vollständig belegt. Daher ist das Potential von WDM nahezu ausgeschöpft. Um bei gleichbleibendem Bandbreitenverbrauch dennoch eine höhere Übertragungskapazität zu erzielen, werden inzwischen höhere Modulationsformate mit besserer spektraler Effizienz eingesetzt. Diese sind jedoch anfälliger gegenüber Rauschen. Aufgrund des Rauschens in jedem Übertragungssystem, sind dieser Methode daher ebenfalls Grenzen gesetzt [2].

Um die Kapazität der Übertragungssysteme weiterhin in ausreichendem Maß zu steigern, ist die Verwendung neuer Technologien notwendig. Die simple Vervielfältigung der bestehenden Hardware ist nicht zukunftsfähig. Eine tragfähige Technologie muss in der Lage sein die Kosten pro Bit signifikant zu reduzieren, um den wirtschaftlichen Betrieb der Netze bei höherem Datenaufkommen zu gewährleisten. Eine vielversprechende Technologie, um dieses Ziel zu erreichen, sind Raummultiplexverfahren. Dabei werden mehrere räumlich getrennt Kanäle in einer Glasfaser verwendet um unabhängige Datenströme zu übertragen. Raummultiplexbetrieb (Space Division Multiplexing, SDM) kann generell auf zwei verschiedene Arten umgesetzt werden. Einerseits können Glasfasern mit mehreren parallelen lichtleitenden Kernen hergestellt werden. Andererseits können auch verschiedene räumliche Feldverteilungen des Lichts, sogenannte Moden, innerhalb eines lichtführenden Kerns verwendet werden. Dazu werden ebenfalls spezielle Fasern, sogenannte Mehrmodenfasern (Multi-Mode Fiber, MMFs) benötigt. Damit SDM erfolgreich zum Einsatz kommen kann ist jedoch vor allem die Entwicklung kostengünstiger aktiver und passiver Komponenten zum Verstärken und Filtern von Signalen in MMFs notwendig [3, 4].

Diese Arbeit beschäftigt sich mit der Simulation der Signalausbreitung in Mehrmodenfasern. Die Motivation hinter der Simulation eines Systems kann vielfältig sein. In der Forschung erlauben Simulationen die isolierte Betrachtung einzelner physikalischer Effekte, wie es in der Realität nicht möglich wäre. In der Entwicklung und Systemplanung erlauben Simulationen es wiederum mit sehr geringem Kostenaufwand einzelne Komponenten oder gesamte Netzwerke zu analysieren. Ein realer Testaufbau kann aufgrund der damit verbundenen Kosten oder fehlender Technologien gegebenenfalls nicht umgesetzt werden. Simulationen erfordern jedoch meist sehr viele Rechenoperationen, resultierend in einem hohen Zeitaufwand. Der Einsatz spezialisierter Computerhardware kann eine Simulation jedoch signifikant beschleunigen. In vielen Fällen sind sogar handelsübliche Grafikkarten (Graphics Processing Unit, GPUs) dazu geeignet. Die GPU-gestützte Simulation der Signalausbreitung in MMFs wurde bereits in [5] und [6] demonstriert. Dabei konnte die Berechnung in realistischen Einsatzszenarien um das bis zu 80-fache beschleunigt werden. Derzeit verfügbare GPUs haben jedoch einen begrenzten internen Speicher. Da bei der Simulation der Signalausbreitung in MMFs sehr große Datenmengen anfallen können, reicht der Speicherplatz der GPU gegebenenfalls nicht aus um Simulationen mit langen Signalen und zahlreichen Moden durchzuführen. Hierin begründet sich die Motivation dieser Arbeit, den Speicherverbrauch einer am Lehrstuhl für Hochfrequenztechnik bestehenden Simulationssoftware zu reduzieren. Das primäre Ziel der Arbeit ist es, den Speicherverbrauch des Algorithmus von der Anzahl der simulierten Moden zu entkoppeln. Damit werden Simulationen mit einer deutlich höheren Anzahl von Moden ermöglicht.

In den folgenden Kapiteln wird zuerst ein Überblick über die theoretischen Grundlagen der Arbeit gegeben. Dazu wird in Kapitel 2 eine Einführung in die optischen Übertragungstechnik gegeben. Die physikalischen Eigenschaften von Glasfasern werden beschrieben und es wird ein mathematische Modell zur Beschreibung der Signalausbreitung vorgestellt. Weiterhin wird ein Verfahren zur numerischen Berechnung des vorgestellten Modells präsentiert. Im Anschluss wird in Kapitel 3 die Architektur der verwendeten GPU erläutert. In Kapitel 4 wird die bestehende Implementierung der Simulationssoftware vorgestellt und der im Rahmen dieser Arbeit entworfene Algorithmus beschrieben. Ein Vergleich der ursprünglichen Implementierung und der überarbeiteten Version hinsichtlich Speicherverbrauch und Performance wird in Kapitel 5 vorgenommen. Schließlich wird in Kapitel 6 eine Zusammenfassung der Arbeit präsentiert.

2. Grundlagen der optischen Übertragungstechnik

Zur Simulation der Signalausbreitung in Glasfasern ist ein mathematisches Modell notwendig, das die Lichtausbreitung in dem zu untersuchenden Medium beschreibt. Das folgende Kapitel beginnt daher mit einem Überblick über die physikalischen Grundlagen, die zur Beschreibung der Lichtausbreitung in Glasfasern notwendig sind. Danach wird die zur Modellierung verwendete Manakov-Gleichung (Manakov Equation, ME) vorgestellt. Schließlich wird ein Algorithmus zur numerischen Lösung der ME präsentiert und erläutert.

Diese Arbeit konzentriert sich auf die Simulation der Signalausbreitung in Glasfasern zur Datenübertragung über große Distanzen. Weitere Anwendungsgebiete der Faseroptik werden in [7] vorgestellt.

2.1. Physikalische Grundlagen

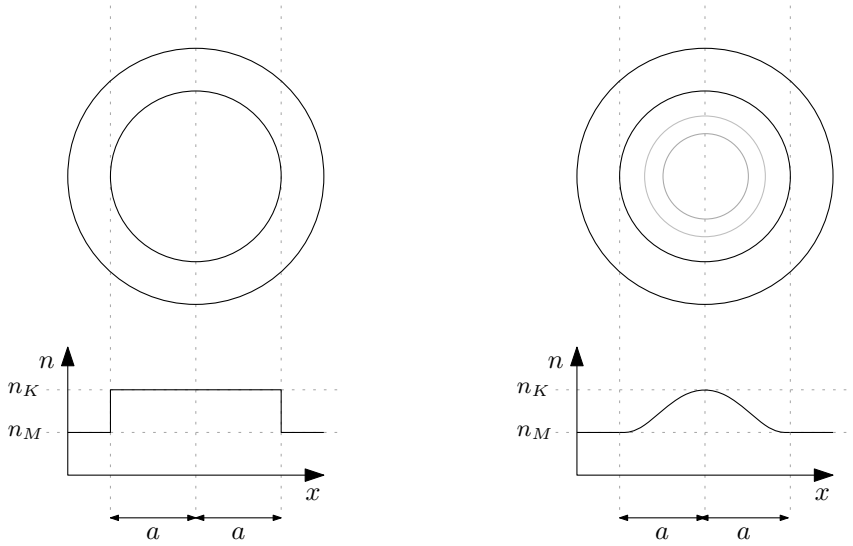
Licht kann auf verschiedene Arten beschrieben werden: als Lichtstrahl, als Teilchen (Photon) oder als eine elektromagnetische Welle (EM-Welle). Das dieser Arbeit zugrundeliegende Modell basiert auf der Beschreibung als EM-Welle. EM-Wellen sind im mathematischen Sinne Lösungen der Wellengleichung, die sich aus den Maxwell-Gleichungen und weiteren Randbedingungen ergibt. Die Maxwell-Gleichungen sind allgemeingültige Gleichungen, die die Zusammenhänge von elektrischen Feldern, magnetischen Feldern und Ladungsträgern beschreiben. Die Randbedingungen ergeben sich aus den Eigenschaften der Materie in der Umgebung der Welle. Daher wird im Folgenden der Aufbau von Glasfasern vorgestellt.

2.1.1. Aufbau von Glasfasern

Lichtführende Fasern werden im Allgemeinen als Lichtwellenleiter bezeichnet. Sie können aus verschiedenen Materialien gefertigt werden, unter anderem aus Kieselglas (amorphem Silizium-Dioxid) oder verschiedenen Kunststoffen. Fasern aus Kieselglas, die sogenannten Glasfasern, werden bei langen Übertragungsstrecken aufgrund ihrer geringen optischen Dämpfung am häufigsten verwendet. Daher beschränkt sich diese Arbeit ausschließlich auf Eigenschaften von Glasfasern.

Eine Glasfaser ist eine zylindrischer Faser, die aus mehreren konzentrischen Schichten mit unterschiedlichen Brechzahlen besteht. Die verschiedenen Brechzahlen werden durch gezieltes Einbringen von Fremdatomen (Dotierung) in das Grundmaterial, Kieselglas, erzeugt. Im einfachsten Fall werden die Fasern zwei scharf abgegrenzte Schichten hergestellt. Die innere Schicht wird

als Kern bezeichnet, die äußere Schicht als Mantel. Die Brechzahl n_K des Kerns ist dabei etwas größer als die Brechzahl n_M des Mantel. Diese Bauform wird aufgrund der sprunghaften Änderung der Brechzahl an der Grenzfläche zwischen Kern und Mantel als Stufenfaser bezeichnet. Eine andere relevante Bauform von Glasfasern ist die Gradientenfaser. Bei einer solchen Faser verringert sich die Brechzahl vom Kern zum Mantel kontinuierlich. Speziell Gradientenfasern bieten sich für den Einsatz in SDM an, da sie die Laufzeitunterschiede zwischen verschiedenen Signalen reduzieren [8, Kap. 11]. Querschnitte einer Stufen- und einer Gradientenfaser sind in Abbildung 2.1 dargestellt.



(a) Stufenfaser (b) Gradientenfaser

Abbildung 2.1.: Querschnitte verschiedener Bauformen von Glasfasern. Unter jeder Faser ist ihre Brechzahl n als Funktion der x-Koordinate aufgetragen. n_K bezeichnet die Brechzahl des Kerns, n_M die Brechzahl des Mantels. a ist der Kernradius.

Vereinfacht kann die Funktionsweise einer Glasfaser mittels Strahlenoptik erläutert werden. Die Grenzfläche zwischen Kern und Mantel reflektiert aufgrund des Brechzahlsprungs Lichtstrahlen im Inneren des Kerns, die unter einem flachen Einfallswinkel auftreffen. Anstatt die Glasfaser zu verlassen wird der Lichtstrahl also immer wieder innerhalb der Faser hin und her reflektiert. Für die Beschreibung des Lichts als EM-Welle definieren Kern und Mantel hingegen die Randbedingungen aus denen sich die Wellengleichung zur Beschreibung der Feldausbreitung ergibt. Für die Herleitung sei auf [9, Kap. 2] verwiesen.

Im Folgenden werden ausschließlich ungekrümmte Fasern betrachtet. Das verwendete Koordinatensystem ist so ausgerichtet, dass die z-Achse in die Ausbreitungsrichtung der Wellen zeigt und die x- und y-Achsen in der Querschnittebene der Faser liegen.

2.1.2. Modenausbreitung

Die Wellengleichung einer Glasfaser besitzt unendlich viele Lösungen, die als Moden bezeichnet werden. Es beschreiben jedoch nur endlich viele Lösungen Wellen, die im Kern der Faser geführt

werden. Aufgrund der Superpositionseigenschaft von EM-Wellen existieren diese ausbreitungsfähigen Moden prinzipiell unabhängig voneinander. Durch verschiedene Effekte, die durch die Materialeigenschaften der Faser bedingt sind, interagieren sie jedoch miteinander. Die Anzahl der ausbreitungsfähigen Moden hängt von den Eigenschaften der Faser ab. Fasern, die nur einen Modus führen, werden als Einmodenfasern (Single-Mode Fiber, SMFs) bezeichnet. Fasern, in denen mehr als ein Modus ausbreitungsfähig ist, werden Mehrmodenfasern (Multi-Mode Fiber, MMFs) genannt. Um die einzelnen Moden eindeutig unterscheiden zu können werden sie nach verschiedenen Eigenschaften kategorisiert.

Die exakten Lösungen der Wellengleichung werden als Vektormoden bezeichnet, da sie durch vektorielle elektrische (E-) und magnetische (H-) Felder beschreiben werden. Je nachdem, ob diese Felder entlang der z-Achse eine von Null verschiedene E- oder H-Komponente besitzen, werden sie wie folgt benannt:

TE („Transversal elektrisch“) Das E-Feld dieser Moden hat keine z-Komponente.

TM („Transversal magnetisch“) Das H-Feld dieser Moden hat keine z-Komponente.

HE/EH Sowohl das E-Feld, als auch das H-Feld besitzen beide eine z-Komponente. Diese Moden werden daher als hybride Moden bezeichnet. Die Reihenfolge der Buchstaben deutet an, welches Feld die größere z-Komponente aufweist.

Fasern, die nur eine sehr geringe Brechzahldifferenz zwischen Kern und Mantel aufweisen, werden schwach führende Fasern genannt und besitzen einige besondere Eigenschaften. In schwach führenden MMFs existieren mitunter mehrere Moden mit sehr ähnlichen Ausbreitungskonstanten. Das führt dazu, dass die Überlagerungen dieser Moden linear polarisierte Felder erzeugen. Mathematisch ist es möglich die EM-Wellen näherungsweise durch diese Überlagerungen zu beschreiben [10]. Die sich daraus ergebenden Näherungslösungen werden aufgrund der linearen Polarisation als linear polarisierte (LP) Moden bezeichnet.

Die Anzahl der Moden, die sich gleichzeitig in einer Faser ausbreiten können, hängt von einigen Eigenschaften der Faser ab. Diese Eigenschaften werden im sogenannten Faserparameter V zusammengefasst. Der Faserparameter ist als

$$V = \frac{2\pi a}{\lambda} \underbrace{\sqrt{n_K^2 - n_M^2}}_{NA} \tag{2.1}$$

definiert und somit von der Wellenlänge λ des übertragenen Lichts, dem Kernradius der Faser a und der numerischen Apertur NA abhängig. Nach [11, Kap. 5] kann die Anzahl der Moden für große Werte von V mit den Näherungsformeln

$$M_{SF} \approx \frac{V^2}{2} \qquad\qquad M_{GF} \approx \frac{V^2}{4} \tag{2.2}$$

bestimmt werden. Dabei bezeichnet M_{SF} die Anzahl der Moden in schwach führenden Stufenfasern und M_{GF} die Anzahl der Moden in Gradientenfasern mit parabolischem Profil. Liegt ein BV-Diagramm vor, kann dieses ebenfalls verwendet werden um die Anzahl Moden für einen gegebenen Faserparameter zu bestimmen. In einem BV-Diagramm ist der Phasenparameter B als Funktion des Faserparameters aufgetragen. Ein solches Diagramm ist in Abbildung 2.2 dargestellt.

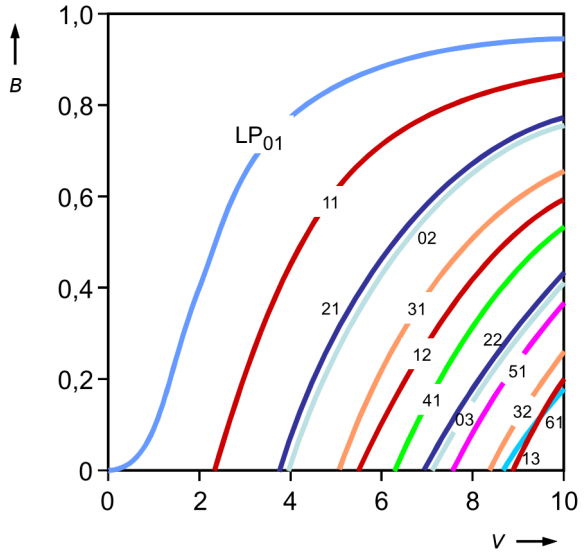


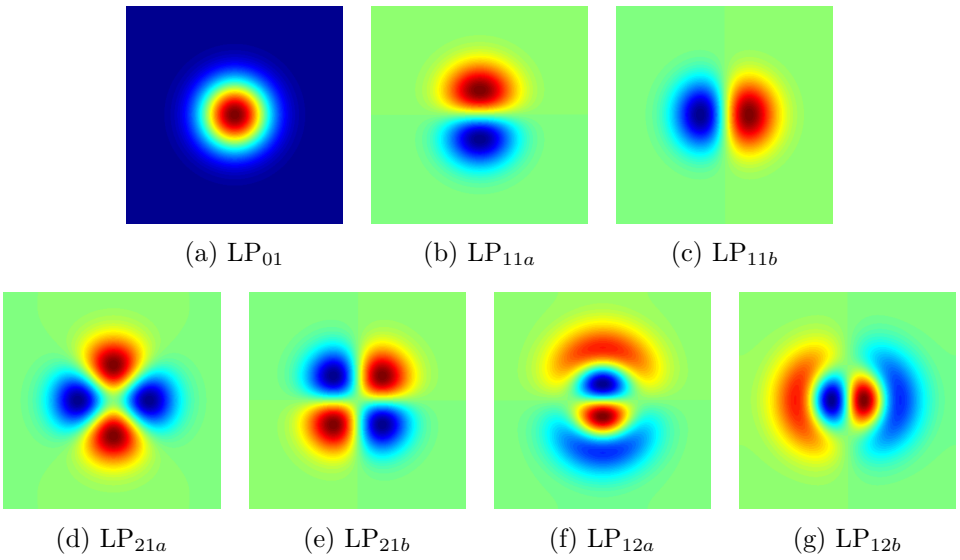
Abbildung 2.2.: Ein BV-Diagramm für die LP-Moden einer schwach führenden Stufenfaser. Die Grafik wurde entnommen aus [12].

Zur weiteren Unterscheidung der einzelnen Moden eines Typs, werden die Moden nach ihrer geometrischen Feldverteilung benannt. So bekommt jeder Modus einen Index zugewiesen, der die Anzahl der Nullstellen der Feldverteilung beschreibt. Die Feldverteilung eines LP_{mn}-Modus besitzt $2m$ Nullstellen in azimutaler Richtung und n Nullstellen in radialer Richtung. Da der azimutale Verlauf zudem durch einen Sinus oder Kosinus beschrieben werden kann, existieren die Moden mit $m > 0$ in zwei entarteten Varianten. Diese werden als gerader und ungerader Modus bezeichnet und mit den Indizes a und b gekennzeichnet. LP-Moden treten weiterhin in zwei verschiedenen Polarisationsrichtungen, der x- und y-Polarisation, auf. Die Feldverteilung einiger LP-Moden ist in Abbildung 2.3 dargestellt.

2.1.3. Lineare Effekte in Glasfasern

In Glasfasern treten durch die Materialeigenschaften der Faser verschiedene Effekte auf. Sie können je nach Anwendung als Störung wirken oder gezielt zur Signalmanipulation eingesetzt werden. Die Effekte werden in zwei Kategorien gegliedert: Lineare und nichtlineare Effekte. Sie unterscheiden sich in ihrer Abhängigkeit von der aktuell in der Faser geführten Leistung. In diesem Abschnitt werden zuerst die linearen Effekte vorgestellt. Die nichtlinearen Effekte folgen im nächsten Abschnitt.

Lineare Effekte sind unabhängig von der Leistung der EM-Welle in der Faser. Sie können jedoch sehr wohl von der Wellenlänge des Lichts oder anderen Eigenschaften abhängen. Lineare Effekte bewirken eine Änderung der Amplitude und/oder Phase der Welle. Die am deutlichsten in Erscheinung tretenden linearen Effekte umfassen die Dämpfung, die Dispersion und in MMFs die lineare Modenkopplung.



(a) LP_{01} (b) LP_{11a} (c) LP_{11b}

(d) LP_{21a} (e) LP_{21b} (f) LP_{12a} (g) LP_{12b}

Abbildung 2.3.: Feldverteilung einiger LP-Moden im Faserquerschnitt. Die Reihenfolge der Ausbreitungsfähigkeit der Moden kann dem BV-Diagramm 2.2 entnommen werden. Die Feldverteilungen wurden mit einem Modenlöser am Lehrstuhl berechnet.

Dämpfung

Mehrere Mechanismen führen zur Dämpfung der EM-Wellen in einer Glasfaser. Dämpfung beschreibt die Reduktion der Amplitude der Welle in Abhängigkeit von der durch die Faser zurückgelegten Strecke. Beschreibt $P(z)$ die Leistung der Welle in Abhängigkeit der z-Koordinate, so bewirkt die Dämpfung eine Leistungsabnahme gemäß

$$P(z) = P(0) \cdot \mathrm{e}^{-\alpha z} \tag{2.3}$$

wobei der Parameter α als Dämpfungskoeffizient bezeichnet wird. Der Dämpfungskoeffizient ist abhängig vom Fasermaterial und der Wellenlänge des Lichts. Bei der häufig verwendeten Wellenlänge von 1550 nm weisen Glasfasern einen Dämpfungskoeffizienten von circa 0,2 dB/km auf. Die Leistung des Signals halbiert sich somit alle 15 km.

Dämpfung wird hauptsächlich durch zwei Mechanismen erzeugt: Absorption und Rayleigh-Streuung. Bei der Absorption wird Energie der Welle in andere Energieformen umgewandelt. Dies kann beispielsweise durch Anregung von Atom- und Molekülschwingungen geschehen oder durch Anhebung von Elektronen in höhere Atomschalen. Im Gegensatz zur Absorption wird bei der Rayleigh-Streuung keine Energie umgewandelt, sondern abgelenkt. Durch Unregelmäßigkeiten im Material entstehen mikroskopische Brechzahlschwankungen, an denen ein Teil der Energie gestreut wird. Diese trägt somit nicht mehr zur Leistung der Welle in Ausbreitungsrichtung bei. Da beide Effekte, Absorption und Rayleigh-Streuung, frequenzabhängig sind, wirkt die Dämpfung unterschiedlich stark auf Licht verschiedener Wellenlängen. Wird allerdings nur ein begrenzter Spektralbereich betrachtet, kann der Dämpfungskoeffizient als konstant angenommen werden.

Dispersion

Die Ausbreitungskonstante β einer Welle in der Faser ist wie die Dämpfungskonstante abhängig von ihrer Wellenlänge und zusätzlich dazu auch abhängig vom betrachteten Modus. Es kann gezeigt werden, dass die Phasengeschwindigkeit v_{ph}, mit der sich ein Punkt konstanter Phase ausbreitet, und die Gruppengeschwindigkeit v_{gr}, mit der sich die Energie eines Wellenpakets ausbreitet, wie folgt mit der Ausbreitungskonstante $\beta(\omega)$ eines Modus verknüpft sind:

$$v_{ph} = \frac{\omega}{\beta(\omega)} \qquad \frac{1}{v_{gr}} = \frac{\partial\beta(\omega)}{\partial\omega} \tag{2.4}$$

Da sich die Information eines Signals mit der Gruppengeschwindigkeit ausbreitet, ist dies die interessante Größe. Die Abhängigkeit der Gruppengeschwindigkeit von verschiedenen Parametern hat daher einen eigenen Namen. Sie wird als Dispersion bezeichnet. Die Dispersion aufgrund der Frequenzabhängigkeit der Ausbreitungskonstante wird chromatische Dispersion genannt. Sie lässt sich an der Ableitung der Gruppengeschwindigkeit und somit an den höheren Ableitungen von $\beta(\omega)$ erkennen. Dagegen wird die Dispersion aufgrund unterschiedlicher Ausbreitungskonstanten verschiedener Moden als Modendispersion bezeichnet. In doppelbrechenden Materialien tritt diese Form der Dispersion sogar bei den unterschiedlichen Polarisationen desselben Modus auf. Dies wird als Polarisationsmodendispersion bezeichnet.

Die chromatische Dispersion bewirkt, dass sich die verschiedenen Frequenzanteile eines Signals unterschiedlich schnell ausbreiten. Abhängig davon, ob sich höhere oder niedrigere Frequenzen schnell ausbreiten, kommt es zu einem Zerlaufen oder Fokussieren des Signals. Ersteres wird als normale Dispersion und zweiteres als anomale Dispersion bezeichnet. Die Modendispersion bewirkt, dass unterschiedliche Moden sich verschieden schnell ausbreiten.

Die Ausbreitungskonstante $\beta(\omega)$ kann weiterhin mittels einer Taylor-Reihe

$$\beta(\omega) = \beta^{(0)} + \beta^{(1)}\,(\omega - \omega_0) + \frac{1}{2!}\,\beta^{(2)}\,(\omega - \omega_0)^2 + \frac{1}{3!}\,\beta^{(3)}\,(\omega - \omega_0)^3 + \ldots \tag{2.5}$$

um die Mittenfrequenz ω_0 dargestellt werden. Dabei ist $\beta^{(i)}$ als

$$\beta^{(i)} = \left.\frac{\partial^i \beta(\omega)}{\partial\omega^i}\right|_{\omega_0} \tag{2.6}$$

definiert. Der Parameter $\beta^{(0)}$ ist somit mit der Phasengeschwindigkeit des Signals assoziiert, während $\beta^{(1)}$ durch $\beta^{(1)} = \frac{1}{v_{gr}}$ mit der Gruppengeschwindigkeit verknüpft ist. Die chromatische Dispersion ist durch $\beta^{(2)}$ gegeben. Alle $\beta^{(i)}$-Terme höherer Ordnung, beschreiben Dispersion höherer Ordnung.

Lineare Modenkopplung

Lineare Modenkopplung ist ein Effekt, der durch lokale Inhomogenitäten der Fasereigenschaften hervorgerufen wird. Diese Inhomogenitäten können bei der Herstellung entstehen oder durch Umwelteinflüsse — wie zum Beispiel Temperatur, Biegung oder Druck — hervorgerufen werden. Durch die Inhomogenitäten entsteht ein Leistungsaustausch zwischen verschiedenen Moden und somit eine gegenseitige Störung der Signale. Diese lässt sich durch den Einsatz von

Multiple-Input Multiple-Output (MIMO) Techniken allerdings gut kompensieren, wie neuere Veröffentlichungen [13] zeigen. Eine ausführliche Beschreibung des Effekts und der daraus resultierenden Folgen ist in [8, Kap. 11] zu finden. Lineare Modenkopplung wird in dieser Arbeit nicht weiter betrachtet und ist im später vorgestellten mathematischen Modell nicht enthalten.

2.1.4. Nichtlineare Effekte in Glasfasern

Nichtlineare Effekte hängen von der Leistung der EM-Welle in der Glasfaser ab. Diese Abhängigkeit kann stören, da sie das Signal verzerrt und die maximale Leistung in einer Faser limitiert. Sie kann jedoch auch in aktiven optischen Komponenten wie Verstärkern oder Schaltern genutzt werden. Nichtlineare Effekte sind in Fasern zur Datenübertragung relativ gering und kommen daher erst bei sehr hohen Leistungen oder sehr langen Übertragungsstrecken zum Vorschein. Die wichtigsten nichtlinearen Effekte sind der Kerr-Effekt, die Raman-Streuung und die Brillouin-Streuung.

Kerr-Effekt

Der Kerr-Effekt beschreibt die nichtlineare Brechzahländerung der Faser durch elektrische Polarisation des Fasermaterials. Die Brechzahl n in Abhängigkeit der elektrischen Feldstärke E ist durch die Gleichung

$$n(E) = n_0 + n_2 \cdot |E|^2 \tag{2.7}$$

gegeben, wobei n_0 die lineare Brechzahl und n_2 die nichtlineare Brechzahl der Faser ist. Der Term $|E|^2$ entspricht der Leistung der Welle.

Die Brechzahlschwankung verursacht eine Phasenänderung der Welle. Da die Brechzahl wiederum durch das Signal moduliert wird, entsteht die sogenannte Selbstphasenmodulation (SPM) des Signals. Breiten sich mehrere Moden in der Faser aus, kommt es zudem zu einer gegenseitigen Beeinflussung der Signale und somit zur Kreuzphasenmodulation (XPM).

Die SPM verursacht ein Zerlaufen des Signalspektrums. In kurzen Fasern kann dies technisch genutzt werden um die Effekte der SPM und anomalen Dispersion gegenseitig zu kompensieren. Die Kreuzphasenmodulation stellt eine große Schwierigkeit bei der Übertragung mehrerer Moden in einer Faser dar, da sie Störungen verursacht, die sich nicht kompensieren lassen [2].

Des Weiteren erzeugt der Kerr-Effekt die sogenannte Vierwellenmischung (FWM). Bei eigentlicher FWM entsteht aus drei verschiedenen Frequenzen ω_1, ω_2, ω_3 in einem Signal nach der Vorschrift

$$\omega_4 = \omega_1 + \omega_2 - \omega_3 \tag{2.8}$$

eine neue Frequenz ω_4. Im Falle entarteter FWM mischen sich nur zwei Frequenzen ω_1, ω_2 durch

$$\omega_3 = 2 \cdot \omega_1 - \omega_2 \tag{2.9}$$

$$\omega_4 = 2 \cdot \omega_2 - \omega_1 \tag{2.10}$$

zu zwei weiteren Frequenzen ω_3, ω_4.

In aktuellen Arbeiten [14] wurde nachgewiesen, dass eigentliche FWM auch zwischen Signalen in verschiedenen Moden auftritt. In dieser Arbeit wird die intermodale FWM nicht weiter betrachtet, da derzeit keine Untersuchungen von WDM in Verbindung mit SDM vorgesehen ist. Intermodale FWM ist daher nicht durch das später vorgestellte mathematische Modell abgedeckt.

Raman- und Brillouin-Streuung

Die Raman- und Brillouin-Streuung sind weitere nichtlineare Effekt in Glasfasern. Sie entstehen durch die Interaktion von Photonen (Lichtquanten) mit Phononen (Energiequanten von Gitterschwingungen in Festkörpen).

Im Fall der Raman-Streuung findet eine Wechselwirkung zwischen den Wellen und Molekülschwingungen im Fasermaterial statt. Dabei wird ein Teil der Energie der Welle in eine andere Energieform überführt und eine neue Welle mit einer größeren Wellenlänge erzeugt. Dieser Effekt kann Störungen verursachen, da er sehr breitbandiges Übersprechen zwischen Frequenzkanälen erzeugt, kann aber auch technisch zur Signalverstärkung genutzt werden.

Die Brillouin-Streuung entsteht durch Wechselwirkung der Wellen mit hochfrequenten Schallwellen in der Glasfaser. Aufgrund der Schallwellen entstehen Brechzahlschwankungen im Fasermaterial, die sich mit der Geschwindigkeit der Schallwelle bewegen. Wird eine Welle an diesen Brechzahlsprüngen reflektiert, entstehen durch den Doppler-Effekt Wellen mit einer veränderten Wellenlänge. Die Schallwellen in der Faser können spontan durch thermische Schwingungen entstehen oder mithilfe des Effekts der Elektrostriktion durch bestimmte Wellen in der Faser angeregt werden. Die Brillouin-Streuung limitiert bei der Signalübertragung die verwendbare Leistung.

Raman- und Brillouin-Streuung sind im folgenden Modell nicht enthalten.

2.2. Mathematisches Modell

Zur Modellierung der EM-Wellen in einer Glasfaser genügt eine Betrachtung des elektrischen Felds. Das magnetische Feld lässt sich mithilfe der Maxwell-Gleichungen wieder eindeutig aus diesem rekonstruieren. Das zeit- und ortsabhängige elektrische Feld $E(t, z)$ wird durch die sogenannte langsam variierende Einhüllende $A(t, z)$ repräsentiert. Die komplexwertige langsam variierende Einhüllende hängt nach [9, Kap. 6] gemäß

$$E(t,z) = \mathrm{Re}\left\{A(t,z) \cdot F(x,y) \cdot \mathrm{e}^{-\mathrm{j}\,\beta^{(0)}\,z} \cdot \mathrm{e}^{\mathrm{j}\,\omega_0\,t}\right\} \tag{2.11}$$

mit dem elektrischen Feld zusammen, wobei $F(x, y)$ die Funktion der transversalen Feldverteilung des jeweiligen Modus ist und $\beta^{(0)}$ wie in Gleichung (2.6) definiert ist.

Die Entwicklung der langsam variierenden Einhüllenden entlang der z-Achse kann durch eine Differentialgleichung (DGL) beschrieben werden. Diese sogenannte Nichtlineare Schrödinger-Gleichung (Nonlinear Schrödinger Equation, NLSE) [15, Kap. 2] lautet

$$\frac{\partial A}{\partial z} = \underbrace{-\frac{\alpha}{2}A}_{\text{Dämpfung}} \underbrace{+\mathrm{j}\,\beta^{(0)}A - \beta^{(1)}\frac{\partial A}{\partial t} - \mathrm{j}\,\frac{\beta^{(2)}}{2}\frac{\partial^2 A}{\partial t^2}}_{\text{Dispersion}} + \underbrace{\mathrm{j}\,\gamma\,|A|^2\,A}_{\text{Kerr-Effekt}} \tag{2.12}$$

mit dem nichtlinearen Parameter $\gamma = \frac{\omega_0\, n_2}{c_0\, A_{eff}}$. Dabei bezeichnet c_0 die Vakuum-Lichtgeschwindigkeit und A_{eff} die effektive Modenfläche. Die Herleitung der NLSE aus der Wellengleichung wird ausführlich in [15, Kap. 2] erläutert.

In [16] wird die NLSE zur Beschreibung der Impulsausbreitung in MMFs erweitert. Diese erfasst damit auch intermodale Kopplung. Gemeint ist dabei nicht die lineare Modenkopplung, sondern die durch den Kerr-Effekte induzierte Wechselwirkung der Moden. Die entstehende verallgemeinerte nichtlineare Schrödinger-Gleichung für Mehrmodenfasern (Multi-Mode Generalized Nonlinear Schrödinger Equation, MM-GNLSE) wird in [17] für stark koppelnde Moden vereinfacht. Diese starke Kopplung tritt durch lineare Modenkopplung zwischen Moden mit sehr ähnlichen Ausbreitungskonstanten auf. Das sind insbesondere Moden in Gradientenfasern und alle LP_{mn} Moden in schwach führenden Stufenfasern, die gleiche Indizes m und n aufweisen. Wird die Kopplung stochastisch beschrieben, kann eine Mittelung der Kopplungsfaktoren in der MM-GNLSE durchgeführt werden. Dadurch ergibt sich die verallgemeinerte Manakov-Gleichung für Mehrmodenfasern (Multi-Mode Generalized Manakov Equation, MM-GME)

$$\frac{\partial \mathbf{A}}{\partial z} = -\frac{\alpha}{2}\mathbf{A} + \mathrm{j}\,\beta^{(0)}\mathbf{A} - \beta^{(1)}\frac{\partial \mathbf{A}}{\partial t} - \mathrm{j}\,\frac{\beta^{(2)}}{2}\frac{\partial^2 \mathbf{A}}{\partial t^2} + \mathrm{j}\,\gamma\,\kappa\,|\mathbf{A}|^2\,\mathbf{A} \tag{2.13}$$

wobei der Vektor $\mathbf{A}$ alle Moden umfasst. Der Kopplungsfaktor κ ergibt sich durch die stochastischen Betrachtungen und berechnet sich aus Überlappungsintegralen der in $\mathbf{A}$ enthaltenen Moden. Die Berechnung wird in [17] ausführlich beschrieben.

Die Autoren von [17] haben diesen Ansatz in [18] weiter verallgemeinert. Die stark koppelnden Moden in einer Faser werden jeweils zu einer Modengruppe $\mathbf{A}_{\mathfrak{a}}$ zusammengefasst. Zusätzlich können die verschiedenen Modengruppen untereinander schwach koppeln. Daraus resultiert das Gleichungssystem

$$\frac{\partial \mathbf{A}_{\mathfrak{a}}}{\partial z} = -\frac{\alpha}{2}\mathbf{A}_{\mathfrak{a}} + \mathrm{j}\,\beta_{\mathfrak{a}}^{(0)}\mathbf{A}_{\mathfrak{a}} - \beta_{\mathfrak{a}}^{(1)}\frac{\partial \mathbf{A}_{\mathfrak{a}}}{\partial t} - \mathrm{j}\,\frac{\beta_{\mathfrak{a}}^{(2)}}{2}\frac{\partial^2 \mathbf{A}_{\mathfrak{a}}}{\partial t^2} + \mathrm{j}\,\gamma\Bigg(\overbrace{\kappa_{\mathfrak{a}\mathfrak{a}}\,|\mathbf{A}_{\mathfrak{a}}|^2}^{\text{intramod. K.}} + \overbrace{\sum_{\mathfrak{b}\neq\mathfrak{a}}\kappa_{\mathfrak{a}\mathfrak{b}}\,|\mathbf{A}_{\mathfrak{b}}|^2}^{\text{intermod. K.}}\Bigg)\mathbf{A}_{\mathfrak{a}} \tag{2.14}$$

wobei $\kappa_{\mathfrak{a}\mathfrak{b}}$ der Kopplungsfaktor der Gruppen $\mathfrak{a}$ und $\mathfrak{b}$ ist. Die Bestandteile der Gleichung, die die intramodale und intermodale Kopplung ausdrücken, sind entsprechend gekennzeichnet. Gleichung 2.14 stellt im Weiteren die Grundlage zur Simulation der Wellenausbreitung in Glasfasern dar.

2.3. Numerische Lösung mithilfe der Split-Step-Fourier-Methode

Die Simulation der Signalausbreitung entspricht der Lösung eines Anfangswertproblems zum DGL-System (2.14). Dabei ist das Signal $A(t, z = 0)$ am Faseranfang bekannt. Dies ist der Anfangswert. Gesucht wird das Signal $A(t, z = l_F)$ am Faserende l_F. Da die DGL nicht analytisch

lösbar ist, muss zur Lösung ein numerisches Verfahren verwendet werden. Als ein mögliches Verfahren hat sich die Split-Step-Fourier-Methode (SSFM) etabliert. Diese bietet im Gegensatz zu anderen Verfahren, wie zum Beispiel Finite-Differenzen-Methoden, insbesondere eine höhere Geschwindigkeit. Die nachfolgenden Erläuterungen orientieren sich an [19].

2.3.1. Die Split-Step-Fourier-Methode

Die SSFM wird der Einfachheit halber anhand der Gleichung (2.13) erläutert. Das Verfahren lässt sich jedoch auch auf mehrere in einer Glasfaser propagierende Modengruppen erweitern.

Allgemeine Funktionsweise

Die Manakov-Gleichung (2.13) kann als

$$\frac{\partial \mathbf{A}(t,z)}{\partial z} = \{\mathcal{L} + \mathcal{N}(t,z)\} \circ \mathbf{A}(t,z) \tag{2.15}$$

mit dem linearen Operator $\mathcal{L}$ und dem nichtlinearen Operator $\mathcal{N}$ dargestellt werden. Diese sind als

$$\mathcal{L} = -\frac{\alpha}{2} + \mathrm{j}\,\beta^{(0)} - \beta^{(1)}\frac{\partial}{\partial t} - \mathrm{j}\,\frac{\beta^{(2)}}{2}\frac{\partial^2}{\partial t^2} \tag{2.16}$$

$$\mathcal{N}(t,z) = \mathrm{j}\,\gamma\,\kappa\,|\mathbf{A}(t,z)|^2 \tag{2.17}$$

definiert, wobei $\{\star\} \circ \mathbf{A}(t,z)$ die Anwendung des Operators $\star$ auf $\mathbf{A}(t,z)$ bezeichnet.
Die exakte Lösung der Gleichung (2.15) lautet

$$\mathbf{A}(t,z+h) = \left\{\exp\left(\int_z^{z+h} \mathcal{L} + \mathcal{N}(t,z')\,\mathrm{d}z'\right)\right\} \circ \mathbf{A}(t,z) \quad . \tag{2.18}$$

Das Integral kann jedoch nicht für eine beliebige Schrittweite h berechnet werden, da $\mathcal{N}(t,z')$ nicht für alle $z' \in [z, z+h]$ bekannt ist. Für kleine $h \to 0$ kann $\mathcal{N}(t,z')$ jedoch als konstant angenommen werden, wodurch sich die Näherung

$$\mathbf{A}(t,z+h) \approx \{\exp\left([\mathcal{L} + \mathcal{N}(t,z)] \cdot h\right)\} \circ \mathbf{A}(t,z) \tag{2.19}$$

ergibt. Da $\mathcal{N}(t,0)$ durch die Anfangswerte bekannt ist, kann durch k-fache Anwendung dieser Gleichung das Signal $\mathbf{A}(t, k\cdot h)$ für beliebige $k \in \mathbb{N}_0$ berechnet werden. Daher wird die simulierte Faser in K Segmente unterteilt, sodass $\mathbf{A}(t,l_F)$ die Bedingung $\mathbf{A}(t,l_F) = \mathbf{A}(t,K\cdot h)$ erfüllt. Die Lösung wird schrittweise für jedes Segment berechnet, bis das Faserende erreicht ist. Dieses Prinzip ist nochmals in Abbildung 2.4 veranschaulicht.

Die Wahl der Schrittweite h beeinflusst maßgeblich die Genauigkeit und die Geschwindigkeit des Verfahrens. Bei großen Schrittweiten wächst der Fehler in Gleichung (2.19) an, da die Approximation durch ein konstantes $N(t,z')$ ungenauer wird. Gleichzeitig steigt mit kleiner Schrittweite die Anzahl der Segmente und somit die Berechnungsdauer der Simulation linear an. Daher sind zahlreiche Ansätze zur Schrittweitenwahl entwickelt worden, die einen Kompromiss zwischen Genauigkeit und Rechenaufwand bilden. In [20] wird ein Überblick über die gängigsten Verfahren gegeben. Viele dieser Ansätze basieren im Gegensatz zu der hier vorgestellten Variante darauf, dass sich die Schrittweite verschiedener Segmente unterscheidet. Im Weiteren wird jedoch eine konstante Schrittweite verwendet.

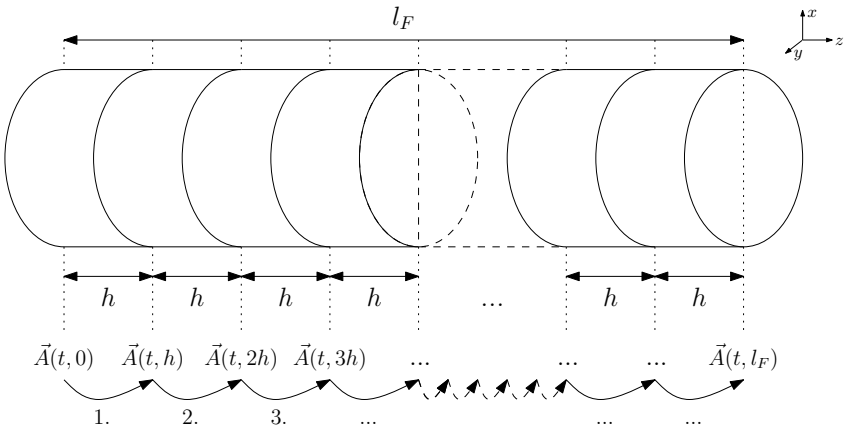


Abbildung 2.4.: Schematische Darstellung der Split-Step-Fourier-Methode. Die simulierte Glasfaser der Länge l_F wird in Segmente der Länge h unterteilt. Ausgehend vom Faseranfang wird die Ausbreitung des Signals Segment für Segment simuliert bis das Faserende erreicht ist.

Simulation eines Segments

Zur Simulation eines Fasersegments muss die Gleichung (2.19) gelöst werden. Da der lineare Operator zeitliche Ableitungen umfasst, lässt sich die entsprechende DGL durch Transformation in den Frequenzbereich lösen. Der nichtlineare Operator muss jedoch im Zeitbereich berechnet werden, da er das Betragsquadrat der Einhüllenden beinhaltet. Für den nichtlinearen Operator $\mathcal{N}(t,z)$ wird nachfolgend die Kurzschreibweise $\mathcal{N}$ verwendet. Der Gesamtoperator $\exp\left([\mathcal{L}+\mathcal{N}]\cdot h\right)$ wird daher zur Trennung der Operatoren durch die Näherung

$$\exp\left([\mathcal{L}+\mathcal{N}]\cdot h\right) \approx \exp\left(\mathcal{L}\cdot h\right)\cdot\exp\left(\mathcal{N}\cdot h\right) \tag{2.20}$$

beschrieben. Der lineare Anteil wird dann im Frequenzbereich durch

$$\mathbf{A}(t,z+h) = \mathcal{F}^{-1}\left\{\mathcal{F}\left\{\mathbf{A}(t,z)\right\}\cdot\exp(\mathcal{F}\left\{\mathcal{L}\right\}\cdot h)\right\} \tag{2.21}$$

gelöst. Der nichtlineare Anteil wird im Zeitbereich als

$$\mathbf{A}(t,z+h) = \mathbf{A}(t,z)\cdot\exp(\mathcal{N}\cdot h) \tag{2.22}$$

berechnet.

Der Ansatz (2.20) ist nur eine Näherung, da $\mathcal{L}$ und $\mathcal{N}$ zwei nicht kommutierende Operatoren sind. Nach der Baker-Campell-Hausdorff-Formel

$$\exp\left(\mathcal{L}\cdot h\right)\cdot\exp\left(\mathcal{N}\cdot h\right) = \exp\left(\underbrace{\mathcal{L}\cdot h+\mathcal{N}\cdot h}_{\text{gewünschter Term}} + \underbrace{\mathcal{L}\mathcal{N}\cdot\frac{h^2}{2}-\mathcal{N}\mathcal{L}\cdot\frac{h^2}{2}+\mathcal{O}(h^3)}_{\text{Separations-Fehler}}\right) \tag{2.23}$$

ergibt sich daher ein lokaler Fehler der Größenordnung $\mathcal{O}(h^2)$.

Aufgrund der Trennung der Operatoren wird die Berechnung eines Segments als *Split-Step* bezeichnet. Daraus — und aus der Fourier-Transformation zur Lösung des linearen Anteils —

ergibt sich der Name Split-Step-Fourier-Methode. Die Berechnung des linearen Anteils nach Gleichung (2.21) wird dabei als linearer Teilschritt und die Berechnung des nichtlinearen Anteils nach Gleichung (2.22) als nichtlinearer Teilschritt bezeichnet.

2.3.2. Verringerung des Separations-Fehlers durch die symmetrische SSFM

Zur Reduktion des durch Trennung der nicht kommutierenden Operatoren $\mathcal{L}$ und $\mathcal{N}$ verursachten Fehlers kann alternativ zum Ansatz (2.20) der Trennungsansatz

$$\exp\left([\mathcal{L}+\mathcal{N}]\cdot h\right) \approx \exp\left(\mathcal{L}\cdot\frac{h}{2}\right)\cdot\exp\left(\mathcal{N}\cdot h\right)\cdot\exp\left(\mathcal{L}\cdot\frac{h}{2}\right) \tag{2.24}$$

gewählt werden. Mithilfe der Baker-Campell-Hausdorf-Formel ergibt sich dann aus

$$\exp\left(\mathcal{L}\cdot\frac{h}{2}\right)\cdot\exp\left(\mathcal{N}\cdot h\right)\cdot\exp\left(\mathcal{L}\cdot\frac{h}{2}\right) = \exp\left(\mathcal{L}\cdot h+\mathcal{N}\cdot h+\mathcal{O}(h^3)\right) \tag{2.25}$$

nur noch ein Fehler der Größenordnung $\mathcal{O}(h^3)$, anstatt des vorherigen Fehlers der Größenordnung $\mathcal{O}(h^2)$. Die Vertauschung des linearen und nichtlinearen Operators führt ebenfalls zu einem Fehler der Größenordnung $\mathcal{O}(h^3)$. Dieses Vorgehen wird aufgrund der Anordnung der Operatoren als symmetrische SSFM bezeichnet.

2.3.3. Verbesserte Approximation des nichtlinearen Operators

Beim Übergang von Gleichung (2.18) zu Gleichung (2.19) wurde das Integral durch

$$\int_z^{z+h} \mathcal{L}+\mathcal{N}(t,z')\,\mathrm{d}z' \approx [\mathcal{L}+\mathcal{N}(t,z)]\cdot h \tag{2.26}$$

angenähert. Somit wurde für $\mathcal{N}(t,z')$ auf dem Intervall $[z,z+h]$ der konstante Wert $\mathcal{N}(t,z)$ angenommen. Stattdessen kann die Näherung des Integrals mithilfe der Trapezregel

$$\int_z^{z+h} \mathcal{N}(t,z')\,\mathrm{d}z' \approx h\cdot\frac{\mathcal{N}(t,z)+\mathcal{N}(t,z+h)}{2} \tag{2.27}$$

verbessert werden. Da $\mathcal{N}(t,z+h)$ jedoch nicht im Voraus bekannt ist, muss jeder Split-Step mehrfach berechnet werden. Beim ersten Durchlauf wird die bekannte Näherungsformel (2.26) verwendet, um eine anfängliche Schätzung für $\mathcal{N}(t,z+h)$ zu erzeugen. Diese wird beim folgenden Durchlauf in der Trapezregel verwendet, um eine bessere Schätzung für $\mathcal{N}(t,z+h)$ zu erhalten. Das Verfahren kann prinzipiell beliebig oft wiederholt werden; laut [15, Kap. 2] sind jedoch zwei Iterationen ausreichend, da sich die Genauigkeit der Ergebnisse durch weitere Wiederholungen nicht mehr erhöht.

3. Einführung in die GPU-Programmierung mit Nvidia CUDA

Nvidia CUDA (Compute Unified Device Architecture, CUDA) [21] ist eine Plattform zur Programmierung aktueller Nvidia-Grafikkarten. Zum Verständnis der Arbeit ist es nötig einige Begriffe und Grundlagen der verwendeten Grafikkartenarchitektur sowie des CUDA Programmiermodells zu kennen. Daher wird diese im folgenden Kapitel zusammengefasst. Das Kapitel beginnt mit einer Erläuterung der Grundidee von GPU-gestütztem Rechnen. Danach wird die CUDA-Architektur vorgestellt. Es wird die Hardware einer CUDA-GPU zusammengefasst und das Grundprinzip der Parallelisierung unter CUDA erläutert. Dann folgt eine detailliertere Vorstellung der unterschiedlichen Speichertypen auf der GPU. Schließlich wird die Nvidia cuFFT-Bibliothek (cuFFT) für schnelle Fourier-Transformationen vorgestellt. Die Erklärungen beschränken sich auf die notwendigen Punkte, um ein ausreichendes Bild der CUDA-Architektur zu erhalten. Weniger bekannte Begriffe der Informatik, die in dieser Arbeit verwendet werden, werden in Anhang C zusammengefasst und erläutert.

3.1. Die Grundidee

Heutige Hauptprozessoren (Central Processing Unit, CPUs) sind darauf optimiert seriellen Code auszuführen. Dazu sind ihre Rechenkerne so entworfen, dass die einzelnen Befehle einer Befehlskette sehr schnell abgearbeitet werden und dass die Ausführung nicht durch Abhängigkeiten zwischen aufeinanderfolgenden Befehlen unterbrochen wird. Um dies zu erreichen sind sehr komplexe — und somit teure — Rechenkerne nötig. Daher werden nur wenige Kerne in einer CPU verbaut. GPUs nutzen hingegen den umgekehrten Ansatz. Sie bestehen aus einfach aufgebauten — und somit günstigen — Rechenkernen, die seriellen Code nur sehr ineffizient abarbeiten können. Im Gegenzug enthalten sie zahlreiche Kerne und können daher parallel eine große Anzahl von Daten verarbeiten. Somit kann die Auslagerung von Berechnungen auf eine GPU immer dann zu einer hohen Beschleunigung führen, wenn sich die bearbeitete Datenmenge in viele unabhängige Teile zerlegt lässt, die parallel verarbeitet werden können.

3.2. Die Architektur

Der Hauptrechner, bestehend aus einer klassischen CPU und Hauptspeicher (Random Access Memory, RAM), wird unter CUDA als *Host* bezeichnet. Die CPU ist die oberste Instanz in der CUDA-Architektur. Sie verwaltet die Ressourcen des Systems und instruiert die Berechnungen auf dem *Device*. Als Device wird unter CUDA die GPU bezeichnet. Eine Übersicht der einzelnen Komponenten auf Host- und Device-Seite ist in Abbildung 3.1 dargestellt.

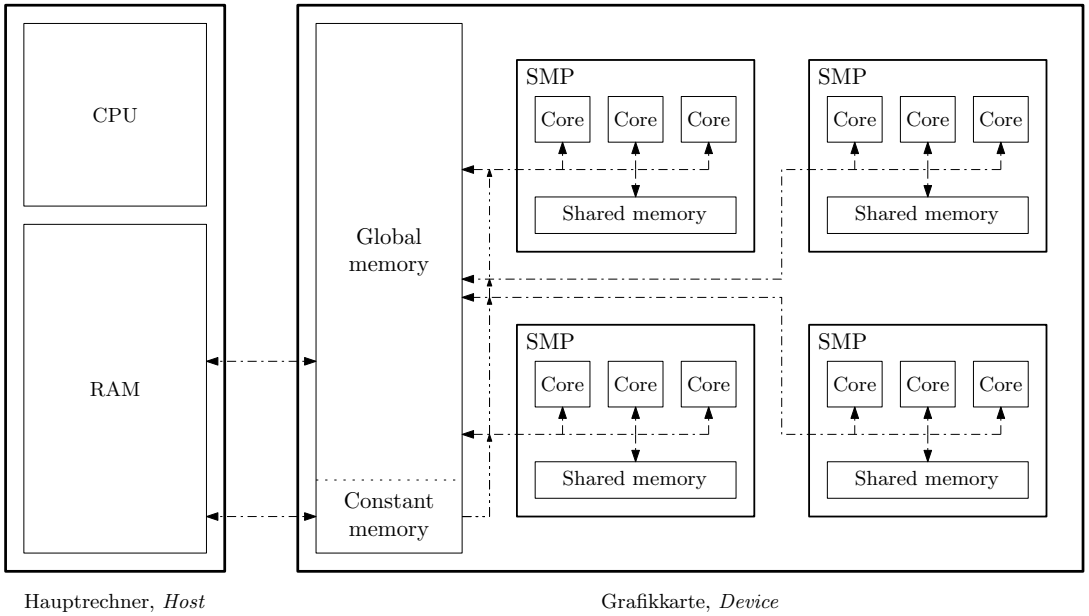


Abbildung 3.1.: Hardware-Komponenten der CUDA-Architektur.

Die Rechenkerne des Device (in der Abbildung mit „Core“ gekennzeichnet) sind in größeren Recheneinheiten, den sogenannten Streaming-Multiprozessoren (SMPs) gruppiert. Alle Kerne innerhalb eines SMP bearbeiten synchron denselben Programm-Teil auf verschiedenen Daten. Sie können zudem durch den *shared memory* Daten untereinander austauschen. Verschiedene SMPs arbeiten dagegen unabhängig voneinander. Sollen Daten zwischen SMPs ausgetauscht werden, dann müssen diese im *global memory* abgelegt werden. Da die SMPs im Allgemeinen asynchron zu einander arbeiten, sind Zugriffe auf den *global memory* jedoch nur mit größerem Aufwand synchronisierbar. Neben dem *global memory* können alle SMPs auch auf einen gemeinsamen *constant memory* zugreifen. Dieser kann vom Device aus jedoch nur gelesen und nicht beschrieben werden. Der Datenaustausch zwischen Host und Device findet statt, indem Daten zwischen RAM und *global memory* hin und her kopiert werden. Auf die Eigenschaften der unterschiedlichen Speichertypen wird später noch genauer eingegangen. Ein typischer Programmablauf sieht daher so aus, dass zuerst alle Daten vom Host in den *global memory* und gegebenenfalls den *constant memory* kopiert werden. Dann wird die Programm-Ausführung auf dem Device gestartet. Jeder Rechenkern liest Daten aus dem *global memory* und *constant memory* aus, verarbeitet sie und speichert sie schließlich wieder im *global memory* ab. Sind alle Berechnungen abgeschlossen, werden die Inhalte des *global memory* zuletzt zurück zum Host kopiert.

3.2.1. Parallelität unter CUDA

Parallelität wird unter CUDA erzeugt, indem zahlreiche *Threads* generiert werden. Ein Thread ist gewissermaßen eine Befehlsfolge, die parallel zu anderen Threads ausgeführt werden kann. Die Befehle beschreiben die Operationen, die auf die Daten angewandt werden sollen. Im einfachsten Fall wird für jedes Datum des zu verarbeitenden Datensatzes ein eigener Thread generiert. Es ist jedoch auch üblich innerhalb eines Threads mehrere Elemente nacheinander zu

verarbeiten. Die Vorteile dieses Vorgehens werden in [22] erläutert. Die Befehlsfolge, die von jedem Thread ausgeführt wird, kann als eine `C/C++` Funktion betrachtet werden, die jeder Thread einmalig durchläuft. Diese „Funktion“ wird unter CUDA als *Kernel* bezeichnet. Ein Kernel hat jedoch keinen Rückgabewert und kann daher nur durch Manipulation des *global memory* Ergebnisse zurückgeben. Außerdem erhält jeder Thread dieselben Funktionsparameter. Damit die Threads voneinander unterschieden werden können, werden diese mit einer aufsteigenden *Thread-ID* durchnummeriert. Diese ID kann im Kernel-Programm abgefragt werden, um beispielsweise durch Zeiger-Arithmetik die zu verarbeitenden Datenbereiche zu bestimmen.

Da meist deutlich mehr Threads erzeugt werden, als Rechenkerne auf dem Device vorhanden sind, können nicht alle Threads parallel ablaufen. Stattdessen werden die Threads in *Thread-Blöcke* eingeteilt. Alle Threads in einem Block werden gleichzeitig von einem SMP ausgeführt und können über den *shared memory* Daten untereinander austauschen. Die verschiedenen Blöcke werden jedoch vollkommen unabhängig voneinander ausgeführt. Es werden immer nur so viele Blöcke gleichzeitig ausgeführt, wie SMPs vorhanden sind. Hat ein SMP einen Block vollständig ausgeführt, beginnt er mit dem nächsten Block, bis alle Blöcke abgearbeitet sind. Die Threads in verschiedenen Blöcken können daher nicht miteinander interagieren. Die Anzahl der Threads in einem Block wird durch die Hardware der SMPs limitiert. Die Anzahl der Blöcke kann in der Praxis als unbegrenzt betrachtet werden.

3.2.2. Speichertypen

Auf dem Device existieren drei wichtige Speichertypen mit unterschiedlichen Eigenschaften und Funktionen. Davon ist der *global memory* der bedeutendste, da er als einziger Speicher unabdingbar ist, um das Device zu verwenden. *shared memory* und *constant memory* dienen lediglich der Erhöhung der Performance, da sie höhere Zugriffsgeschwindigkeiten als der *global memory* bieten.

Global Memory

Der *global memory* dient dem Device als Hauptspeicher, da alle Komponenten auf ihn zugreifen können. Daher kann er mehrere Gigabyte an Daten speichern. Weiterhin ist der *global memory* die Schnittstelle um Daten zwischen Host und Device auszutauschen.

Aufgrund der Bauweise des *global memory* erreicht er seine maximale Datenrate nur, wenn eine bestimmte Menge zusammenhängender Daten zeitgleich gelesen wird. Zusammenhängend bedeutet in diesem Fall, dass die Daten sich lückenlos über einen Speicherbereich erstrecken. Da es nicht möglich ist mit einem einzelnen Thread ausreichend viele Daten zu lesen, müssen mehrere Threads „kooperieren“. Die entsprechenden Threads müssen dabei alle in einem Thread-Block liegen und lückenlos durchnummerierte Thread-IDs aufweisen. Da die Threads synchron auf einem SMP ausgeführt werden, fordern sie alle gleichzeitig Daten an. Der SMP kann dann die einzelnen Anfragen zu einer großen Anfrage an den *global memory* kombinieren und damit einen ausreichend großen zusammenhängenden Speicherbereich anzusprechen.

Der Host erreicht beim Zugriff auf den *global memory* im Vergleich zu den Komponenten des Device nur eine sehr niedrige Datenrate bei vergleichsweise hoher Latenz. Um den Performanceverlust durch diesen Umstand zu mindern, sind auf neueren Nvidia-GPUs mehrere sogenannte

Copy-Engines verbaut. Diese ermöglichen es, dass parallel zu anderen Vorgängen auf dem Device Daten zwischen Host und Device ausgetauscht werden. So ist es beispielsweise möglich einen Datensatz auf das Device zu kopieren, während ein anderer Datensatz vom Device herunter kopiert wird und parallel Berechnungen auf einem dritten Datensatz stattfinden.

Constant Memory

Der *constant memory* ist eine Sonderform des *global memory*. Er umfasst nur einige Kilobyte und kann von Komponenten auf dem Device ausschließlich gelesen, aber nicht beschrieben, werden. Diese Eigenschaft erlaubt es, dass Daten, die von einem Rechenkern aus dem *constant memory* gelesen werden, gecached werden. Caching bedeutet, dass die Daten nach dem ersten Lesen in einem schnellen Zwischenspeicher auf dem SMP abgelegt werden. Will der gleiche Kern oder ein anderer Kern auf demselben SMP nachfolgend dasselbe Datum nochmals lesen, wird es direkt aus dem schnellen Zwischenspeicher geladen. Wiederholte Zugriffe auf Daten im *constant memory* erreichten daher deutlich höhere Datenraten und deutlich geringere Latenzen als bei Zugriffen auf den *global memory* möglich sind.

Shared Memory

Der *shared memory* ist ein einige Kilobyte großer Speicher auf jedem SMP. Wie bereits erwähnt teilen sich alle Kerne auf einem SMP denselben *shared memory* und alle Threads auf diesen Kernen können somit untereinander Daten darüber austauschen. Im Vergleich zum *global memory* erreicht der *shared memory* sehr hohe Datenraten und sehr geringe Latenzen. Vom Host aus kann jedoch nicht auf den *shared memory* zugegriffen werden. Er eignet sich daher vor allem zur Speicherung von Zwischenergebnissen oder dem manuellen Caching von Daten aus dem *global memory*.

3.3. Nvidia cuFFT-Bibliothek

Mithilfe der Nvidia cuFFT-Bibliothek [23] können GPU-gestützt schnelle Fourier-Transformationen (Fast Fourier Transform, FFTs) berechnet werden. Im Folgenden werden die für diese Arbeit relevanten Begrifflichkeiten und Eigenschaften der cuFFT vorgestellt.

Das mithilfe der cuFFT zu transformierende Signal muss vollständig auf dem Device gespeichert sein. Vor der Durchführung einer Transformation muss ein sogenannter cuFFT-Plan erstellt werden. Dieser Plan umfasst die Konfiguration des cuFFT-internen FFT-Algorithmus sowie Pufferspeicher, der während der eigentlichen Transformation verwendet wird. Dies bietet den Vorteil, dass die zeitaufwändige Erstellung einer optimierten Konfiguration und die Allokation von Pufferspeicher nur einmalig durchgeführt werden müssen, selbst wenn die Transformation mehrfach ausgeführt wird. Für die Transformation verschieden langer Signale müssen jedoch unterschiedliche Pläne erstellt werden und ein einzelner cuFFT-Plan kann nicht gleichzeitig für mehrere Transformationen verwendet werden. Der Nachteil der cuFFT-Pläne ist, dass der Pufferspeicher für die FFT auch dann von cuFFT reserviert wird, wenn zwischenzeitlich keine FFT ausgeführt wird.

Für diese Arbeit werden ausschließlich eindimensionale Transformationen von komplexwertigen Abtastwerten durchgeführt. Das cuFFT-Nutzerhandbuch [24] nennt keine exakte Größe des dazu benötigten Pufferspeichers. Eigenen Analysen zufolge entspricht die Puffergröße hierbei jedoch immer der Größe des Signals.

Das Ergebnis einer Transformation kann entweder das transformierte Signal überschreiben (in-place Transformation) oder in einem separaten Speicherbereich gespeichert werden (out-of-place Transformation). Laut cuFFT-Nutzerhandbuch ergibt sich bei der Transformation von reellen Eingangsdaten eine höhere Geschwindigkeit bei Verwendung der out-of-place Transformation. Für den hier relevanten Anwendungsfall der Transformation von komplexwertigen Signalen konnte eigenen Beobachtungen nach jedoch kein signifikanter Geschwindigkeitsunterschied zwischen den Varianten festgestellt werden.

4. Implementierung

Am Lehrstuhl für Hochfrequenztechnik existiert eine Software zur Simulation der Impulsausbreitung in Mehrmodenfasern. Sie wurde im Rahmen einer Masterarbeit [19] entwickelt und später derart erweitert, dass ein Großteil der Berechnungen GPU-gestützt durchgeführt werden kann. Ein integraler Bestandteil dieser Software ist eine SSFM-Implementierung in CUDA. Diese Arbeit beschäftigt sich mit der Optimierung des GPU-seitigen Speicherverbrauchs der SSFM-Implementierung.

In diesem Kapitel wird zuerst die zugrundeliegende Implementierung beschrieben. Dann folgt ein Absatz über die Reduktion des Speicherbedarfs durch Modifikation einzelner Abschnitte der Implementierung. Anschließend wird erläutert, wie sich die SSFM in Teilprobleme zerlegen lässt, die jeweils auf einem abgegrenzten Bereich der Signaldaten arbeiten. Darauf basierend wird ein System eingeführt, das es ermöglicht jederzeit nur einen Teil der Daten auf der GPU speichern zu müssen. Zuletzt wird die finale Implementierung nach Umsetzung der diskutierten Veränderungen vorgestellt.

4.1. Datenformat

Der Algorithmus zur Durchführung der SSFM bekommt die Simulationsparameter, Faserparameter und Signale am Faseranfang als Eingabewerte und gibt die daraus errechneten Signale am Faserende zurück. Die Daten sind wie folgt gespeichert:

- Ein Signal wird durch einen komplexen Zeilenvektor von N Abtastwerten der komplexen Einhüllenden repräsentiert. Alle M Signale zusammen ergeben eine Signalmatrix der Dimension $M \times N$.
- Der lineare Operator der Faser wird ebenfalls durch eine $M \times N$ Matrix repräsentiert. Diese ist identisch zu den Signalmatrizen aufgebaut, sodass ein Element $\mathcal{L}_{mn}$ des linearen Operators bei der Verrechnung im Frequenzbereich auf das entsprechende Element $\mathbf{A}_{mn}^{\text{freq}}$ des Signals im Frequenzbereich angewandt wird.
- Der nichtlineare Operator ist abhängig von der momentanen Leistung der Signale und somit abhängig vom aktuellen Signal in jedem Zwischenschritt. Er kann daher nicht im Voraus berechnet werden und muss während jeder Durchführung des nichtlinearen Schritt bestimmt werden. Neben dem aktuellen Signal sind dazu auch die Kopplungsfaktoren κ notwendig. Diese werden als Matrix der Größe $M \times M$ gespeichert.
- Alle weiteren Parameter, wie beispielsweise die Schrittweite h und der Faserparameter γ, sind skalare Werte.

4.2. Vorstellung der vorliegenden Implementierung

Wenn im Folgenden von der „vorliegenden“ oder „ursprünglichen“ Implementierung die Rede ist, ist eine leicht modifizierte Variante der zu Beginn der Arbeit bereitgestellten Implementierung gemeint. Diese Variante enthält kleine Korrekturen und Optimierungen, deren Beschreibung im Rahmen dieser Arbeit nicht von Interesse ist. Dies dient vor allem dem fairen Vergleich der Implementierungen bei der Analyse in Kapitel 5. Die Beschreibung der Implementierung beschränkt sich auf die relevanten Elemente, um die Funktionsweise zu verstehen und die später durchgeführten Veränderungen einzuordnen.

4.2.1. Programm-Ablauf

Die Implementierung verwendet die in Abschnitt 2.3 vorgestellte symmetrische SSFM mit Anwendung der Trapezregel zur Lösung des nichtlinearen Schritts. Der im Folgenden erläuterte Programm-Ablauf ist grafisch in Abbildung 4.1 dargestellt. Alle im *global memory* allokierten Ressourcen sind in Tabelle 4.1 zusammengefasst. Die Matrix der nichtlinearen Kopplungsfaktoren κ ist dort nicht aufgeführt, da sie im *constant memory* des Device gespeichert wird.

Tabelle 4.1.: Auflistung der durch die vorliegende Implementierung allokierten Speicherbereiche im *global memory*.

Variablenname	Enthaltene Daten
LINEAR	Der lineare Operator (konstant)
LINEAR_EFFECT	Die linearen Effekte über eine halbe Schrittweite
AA	Zumeist die gerade berechneten Signale
A_FREQ	Die Signale im Frequenzbereich (während eines linearen Halbschritts)
A_MID	Die Signale nach dem ersten linearen Halbschritt
A0_SQABS	Die Betragsquadrate am Anfang des Split-Step
MEAN_SQABS	Die gemittelten Betragsquadrate (Trapezregel)

Die Implementierung der SSFM besteht zum Großteil aus einer Schleife, die über alle Split-Steps iteriert und dabei jeweils die Signalausbreitung durch ein Segment simuliert. Nach dem letzten Schleifendurchlauf ist somit das Signal am Ende des letzten Split-Step bekannt und kann zurückgegeben werden. Da ursprünglich alle Daten im Host-Speicher liegen und das Ergebnis ebenfalls wieder dort gespeichert sein soll, werden zu Beginn der Simulation das Eingangssignal und der lineare Operator auf das Device kopiert und zum Schluss das Ergebnis wieder zurück kopiert. Dazwischen findet kein Datentransfer zwischen Host und Device statt. Der Algorithmus zur Simulation eines Split-Step läuft, analog zu Abbildung 4.1, wie folgt ab. Die Nummerierung der Aufzählung entspricht dabei der Nummerierung auf der linken Seite der Abbildung.

1. Zuerst werden die linearen Effekte LINEAR_EFFECT über die halbe Schrittweite berechnet. Da die Schrittweiten in einer Simulation oft konstant gewählt werden, erfolgt ihre Neuberechnung nur bei einer Änderung der Schrittweite.

2. Dann werden die Betragsquadrate A0_SQABS der Signale am Anfang des Split-Steps berechnet, da diese später nicht mehr ermittelt werden können, wenn AA überschrieben wurde.

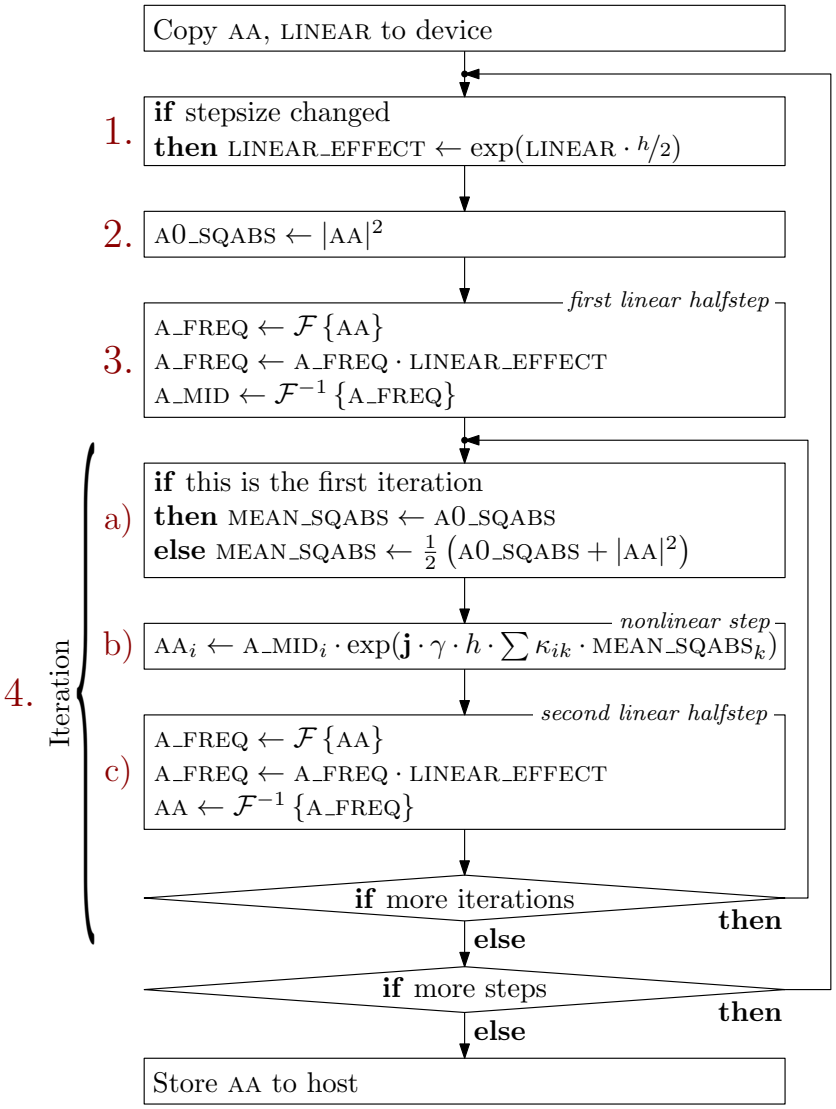


Abbildung 4.1.: Das Ablaufdiagramm der vorliegenden Implementierung.

3. Danach wird der erste lineare Halbschritt durchgeführt. Zuerst wird die Fourier-Transformierte von AA gebildet und in A_FREQ zwischengespeichert. A_FREQ wird dann elementweise mit LINEAR_EFFECT multipliziert. Schließlich wird die inverse Fourier-Transformation von A_FREQ berechnet und das Ergebnis in A_MID gespeichert.

4. Nach dem ersten linearen Halbschritt erfolgt laut SSFM die Berechnung des nichtlinearen Schritt. Aufgrund der Trapezregel wird die Berechnung des nichtlinearen Schritt und des zweiten linearen Halbschritt dabei in der folgenden Schleife wiederholt:

   a) Zuerst wird das gemittelte Betragsquadrat MEAN_SQABS bestimmt. In der ersten Iteration entspricht dies per Definition A0_SQABS. In allen folgenden Iterationen wird der Mittelwert von A0_SQABS und AA gebildet. AA enthält zu diesem Zeitpunkt aufgrund der vorhergehenden Iterationen bereits eine Schätzung der Signale am Ende des aktuellen Split-Steps.

b) Dann wird der nichtlineare Schritt durchgeführt. Dazu wird der nichtlineare Operator aus A_MID und MEAN_SQABS gebildet und auf A_MID angewandt. Das Ergebnis wird in AA gespeichert. Eine genauere Erklärung dieses Schrittes folgt in Abschnitt 4.2.2.

c) Schließlich wird der zweite lineare Halbschritt durchgeführt. Dieser verläuft analog zum ersten linearen Halbschritt, allerdings mit dem Unterschied, dass das Endergebnis wieder in AA gespeichert wird. Somit enthält AA die aktuellste Schätzung der Signale am Faserende und es können genauere gemittelte Betragsquadrate bestimmt werden oder es wird mit dem nächsten Split-Step fortgefahren.

Der erläuterte Programm-Ablauf wird durch Code auf dem Host gesteuert, indem verschiedene Kernel auf dem Device gestartet werden. Die interne Funktionsweise dieser Kernel wird im folgenden Abschnitt erläutert.

4.2.2. Subroutinen auf der GPU

Es gibt fünf Kernel, die jeweils eine der folgenden Operationen ausführen:

- Die Berechnung der linearen Effekte über eine halbe Schrittweite.
- Die Anwendung der linearen Effekte auf das Signal.
- Die Berechnung der Betragsquadrate am Anfang eines Split-Steps.
- Die Berechnung der gemittelten Betragsquadrate während der Iteration.
- Die Durchführung des nichtlinearen Schritt.

Hinzu kommen noch die Kernel zur Berechnung der (inversen) Fourier-Transformation. Da diese durch die cuFFT-Bibliothek bereitgestellt werden, werden sie nicht weiter erläutert.

Während der Kernel zur Durchführung des nichtlinearen Schritts verhältnismäßig komplex aufgebaut ist, sind die anderen Kernel vergleichsweise einfach und ähneln sich in ihrer Funktionsweise stark. Zweitere werden daher im Folgenden gemeinsam erläutert.

Einfach aufgebaute Kernel

Die einfach aufgebauten Kernel führen Operationen aus, für die jedes Element der Signale isoliert betrachtet werden kann. Damit die größtmögliche Datenrate bei Speicherzugriffen auf den *global memory* erreicht wird, müssen mehrere Threads — wie im Abschnitt 3.2.2 erläutert — gemeinsam auf einen linearen Speicherbereich zugreifen. Daher führt jeder SMP einen Block mit zahlreichen Threads aus, die alle einen zusammenhängenden Ausschnitt der Signale verarbeiten. Die entsprechende Zuordnung von Thread-Blöcken zu Datenbereichen ist in Abbildung 4.2 dargestellt. Da dabei deutlich weniger Threads gestartet werden, als Elemente zu verarbeiten sind, iteriert jeder Thread über mehrere Elemente. Dieses Prinzip ist als *grid-strided-loops* [22] bekannt und lässt sich in Abbildung 4.2 an der Zuordnung eines Thread-Blocks zu mehreren Signalausschnitten erkennen.

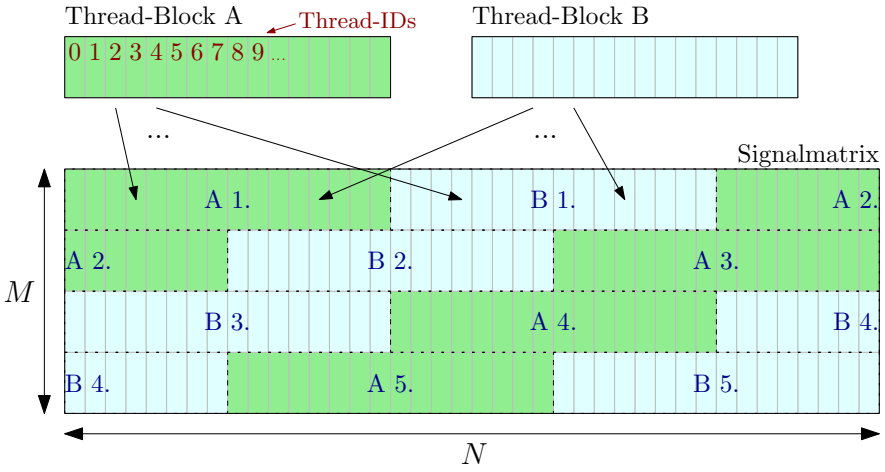


Abbildung 4.2.: Zuweisung der Thread-Gruppen zu Signal-Elementen in einfachen Kerneln.

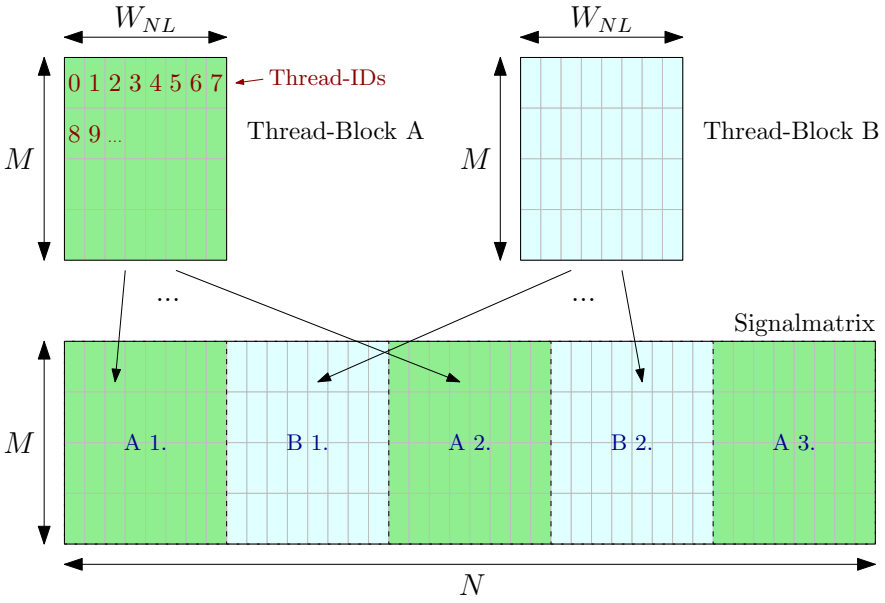


Abbildung 4.3.: Zuweisung der Thread-Gruppen zu Signal-Elementen im nichtlinearen Kernel.

Nichtlinearer Kernel

Der Kernel zur Durchführung des nichtlinearen Schritts berechnet den nichtlinearen Operator und wendet ihn auf die Signale an.

Der nichtlineare Operator ergibt sich jeweils aus der momentanen Leistung, berechnet durch das Betragsquadrat, aller Signale zum gleichen Abtastzeitpunkt. Somit interagieren jeweils alle Elemente einer Spalte der Signalmatrix miteinander. Da die Berechnung des nichtlinearen Operator einmalig für jedes Element stattfinden muss, wird jedes Betragsquadrat somit M mal verrechnet. Damit weniger Speicherzugriffe auf den *global memory* erfolgen, wird immer eine gesamte Spalte der Signalmatrix gleichzeitig durch den Kernel bearbeitet und die geladenen Betragsquadrate werden im *shared memory* zwischengespeichert.

Weiterhin besteht das Problem, dass die Signalmatrix im *row-major* Format gespeichert ist. Das bedeutet, dass die Elemente einer Zeile zusammenhängend im Speicher abgelegt sind und die einzelnen Zeilen hintereinander im Speicher liegen. Die Elemente in einer Spalten liegen somit weit auseinander. Dadurch kann beim Speicherzugriff auf die Elemente einer Spalte nicht die maximale Datenrate erreicht werden. Deshalb werden die Daten — anstatt spaltenweise — in zwei-dimensionalen Kacheln verarbeitet. Eine Kachel hat die Größe $M \times W_{NL}$, wobei die Kachelbreite W_{NL} nicht beliebig wählbar ist und später diskutiert wird. Jede Kachel wird von einem Thread-Block mit der entsprechenden Anzahl Threads verarbeitet. Die verschiedenen Spalten sind weiterhin logisch voneinander getrennt, doch durch diese Form der Kacheln wird erreicht, dass mehrere Threads mit aufsteigenden IDs auf einen zusammenhängenden Speicherbereich zugreifen. Die Einteilung des Signals in Kacheln und die Zuweisung von Threads zu Elementen ist in Abbildung 4.3 dargestellt. Wie dort ebenfalls zu erkennen ist, wird auch beim nichtlinearen Kernel das Prinzip der *grid-strided-loops* angewandt, da ein Thread-Block mehreren Abschnitten zugewiesen wird.

Die Berechnung im nichtlinearen Kernel läuft wie folgt ab:

1. Zuerst werden die Betragsquadrate der gesamten Kachel in den *shared memory* kopiert.
2. Dann berechnet jeder Thread für das entsprechende Element $\mathcal{N}_{ij}$ des nichtlinearen Operators die Summe $S_{ij} = \sum_{k=1}^{M} \kappa_{ik} \cdot \text{MEAN\_SQABS}_{kj}$ durch eine Schleife über alle Signale. An dieser Stelle ergibt sich der Vorteil durch die Zwischenspeicherung der Betragsquadrate im *shared memory* und der Speicherung der Kopplungsfaktoren κ im *constant memory*, da sowohl auf die Betragsquadrate als auch auf die Kopplungsfaktoren mehrfach zugegriffen wird.
3. Schließlich wird in jedem Thread der Term $\exp(\mathrm{j} \cdot \gamma \cdot S_{ij})$ berechnet und auf das entsprechende Element von A_MID multipliziert. Das Ergebnis wird in AA gespeichert.

Wie zuvor erwähnt unterliegt die Wahl der Kachelbreite W_{NL} bestimmten Beschränkungen. Da ein Thread-Block $M \cdot W_{NL}$ Threads enthält, wächst die Anzahl der Threads linear mit der Kachelbreite. Da die Anzahl der Threads pro Block durch die verwendete Hardware auf einen festen Wert T_{max} limitiert ist, ergibt sich für W_{NL} eine obere Grenze von $\frac{T_{max}}{M}$. Aus dem gleichen Grund besteht für M eine globale obere Schranke von T_{max}. Prinzipiell wäre die Größe eines Blocks zusätzlich durch die Größe des *shared memory* beschränkt, jedoch ist bei allen bisherigen CUDA-GPUs die Größe des *shared memory* ausreichend um mit dem verwendeten Algorithmus die maximale Anzahl an Threads nutzen zu können.

4.3. Reduktion des Speicherbedarfs

Durch die in diesem Abschnitt erläuterten Modifikationen des Programmablaufs können einige der allokierten Speicherbereiche ersatzlos entfernt werden. Dadurch verringert sich der allgemeine Speicherbedarf, ohne dass die Laufzeit beeinflusst wird.

Da die linearen Effekte über eine halbe Schrittweite sich jederzeit aus LINEAR berechnen lassen, besteht keine Notwendigkeit sie in LINEAR_EFFECT zwischenzuspeichern. Werden die linearen Effekte erst direkt bei ihrer Anwendung im entsprechenden Kernel berechnet, müssen sie überhaupt nicht zwischengespeichert werden. Dieses Vorgehen hat jedoch auf den ersten Blick den

Nachteil, dass dieselbe Berechnung bei jedem linearen Halbschritt erneut ausgeführt werden muss, da das Ergebnis nicht wiederverwendet werden kann. Insbesondere bei Simulationen mit konstanter Schrittweite ist dies ein erheblicher Mehraufwand. Tatsächlich beeinflusst dies die Laufzeit des Programms jedoch nicht, da die Geschwindigkeit des Kernels zur Anwendung der linearen Effekte bisher stark durch die maximale Datenrate des *global memory* begrenzt war. Die zusätzlichen Rechenoperationen werden ausgeführt, während die Rechenkerne auf weitere Daten aus dem *global memory* warten [25, Kap. 4.7].

Die gemittelten Betragsquadrate werden jeweils genau einmal im nichtlinearen Schritt verwendet, bevor sie durch die Trapezregel neu berechnet werden. Somit besteht keine Notwendigkeit sie zwischenzuspeichern. Ihre Berechnung kann direkt im nichtlinearen Kernel erfolgen, anstatt zuvor in einem separaten Kernel.

Bei der Durchführung des linearen Halbschritts werden die Signale im Frequenzbereich in A_FREQ gespeichert. Währenddessen sind in AA keine Daten enthalten, die später nochmals benötigt werden. Somit ist es problemlos möglich die Signale im Frequenzbereich ebenfalls in AA abzuspeichern. Dazu wird eine *in-place* Fourier-Transformation verwendet, was — wie in Abschnitt 3.3 beschrieben — im vorliegenden Anwendungsszenario ohne Laufzeit-Einbußen möglich ist.

Zusammengefasst lässt sich durch die Modifikationen der Speicher für AA_FREQ, MEAN_SQABS und LINEAR_EFFECT einsparen.

4.4. Entkopplung von Speicherbedarf und Gesamtdatenmenge

Im vorhergehenden Abschnitt wurde analysiert, wie der Speicherbedarf unter Beibehaltung des allgemeinen Programm-Ablaufs reduziert werden kann. Die Problemgröße wird jedoch weiterhin dadurch beschränkt, dass alle Daten gleichzeitig im Device-Speicher abgelegt werden müssen. Um diese Beschränkung aufzuheben, kann der Algorithmus in Teilprobleme zerlegt werden, die nur auf kleinen Ausschnitten der Daten operieren. Werden nur einige Teilprobleme gleichzeitig bearbeitet, wird auch nur so viel Speicherplatz auf dem Device verbraucht, wie zur Zwischenspeicherung der entsprechenden Datenausschnitte notwendig ist.

Dies bedeutet, dass die einzelnen Datenausschnitte zwischen Host und Device ausgetauscht werden müssen. Da Datentransfers zwischen Host und Device vergleichsweise langsam sind, sollen so wenige Transfers wie möglich stattfinden. Daher werden alle Operationen, die direkt nacheinander auf den selben Datenausschnitten operieren können, zu einem Teilproblem zusammengefasst. Ist ein Datensatz vom Host geladen, werden alle Operationen des Teilproblem darauf angewandt. Erst dann wird das Ergebnis wieder zum Host übertragen.

Problematisch ist im Fall der SSFM, dass nicht alle Operationen auf die gleichen Datenausschnitte angewandt werden können. Die Zerlegung der Kernel in unabhängige Teile ist bereits durch die in Abschnitt 4.2.2 beschriebenen Thread-Gruppen gegeben. Die Fourier-Transformation lässt sich derweil jeweils auf ein vollständiges einzelnes Signal anwenden. Da die Fourier-Transformation somit immer eine gesamte Zeile der Signalmatrix benötigt und der nichtlineare Schritt eine gesamte Spalte der Signalmatrix benötigt, ist ausgeschlossen, dass beide Schritte auf einem gemeinsamen Datenausschnitt arbeiten. Die eben erwähnte Zusammenfassung der Operationen erfolgt daher in zwei verschiedene Teile. Der eine Teil enthält nur den

nichtlinearen Kernel und operiert auf den in Abschnitt 4.2.2 beschriebenen Kacheln. Der andere Teil enthält alle übrigen Operationen und operiert aufgrund der FFT auf vollständigen Zeilen der Signalmatrix. In Anlehnung an die Datenausschnitte auf denen sie arbeiten, wird der erste Teil im Folgenden als vertikaler Teil bezeichnet und der zweite Teil als horizontaler Teil.

Da sich vertikaler und horizontaler Teil sehr häufig abwechseln, müssen oft Daten in verschiedenen Konfigurationen zwischen Host und Device übertragen werden. Aufgrund der Fähigkeit von CUDA-GPUs Datentransfers zwischen Host und Device und davon unabhängige Berechnungen zeitgleich auszuführen, fällt der zu erwartende Anstieg der Laufzeit jedoch geringer aus, wenn mehrere Teilprobleme parallel berechnet werden. Dieser Effekt wird in Abschnitt 5.2.1 eingehender untersucht.

Zur weiteren Reduktion der transferierten Datenmenge ist es schließlich möglich eine beliebige Anzahl an Signalen statisch auf dem Device zu speichern, sodass diese nicht mehr zwischen Host und Device ausgetauscht werden. Dies erlaubt im Extremfall, dass alle Daten auf dem Device verbleiben und keine Datentransfers mehr durchgeführt werden, die die Laufzeit verschlechtern.

4.5. Vorstellung der finalen Implementierung

Analog zur ursprünglichen Implementierung wird im Folgenden die Implementierung nach Umsetzung der vorgestellten Verbesserungsmöglichkeiten erläutert. Dazu wird zuerst ein System zur Verwaltung der Device-Ressourcen und der SSFM-Teilprobleme beschrieben und anschließend der Programm-Ablauf erläutert.

Um die korrekte Berechnung der Simulationsergebnisse sicherzustellen, wurden während der Entwicklung regelmäßig bereitgestellte Testsimulationen durchgeführt, deren Ergebnisse mit Ergebnissen der ursprünglichen Implementierung verglichen wurden.

4.5.1. Ressourcen- und Aufgabenverwaltung

Damit bei der Ressourcenvergabe für die einzelnen Teilprobleme keine Konflikte entstehen, wird ein System zur Ressourcen-Verwaltung eingeführt. Es verwendet die folgende Analogie: Ein Betrieb mit *mehreren Arbeitern* will ein Produkt fertigen. Dazu müssen *mehrere Prozessschritte* durchlaufen werden, von denen *einige parallel* ablaufen können, während *andere nacheinander* ausgeführt werden müssen. Jedem Arbeiter in dem Betrieb werden *mehrere gleichartige Aufgaben zugeteilt*, die er während seiner *Schicht* abarbeiten muss. Die Ressourcen, die im Betrieb vorhanden sind, werden gleichmäßig auf alle Arbeiter einer Schicht verteilt. Jeder bekommt dabei ausreichend Ressourcen zugeteilt um *genau eine Aufgabe gleichzeitig* zu bearbeiten. Der Arbeiter kann seine Aufgaben anschließend *unabhängig von den anderen Arbeitern seiner Schicht* verrichten. Dazu besorgt er zuerst das Ausgangsmaterial, das er entweder aus dem *angeschlossenen Lager* holen kann oder aus dem *entfernten Lager* liefern lassen muss. Wenn er das Material verarbeitet hat, bringt er es wieder zurück in das ursprüngliche Lager. Wenn alle Arbeiter einer Schicht fertig sind, geben sie ihre Ressourcen ab und es beginnt die nächste Schicht. Die Arbeiten in einer neuen Schicht bauen auf den Ergebnissen der vorhergehenden Schicht auf. So wird schließlich in mehreren Schichten das Produkt fertig gestellt, ohne dass Ressourcen ungenutzt bleiben oder verschiedene Arbeiter miteinander in Konflikt geraten.

Die Aufgaben in diesem Modell entsprechen den Teilproblemen der SSFM. Eine Schicht entspricht dem vertikalen oder horizontalen Teil. Das angeschlossene Lager ist der Device-Speicher und das entfernte Lager der Host-Speicher. Ein Arbeiter schließlich ist eine logische Einheit, die mehrere Ressourcen — also Pufferspeicher auf dem Device und FFT-Pläne — bündelt. Dabei muss es offensichtlich für verschiedene Aufgabentypen unterschiedliche Arbeiter geben. In Anlehnung an die berechneten Teilprobleme werden die Aufgaben und Arbeiter ebenfalls in die Kategorien vertikal und horizontal unterteilt. Da die Arbeiter, die nur Daten verarbeiten, die statisch auf dem Device liegen, keinen temporären Pufferspeicher benötigen, gibt es wiederum zwei Arten von horizontalen Arbeitern: solche mit Puffer und solche ohne. Bei den vertikalen Arbeitern gibt es diese Unterscheidung nicht, da diese immer teils auf den Device- und teils auf den Host-Daten arbeiten.

4.5.2. Programm-Ablauf

Der Programm-Ablauf der finalen Implementierung wird analog zur zugrundeliegenden Implementierung beschrieben. In Abbildung 4.4 ist das Ablaufdiagramm der neuen Implementierung zu finden. Dort sind wie zuvor die Initialisierung und Freigabe der Ressourcen zur Verbesserung der Übersichtlichkeit vernachlässigt worden.

Die neue Implementierung nimmt einige weitere Parameter entgegen, die die Anzahl der auf dem Device gespeicherten Signale, die Größe der Aufgaben und die Anzahl der Arbeiter festlegen. Aus diesen Parametern wird zuerst berechnet, wie viel Speicherplatz benötigt wird, und der entsprechende Speicher wird allokiert. Anschließend werden die Arbeiter generiert und ihnen wird jeweils ein Teil des Speichers als Pufferspeicher zugewiesen. Für jeden vertikalen Arbeiter wird außerdem ein cuFFT-Plan generiert. Schließlich werden die Aufgaben definiert und es wird eine Zuordnung von Aufgaben zu Arbeitern festgelegt. Nachdem alle Daten, die statisch auf dem Device verbleiben, dorthin kopiert wurden, beginnt die Simulation. Diese läuft wie zuvor in einer Schleife über alle Split-Steps ab. Allerdings wird in einem Schleifendurchlauf — im Gegensatz zur ursprünglichen Implementierung — nicht ein zusammenhängender Split-Step simuliert, sondern es wird zuerst der zweite lineare Halbschritt eines Split-Step und dann der folgende erste lineare Halbschritt und nichtlineare Schritt des nächsten Split-Step durchgeführt. Die Motivation hinter dieser Funktionsweise ist eine bessere Zusammenfassung der Operationen im horizontalen Teil.

Innerhalb der Schleife erfolgt die Simulation der Signalausbreitung nach folgendem Schema. Dabei umfasst ein Lese- oder Schreibzugriff auf die Daten auch den Transfer der Daten zwischen Host und Device, falls dies notwendig ist. Die Nummerierung der folgenden Aufzählung entspricht der Nummerierung links in Abbildung 4.4.

1. Außer im ersten Schleifendurchlauf wird zu Beginn der zweite lineare Halbschritt des vorhergehenden Split-Steps auf AA durchgeführt.
2. An dieser Stelle entspricht das Signal in AA dem Signal am Anfang des neuen Split-Step. Daher wird nun die Berechnung der Betragsquadrate am Anfang des Split-Step durchgeführt und das Ergebnis in A0_SQRABS gespeichert.
3. Es folgt die Durchführung des ersten linearen Halbschritt in diesem Split-Step. Das Ergebnis wird in A_MID gespeichert.

4. In der ursprünglichen Implementierung beginnt an dieser Stelle die Iteration über den nichtlinearen Schritt und den zweiten linearen Halbschritt. Da der letzte zweite lineare Halbschritt jedoch erst zu Beginn des nächsten Schleifendurchlauf berechnet werden soll, muss analog dazu die erste Durchführung des nichtlinearen Schritt aus der Iteration heraus genommen werden. Somit wird an dieser Stelle nur der erste nichtlineare Schritt aus AA durchgeführt.

5. Nun folgt die Schleife über alle verbleibenden Iterationen. Es wird immer zuerst ein zweiter linearer Halbschritt und anschließend ein nichtlinearer Schritt auf AA ausgeführt. Da die Iteration mit einem nichtlinearen Schritt endet, ist auch die anfangs postulierte Schleifeninvariante erfüllt.

Zuletzt wird unter Punkt 6 einmalig ein abschließender zweiter linearer Halbschritt durchgeführt, der notwendig ist um das Signal am Faserende zu erhalten. Schließlich wird der auf dem Device verbliebene Teil des Ergebnisses zum Host kopiert, bevor die Simulation beendet ist.

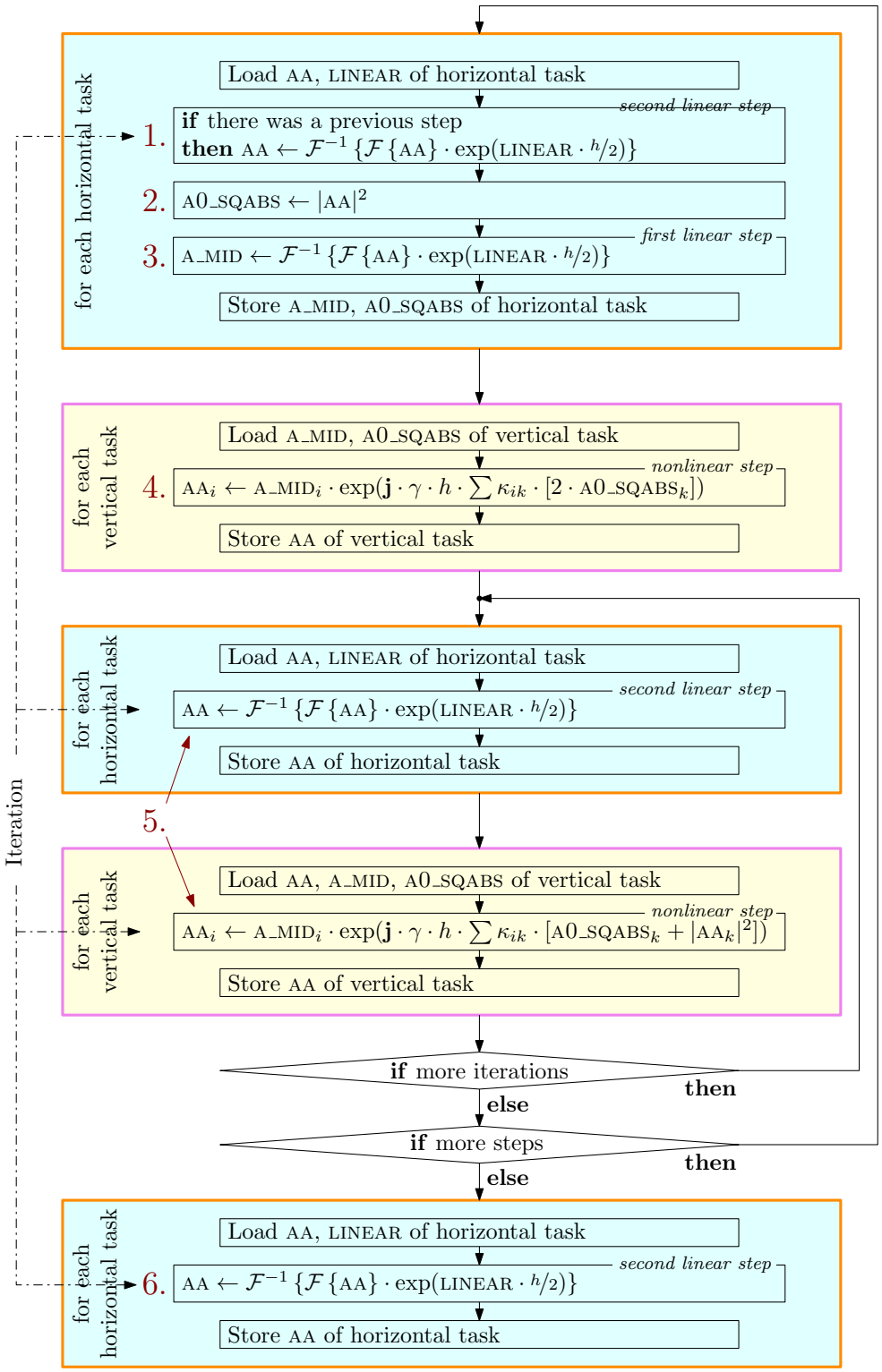


Abbildung 4.4.: Ablaufdiagramm der finalen Implementierung

5. Speicherbedarfs- und Laufzeit-Analyse

In diesem Kapitel werden der Speicherbedarf und die Laufzeit der ursprünglichen und überarbeiteten Implementierung analysiert und verglichen. Im ersten Abschnitt wird eine geschlossene Formel zur Bestimmung des Speicherbedarfs aufgestellt und durch mehrere Simulationen verifiziert. Im zweiten Abschnitt wird die Überlappung von Datentransfers und Rechenoperationen in der neuen Implementierung analysiert und die Laufzeit der Implementierungen bei verschiedenen Problemgrößen und Parametrisierungen verglichen.

5.1. Speicheranalyse

Damit es möglich ist die Kapazitätsgrenzen bestehender Hardware einzuschätzen beziehungsweise die durch eine bestimmtes Problem gestellten Hardwareanforderung zu ermitteln, wird eine Formel zur Berechnung des Speicherverbrauchs des Programms benötigt. Zum Vergleich der beiden Implementierungen wird im Folgenden sowohl für die ursprüngliche als auch für die überarbeitet Implementierung eine solche Formel ausgestellt. Da die Simulation mit Gleitkommazahlen in einfacher oder doppelter Genauigkeit stattfinden kann, werden die Formeln in Abhängigkeit der Größe S_{real} des verwendeten Gleitkommatyps angegeben. Um die Formeln zu verifizieren, werden abschließend mehrere Simulationen mit unterschiedlicher Parametrisierung durchgeführt und der berechnete Speicherbedarf wird mit dem gemessenen Speicherverbrauch verglichen.

5.1.1. Speicheranalyse der ursprünglichen Implementierung

In der ursprünglichen Implementierung wird der Speicherverbrauch nur die Parameter M und N beeinflusst. Tabelle 5.1a listet die Größen aller allokierten Speicherbereiche auf. Da keine Überlappung der Speicherbereiche stattfindet, ergibt sich der Gesamtspeicherbedarf S_{alt} als die Summe der einzelnen Speichergrößen. Die vollständige Formel

$$S_{alt} = 14 \cdot M \cdot N \cdot S_{real} \tag{5.1}$$

ergibt sich schließlich nach dem Zusammenfassen aller Summenterme. Dies kann weiterhin umgeschrieben werden zu

$$S_{alt} = 7 \cdot S_{signal} \tag{5.2}$$

mit dem Speicherbedarf S_{signal} einer Signalmatrix. Dieser lässt sich durch

$$S_{signal} = 2 \cdot M \cdot N \cdot S_{real} \tag{5.3}$$

berechnen.

Tabelle 5.1.: Größe der durch die beiden Implementierungen verwendeten Speicherbereiche. S_{real} bezeichnet die Speichergröße einer reellwertigen Zahl bei der verwendeten Genauigkeit.

(a) Ursprüngliche Implementierung.

	Anzahl der Elemente	Größe eines Elements
cuFFT-Pläne	$M \cdot N$	$2 \cdot S_{real}$
LINEAR	$M \cdot N$	$2 \cdot S_{real}$
LINEAR_EFFECT	$M \cdot N$	$2 \cdot S_{real}$
AA	$M \cdot N$	$2 \cdot S_{real}$
A_FREQ	$M \cdot N$	$2 \cdot S_{real}$
A_MID	$M \cdot N$	$2 \cdot S_{real}$
A0_SQABS	$M \cdot N$	S_{real}
MEAN_SQABS	$M \cdot N$	S_{real}

(b) Überarbeitete Implementierung.

		Anzahl der Elemente	Größe eines Elements
static	LINEAR	$M_{static} \cdot N$	$2 \cdot S_{real}$
	AA	$M_{static} \cdot N$	$2 \cdot S_{real}$
	A_MID	$M_{static} \cdot N$	$2 \cdot S_{real}$
	A0_SQABS	$M_{static} \cdot N$	S_{real}
horz,local	cuFFT-Pläne	$W_{horz,local} \cdot M_{horz,local} \cdot N$	$2 \cdot S_{real}$
horz,remote	cuFFT-Pläne	$W_{horz,remote} \cdot M_{horz,remote} \cdot N$	$2 \cdot S_{real}$
	LINEAR	$W_{horz,remote} \cdot M_{horz,remote} \cdot N$	$2 \cdot S_{real}$
	AA und A_MID	$W_{horz,remote} \cdot M_{horz,remote} \cdot N$	$2 \cdot S_{real}$
	A0_SQABS	$W_{horz,remote} \cdot M_{horz,remote} \cdot N$	S_{real}
vert	AA	$W_{vert} \cdot (M - M_{static}) \cdot N_{vert}$	$2 \cdot S_{real}$
	A_MID	$W_{vert} \cdot (M - M_{static}) \cdot N_{vert}$	$2 \cdot S_{real}$
	A0_SQABS	$W_{vert} \cdot (M - M_{static}) \cdot N_{vert}$	S_{real}

5.1.2. Speicheranalyse der überarbeiteten Implementierung

In der überarbeiteten Implementierung hängt der Speicherverbrauch, neben M und N, noch von den Parametern zur Einstellung der Aufgaben- und Arbeiter-Verwaltung ab:

- M_{static}: Die Anzahl der Signale, die statisch auf dem Device verbleiben.
- $M_{horz,local}$: Die Anzahl der Signale in einer horizontalen Aufgabe auf dem Device.
- $W_{horz,local}$: Die Anzahl der Arbeiter für horizontale Aufgaben auf dem Device.
- $M_{horz,remote}$: Die Anzahl der Signale in einer horizontalen Aufgabe auf dem Host.
- $W_{horz,remote}$: Die Anzahl der Arbeiter für horizontale Aufgaben auf dem Host.
- N_{vert}: Die Anzahl der Abtastwerte pro Signal in einer vertikalen Aufgabe.
- W_{vert}: Die Anzahl der Arbeiter für vertikale Aufgaben.

Eine Auflistung des Speicherbedarfs aller verwendeten Speicherbereiche befindet sich in Tabelle 5.1b . Der Gesamtspeicherverbrauch ist in diesem Fall nicht gleich der Summe aller aufgelisteten Speichergrößen. Dies liegt daran, dass der Pufferspeicher der horizontalen und vertikalen Arbeiter überlappt. Die Größe S_{buffer} des Puffers berechnet sich somit als das Maximum der beiden Puffer. Der Gesamtspeicherverbrauch S_{neu} der überarbeiteten Implementierung setzt sich daher zusammen aus

$$S_{neu} = S_{static} + S_{cuFFT} + \max\left(S_{horz,buffer},\, S_{vert,buffer}\right) \tag{5.4}$$

mit

$$S_{static} = 7 \cdot N \cdot M_{static} \cdot S_{real} \quad , \tag{5.5}$$

$$S_{cuFFT} = 2 \cdot N \cdot W_{horz,local} \cdot M_{horz,local} \cdot S_{real} \quad , \tag{5.6}$$

$$+ 2 \cdot N \cdot W_{horz,remote} \cdot M_{horz,remote} \cdot S_{real} \quad , \tag{5.7}$$

$$S_{horz,buffer} = 5 \cdot N \cdot W_{horz,remote} \cdot M_{horz,remote} \cdot S_{real} \quad , \tag{5.8}$$

$$S_{vert,buffer} = 5 \cdot N_{vert} \cdot W_{vert} \cdot (M - M_{static}) \cdot S_{real} \quad . \tag{5.9}$$

Maximaler Speicherverbrauch

Besonders interessant ist die Parametrisierung, bei der das Verhalten der überarbeiteten Implementierung der ursprünglichen Implementierung entspricht. Dies ist der Fall, wenn alle Daten auf dem Device gespeichert sind und der vertikale und horizontale Teil nicht in mehrere Aufgaben geteilt wird. Dazu muss die Parametrisierung $O_{klassisch}$ mit

$$M_{static} = M,\, M_{horz,local} = M,\, W_{horz,local} = 1,\, W_{horz,remote} = 0,\, N_{vert} = N,\, W_{vert} = 1$$

verwendet werden. Gleichung (5.4) reduziert sich dann durch Einsetzen und Vereinfachen zu

$$S_{neu}\Big|_{O_{klassisch}} = 9 \cdot M \cdot N \cdot S_{real} = 4{,}5 \cdot S_{signal} \quad . \tag{5.10}$$

Das entspricht zugleich der oberen Schranke für den Speicherverbrauch, wie durch die folgenden Abschätzungen bewiesen wird. Der Puffer für die horizontalen und vertikalen Arbeiter kann nicht größer werden als die Daten, die auf dem Host gespeichert sind. Mehr Puffer zu reservieren als Daten zur Pufferung vorhanden sind, erzeugt keinen Vorteil. Dies wird durch die Ungleichung

$$\max\left(S_{horz,buffer},\, S_{vert,buffer}\right) \leq 5 \cdot (M - M_{static}) \cdot N \cdot S_{real} \tag{5.11}$$

ausgedrückt. Die gemeinsame Größe aller cuFFT-Pläne ist wiederum beschränkt auf die Größe der Signalmatrix. Auch hier entsteht kein Vorteil, wenn mehr cuFFT-Pläne erzeugt werden als Signale vorhanden sind. Das wird durch die Ungleichung

$$S_{cuFFT} \leq 2 \cdot M \cdot N \cdot S_{real} \tag{5.12}$$

ausgedrückt.

Werden die Abschätzungen (5.11) und (5.12) in die Gleichung (5.4) zur Berechnung von S_{neu} eingesetzt, ergibt sich schließlich die Abschätzung des Speicherbedarfs der überarbeiteten Implementierung als

$$\begin{aligned} S_{neu} &= S_{static} + S_{cuFFT} + \max\left(S_{horz,buffer},\, S_{vert,buffer}\right) \\ &\leq (7 \cdot M_{static} + 5 \cdot (M - M_{static}) + 2 \cdot M) \cdot N \cdot S_{real} \\ &\leq (7 \cdot M_{static} + 7 \cdot (M - M_{static}) + 2 \cdot M) \cdot N \cdot S_{real} \\ &= (7 \cdot M + 2 \cdot M) \cdot N \cdot S_{real} \\ &= 9 \cdot M \cdot N \cdot S_{real} \\ &= 4{,}5 \cdot S_{signal} \quad . \end{aligned} \tag{5.13}$$

Das entspricht dem zuvor berechneten Speicherverbrauch (5.10) unter der Parametrisierung $O_{klassisch}$.

Minimaler Speicherverbrauch

Neben der Parametrisierung $O_{klassisch}$ ist vor allem der Fall interessant, dass keine Signale auf dem Device gespeichert werden. Wird dabei zusätzlich N_{vert} antiproportional zur Anzahl der Signale gewählt, ergibt sich die unvollständige Parametrisierung $O_{host-only}$ mit

$$M_{static} = 0,\ W_{horz,local} = 0,\ N_{vert} = \frac{N'_{vert}}{M} \quad .$$

und den freien Parametern $W_{horz,remote}$, $M_{horz,remote}$, W_{vert}. Die Gleichungen (5.4) bis (5.9) werden dadurch zu

$$S_{neu}\Big|_{O_{host-only}} = S'_{cuFFT} + \max\left(S'_{horz,buffer},\, S'_{vert,buffer}\right) \tag{5.14}$$

mit

$$S'_{cuFFT} = 2 \cdot N \cdot W_{horz,remote} \cdot M_{horz,remote} \cdot S_{real} \tag{5.15}$$

$$S'_{horz,buffer} = 5 \cdot N \cdot W_{horz,remote} \cdot M_{horz,remote} \cdot S_{real} \tag{5.16}$$

$$S'_{vert,buffer} = 5 \cdot N'_{vert} \cdot W_{vert} \cdot S_{real} \tag{5.17}$$

reduziert. Da keiner dieser Terme mehr die Anzahl M der Signale enthält, ist der Speicherverbrauch der neuen Implementierung bei geeigneter Parameterwahl somit unabhängig von der Anzahl der Signale, wohl aber von der Länge der Signale.

Der minimale Speicherverbrauch der überarbeiteten Implementierung ergibt daraus schließlich bei der Parametrisierung O_{min} mit

$$M_{static} = 0,\ W_{horz,local} = 0,\ M_{horz,remote} = 1,\ W_{horz,remote} = 1,\ N_{vert} \leq \frac{N}{M},\ W_{vert} = 1$$

zu

$$S_{neu}\Big|_{O_{min}} = 7 \cdot N \cdot S_{real} \quad . \tag{5.18}$$

5.1.3. Vergleich beider Implementierungen

Die überarbeitete Implementierung zeigt bezüglich des Speicherbedarfs im Vergleich zur ursprünglichen Implementierung einige wünschenswerte Eigenschaften. Im Gegensatz zum Speicherbedarf der ursprünglichen Implementierung von $14 \cdot S_{signal}$ liegt der maximale Speicherbedarf der überarbeiteten Implementierung mit $9 \cdot S_{signal}$ um einen Faktor $\frac{5}{14}$ ($\approx 36\%$) niedriger. Außerdem ist es möglich den Speicherbedarf der überarbeiteten Implementierung durch Wahl weiterer Parameter zu senken. Letztlich ist der Speicherverbrauch bei geeigneter Parameterwahl unabhängig von der Anzahl der Signale, sodass trotz beschränktem Speicherplatz Simulationen mit sehr vielen Signalen ermöglicht werden. Simulationen mit längeren Signalen werden dadurch ebenfalls ermöglicht, da die maximale Signallänge bei gegebenen Speicher nunmehr ausschließlich dadurch dadurch begrenzt wird, dass ein einzelnes Signal vollständig im *global memory* gespeichert werden muss.

5.1.4. Verifikation der aufgestellten Formeln

Zur Verifikation der aufgestellten Speicherverbrauchs-Formeln (5.1) und (5.4) werden mehrere Simulationen durchgeführt, bei denen die verwendete Speichermenge ermittelt wird. Dazu wird mithilfe des „nvidia-smi“ Tools der maximale Speicherverbrauch während der Programm-Ausführung aufgezeichnet. Das Tool erlaubt dabei Messungen mit einer Genauigkeit von einem Mebibyte (MiB; $1\,\text{MiB} \mathrel{\hat{=}} 1024^2$ Byte). Die Gegenüberstellung der Ergebnisse der Messungen mit den berechneten Werten ist auszugsweise in Anhang A verzeichnet. In Abbildung 5.1 und 5.2 sind einzelne Versuchsreihen grafisch dargestellt.

Es ergibt sich, dass der tatsächliche Speicherverbrauch sowohl in der alten, als auch der neuen, Implementierung annähernd konstant um 94 MiB bis 101 MiB nach oben vom berechneten Speicherverbrauch abweicht. Dies lässt sich einerseits direkt aus den Tabellen in Anhang A ablesen. Andererseits ist auch in Abbildung 5.1 erkennbar, dass sich sich der Speicherverbrauch für Simulationen mit geringem Datenvolumen einer Untergrenze von circa 100 MiB annähert.

Eigene Analysen haben ergeben, dass circa 20 MiB des überschüssigen Speichers durch die Einbindung der cuFFT-Bibliothek verursacht werden. Der restliche überschüssige Speicher wird selbst dann verbraucht, wenn keine Operationen auf dem Device durchgeführt werden und kein Speicher auf dem Device allokiert wird. Somit ist dies möglicherweise ein undokumentierter Overhead der CUDA-Architektur beim Initialisieren der Ausführungs-Umgebung.

Von besonderem Interesse bezüglich des Speicherbedarfs ist die Entkoppelung des Speicherverbrauchs von der Gesamtzahl der Signale. In Abbildung 5.2b ist der Speicherverbrauch der überarbeiteten Implementierung bei unterschiedlicher Anzahl von Signalen, aber gleichbleibender Anzahl von Arbeitern, dargestellt. Im Vergleich dazu ist in Abbildung 5.2a der Speicherverbrauch bei einer mitwachsenden Anzahl von Arbeitern dargestellt. Es ist zu erkennen, dass der Speicherverbrauch bei konstanter Arbeiterzahl wie erwartet unabhängig von der Gesamtzahl der Signale ist.

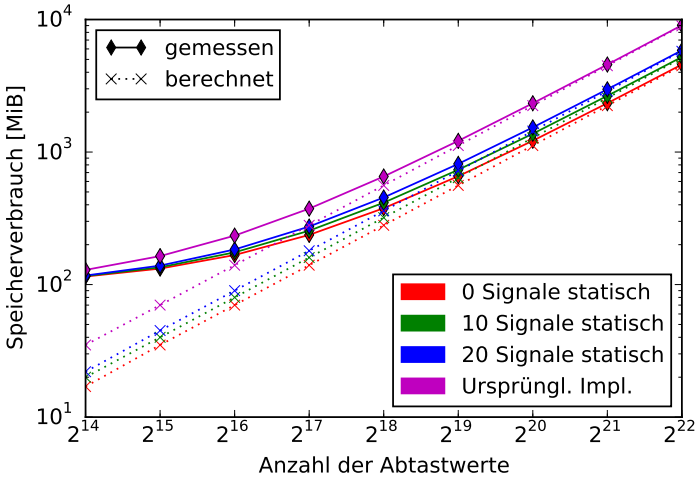
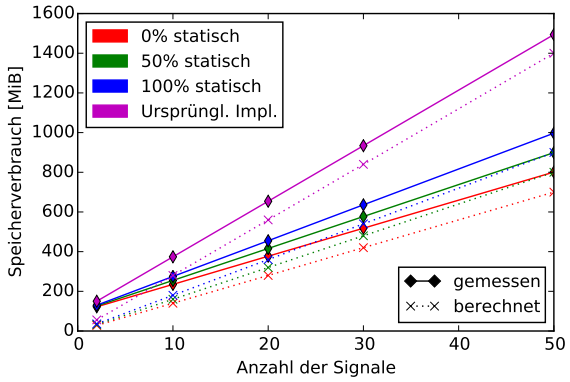
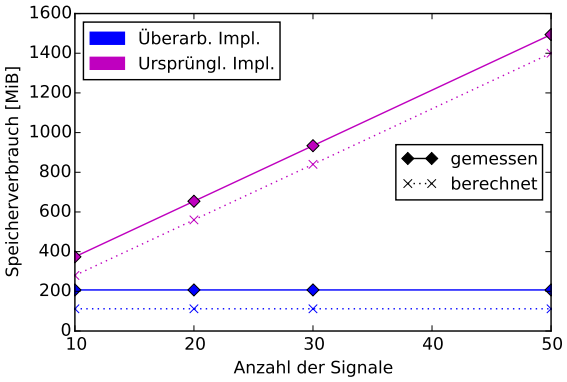


Abbildung 5.1.: Speicherverbrauch beider Implementierungen bei variierter Signallänge. Im Fall der neuen Implementierung dargestellt für verschiedene Anzahlen von Signalen, die statisch auf dem Device liegen. Simulationen jeweils mit 20 Signalen und Setup gemäß Tabelle A.1.



(a) Anzahl der Arbeiter in der überarbeiteten Implementierung wächst mit der Anzahl der Signale gemäß dem Setup aus Tabelle A.1. Dargestellt für verschiedene Anteile der Daten statisch auf dem Device.



(b) Konstante Anzahl der Arbeiter bei allen Simulationen. Es liegen keine Daten statisch auf dem Device; es gibt acht horizontale Arbeiter, die je ein Signal bearbeiten, und zwei vertikale Arbeiter, die je 2^{14} Abtastwerte bearbeiten.

Abbildung 5.2.: Speicherverbrauch beider Implementierungen bei variierter Anzahl von Signalen. Simulationen jeweils mit 2^{18} Abtastwerten pro Signal.

5.2. Laufzeitanalyse

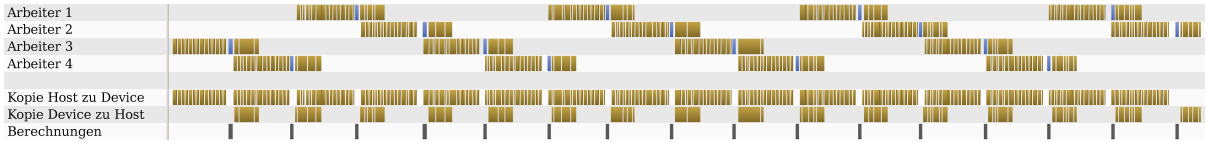
Durch Einführen der regelmäßigen Datenkopien zwischen Host und Device ist eine Reduktion der Ausführungsgeschwindigkeit des Programms zu erwarten. Daher wird im Folgenden die Veränderung der Laufzeit durch die überarbeitete Implementierung untersucht. Zuerst wird die in Abschnitt 4.4 angesprochene Überlappung der Berechnungen und Datentransfers betrachtet. Danach wird die Laufzeit der ursprünglichen und der überarbeiteten Implementierung durch mehrere Simulationsreihen verglichen.

5.2.1. Überlappung von Operationen

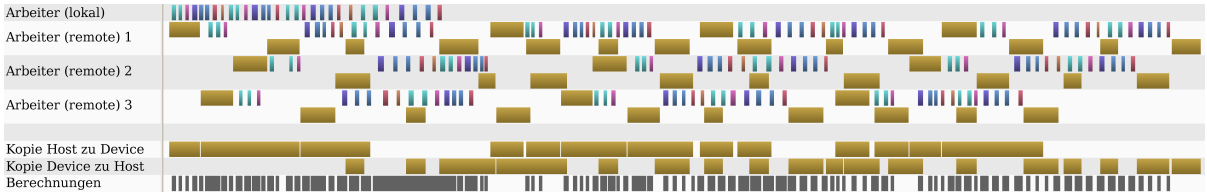
Zur Analyse der Überlappung von Operationen wird der Nvidia Visual Profiler [26] verwendet. Dieser registriert während einer Programmausführung alle Vorgänge auf dem Device und stellt den zeitlichen Verlauf grafisch dar. Dabei wird zwischen Datentransfers vom und zum Device sowie den verschiedenen Kerneln unterschieden. Es wird eine Simulation mit zwölf Signalen zu je 2^{17} Abtastwerten durchgeführt, wobei zwei Signale statisch auf dem Device gespeichert werden.

Es werden vertikale Aufgaben mit je 2^{13} Abtastwerten definiert. Dadurch ergeben sich insgesamt 2^{17-13}, also sechzehn, vertikale Aufgaben. Diese werden von vier Arbeitern erledigt. In Abbildung 5.3a ist die Profiler-Ausgabe für die Bearbeitung des vertikalen Teils dargestellt. Im oberen Teil der Abbildung sind die vier Arbeiter zu erkennen. Wie zu sehen ist, bearbeitet jeder von ihnen nacheinander vier Aufgaben. Im unteren Teil der Abbildung sind die Datentransfers zum und vom Device sowie die Kernelausführungen zusammengefasst dargestellt. Wie deutlich zu erkennen ist, ist die Überlappung von Datentransfers und Berechnungen sehr gering. Hieraus entstehen deutliche Geschwindigkeits-Einbußen. Die Datentransfers in die verschiedenen Richtungen überlappen jedoch vollständig.

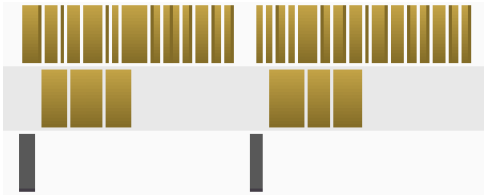
Eine horizontale Aufgabe wird so definiert, dass sie jeweils ein Signal umfasst. Für die zwei Aufgaben, deren Daten somit statisch auf dem Device liegen, steht ein Arbeiter bereit. Für die restlichen zehn Aufgaben, die zwischen Host und Device übertragen werden, stehen drei Arbeiter bereit. In Abbildung 5.3b ist die Profiler-Ausgabe während der Berechnung des horizontalen Teils dargestellt. Im oberen Teil der Abbildung sind die verschiedenen Arbeiter zu erkennen. Der „lokale Arbeiter“ erledigt die Aufgaben, deren Daten statisch auf dem Device liegen. Daher finden in seiner Zeitleiste keine Datentransfers statt. Da die beiden Aufgaben direkt nacheinander ausgeführt werden, ist nicht zu erkennen, wann eine Aufgabe endet und die nächste beginnt. Die anderen drei Arbeiter führen verschiedene Datentransfers durch. Die einzelnen Aufgaben sind hier ebenfalls schwierig zu unterscheiden, können jedoch an den Datentransfers (in gelb dargestellt) ausgemacht werden. So lässt ich erkennen, dass der erste Arbeiter vier Aufgaben bearbeitet, während die anderen zwei Arbeiter jeweils drei Aufgaben bearbeiten. Im unteren Teil der Abbildung sind wieder jeweils die Datentransfers in beide Richtungen und die Kernellaufzeiten zusammengefasst. Es ist zu erkennen, dass die unterschiedlichen Datentransfers nur zum Teil überlappen. Die Kernel und Datentransfers überlappen jedoch sehr vollständig. Im Vergleich zum Idealfall, einer kontinuierlichen Kernelausführung, geht somit nur wenig Zeit durch die Datenbewegungen verloren.



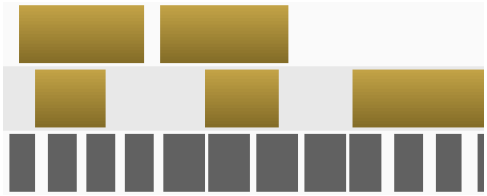
(a) Vertikaler Teil. Der dargestellte Ausschnitt umfasst eine Laufzeit von circa 9 ms.



(b) Horizonaler Teil. Der dargestellte Ausschnitt umfasst eine Laufzeit von circa 8 ms.



(c) Detailausschnitt der Überlappung des vertikalen Teils.



(d) Detailausschnitt der Überlappung des horizontalen Teils.

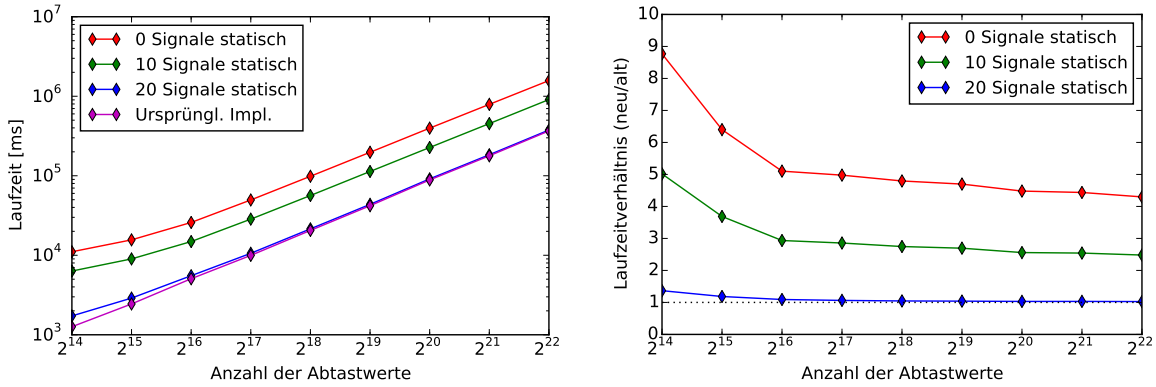
Abbildung 5.3.: (a),(b) Profiler-Anzeige für den vertikalen und horizontalen Teilschritt. Datentransfers sind gelb hervorgehoben, Kernelausführungen bunt. Im jeweils unteren Teil der Abbildungen sind die Datentransfers und Kernelausführungen aller Arbeiter zusammengefasst. Kernel sind dort dunkelgrau markiert.
(c),(d) Detailausschnitte der Überlappung im unteren Bereich von (a) und (b).

Insgesamt lässt sich feststellen, dass die Einführung der Datentransfers die Laufzeit des horizontalen Teils nur leicht beeinflusst, während die Laufzeit des vertikalen Teils dadurch sehr stark ansteigt. Die Ursache dafür ist der geringe Rechenaufwand im vertikalen Teil. Die Rechenzeit einer Aufgabe reicht nicht aus um die Zeit zu füllen, in der eine andere Aufgabe auf Daten wartet.

5.2.2. Performance-Vergleich

Zur Messung des realen Laufzeitunterschieds zwischen der ursprünglichen und der überarbeiteten Implementierung, werden mehrere Simulationen durchgeführt. Die Laufzeit der Berechnungen auf der GPU wird dabei mithilfe einer GPU-internen Uhr gemessen. Die Daten enthalten daher keine Information über die Laufzeit des Host-seitigen Teils der Simulation. Dieser umfasst mitunter die Berechnung des linearen Operators und der Kopplungsfaktoren. Der Vorteil dieses Vorgehens ist, dass nur die Laufzeit des überarbeiteten Teils des Algorithmus betrachtet wird und Veränderungen in diesem Teil somit deutlicher zu erkennen sind. Zur Simulationen wurde eine Nvidia Tesla K40c GPU unter einem Linux Betriebssystem mit einer Intel Core i5-4570 CPU und 24 GB RAM verwendet. Die Taktrate der GPU wurde für die Dauer der Testreihen auf den höchstmöglichen Wert festgelegt.

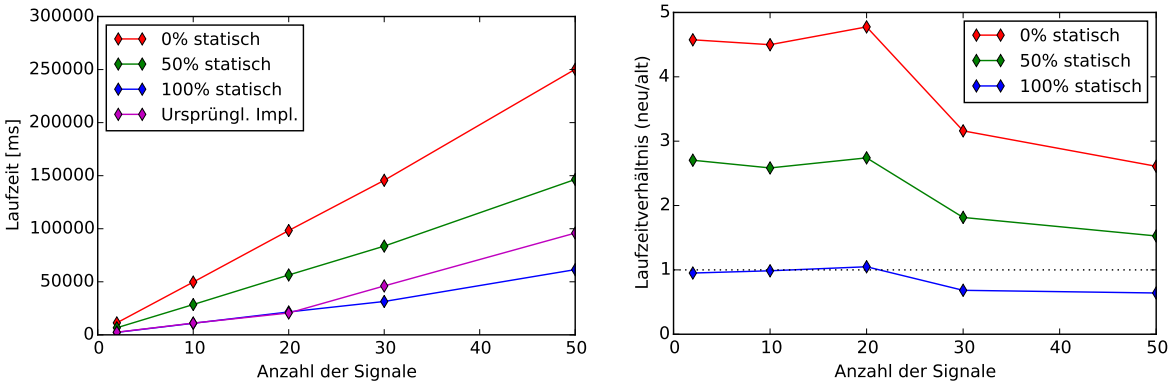
In den Abbildungen 5.4a und 5.5a sind jeweils die absoluten Laufzeiten der Implementierungen und Parametrisierungen zu erkennen. In den Abbildungen 5.4b und 5.5b sind die Laufzeitquotienten der beiden Implementierung dargestellt. Der Laufzeitquotient berechnet sich aus der Division der Laufzeit der überarbeiteten Implementierung durch die Laufzeit der ursprünglichen Implementierung.



(a) Absolute Laufzeiten.

(b) Laufzeitquotienten.

Abbildung 5.4.: Gegenüberstellung der Laufzeiten bei variierter Signallänge. Dargestellt jeweils für verschiedene Anzahlen von Signalen statisch auf dem Device. Die Simulationen wurden mit 20 Signalen und Setup gemäß Tabelle A.1 durchgeführt.



(a) Absolute Laufzeiten.

(b) Laufzeitquotienten.

Abbildung 5.5.: Gegenüberstellung der Laufzeiten bei variierter Anzahl von Signalen. Dargestellt jeweils für verschiedene Anteile der Daten statisch auf dem Device. Die Simulationen wurden mit 2^{18} Abtastwerten und Setup gemäß Tabelle A.1 durchgeführt.

In Abbildung 5.4b ist zu erkennen, dass sich die Laufzeiten der ursprünglichen und der überarbeiteten Implementierung bei größeren Signallängen um einen näherungsweise konstanten Faktor unterscheiden. Werden in der überarbeiteten Implementierung alle Daten statisch auf dem Host gespeichert, so ist die Laufzeit quasi identisch mit der Laufzeit der ursprünglichen Implementierung. Werden dagegen in der überarbeiteten Implementierung keine Daten auf dem Host gespeichert, beträgt die Laufzeit das vier- bis fünffache der Laufzeit der ursprünglichen

Implementierung. Bei kurzen Signallängen von unter 2^{16} Abtastwerten steigt die Laufzeit der überarbeiteten Implementierung im Vergleich deutlich stärker an. Die Ursache hierfür konnte aufgrund der begrenzten zur Verfügung stehenden Zeit nicht ermittelt werden und bedarf noch weiterer Betrachtung.

Abbildung 5.5a zeigt, dass die Laufzeit der ursprünglichen Implementierung mehr als linear mit der Anzahl der Signale ansteigt. Dies lässt sich darauf zurückführen, dass die Anzahl der Interaktionen im nichtlinearen Schritt quadratisch mit der Anzahl der Signale wächst. Jedes der M Signale koppelt mit den $M-1$ anderen Signalen. Insgesamt finden somit M^2-M Interaktionen statt. Dass das quadratische Wachstum nicht bei der überarbeiteten Implementierung festzustellen ist, lässt durch die schlechte Überlappung von Datentransfers und Berechnungen im vertikalen Teil erklären. Die Menge der übertragenen Daten steigt nur linearen mit der Anzahl der Signale an. Daraus folgt, dass auch die Laufzeit der Datentransfers nur linear wächst. Solange die Berechnungen weiterhin schneller abgeschlossen werden als die Datentransfers, wirkt sich das quadratische Wachstum der Kernellaufzeit daher nicht auf die Gesamtlaufzeit aus. Umgekehrt lässt sich daraus folgern, dass die Überlappung bei einer größeren Anzahl Signale besser ausfällt, als in Abschnitt 5.2.1 beschrieben. Das geringe Wachstum der Laufzeit der überarbeiteten Implementierung bei vollständiger Speicherung der Daten auf dem Device lässt sich allerdings hierdurch nicht erklären, da keine Datentransfers stattfinden.

Zusammenfassend lässt sich feststellen, dass die Laufzeit der überarbeiteten Implementierung bei vollständiger Speicherung der Daten auf dem Device annähernd gleich der Laufzeit der ursprünglichen Implementierung ist. Werden hingegen keine Daten auf dem Device gespeichert, liegt die Laufzeit der überarbeiteten Implementierung bei ausreichend großen Datensätzen um einen Faktor drei bis fünf höher.

6. Fazit

Im Rahmen dieser Arbeit wurde eine speicheroptimierte Implementierung der Split-Step-Fourier-Methode für Nvidia CUDA Grafikkarten geschaffen. Dazu wurde eine am Lehrstuhl bestehende Implementierung aufgegriffen und überarbeitet. In einem ersten Schritt wurde der grundlegende Speicherverbrauch der Implementierung durch minimale Änderungen am Datenfluss reduziert. Dabei konnte eine Reduktion des Speicherbedarfs um mehr als ein Drittel erreicht werden. Im zweiten Überarbeitungsschritt, wurde es ermöglicht einen Teil oder das gesamte Datenvolumen der Simulation vom Device-Speicher auf den Host-Speicher auszulagern. Dazu musste ein System zur Ressourcenverwaltung auf dem Device entworfen werden, das den flexiblen und sicheren Zugriff auf verschiedene Ressourcen erlaubt. Damit ist es ermöglicht worden die Anzahl der simulierten Signale zu erhöhen ohne den Speicherverbrauch zu verändern. Dies gestattet es die Vorteile einer GPU-beschleunigten Simulation auch für SDM-Systeme mit zahlreichen Moden zu nutzen.

Durch die flexibel gestaltete Ressourcenverwaltung, wird bei der Simulationen, die auch mit der ursprünglichen Implementierung durchgeführt werden konnten, dieselbe Laufzeit erreicht. Simulationen, die aufgrund von begrenztem Speicherplatz zuvor gar nicht durchgeführt werden konnten, sind mit der überarbeiteten Implementierung bei akzeptablen Geschwindigkeitseinbußen möglich. In den durchgeführten Versuchsreihen hat sich ein Laufzeitanstieg um einen Faktor von weniger als fünf gezeigt.

Zur weiteren Verringerung des Speicherverbrauchs und der Laufzeit kann der Algorithmus zukünftig um eine verbesserte Behandlung der Modengruppen in der zugrundeliegenden Gleichung (2.14) erweitert werden. Bisher findet in der überarbeiteten Implementierung, wie auch in der ursprünglichen Implementierung, keine gesonderte Behandlung von Modengruppen statt, sondern die Gruppen werden *virtuell* durch eine entsprechend angepasste Matrix der Kopplungsfaktoren dargestellt. Eine vollständige Umsetzung der Unterteilung kann den Speicherverbrauch der Betragsquadrate geringfügig senken und die Anzahl der Interaktionen im nichtlinearen Schritt signifikant verringern. Zudem würde die Matrix der Kopplungsfaktoren verkleinert, was vorteilhaft ist, da die Größe des *constant memory* stark begrenzt ist.

Die geschaffene Implementierung bietet damit erstmals die Möglichkeit GPU-beschleunigte Simulationen der Signalausbreitung in Mehrmodenfasern durchzuführen, wenn die Datenmenge der Signale die Speicherkapazität der GPU übersteigt. Es steht zu hoffen, dass die Implementierung somit in der Lage ist auf längere Zeit eine Grundlage für die Untersuchung von Mehrmodenfasern an der TU Dortmund zu bilden.

Literaturverzeichnis

[1] Essiambre, René-Jean ; Tkach, Robert W.: Capacity trends and limits of optical communication networks. In: *Proceedings of the IEEE* 100 (2012), Nr. 5. – ISSN 0018–9219

[2] Ellis, A. D. ; Zhao, Jian ; Cotter, D.: Approaching the Non-Linear Shannon Limit. In: *J. Lightwave Technol.* 28 (2010), Februar, Nr. 4. – ISSN 0733–8724

[3] Richardson, D. J. ; Fini, J. M. ; Nelson, L. E.: Space-division multiplexing in optical fibres. In: *Nature Photonics* 7 (2013), April, Nr. 5. – ISSN 1749–4885

[4] Krummrich, Peter M.: Optical amplification and optical filter based signal processing for cost and energy efficient spatial multiplexing. In: *Optics Express* 19 (2011), August, Nr. 17. – ISSN 1094–4087

[5] Hellerbrand, Stephan ; Hanik, Norbert: Fast Implementation of the Split-Step Fourier Method Using a Graphics Processing Unit. In: *Optical Fiber Communication Conference*, Optical Society of America (OSA), März 2010. – ISBN 978–1–55752–885–8

[6] Pachnicke, S. ; Chachaj, A. ; Helf, M. ; Krummrich, Peter M.: Fast parallel simulation of fiber optical communication systems accelerated by a graphics processing unit. In: *12th International Conference on Transparent Optical Networks*, Institute of Electrical and Electronics Engineers (IEEE), Juni 2010. – ISBN 978–1–4244–7799–9

[7] Agrawal, Govind P.: Nonlinear fiber optics: its history and recent progress. In: *Journal of the Optical Society of America B* 28 (2011), Dezember, Nr. 12. – ISSN 0740–3224

[8] Kaminow, Ivan (Hrsg.) ; Li, Tingye (Hrsg.) ; Willner, Alan E. (Hrsg.): *Optical Fiber Telecommunications VIB*. Academic Press. – ISBN 978–0–12–396960–6

[9] Engelbrecht, Rainer: *Nichtlineare Faseroptik*. 1. Auflage. Springer Berlin Heidelberg, 2014. – ISBN 978–3–642–40967–7

[10] Gloge, D.: Weakly Guiding Fibers. In: *Applied Optics* 10 (1971), Oktober, Nr. 10. – ISSN 0003–6935

[11] Unger, Hans-Georg: *Optische Nachrichtentechnik, Teil I: Optische Wellenleiter*. 2. Auflage. Hüthig Buch Verlag Heidelberg, 1990. – ISBN 3–7785–1752–X

[12] Krummrich, Peter M. ; Westhäuser, Matthias ; Brehler, Marius: *Skript: Optische Übertragungstechnik*. Juli 2015

[13] Winzer, Peter J. ; Foschini, Gerard J.: MIMO capacities and outage probabilities in spatially multiplexed optical transport systems. In: *Optics Express* 19 (2011), August, Nr. 17. – ISSN 1094–4087

[14] Essiambre, René-Jean ; Mestre, Miquel A. ; Ryf, Roland ; Gnauck, Alan H. ; Tkach, Robert W. ; Chraplyvy, Andrew R. ; Sun, Yi ; Jiang, Xinli ; Lingle, Robert: Experimental Investigation of Inter-Modal Four-Wave Mixing in Few-Mode Fibers. In: *IEEE Photonics Technology Letters* 25 (2013), März, Nr. 6. – ISSN 1041–1135

[15] Agrawal, Govind P.: *Nonlinear Fiber Optics*. 5th ed. Academic Press, 2012. – ISBN 978–0–12397–023–7

[16] Poletti, Francesco ; Horak, Peter: Description of ultrashort pulse propagation in multimode optical fibers. In: *Journal of the Optical Society of America B* 25 (2008), Oktober, Nr. 10. – ISSN 0740–3224

[17] Mecozzi, Antonio ; Antonelli, Cristian ; Shtaif, Mark: Nonlinear propagation in multimode fibers in the strong coupling regime. In: *Optics Express* 20 (2012), Mai, Nr. 11. – ISSN 1094–4087

[18] Mecozzi, Antonio ; Antonelli, Cristian ; Shtaif, Mark: Coupled Manakov equations in multimode fibers with strongly coupled groups of modes. In: *Optics Express* 20 (2012), Oktober, Nr. 21. – ISSN 1094–4087

[19] Brehler, Marius: *Modellierung und Simulation der Impulspropagation in einer Mehrmodenfaser für räumliches Multiplexen*, Technische Universität Dortmund, Lehrstuhl für Hochfrequenztechnik, Masterarbeit, November 2013

[20] Sinkin, O. V. ; Holzlohner, R. ; Zweck, John ; Menyuk, C. R.: Optimization of the split-step fourier method in modeling optical-fiber communications systems. In: *J. Lightwave Technol.* 21 (2003), Januar, Nr. 1. – ISSN 0733–8724

[21] *Nvidia CUDA Website*. `https://developer.nvidia.com/cuda-zone`, Abruf: 2. Sept. 2015

[22] Harris, Mark: *CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops*. `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/`. Version: April 2013, Abruf: 2. Sept. 2015

[23] *Nvidia cuFFT Website*. `https://developer.nvidia.com/cuFFT`, Abruf: 2. Sept. 2015

[24] *cuFFT Library User's Guide*. März 2015. – Version: DU-06707-001_v7.0

[25] Kirk, David ; Hwu, Wen-mei: *Programming Massively Parallel Processors: A Hands-on Approach*. 2nd. ed. 2012. – ISBN 9780123914187

[26] *Nvidia Visual Profiler Website*. `https://developer.nvidia.com/nvidia-visual-profiler`, Abruf: 2. Sept. 2015

A. Laufzeit- und Speicherverbrauchs-Werte

Tabelle A.1.: Vergleich des berechneten und gemessenen Speicherbedarfs sowie der Laufzeit bei variierter Signallänge und Anzahl der Signale. Wenn nicht anders in der jeweiligen Zeile angegeben, gehören die Daten zur überarbeiteten Implementierung. Grundlegendes Setup: $S_{real} = 64\,\text{bit}$, $M_{horz,local} = M_{static}$, $W_{horz,local} = 1$, $M_{horz,remote} = 1$, $W_{horz,remote} = M - M_{loc}$, $N_{vert} = N$ und $W_{vert} = 1$.

M	$\log_2(N)$	M_{static}	Speicher berechnet	Speicher gemessen	Speicher Differenz	Laufzeit
2	18	alte Impl	150 MiB	56 MiB	94 MiB	2463 ms
		0	122 MiB	28 MiB	94 MiB	11 287 ms
		1	126 MiB	32 MiB	94 MiB	6670 ms
		2	130 MiB	36 MiB	94 MiB	2346 ms
10		alte Impl	374 MiB	280 MiB	94 MiB	11 052 ms
		0	235 MiB	140 MiB	95 MiB	49 724 ms
		5	255 MiB	160 MiB	95 MiB	28 560 ms
		10	275 MiB	180 MiB	95 MiB	10 897 ms
20	16	alte Impl	140 MiB	234 MiB	94 MiB	5068 ms
		0	70 MiB	167 MiB	97 MiB	25 859 ms
		10	80 MiB	175 MiB	95 MiB	14 873 ms
		20	90 MiB	184 MiB	94 MiB	5531 ms
	19	alte Impl	1120 MiB	1214 MiB	94 MiB	41 995 ms
		0	560 MiB	657 MiB	97 MiB	197 296 ms
		10	640 MiB	735 MiB	95 MiB	113 154 ms
		20	720 MiB	814 MiB	94 MiB	43 668 ms
	22	alte Impl	8960 MiB	9054 MiB	94 MiB	366 257 ms
		0	4480 MiB	4577 MiB	97 MiB	1 573 961 ms
		10	5120 MiB	5215 MiB	95 MiB	908 442 ms
		20	5760 MiB	5854 MiB	94 MiB	375 938 ms
30	18	alte Impl	934 MiB	840 MiB	94 MiB	46 074 ms
		0	518 MiB	420 MiB	98 MiB	145 613 ms
		15	577 MiB	480 MiB	97 MiB	83 614 ms
		30	636 MiB	540 MiB	96 MiB	31 441 ms
50		alte Impl	1494 MiB	1400 MiB	94 MiB	95 910 ms
		0	801 MiB	700 MiB	101 MiB	250 281 ms
		25	899 MiB	800 MiB	99 MiB	146 456 ms
		50	997 MiB	900 MiB	97 MiB	61 469 ms

Tabelle A.2.: Vergleich des berechneten und gemessenen Speicherbedarfs sowie der Laufzeit bei variierter Größe und Anzahl der horizontalen Aufgaben und Arbeiter. Die Indizes *horz*, *local* und *remote* sind als h, l und r abgekürzt. Alle Daten gehören zur überarbeiteten Implementierung. Grundlegendes Setup: $S_{real} = 64\,\text{bit}$, $M = 12$, $N = 2^{20}$, $N_{vert} = N$ und $W_{vert} = 1$.

M_{static}	$M_{h,l}$	$W_{h,l}$	$M_{h,r}$	$W_{h,r}$	Speicher berechnet	Speicher gemessen	Speicher Differenz	Laufzeit
0	0	0	1	12	768 MiB	672 MiB	96 MiB	239 611 ms
0	0	0	2	6	767 MiB	672 MiB	95 MiB	242 517 ms
0	0	0	3	4	766 MiB	672 MiB	94 MiB	246 197 ms
0	0	0	4	3	766 MiB	672 MiB	94 MiB	248 287 ms
0	0	0	6	2	766 MiB	672 MiB	94 MiB	257 782 ms
0	0	0	12	1	766 MiB	672 MiB	94 MiB	278 651 ms
6	1	2	1	4	767 MiB	672 MiB	95 MiB	129 212 ms
6	1	2	1	5	783 MiB	688 MiB	95 MiB	131 169 ms
6	1	2	1	6	799 MiB	704 MiB	95 MiB	132 087 ms
6	1	3	1	1	734 MiB	640 MiB	94 MiB	156 505 ms
6	1	3	1	2	750 MiB	656 MiB	94 MiB	136 665 ms
6	1	3	1	3	767 MiB	672 MiB	95 MiB	132 723 ms
12	1	12	0	0	959 MiB	864 MiB	95 MiB	51 410 ms
12	2	6	0	0	958 MiB	864 MiB	94 MiB	51 379 ms
12	3	4	0	0	958 MiB	864 MiB	94 MiB	51 389 ms
12	4	3	0	0	958 MiB	864 MiB	94 MiB	51 434 ms
12	6	2	0	0	958 MiB	864 MiB	94 MiB	51 488 ms
12	12	1	0	0	958 MiB	864 MiB	94 MiB	52 042 ms

Tabelle A.3.: Vergleich des berechneten und gemessenen Speicherbedarfs sowie der Laufzeit bei variierter Größe und Anzahl der vertikalen Aufgaben und Arbeiter. Die Indizes *horz*, *local* und *remote* sind als h, l und r abgekürzt. Alle Daten gehören zur überarbeiteten Implementierung. Grundlegendes Setup: $S_{real} = 64\,\text{bit}$, $M = 12$, $M_{static} = 6$, $N = 2^{20}$, $M_{horz,local} = 1$, $M_{horz,remote} = 1$

$W_{h,l}$	$W_{h,r}$	$\log_2(N_{vert})$	W_{vert}	Speicher berechnet	Speicher gemessen	Speicher Differenz	Laufzeit
1	1	20	1	702 MiB	608 MiB	94 MiB	151 197 ms
1	1	18	4	702 MiB	608 MiB	94 MiB	139 549 ms
1	1	16	16	703 MiB	608 MiB	95 MiB	136 120 ms
1	1	14	64	706 MiB	608 MiB	98 MiB	139 400 ms
1	6	16	1	863 MiB	768 MiB	95 MiB	139 342 ms
1	6	16	4	863 MiB	768 MiB	95 MiB	116 949 ms
1	6	16	7	863 MiB	768 MiB	95 MiB	115 739 ms
1	6	16	10	863 MiB	768 MiB	95 MiB	115 727 ms

B. Abkürzungsverzeichnis

CPU Hauptprozessor (Central Processing Unit)

CUDA Nvidia CUDA (Compute Unified Device Architecture)

cuFFT Nvidia cuFFT-Bibliothek

DGL Differentialgleichung

EM-Welle elektromagnetische Welle

FFT schnelle Fourier-Transformation (Fast Fourier Transform)

FWM Vierwellenmischung

GPU Grafikkarte (Graphics Processing Unit)

ME Manakov-Gleichung (Manakov Equation)

MIMO Multiple-Input Multiple-Output

MM-GME verallgemeinerte Manakov-Gleichung für Mehrmodenfasern (Multi-Mode Generalized Manakov Equation)

MM-GNLSE verallgemeinerte nichtlineare Schrödinger-Gleichung für Mehrmodenfasern (Multi-Mode Generalized Nonlinear Schrödinger Equation)

MMF Mehrmodenfaser (Multi-Mode Fiber)

NLSE Nichtlineare Schrödinger-Gleichung (Nonlinear Schrödinger Equation)

RAM Hauptspeicher (Random Access Memory)

SDM Raummultiplexbetrieb (Space Division Multiplexing)

SMF Einmodenfaser (Single-Mode Fiber)

SMP Streaming-Multiprozessor

SPM Selbstphasenmodulation

SSFM Split-Step-Fourier-Methode

WDM Wellenlängenmultiplexbetrieb (Wavelength Division Multiplexing)

XPM Kreuzphasenmodulation

C. Informatik-Glossar

Allokieren Reservieren von Speicherplatz für eine bestimmte Aufgabe.

C/C++ Zwei verbreitete Programmiersprachen.

Caching Das Zwischenspeichern von Daten in einem speziellen Cache-Speicher. Dient dem schnelleren Lese- und Schreibzugriff bei wiederholten Zugriffen auf das selbe Datum.

Device Bezeichnung der Grafikkarte unter CUDA.

Host Bezeichnung des Hauptrechners unter CUDA.

Iteration „Wiederholung“. Bezeichnet insbesondere das Durchlaufen einer Programm-Schleife.

Kern Siehe Rechenkern.

Kernel Bezeichnet unter CUDA eine Funktion, die auf dem Device ausgeführt wird. Zahlreiche Threads auf dem Device führen zeitgleich denselben Kernel aus.

Prozessor Ausführungseinheit mit einem oder mehreren Rechenkernen. Enthält außerdem Cache-Speicher und gegebenenfalls weitere Komponenten.

Puffer Speicher der verwendet wird um Daten temporär zwischenzuspeichern.

Rechenkern Enthält mehrere Funktionseinheiten um logische und arithmetische Operationen auszuführen. Bestandteil eines Prozessors.

Row-major Ein Speicherformat für Matrizen in eindimensionalem Speicher. Die Matrix wird in ihre Zeilen zerlegt; die Zeilen werden hintereinander gespeichert. Analog dazu wird spaltenweises Speichern als *column-major* bezeichnet.

Thread Zu deutsch „Faden“. Eine Abfolge von seriellen Befehlen. Wird durch einen einzelnen Rechenkern ausgeführt.

Thread-Block Threads auf dem Device werden unter CUDA in Blöcken gruppiert. Die Threads eines Block laufen zeitgleich auf einem SMP.

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift