DECKBLATT

EIGENSTÄNDIGKEITSERKLÄRUNG

ZUSAMMENFASSUNG

INHALTSVERZEICHNIS

# Einführung

## Motivation

Software für Automobile muss eine reihe von Test und Kriterien erfüllen um zugelassen zu werden. Eine dieser Kriterien ist, dass die Software in einem Testfahrzeug mehrere Hunderttausend Kilometer gefahren sein muss. Zu diesen gefahren Kilometern werden auch Kilometer gefahren in einer Simulation gezählt. In Automotive Anwendungen werden Hardwarebeschleuniger verwendet um Rechenintensive algos auszuführen. In Simulationen ist diese verwendung von Hardwarebeschleinigern nicht wünschenswert (warum?). Da Simulationen auf Rechenmaschinen mit GPUs ausgeführt werden, können diese zur Beschleunigung verwendet werden. Die Algorithmen können aber nicht einfach so auf einer GPU ausgeführt werden. Es braucht …

Nvidia, ein GPU Hersteller, (stellte vor? ist jetzt im Kommen?) die Parallel Algorithms. Mit welchen algorithmen einfach für GPU offloading kompiliert werden können.

## Zielsetzung

Mithilfe von C++17 und dem Nvidia toolkit soll ein im bereich Automotive verwendeter Algorithmus parallelisiert werden. Dieser Algorithmus soll dann auf einer GPU laufen. Ziel ist es dadurch die (möglichkeit, brauchbarkeit) der von C++17 eingeführten Algorithmen mit Execution Policies unter verwendung des Nvidia Toolkits zu (erfassen, bewerten, analysieren).

## Kapitelübersicht

Die Arbeit besteht aus folgenden Kapiteln:

1. **Einleitung**
2. **Grundlagen**

In diesem Kapitel werden Themen und Begriffe eingeführt, auf denen das Lösungskonzept aufbaut. Hierbei werden Themen zu Parallelem Programmieren, (GPUs / Nvidia) und zum verwendeten Algorithmus behandelt.

1. **Lösungsansatz**

Dieses Kapitel beinhaltet die Analyse der Gegebenheiten vor der Parallelisierung des Algorithmus. Außerdem werden zwei Lösungsansätze vorgestellt um den Algorithmus zu Parallelisieren. Die Ansätze werden bewertet und es wird Ansatz ausgewählt. Dieser Ansatz wird im Detail verfolgt, aufkommende Probleme beschrieben und eine endgültige Lösung des Ansatzes präsentiert.

1. **Ergebnisse**

In diesem Kapitel wird die erarbeitete Lösung validiert und die aus der Lösung erhaltenen Ergebnisse dargestellt und bewertet.

1. **Zusammenfassung**

Es werden erarbeitete Erkenntnisse, aufgetretene Probleme und erreichte Ergebnisse zusammengefasst. (Ausblick auf neue Aufgaben?).

1. **Literaturverzeichnis**
2. (Anlagen)

# Grundlagen

## Paralleles Programmieren

### Daten Parallelität

Es gibt zwei Arten von Parallelität. Parallelität kann erreicht werden indem Ausführungsschritte eines Programms voneinander gekapselt werden und dann gleichzeitig ausgeführt werden. Hierbei wird jedem Ausführungsschritt ein Thread oder manchmal auch ein ganzer Prozessorkern zugewiesen. Diese Art der Parallelität wird Taskparallelität genannt. Um Taskparallelität zu erreichen muss darauf geachtet werden, keine racing conditions oder deadlocks einzuführen. Außerdem sollte ein Prozessor möglichst effizient ausgelastet werden um maximale Performance zu erreichen. Um dies möglich zu machen werden Ausführungsschritte oft in kleinere Unterschritte aufgeteilt um sie besser auf die Anzahl von Threads und Prozessorkernen zu verteilen. Je nach Natur des Programms kann sich dies aber durchaus als schwierig oder unmöglich herausstellen.

Die zweite Art der Parallelität wird Datenparallelität genannt. Hierbei werden Threads Datenelemente zugewiesen. Anstatt die Ausführungsschritte des Programms aufzuteilen, wird auf die Menge der zu bearbeitenden Daten geachtet. So werden die einzelnen Datenelemente, welche meist unabhängig voneinander sind, auf verfügbare Threads auf den Prozessorkernen verteilt. (Boyd, 2008)

Datenparallelität spielt vor allem eine Rolle in Programmen welche sich mit Bild- und Videoverarbeitung befassen. In diesen Bereichen werden oft tausende von Datenelementen verarbeitet, welche unabhängig voneinander erfasst und bearbeitet werden können.

Compute bound and memory bound

The most important characteristic of the GPU memory subsystem is the cache architecture. Unlike a CPU, the GPU has hardly any read/write cache. It is assumed that so much data will be streaming through the processor that it will overflow just about any cache. As a result, the only caches present are separate read-through and write-through buffers that smooth out the data flow. Therefore, it is critical to select algorithms that do not rely on reuse of data at scales larger than the few local registers available.

### Parallel STL

#### Standard Algorithms

Klarer machen, dass es sich hier nur um einen kleinen Teil der STL handelt. -> ich behandle, was mit C++17 als parallel STL dazugekommen ist.

Die C++ Standard Template Library (STL) ist eine Menge von Template Klassen, welche häufig verwendete Programmier- und Datenstrukturen und Funktionen zur Verfügung stellt. Es ist eine Bibliothek von Klassen, Algorithmen und Iteratoren.

Die Algorithmen der C++ STL sind Funktionen, die von der Algorithms library definiert werden. Diese Funktionen agieren dann auf einem Bereich von Elementen. Beispiele hierfür sind das Sortieren, Suchen, Zählen oder Modifizieren von Elementen einer Liste oder eines Arrays.(Algorithms library - cppreference.com, 2021)

Die Parallel STL ist ein kleiner Teil dieser Algorithms Library. Dieser mit C++17 dazugekommene Teil beinhaltet die Algorithmen „std::for\_each“, „std::for\_each\_n“, „std::transform“ und „std::transform\_reduce“, welche im folgenden näher behandelt werden.

1. std::for\_each, std::for:\_each\_n

Was macht das ding ? -> anwender der Funktion auf alle elemente

Der Algorithmus „std::for\_each“ wird verwendet um eine Funktion auf alle Elemente eines Bereichs anzuwenden. Er besitzt mehrere Interfaces und das für diese Arbeit wichtige Interface sieht wie folgt aus:



„std::for\_each“ akzeptiert als ersten Parameter ein Objekt der Klasse „ExecutionPolicy“, auf welches in einem späteren Kapitel genauer eingegangen wird. Weitere akzeptierte Parameter sind Objekte der Klasse „ForwardIt“. Hierbei handelt es sich um Iteratoren die den Bereich der von „std::for\_each“ behandelt wird bestimmen, indem ein Iterator für das erste und das letzte Element des Bereichs angegeben werden. Als letzten Parameter muss eine Funktion angegeben werden, die für jedes Element des Bereichs ausgeführt wird. (std:for\_each - cppreference.com, 2022)

Neben dem „std::for\_each“ Algorithmus, gibt es noch dem „std::for\_each“ sehr ähnlichen Algorithmus „std::for\_each\_n“. Das für diese Arbeit wichtige Interface sieht wie folgt aus:



Dieses unterscheidet sich im Wesentlichen nicht besonders vom Interface von „std::for\_each“. Anstatt des Iterators für das letzte Element, wird ein Wert n für die Anzahl der Elemente des zu bearbeitenden Bereichs angegeben. (std:for\_each\_n - cppreference.com, 2022)

1. std::transform

Zu anfang erwähen dass es zwei varianten gibt. -> Warum gibt es die ? -> was machen unary op und binary op

Der Algorithmus „std::transform“ appliziert eine einstellige Verknüpfung „unary\_op“ auf einen Bereich festgelegt durch die Iteratoren „first1“ und „last1“. Das Ergebnis wir dann in einem Bereich gespeichert, der vom Iterator „d\_first“ bestimmt wird. Grundsätzlich ähneln die akzeptierten Parameter von „std::transform“ denen von „std::for\_each“. Im Gegensatz zu „std::for\_each“ garantiert „std::transform“ die Reihenfolge der Applizierung des Operators nicht. Die wichtigen Interfaces für „std::transform“ sind:



Das besondere von „std::transform“ ist, dass es ein Interface hat welches es ermöglicht zwei Eingangsbereiche anzugeben. Dieses Interface unterscheidet sich zum Obigen in zwei Aspekten. Erstens wird ein weiterer Parameter „first2“ akzeptiert. Dieser Iterator bestimmt den Anfang des zweiten Eingangsbereichs. Hierbei wird die Größe des Bereichs vom ersten Eingangsbereich abgeleitet. Der zweite Aspekt ist der applizierte Operator. In diesem Interface wird eine binäre Verknüpfung angewendet, welche die Elemente der beiden Eingangsbereiche verknüpft.

Das Interface sieht wie folgt aus:



Bei „std::transform“ ist zu beachten, dass die Reihenfolge der Operationen nicht garantiert ist. Möchte man die Reihenfolge der Operationen garantiert haben, sollte „std::for\_each“ stattdessen verwendete werden.

(std:transform - cppreference.com, 2022)

1. std::transform\_reduce

„std::transform\_reduce“ fungiert im Prinzip genauso wie „std::transform“, aber zusätzlich werden die Ergebnisse der „transform“ Operation noch aufsummiert. Für der Wert der Summe wird ein Initialwert „init“ als Parameter akzeptiert, auf welchen dann die Ergebnisse addiert werden. Es kann jedoch nicht nur eine Summe produziert werden, sondern z.B. auch ein Produkt. Dies macht der Parameter „reduce“ der Klasse „BinaryReductionOp“ möglich.



Wie auch zuvor bei „std::transform“ akzeptiert „std::transform\_reduce“ auch zwei Eingansbereiche



(std:transform\_reduce - cppreference.com, 2022)

#### Execution Policies

Mit C++17 wurden die Standard Algorithms mit sogennanten execution policies erweitert. Diese ermöglichen es Standard Allgorithms zu parallelisieren. Viele Algorithmen besitzen nun Überladungen, welche Objekte dieser execution policies akzeptieren. Die im vorherigen Kapitel dargestellten Interfaces der Standard Algorithms sind alle solche Überladungen.

Zu den execution policies gehören:

* sequenced\_policy
* parallel\_policy
* parallel\_unsequenced\_policy

(std:execution:sequenced\_policy, std:execution:parallel\_policy, std:execution:parallel\_unsequenced\_policy, std:execution:unsequenced\_policy - cppreference.com, 2022)

Wird das Objekt „seq“ der sequenced\_policy Klasse übergeben, wird die Funktion in richtiger Reihenfolge (sequenziell) ausgeführt. Dennoch darf der Compiler den Algorithmus parallelisieren, wenn dies für das Programm nicht sichtbar ist.

Beim übergeben des „par“ Objekts der parallel\_policy Klasse wird dem Algorithmus signalisiert, dass dieser die Ausführung auf mehrere Threads aufteilen darf. Dennoch werden die Operationen innerhalb eines Threads sequenziell abgearbeitet.

Wird das „par\_unseq“ Objekt der Klasse parallel\_unseq\_policy übergeben, wird die Ausführung auf mehrere Threads aufgeteilt und vektorisiert.(Voss et al., 2019)

was ist vectorisieren?

#### Typische Anwendung von Parallelem Code mit Parallel STL

Stencil operations(for\_each), Bildverarbeitung -> transform, transform\_reduce anwendung?

In den folgenden Beispielen werden Lambdaausdrücke, auch Lambdafunktionen genannt, verwendet. Dies sind anonyme Funktionen welche nicht an einen Bezeichner gebunden sind. Sie eignen sich sehr gut als Funktionen für die Parallel STL Algorithmen.

1. Anwendungsfall 1: Ändern der Helligkeit in einem Bild.

Angenommen man möchte die Helligkeit eines Bildes ändern, so muss jeder Pixelwert mit einem Wert skaliert werden. Ist der Wert größer als 1, wird die Helligkeit des Bildes erhöht. Ist der Wert kleiner als 1, wird die Helligkeit gesenkt. Für diese Art der Bildverarbeitung bietet sich der „std::transform“ Algorithmus an. Hierzu ein Beispiel:



Ein Graufstufenbild wird im eindimensionalen Vector „image“ gespeichert. Das Ergebnis wird in einem ebenfalls eindimensionalen Vector „image2“ gespeichert. Durch den Lambda-Ausdruck in Zeile 5 wird jeder Wert „a“ aus dem Vector „image“ mit dem Faktor 3 multipliziert und an passender Stelle in den Vector „image2“ geschrieben.

1. Anwendungsfall 2: Skalarprodukt berechnen.

Bei der parallelen Berechnung des Skalarprodukts zweier Vektoren können Racingconditions eingeführt werden, da für die Summierung der Produkte jeweils der aktuelle Wert der Summe gelesen werden muss bevor die Summe aktualisiert werden kann. Um Racingconditions zu vermeiden sollte „std::transform\_reduce“ verwendet werden:



Hier wird das Skalarprodukt der beiden Vektoren „vec1“ und „vec2“ gebildet und in der Variablen „sum“ gespeichert.

1. Anwendungsfall 3: Stencil Operationen

Stencil Operationen brauchen meist Informationen zu benachbarten Werten in einem zweidimensionalen Array um Berechnungen durchzuführen oder Arraywerte miteinander zu vergleichen. Werden Stencil Operationen mit einem Algorithmus der Parallel STL umgesetzt, kann folgendes Hindernis entstehen. Die verwendete Lambdafunktion besitzt nur eine Referenz auf das benutzte Element, aber hat keinerlei Informationen über die Position des Elements innerhalb des Bereichs. Somit können auch keine Informationen über benachbarte Elemente erhalten werden. Um dieses Hindernis zu umgehen muss mit Hilfe von Pointer Arithmetik der Index des Elements berechnet werden. Dieser kann dann verwendet werden um benachbarte Elemente zu adressieren.

Die Verwendung einer Stencil Operation und die Umgehung eines solchen Hindernisses wird mit Hilfe folgenden Beispiels erläutert:

 Bei diesem Beispiel handelt es sich um eine vereinfachte Implementierung von Conway's "Game of Life".

Hier werden für jede Zelle in einem Bereich entschieden, ob diese weiterhin existieren werden. Eine Zelle kann nur dann existieren, wenn sie mindestens 3 Zellen als direkte Nachtbaren hat. Zusätzlich werden Felder zu Zellen wenn das Feld von 3 oder mehr Zellen umgeben ist. Um über den N x N großen Bereich „field“ zu iterieren wird „std::for\_each“ verwendet.

In Zeile 3 des Beispiels findet die Berechnung des Indexes statt. Es wird die Adresse des Bereichs, welches gleichzeitig die Adresse des ersten Elements ist, von der Adresse des gerade verwendeten Elements subtrahiert. Als Ergebnis erhält man den Abstand des Elements zum ersten Element und somit den Index des Elements innerhalb des Bereichs.

Die eigentlich Stencil Operation findet in den Zeilen 12 – 15 statt. Hier werden die Werte benachbarten Zellen mit Hilfe des zuvor berechneten Index summiert.

## (GPU und Nvidia HPC)(Heterogene Systeme)

### Graphics Processing Unit

##### Hardwarearchitektur

Eine CPU ist so entworfen, dass sie eine Sequenz von Operationen so schnell wie möglich ausführen kann. Dahingegen ist die GPU darauf ausgelegt tausende von Operationen gleichzeitig auszuführen. Deshalb unterscheidet sich die Hardwarearchitektur der GPU von der einer CPU.(Programming Guide : CUDA Toolkit Documentation, 2022)

Die Grundarchitektur einer GPU unterscheidet von der einer CPU in mehreren Aspekten. Während eine CPU aus einer Handvoll von komplexen Prozessorkernen besteht, ist eine GPU mit hunderten einfacheren Prozessorkernen ausgestattet. Diese Kerne besitzen jeweils tausende parallel laufende Hardware Threads.(Brodtkorb et al., 2013, 2013) Abbildung 1 veranschaulicht den Aufbauunterschied.

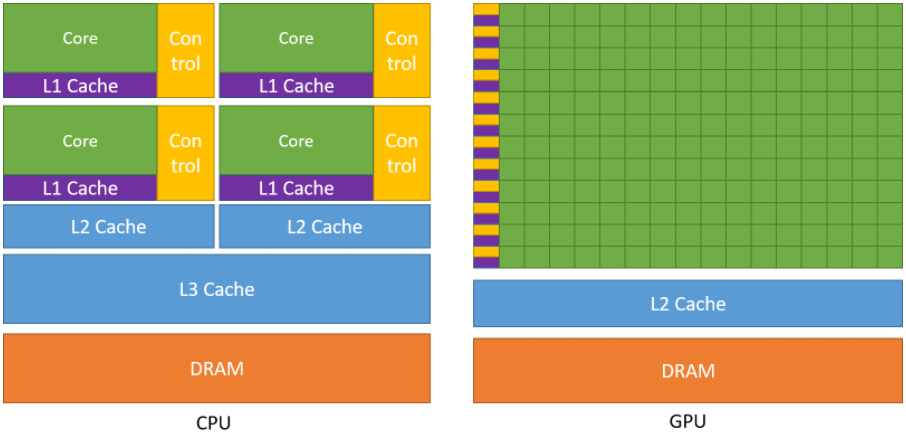


Abbildung . Die GPU dediziert mehr Kerne für die Datenverarbeitung. (Programming Guide : CUDA Toolkit Documentation, 2022)

##### Speicherverwaltung / Heterogene Systeme

Einer der entscheidenderen Unterschiede in Bezug auf Speicher von CPU und GPU ist die Nutzung von Cache Speicher

GPUs sind darauf ausgelegt die gleichen Instruktionen auf mehreren Threads gleichzeitig auszuführen. Dieses Prinzip wird Single Instruction Multiple Thread (SIMT) genannt. Deshalb ist ein kleinerer Cache Speicher pro Prozessorkern ausreichend. Diese Cache Speicher werden von einem gerätübergreifenden Cache Speicher bedient, welcher der letzte Cache Speicher in der Speicherhirarchie ist. Dieser Cache Speicher unterstützt sowohl Lese- als auch Schreibbefehle.

Moderne GPUs besitzen einen gemeinsam genutzten Speicher, auch Shared Memory genannt, welcher dafür genutzt wird Daten zwischen Threads zu teilen. Shared Memory ist kein Cache Speicher, sondern ein sogenannter Scratchpad Speicher. Dieser wird im Gegensatz zu Cache Speicher von der Anwendung und nicht automatisch verwaltet. Somit können Daten explizit zwischen Speicher und Shared Memory transferiert werden. (Understanding GPU caches – RasterGrid, 2022)

### CUDA

CUDA ist eine Programmierschnittstelle entwickelt von Nvidia. Diese ermöglicht es dem Programmierer Anwendungssoftware zu schreiben, die ihre Parallelität skaliert um die steigende Anzahl an Prozessorkernen vollends auszunutzen. CUDA stellt drei Abstraktionen bereit, welche es ermöglichen ein Problem in Teilprobleme aufzuteilen. Abstrahiert wird unteranderem die zu Verfügung stehenden Threads. Threads werden zu gleichgroßen Blöcken zusammengefasst. Teilprobleme können dann unabhängig voneinander parallel gelöst werden, indem jedem Teilproblem ein Block mit Threads zugeteilt wird. Die Anzahl von Threads in einem Block ist limitiert durch die Anzahl der Threads die sich auf einem Prozessorkern befinden. Auf aktuellen GPUs kann ein Threadblock bis zu 1024 Threads beinhalten. Threadblöcke sind Teile eines sogenannten Gitters, welches die Threadblöcke organisiert. (Programming Guide : CUDA Toolkit Documentation, 2022) Abbildung 2 zeigt die logische Unterteilung von Threads in Blöcke und Gitter.

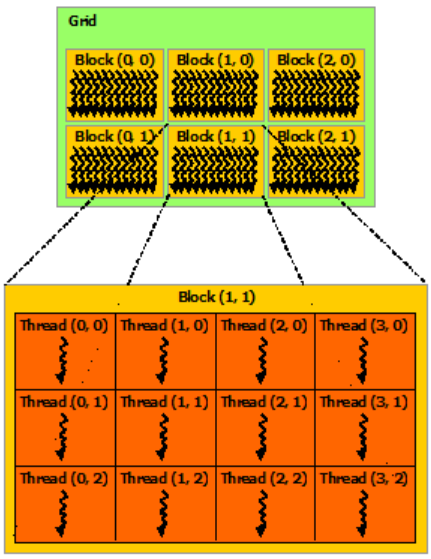


Abbildung . Ein Gitter (engl. Grid) ist unterteilt in Blöcke (engl. Blocks) von Threads. (Programming Guide : CUDA Toolkit Documentation, 2022)

CUDA erlaubt es dem Programmierer bestimmte Funktionen zu definieren, welche dann parallel von CUDA Threads ausgeführt werden. Diese Funktionen werden Kernel genannt. Einem Kernel wird definiert wie viele CUDA Threads diesen ausführen. Hierbei ist die Anzahl der Threads nicht limitiert, da ein Kernel von mehreren Threadblöcken ausgeführt werden kann. Bei der Definition eines Kernels wird die Anzahl der Threads pro Block und die Anzahl von Blöcken pro Gitter angegeben.(Programming Guide : CUDA Toolkit Documentation, 2022)

Nvidias GPU Architektur ist um eine Anordnung von Streaming Multiprozessoren (SMs) konzipiert. Jeder Streaming Multiprozessor kann einen Threadblock nach dem anderen ausführen. Die Anwendungssoftware wird auf die entsprechende GPU Hardwarearchitektur skaliert, indem die auszuführenden Threadblöcke auf die Anzahl der verfügbaren Streaming Multiprozessoren gleichmäßig verteilt werden. So kann eine GPU mit vier Streaming Multiprozessoren vier Threadblöcke gleichzeitig ausführen, während eine GPU mit zwei Streaming Multiprozessoren in der gleichen Zeitspanne nur zwei Threadblöcke ausführen kann. (Programming Guide : CUDA Toolkit Documentation, 2022)

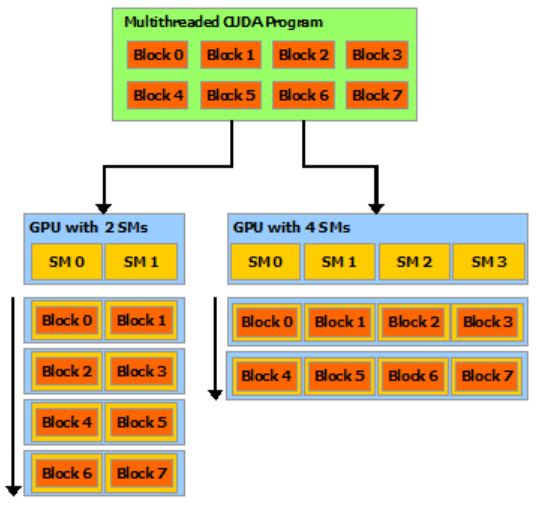


Abbildung 3. Automatische Skalierung durch Aufteilung der Blöcke auf die Verschiedenen Streaming Multiprozessoren. (Programming Guide : CUDA Toolkit Documentation, 2022)

#### CUDA Unified Memory

Unified Memory bezeichnet einen Speicheradressbereich, auf welchen jeder Prozessor in einem System zugreifen kann.

Veranschaulicht durch Abbildung 4:

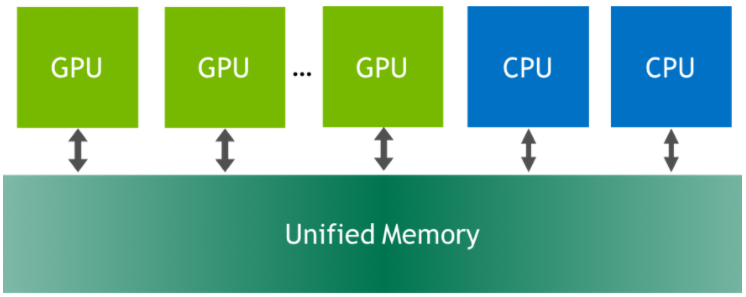


Abbildung . Unified Memory von allen Prozessoren abrufbar. (NVIDIA Developer Blog, 2017a)

Diese Technologie ermöglicht es Speicher/Daten zu allocieren, welche von sowohl CPU als auch GPU gelesen und geschrieben werden können. Um Unified Memory anzulegen werden aufrufe zu malloc() oder new mit aufrufen zu cudaMallocManaged() ersetzt. cudaMallocManaged() ist eine allocation function, welche einen Pointer zugänglich für jeden Prozessor zurückgibt. Greift ein Programm auf Daten zu die auf diese Art angelegt wurden, kümmert sich die CUDA Systemsoftware und oder die Hardware um das migrating der memory pages auf den Speicher des zugreifenden Prozessors.(NVIDIA Developer Blog, 2017b) -> virtueller Speicher mit paging für GPUs, gibt bei CPUs schon länger.

Greift die GPU auf Speicher zu der von cudaMallocManaged angelegt wurde werden folgende schritte ausgeführt. Zuerst werden neue pages auf der GPU angelegt. Dann werden die alten pages auf der CPU unmapped. Daten werden dann von der CPU zur GPU kopiert und die neuen pages auf ger GPU werden gemapped. Zuletzt werden die alten CPU pages freigegeben/gelöscht.

Wenn neuere GPUs auf eine page zugreifen, welche sich nicht auf dem lokalen Speicher der GPU befindet, wird beim übersetzen der page eine fault message generiert. Die GPU kann mehrere solcher fault messages gleichzeitig generieren. Der Unified Memory Treiber verarbeitet die faults, beseitigt Duplikate, bringt die mappings auf den neusten stand und transferiert die Daten. Wichtig zu erwähnen ist, dass Unified Memory eine Entwicklung von NVIDIA ist und deshalb auf nur auf NVIDIA GPUs verfügbar ist.(NVIDIA Developer Blog, 2017a)

### (Nvidia HPC)

Position in der Gliederung noch nicht sicher.

Erklären wie nvc++ c++ in cuda kernel übersetzt

Was stellt die SDK zur verfügung (compiler etc.) -> bedingungen um diese zu nutzen. Bedingung für CUDA Unified Mem.

Das Nvidia HPC Software Development Kit (SDK) beinhaltet sowohl C, C++ und Fortan Compiler als auch Bibliotheken und Tools für die Verbesserung von Leistung und Portabilität von High Performance Computing (HPC) Anwendungen. (NVIDIA Developer, 2020)

Der NVIDIA HPC C++ Compiler NVC++ unterstützt den C++17 Standard und wird genutzt um Algorithmen mit Execution Policies wie „std::execution::par“ oder „std::execution::par\_unseq“ für die Ausführung auf NVIDIA GPUs zu kompilieren. Für die Aktivierung der GPU-Beschleunigung werden keinerlei zusätzliche Spracherweiterungen oder Bibliotheken benötigt.

GPU-Beschleunigung kann aktiviert werden, indem der Comandozeilenbefehl „-stdpar“ an den Compiler übergeben wird. Die Datenübertragung zwischen dem Host Speicher und dem GPU Speicher geschieht automatisch unter der Verwendung von CUDA Unified Memory. Durch die Verwendung des NVC++ Compilers wird die GPU-Beschleunigung von Parallelem C++ Code vereinfacht. Dennoch gibt es Einschränkung für die Verwendungen von NVC++ zur GPU-Beschleunigung, welche nun genannt werden

Der NVC++ erkennt automatisch, welche Funktionen für die Ausführung auf einer GPU kompiliert werden müssen. Dies ist aber nur möglich wenn die Funktionsdefinitionen im selben source file liegen wie der Aufrufe der Funktionen.

-> Das spielt auf Funktionen an die du aus dem stdpar aus aufrufst? Würd ich dazu schreiben, ist sonst etwas unklar. Der Hintergrund ist ja das der code für den stdpar aufruf, also das lambda oder ähnlich, in gpu code übersetzt werden, also muss es auch gpu code geben für alles was das lambda benutzt?

NVC++ vertraut auf CUDA Unified Memory um Daten automatisch zwischen CPU und GPU zu übertragen. Dies ist aktuell nur möglich, wenn diese Daten dynamisch auf dem Heap Speicher der CPU allokiert sind und vom NVC++ Compiler kompiliert wurden. Das bedeutet, dass jeder Pointer der dereferenziert und jedes C++ Objekt das referenziert wird muss auf Daten auf dem CPU-Heap referieren. Außerdem müssen diese Daten von einer Anwendung, welche vom NVC++ Compiler kompiliert wurde, allokiert worden sein.

Ebenso wie für verwendete Daten, gibt es Limitierungen bei der Verwendung von Funktionen in Algorithm Aufruf. So können keine Funktionspointer an einen Algorithm Aufruf als Parameter übergeben werden. Nur Funktionsobjekte oder Lambdas können übergeben werden.

Die letzte wichtige Einschränung ist, dass Exceptions in Code welcher auf der GPU ausgeführt wird nicht verwendet werden sollte. Der Exceptioncode kompiliert zwar, aber hat nicht das erwarte Verhalten. So werden Catch-Bereiche ignoriert und das werfen einer Exception führt dazu, dass der GPU-Kernel abgebrochen wird. (C++ Parallel Algorithms Version 22.2 for ARM, OpenPower, x86, 2022)

Es ist außerdem zu erwähnen, dass die Nvidia HPC SDK zum Zeitpunkt dieser Arbeit nur für Linux-Systeme zur Verfügung steht.

## **(Algorithmus**)

Kurz (halbe seite)! Was macht der Algorithmus

OpenCV ist eine open source Library für Computer Vision. Für diese Arbeit wichtig ist das Modul zur Berechnung des Optischen Flusses. Der von OpenCV implementierte Algorithmus zur Berechnung des Optischen Flusses basiert auf dem Paper „Two-Frame Motion Estimation Based on Polynomial Expansion“ von Gunnar Farnebäck.

In diesem Paper stell Farnebäck einen Algorithmus zur Bewegungsabschätzung vor, welcher auf Distributivgesetzen basiert. So wird im Algorithmus zuerst jeweils eine Umgebung (Neighbourhood) für zwei Frames mit quadratischen Polynomen angenähert. Dann wird die Verschiebung mit Hilfe der polynomialen Erweiterungskoeffizienten abgeschätzt. (Farnebäck, 2003)

# Lösungsansatz / Versuchsaufbau

## Anforderungsanalyse / IST Zustand /

Enscheidung was optimiert wird. -> Messung des Algos von opencv -> wie wird gemessen? Was wird gemessen (Datensatzt / Video)?

Ergebnisse der Messung -> Auswertung -> Entscheidung was optimiert werden soll

Um entscheiden zu können welchen Teil der OpenCV Implementierung am meisten von einer Parallelisierung profitiert, ist es notwendig herauszufinden welche Teile der Implementierung am meisten Zeit für ihre Ausführung benötigen. Diese Ausführungszeit kann durch Komplexität oder Anzahl der Rechenoperationen und die Menge an Speicherzugriffen beeinflusst werden. Danach muss analysiert werden ob Parallelisierung dieses Teils möglich ist. Vor allem sollte auf mögliche Datenparallelität geachtet werden.

Die Implementierung von OpenCV teilt die Berechnung des Optischen Flusses in mehrere Unterfunktionen auf. Messungen sollen zeigen, wie viel Zeit die einzelnen Unterfunktionen jeweils in Anspruch nehmen. Für die Zeitenmessung wird ein Video , welches in einzelne 300 Framebilder aufgeteilt ist, verwendet. Zeiten werden mithilfe der „chrono::steady\_clock“ Klasse aus der Standard Library von C++ genommen. Diese ermöglicht es Zeitpunkte zu speichern und daraus dann Zeitintervalle zu berechnen.

(Berechnung veranschaulichen?)

Um das Input Video einzulesen und den daraus resultierenden Optischen Fluss darzustellen wird das Programm „densFlow“ aus Anhang A verwendet. „denseFlow“ basiert auf dem Beispielcode zur Berechnung von Optischem Fluss der OpenCV Dokumentation. Die Ausführungszeiten folgender Funktionen werden gemessen:

* calcOpticalFlowFarneback()
* FarnebackPolyExp()
* FarnebackUpdateMatricies()
* FarnebackUpdateFlow\_Blur()

Ausgewählt wurden diese Funktionen aus folgenden Gründen. calcOpticalFlowFarneback wird in „denseFlow“ verwendet um den Optischen Fluss zu berechnen. Der Restliche code aus „denseFlow“ beschäftigt sich mit dem einlesen und anzeigen von Videoframes bzw. Flussbildern. FarnebackPolyExp, FarnebackUpdateMatricies und FarnebackUpdateFlow\_Blur werden jeweils von calcOpticalFlowFarneback aufgerufen und werden zur Flussberechnung verwendet.

Es ist zu erwähnen, dass zuvor genannte Funktionen mehr als einmal pro Aufruf von calcOpticalFlowFarneback aufgerufen werden. Die Anzahl der Aufrufe ist abhängig von den gewählten Parametern für calcOpticalFlowFarneback. Unter Verwendung des oben genannten Beispielcodes und der Videosequenz aus (…) werden die Durchschnittlichen Ausführungszeiten für die Berechnung der Flussbilder ermittelt. Diese Zeiten werden wie folgt berechnet:

a : Anzahl der Funktionsaufrufe pro Aufruf von calcOpticalFlowFarneback.

n : Anzahl der Aufrufe von calcOpticalFlowFarneback Aufrufen.

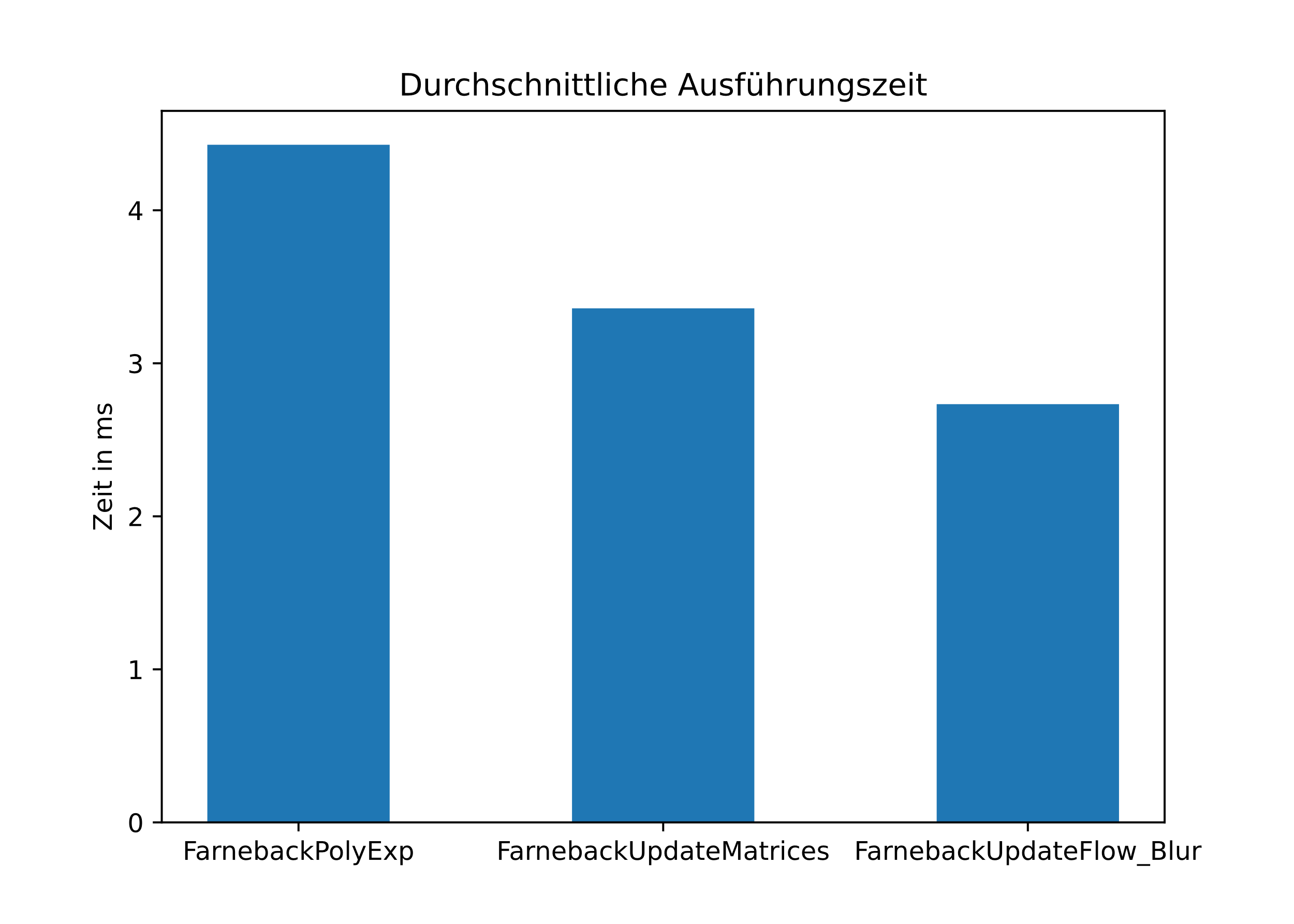
t : Ausführungszeit eines Funktionsaufrufs.

ist das arithmetische Mittel der Ausführungszeit eines Funktionsaufrufs.

Sind die Ausführungszeiten erfasst, sollte der Code der ausgewählten Funktion genauer angesehen werden. Besonders sollte auf Schleifen geachtet werden, welche in der Regel einfach zu parallelisieren sind. Schleifen welche über die Dimensionen des Input Videoframes iterieren sind Anzeichen für mögliche Datenparallelität.

Analyse der Messergebnisse

Messungen werden auf einer Intel(R) Core(TM) i7-9850H CPU abgenommen. Jeder Input Videoframe besitzt eine Dimension von 768x576 Pixeln. Alle Messergebnisse stammen aus einer einzigen Ausführung von „denseFlow“. Die Messergebnisse der durchschnittlichen Ausführungszeiten werden durch folgendes Balkendiagramm dargestellt:

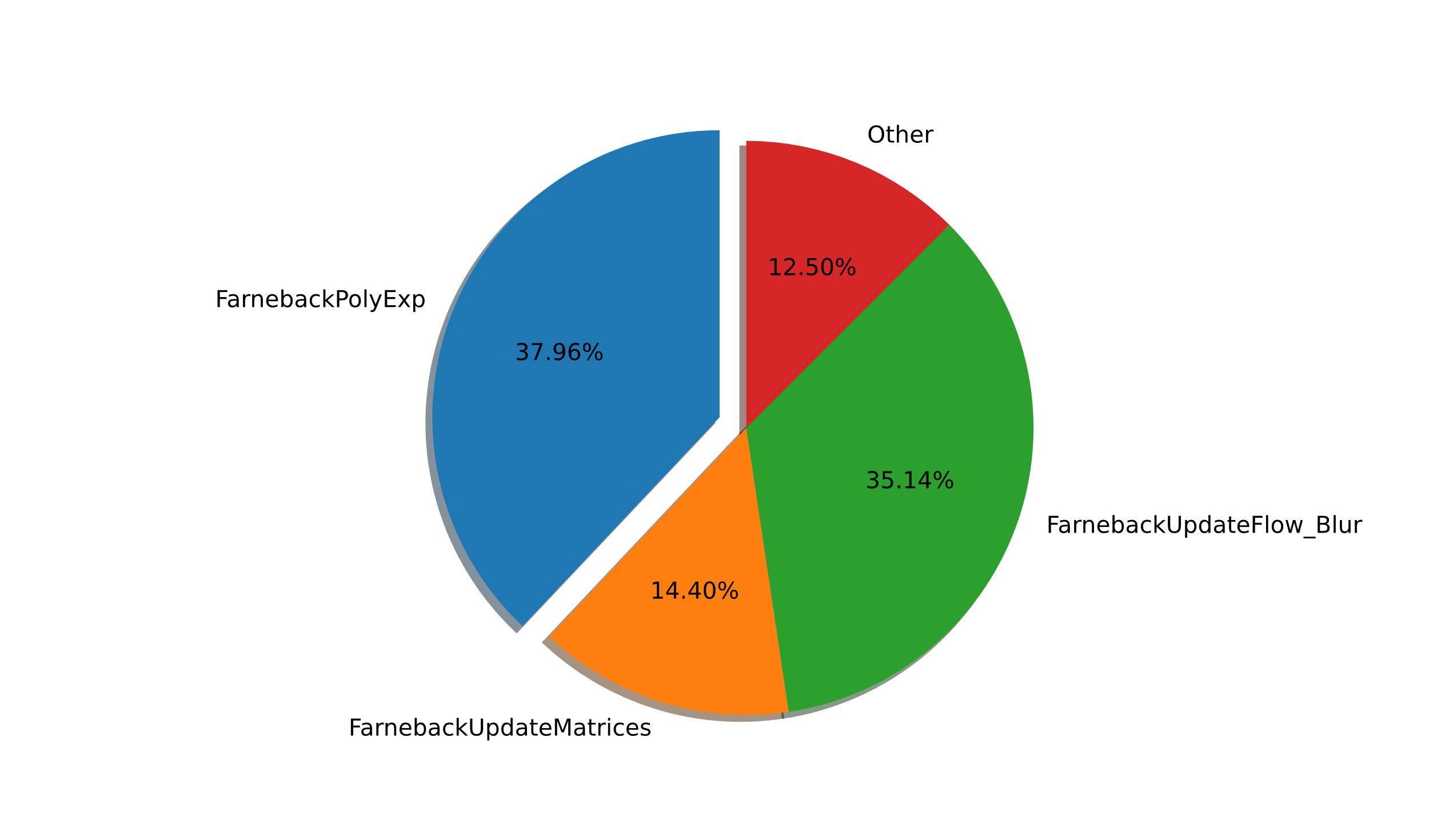
Pro Frame -> kurz erklären, was gemessen wird mit Bildbeschreibungstext -> Profiler oder selber gemessen ?

Die genauen berechneten Ausführungszeiten der Funktionen sind in folgender Tabelle festgehalten:

|  |  |
| --- | --- |
| Funktion | berechnete Ausführungszeit |
| FarnebackPolyExp | 4,4280076923076880 ms |
| FarnebackUpdateMatrices | 3,3591379598662217 ms |
| FarnebackUpdateFlow\_Blur | 2,7328794593088075 ms |

Auf den ersten Blick ist FarnebackPolyExp die Funktion mit der längsten durchschnittlichen Ausführungszeit. Dennoch sollte auch in Betracht gezogen werden, dass die Funktionen mehr als einmal von calcOpticalFlowFarneback aufgerufen werden. Deshalb wird neben den durchschnittlichen Ausführungszeiten auch die Verteilung der Ausführungszeit von calcOpticalFlowFarneback auf die einzelnen Funktionen betrachtet.

Die Verteilung der Ausführungszeit wird in folgendem Diagramm dargestellt:



Other bezeichnet hier Code welcher von calcOpticalFlowFarneback direkt ausgeführt wird und nicht in eine andere Funktion ausgelagert ist. Zu bemerken ist, dass FarnebackUpdateFlow\_Blur einen signifikanten Anteil der Ausführungszeit von calcOpticalFlowFarneback zu verschulden hat. Obwohl FarnebackUpdateFlow\_Blur die niedrigste durchschnittliche Ausführungszeit vorzeigt, hat sie einen ähnlich großen Anteil der Ausführungszeit von calcOpticalFlowFarneback wie FarnebackPolyExp. Dies kann begründet werden durch die unterschiedliche Anzahl der jeweiligen Funktionsaufrufe innerhalb eines Aufrufs von calcOpticalFlowFarneback. Mit den in der Messung verwendeten Parametern von calcOpticalFlowFarneback wird FarnebackPolyExp acht male, FarnebackUpdateMatrices vier male und FarnebackUpdateFlow\_Blur zwölf male aufgerufen.

Auch nach mehreren Ausführungen von „densFlow“ sind keine nennenswerten Änderungen der Messergebnisse zu beobachten. Daraus lässt sich schließen, dass im Durchschnitt die Funktion FarnebackPolyExp die meiste Zeit für ihre Ausführung benötigt.

Aufgrund der erfassten Gegebenheiten wird die Funktion FarnebackPolyExp auf mögliche Parallelisierung geprüft. Hierbei soll besonders Wert auf Datenparallelität gelegt werden.

Als erstes sollen verwendete Schleifen analysiert werden, da diese in der Regel am meisten Zeit in Anspruch nehmen.

Zeigen wie schleifen angeordnet sind – erklären, dass schon optimiert ist -> Farnebacks zitat zur seprable Convolutions

FarnebackPolyExp beinhaltet insgesamt sieben Schleifen, welche wie folgt verschachtelt sind:



Die Schleife in Zeile 132 iteriert über die Höhe des übergebenen Frames. Innerhalb dieser Schleife wird dann mehrmals über die Breite des Frames iteriert. Hier in Zeile 138, Zeile 150 und Zeile 169 zu sehen. Außerdem sind Schleifen vorhanden, die in Bezug zur Variable n ausgeführt werden. Die Variable n beschreibt hier den Radius des Neighbourhoods welches verwendet wird und liegt in realistischen Anwendungen bei Ganzzahlwerten zwischen fünf und sieben. Wie in Kapitel B/III angesprochen berechnet FarnebackPolyExp die Koeffizienten des Polynoms mithilfe eines Neighbourhoods. FarnebackPolyExp ist soweit optimiert, dass das iterieren über den kompletten Inputframe in einen horizontalen und einen vertikalen Teil aufgeteilt wird. Gunnar Farnebäck beschreibt in (…) wie solche Rechnungen aufgeteilt werden können um (…) zu sparen (Farnebäck, 2003). Dies ist auch bei FarnebackPolyExp der Fall. Aufgrund dieser Gegebenheiten kann die Schleifenstruktur von FarnebackPolyExp so zusammengefasst werden, dass die Schleifen im Grunde über den ganzen Inputframe iterieren und Berechnungen mit dem Neigbourhood um einen Pixelwert herum machen. Dies deutet stark auf Datenparallelität hin.

## Lösungsansätze / Probleme

Beschreibung der kritischen funktion -> was macht sie kritisch ->

Die ausgewählte Funktion FarnebackPolyExp

Erster Lösungsansatz ->schleifen übersetzen in STL befehle

Im Ersten Lösungsansatz soll die Reihenfolge der Operationen beibehalten werden um eine mögliche Einführung von Fehlern in der Berechnung zu verhindern. Außerdem soll der Grundgedanke der seperablen Convolution beibehalten werden. Da es sich bei den Schleifen von FarnebackPolyExp um einfache for-Schleifen handelt, können diese in Befehle der Allgorithms Library übersetzt werden. Diese können dann unter Verwendung von Execution Policies parallelisiert werden. Vor allem Befehle wie „std::for\_each“ oder „std::transform“ sind dafür geeignet.

Zweiter Ansatz -> Berechnung in pro Pixel umwandeln. -> dann pro pixel in STL.

Der zweite Lösungsansatz greift mehr in die Struktur des originalen Codes ein. Die originale Implementierung von FarnebackPolExp berechnet die Polynomkoeffizienten Bildzeile für Bildzeile und nicht Pixel für Pixel. Da es sich um eine Implementierung von OpenCV handelt, werden viele OpenCV eigene Klassen verwendet. Dies könnte ein Grund für die Aufteilung in Bildzeilen sein. Um die Möglichkeiten einer GPU auszuschöpfen, soll in diesem Ansatz die Berechnung Pixel für Pixel von statten gehen. In dieser Weise wird dann auch von der Datenparallelität gebrauch gemacht.

Evaluierung der Ansätze ->

erster ansatz ineffiezient -> orginal code ist optimiert und komplex.-> kleine schleifen zu parallelisieren macht keinen Sinn.  
zweiter ansatz -> pro pixel ist in verbindung mit datenparallelität gut geeignet um parallisiert zu werden. -> aufwandt code umzuschreiben,

Evaluierung der Ansätze

Erster Ansatzt rumprobiert -> Laufzeiten war schlecht

Ein Hauptproblem des ersten Ansatzes ist die komplexität des orginalen Codes. Wird der originale Code in Befehle der Allgorihms Library übersetz, kann es zu Performance-Einbußen kommen. Für jeden Befehl der Allgorithms Library, der mit einer Execution Policy versehen ist, wird ein Kernel gestartet und auf der GPU ausgeführt. Wenn für jede Zeile des Bildes ein neuer Kernel aufgerufen wird besteht die Möglichkeit, dass mehr Zeit gebraucht wird um die Kernel zu starten als sie auszuführen. Diese Möglichkeit entsteht durch das Synchronisieren der Daten zwischen den Kernelaufrufen. Deshalb ist es sinnvoller nur einen einzigen Kernelaufruf zu haben, welcher jeden Pixel des Bildes verwendet. Der zweite Lösungsansatz macht genau dies, indem nur ein Algorithms Library Befehl aufgerufen wird. Da es Ziel dieses Ansatzes ist die Berechnung für jeden Pixel einzeln zu machen, wird nur ein Algorithms Library befehl benötigt. Da der originale Code signifikant geändert werden muss, wird der zweite Lösungsansatz einen höheren Aufwand bedeuten.

!! Wo kommt das verständnis der Funktion unter? Versuchsaufbau oder durchführung

## Ausgewählter Ansatz und Durchführung

Ausgewählt wird der zweite Lösungsansatz, den originalen Code in eine Pixel für Pixel Berechnung umzuschreiben und zu parallelisieren. Hierfür ist folgende Vorgehensweise geplant. Zuerst soll der originale Code in eine Pixel für Pixel Berechnung übersetzt werden. Ist dies gelungen, wird mithilfe der von Algorithms Library bereitgestellten Befehle und Execution Policies parallelisiert. Danach soll der parallelisierte Code mit dem nvc++ Compiler der Nvidea HPC SDK compiliert und gebaut werden, um ihn auf der GPU auszuführen.

Um die Berechnung in eine Pixel für Pixel Berechnung umzuschreiben, wird im Detail analysiert, wie der originale Code die Polynomfaktoren berechnet. Sämtliche Berechnung findet innerhalb der Schleife in Zeile 132 statt. Diese Schleife nimmt die variable y = 0 als Startwert und inkrementiert sie bis zur Höhe des Inputframes. In Zeile 135 wird die variable y dann verwendet um Pointer auf korrespondierenden Zeilen des Inputframes zu erhalten. Der Inputframe ist in der Matrix „src“, welche als Parameter an die Funktion übergeben wurde, gespeichert. In Zeile 136 wird ein Pointer auf die Zeile einer weiteren Matrix „dst“ gespeichert. Auch diese wurde als Parameter übergeben und dient später zur Speicherung der Polynomfaktoren.



Die Schleife in Zeile 138 nimmt die variable x = 0 als Startwert und inkrementiert sie bis zur Breite des Inputframes. Genutzt wird sie zum befüllen eines Buffers „row“ mit Initialwerten. Es ist zu beobachten, dass der Buffer für jeden x-Wert drei Werte speichert.



Direkt im Anschluss folgt in Zeile 143 eine Schleife, welche mit k = 1 startet und bis zur variable n inkrementiert. Innerhalb dieser Schleife befindet sich in Zeile 149 eine weiter Schleife, die wie in Zeile 138 zuvor über Werte gleich der Breite des Inputframes iteriert. In den Zeilen 146 und 147 werden jeweils Pointer zu Inputframezeilen über und unter der anfangs in Zeile 135 genannten Zeile akquiriert. Hierbei wird der Abstand durch die variable k festgelegt. Dies deutet auf die Berechnung innerhalb eines Neighbourhoods hin. Die eben erwähnte Schleife in Zeile 149 berechnet dann Werte mithilfe der zwei in Zeilen 146 und 147 erhaltenen Inputframezeilen und addiert diese jeweils zu den bereits vorhandenen Werten im „row“ Buffer. Auch hier werden wieder drei Werte für jeden x-Wert berechnet.



Wichtig festzuhalten ist, dass bis zu diesem Punkt Werte aus einem vertikalen Neigbourhood in „row“ zusammengefügt wurden. Außerdem werden die Ränder des Inputframes bei der Akquisition der Inputframezeilen im Neigbourhood ebenfalls beachtet. „std::max“ und „std::min“ sorgen dafür, dass falls das Neighborhood auf eine Inputframezeile außerhalb des Inputframes zugreifen möchte, die Randzeile des Inputframes stattdessen genommen wird. Somit wird die Randzeile effektiv kopiert und imaginär an den Rand des Inputframes angehängt. Zusammengefasst wurde in vertikaler Richtung das Neigbourhood für jeden x-Wert der Inputframezeile bearbeitet.

Um für die Randbehandlung die gleiche Funktionalität wie bisher bieten zu können, werden mit der Schleife in Zeile 163 die ersten drei Werte am Anfang des Buffers n male an den Anfang angefügt und die letzten drei Werte des Buffer n male an das Ende angefügt.



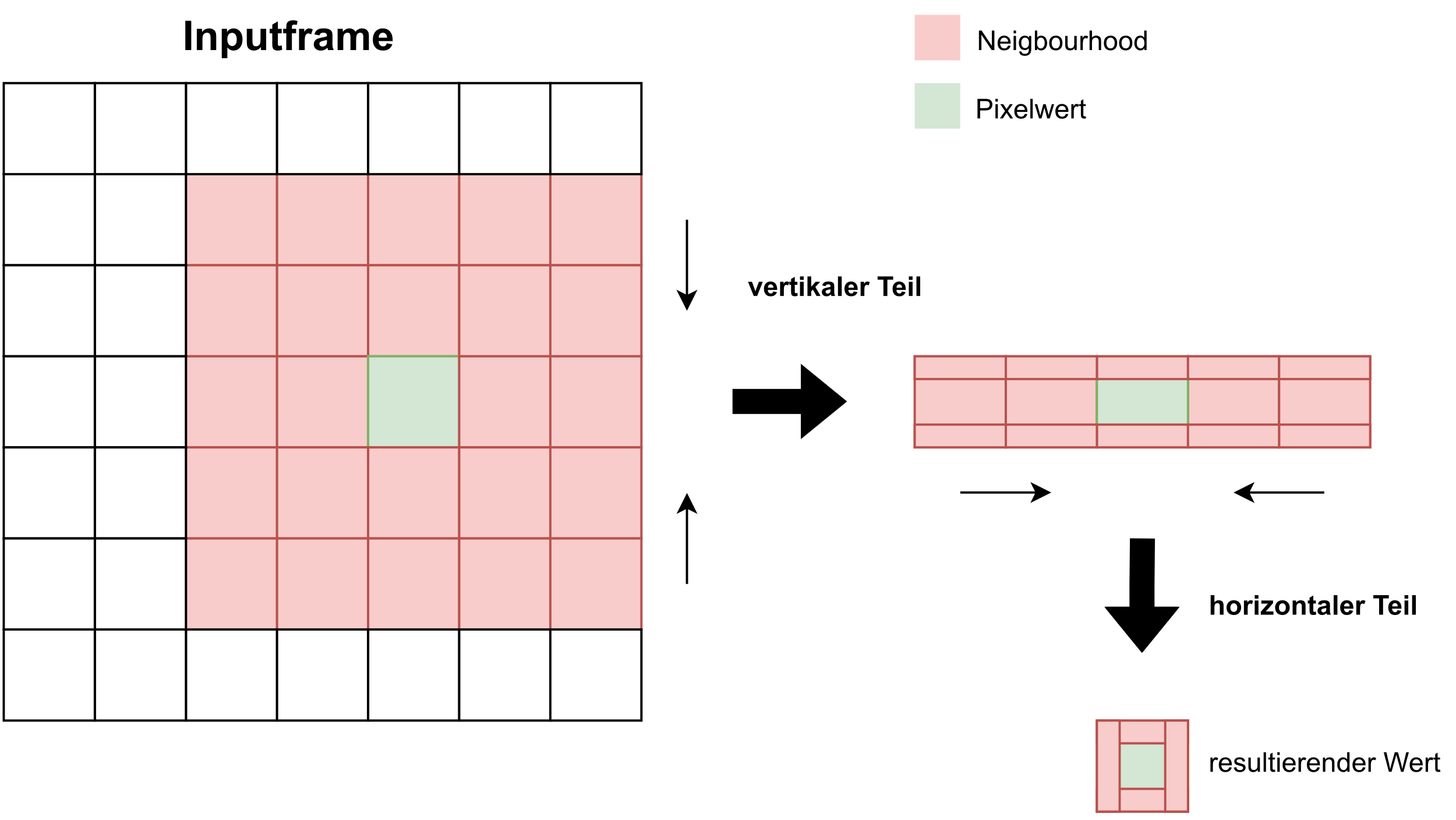
Es ist festzuhalten, dass die Randbehandlung sowohl in vertikaler als auch in horizontaler Richtung durch kopieren der Randwerte realisiert ist.

Da der vertikale Teil der Berechnung für eine Inputframezeile abgearbeitet ist, muss nun der horizontale Teil folgen. Die Schleife in Zeile 168 ist dafür zuständig und initialisiert in Zeile 172 und Zeile 173 variablen für die zu berechnenden Polynomfaktoren. Die Schleife in Zeile 175 wird verwendet um das horizontale Neighbourhood jedes x-Wertes zu bearbeiten. Die aus diesem Neigbourhood berechneten Werte werden dann in der jeweiligen Faktorvariable gespeichert. Nachdem das Neigbourhood für einen x-Wert abgearbeitet ist werden die endgültigen Polynomfaktoren berechnet und in der Matrixzeile „drow“ der 5-Chanel Matrix „dst“ gespeichert. Dies geschieht in Zeile 187 – 191. Damit ist dann eine Iteration der ersten Schleife aus Zeile 132 beendet.



Zusammenfassend wird im originalen Code der Inputframe Zeile für Zeile eingelesen. Dann werden über ein Neigbourhood für jeden Pixel der Inputframezeile drei Werte berechnet. Mit Hilfe dieser drei Werte werden dann die Polynomfaktoren berechnet und in der Zielmatrix gespeichert.

Das zuvor beschriebene verhalten wird nun in eine Pixel für Pixel Berechnung übersetzt. Der vollständige Code ist in Anhang A zu finden. Die Eigenschaft der Aufteilung der Berechnung in einen vertikalen und horizontalen Teil soll dabei beibehalten werden. Das Neibourhood soll in folgendem Schema bearbeitet werden:



Zuerst wird das Neigbourhood in vertikaler Richtung in eine Zeile zusammengefügt. Danach wird Diese Zeile in horizontaler Richtung zusammengefügt

Um über den kompletten Inputframe iterieren zu können werden zwei verschachtelte Schleifen benötigt. Die Variablen „height“ und „width“ entsprechen hier der Höhe und der Breite des Inputframes. Außerdem wird ein Buffer benötigt, welcher dreimal so groß ist wie das Neigbourhood, um alle benötigten Werte zwischenspeichern zu können.



Danach wird der vertikale Teil des Neigbourhoods um einen Pixel bearbeitet. Dies geschieht durch zwei verschachtelte Schleifen. Die äußere Schleife iteriert über die Breite des Neighbourhoods, welches durch den Term 2\*n+1 beschrieben werden kann. Außerdem wird eine neue variable „neighX“ eingeführt, welche als x-Koordinate für das Neighbourhood dient. Die innere Schleife wird verwendet um Pixel in vertikaler Richtung innerhalb des Neigbourhoods abzuarbeiten. Zuerst werden neue variablen „neighY0“ und „neightY1“ eingeführt um die y-Koordinaten im Neighbourhood zu bestimmen. Dann werden für jede x-Koordinate im Neighbourhood drei Werte berechnet und im Buffer gespeichert. Wie auch im originalen Code wird die Randbehandlung durch „std::min“ und „std::max“ durchgeführt. Falls das Neighbourhood auf Koordinaten außerhalb des Inputframes zugreifen würde, wird stattdessen die auf die Werte des Randes zugegriffen.



Nachdem das Neigbourhood eines Pixels im vertikalen Teil auf eine Zeile des Neigbourhoods zusammengefasst wurde, wird nun der horizontale Teil behandelt. Hierbei werden die Variablen „b1“ – „b6“ verwendet um nach dem Zusammenfassen der Neigbourhoodzeile direkt die Polynomfaktoren berechnen zu können.



Die in den Variablen „b1“ – „b6“ gespeicherten Werte werden dann, wie im originalen Code zuvor, dazu verwendet die Polynomfaktoren für einen Pixel zu berechnen und in der Matrix „dst“ zu speichern.

Parallelisierung der per Pixel Berechnung.

Die Funktion FarnebackPolyExp ist nun auch als per Pixel Berechnung implementiert. Unter Verwendung der Algorithms Library kann nun mit geringem Aufwand paralleler Code entstehen. Hierfür werden die zwei äußersten Schleifen, welche über den Inputframe iterieren, durch einen „std::foreach“ Aufruf ersetzt. Da für jedes Pixelelement innerhalb der Schleifen mehr als nur eine Operation benötigt wird, sind „std::transform“ und „std::transform\_reduce“ nicht geeignet. Die übergebene Execution Policy signalisiert, dass der Aufruf parallelisiert werden darf.



Innerhalb der Lambda-Funktion befindet sich fast derselbe Code wie innerhalb der ersetzen zweifach verschachtelten Schleifen. Da nun bei jeder Iteration nur ein Pixel des Inputframes zur Verfügung steht und keine Informationen zur Position des Pixels, muss der Inhalt der Lambda-Funktion angepasst werden. Um auf Pixel des Neighbourhoods zuzugreifen werden x- und y-Koordinaten benötigt. Diese werden berechnet indem der Index des Pixels mit Hilfe von Pointer-Arithmetik berechnet wird. Die Speicheradresse auf die der Pointer des Inputframes zeigt wird subtrahiert von der Speicheradresse des aktuellen Pixels. Mit dem dadurch berechneten Index wird x- und y-Koordinate berechnet.



Diese Berechnung ist nur dann möglich, wenn der Inputframe konsekutiv im Speicher liegt.

Nachdem der parallele Code implementiert ist, kann dieser für die Ausführung auf einer GPU kompiliert und getestet werden. Hierfür wird der NVC++ Compiler der NVIDIA HPC SDK verwendet.

Versuche den Code mit Hilfe des NVC++ Compilers zu kompilieren ergeben mehrere Probleme. Diese können zurückgeführt werden auf die Verwendung von OpenCV Klassen. Der NVC++ Compiler kann die Definitionen der OpenCV Klassen nicht sehen, da die Klassen über die OpenCV Library inkludiert werden. Somit ist die Nutzung von CUDA Unified Memory nicht möglich und der Code kann aufgrund dessen nicht erfolgreich ausgeführt werden.

^^^^Probleme wegen der Nutzung von Opencv – klassen -> bessere beschreibung des Problems

abstrahierung des Problems ->

Die zuvor Beschriebenen Probleme erfordern umfassende Änderung der originalen Implementierung von OpenCV um sie zu lösen. Da die Parallelisierung mit der Parallel STL und die dadurch resultierende Performance im Vordergrund dieser Arbeit steht, wird die Funktion FarnebackPolyExp von der OpenCV Implementierung gekapselt. Hierfür wird ein neues Programm erstellt, welches zur Berechnung Platzhalterdaten verwendet. Diese Platzhalterdaten sollen einem Videoframe ähneln. Auch die ursprüngliche Hauptfunktionalität soll beibehalten werden.

->Unterschiede zu OpenCV code erläutern -> erste messungen sind ernüchternd -> Optimierungen möglich, da CPU und GPU hardware sich unterscheiden, besser auf GPU hardware optimieren -> optimierung vorstellen -> Messung vergleichen

Um die in „cv::Mat“ Klassen gespeicherten Videoframedaten zu ersetzen, geniert das neu erstellte Programm „polyExp-stl“ Platzhalterdaten und speichert diese in einem eindimensionalen „std::vector“ Array. „std::vector“ wird verwendet, da der Speicher für diesen Vektor auf dem CPU Heap allokiert wird. Dies ist Voraussetzung um die Daten mit CUDA Unified Memory nutzen zu können. Die berechneten Polynomkoeffizienten werden ebenfalls ein einem „std::vector“ gespeichert. Die zuvor erarbeitete Per-Pixel-Implementierung der FarnebackPolyExp Funktion wird von „polyExp-stl“ aufgerufen. Anstatt der Videoframedaten werden nun die Platzhalterdaten als Parameter übergeben. (ig values static)

Erstmals kann nun die neue FarnebackPolyExp Implementierung auf der GPU ausgeführt werden. Erste Performance Messungen ergaben folgendes Ergebnis:

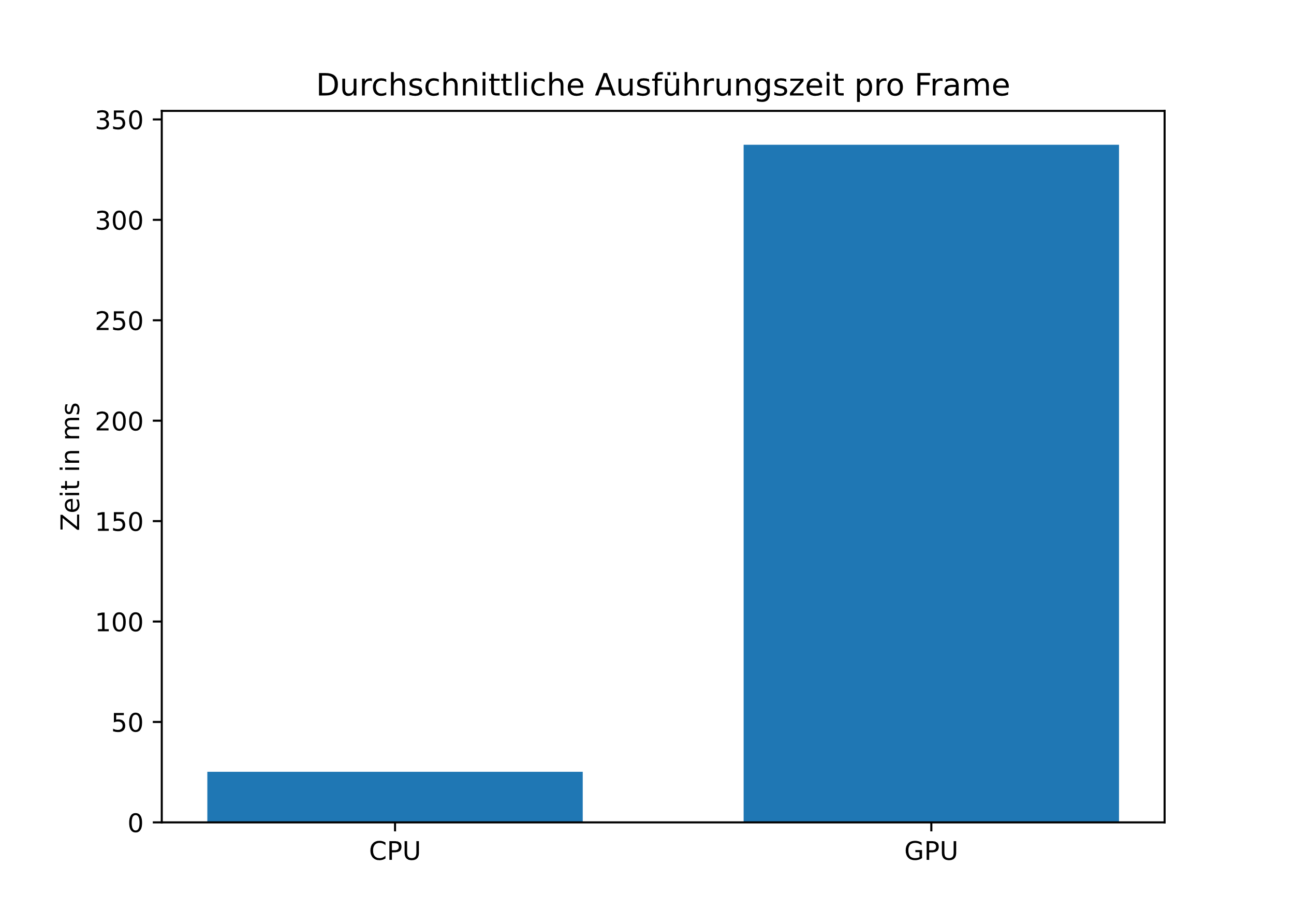


Abbildung (): Performance gemessen auf einer Nvidia Quadro P2200 GPU. Das Balkendiagramm zeigt die Durchschnittliche Ausführungszeit von FarnebackPolyExp pro Frame.

Die erhaltenen Messergebnisse zeigen einen deutlichen Unterschied in der Ausführungszeit zwischen CPU und GPU. Um die unerwartet lange Ausführungszeit begründen zu können wird mit Hilfe von einem Profiling-Tool der Nvidia HPC SDK die Ausführung genauer analysiert.

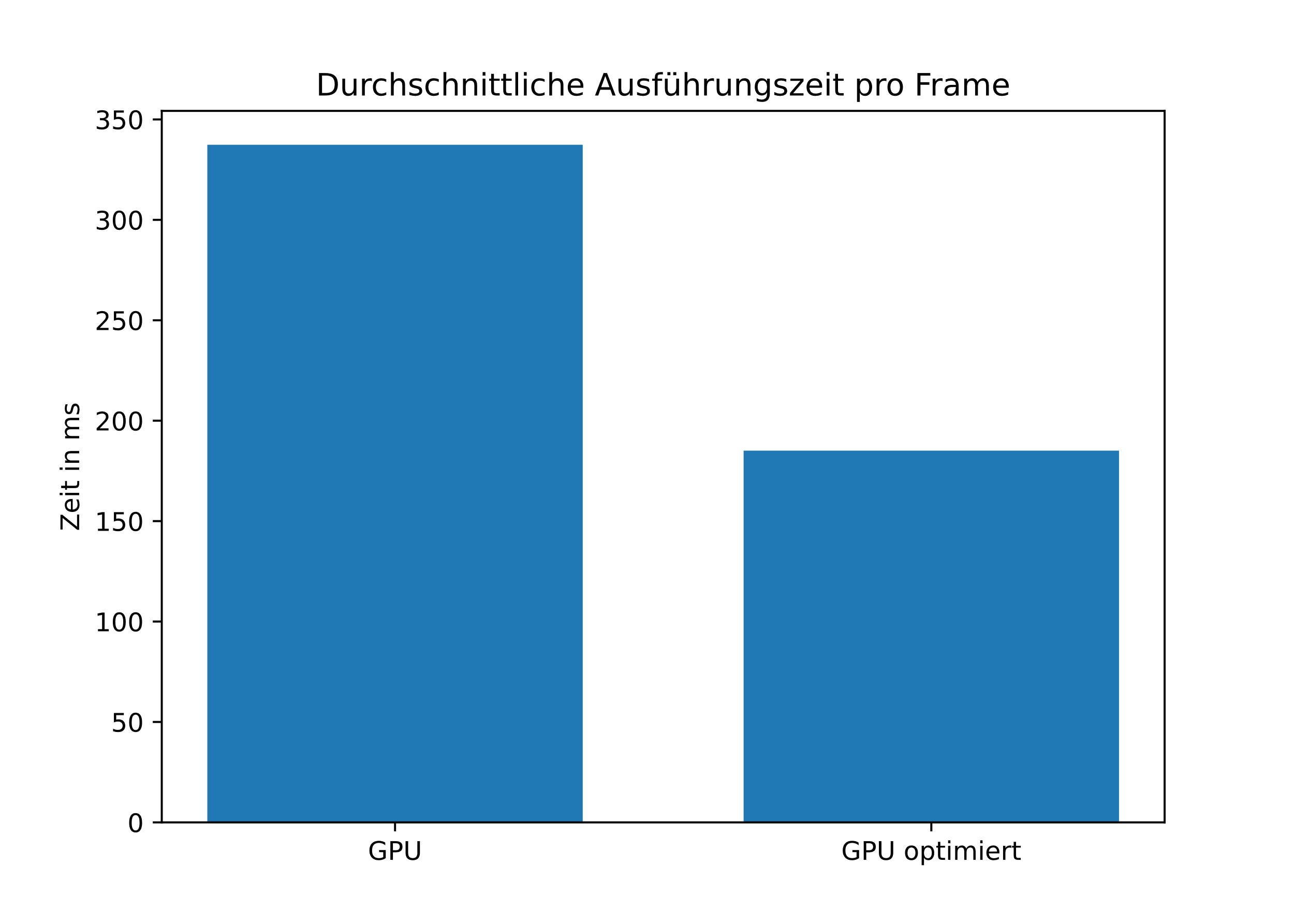
Profilingergebnisse -> was passiert -> falls profiling nicht geht -> Vermutung auf ineffiziente speicherzugriffe

Messergebnisse darstellen -> auf welcher hardware wurde getestet -> Vermutung warum ergebnisse so sind-> Profiling ergebnisse maybe um zu sehen wo optimiert werden kann.

Optimierung vorstellen -> Messergebnisse darstellen -> warum ist es jetzt schneller.

Als Optimierungsschritt soll die Anzahl der verwendeten Schleifen innerhalb des Kernels reduziert werden. Um die Berechnung der Polynomkoeffizienten effizienter zu machen, wird auf den Zwischenspeicher für die horizontale Abarbeitung des Neigbourhoods verzichtet. Anstatt alle Werte des vertikalen Neighbourhoods zwischenzuspeichern wird jeder erhaltene Wert mit dem richtigen Faktor multipliziert und je nach Koeffizienten und Position innerhalb des Neigbourhoods auf die Variablen der Polynomkoeffizienten addiert oder subtrahiert.

* Code



# Ergebnisse

(Validierung dass mein code immernoch das gleiche macht wie im orginal?) -> mittlere summierte Absolute differnz berechnen zwischen den Schritten von original zu anderen.

## Validierung der Ergebnisse und Korrektheit der Implementierung

Um die Korrektheit der FarnebackPolyExp Implementierungen gewärleisten zu können wird die mittlere summierte absolute Differenz der Polynomkoeffizienten aus der Orginal-Implementierung und den Per-Pixel-Implementierungen wie folgt berechnet:

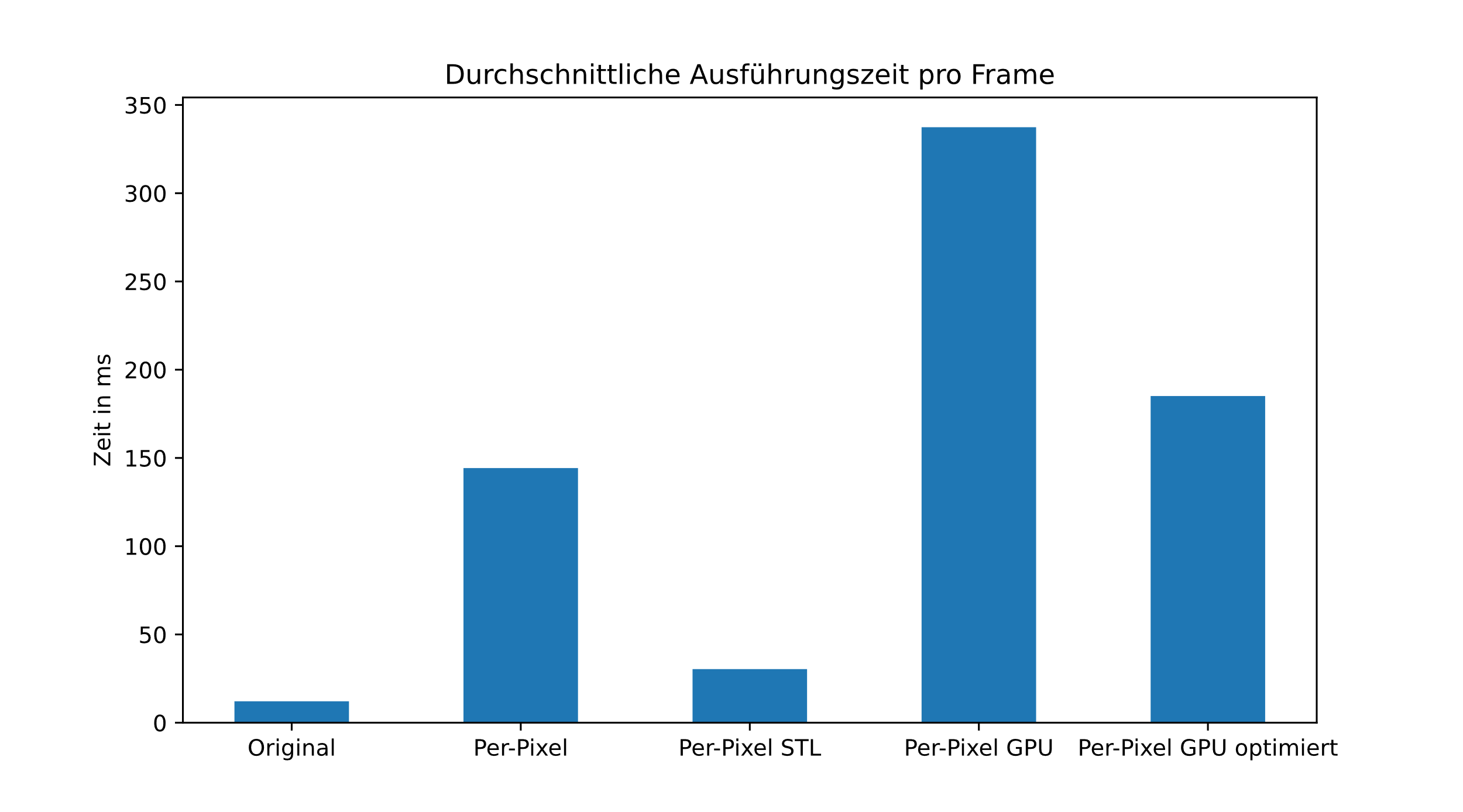
Mathematische formel ?

Code zur berechnung

Darstellung der messergebnisse

## Darstellung der Ergebnisse

Sowohl die Original Implementierung, als auch die Per-Pixel Implementierung wurden auf einem einzigen CPU-Kern ausgeführt. Durch die Einführung eines Parallel STL Algorithmus mit Execution Policy wurde die Per-Pixel Implementierung dann auf Mehreren CPU-Kernen ausgeführt. Die Per-Pixel Implementierung mit Parallel STL wurde mit Hilfe des Nvidia NVC++ Compilers für die Ausführung auf einer GPU kompiliert. Um die Ausführungszeit zu reduzieren, wurde die Implementierung optimiert. Folgendes Diagramm zeigt die verschiedenen Implementierungen:



|  |  |
| --- | --- |
| Implementierung | Durchschnittliche Ausführungszeit |
| Original | 12,122794314381263 ms |
| Per-Pixel | 144,2569431438129 ms |
| Per-Pixel STL | 30,359591973244154 ms |
| Per-Pixel STL auf GPU | 337,3794033333336 ms |
| Per-Pixel STL auf GPU optimiert | 185,07826333333313 ms |

Die mit einer GPU parallelisierten Implementierungen haben beide eine höhere durschnittliche Ausführungszeit als die nicht parallelisierten Implementierungen. Dennoch kann die Ausführungszeit noch verbessert werden. Dies ist zu sehen, wenn die durchnittliche Ausführungszeit der Per-Pixel STL auf GPU Implementierung und die durchnittliche Ausführungszeit der optimierten Per-Pixel STL auf GPU Implementierung verglichen wird.

Übersicht über die ergebnisse -> Bewertung

# Zusammenfassung

References

Algorithms library - cppreference.com. https://​en.cppreference.com​/​w/​cpp/​algorithm (accessed December 7, 2021).

Boyd, C. Data-parallel computing. In *ACM SIGGRAPH 2008 classes on - SIGGRAPH '08*; Unknown, Ed.; ACM Press: New York, New York, USA, 2008; p 1. DOI: 10.1145/1401132.1401150.

Brodtkorb, A. R.; Hagen, T. R.; Sætra, M. L. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing* [Online] **2013,** *73* (1), 4–13. https://​www.researchgate.net​/​publication/​257252061\_Graphics\_processing\_unit\_GPU\_programming\_strategies\_and\_trends\_in\_GPU\_computing.

C++ Parallel Algorithms Version 22.2 for ARM, OpenPower, x86. https://​docs.nvidia.com​/​hpc-sdk/​compilers/​c++-parallel-algorithms/​index.html (accessed February 27, 2022).

Farnebäck, G. Two-Frame Motion Estimation Based on Polynomial Expansion. In ; Springer, Berlin, Heidelberg, 2003; pp 363–370. DOI: 10.1007/3-540-45103-X\_50.

NVIDIA Developer. HPC SDK | NVIDIA. https://​developer.nvidia.com​/​hpc-sdk (accessed February 27, 2022).

NVIDIA Developer Blog. Maximizing Unified Memory Performance in CUDA | NVIDIA Developer Blog. https://​developer.nvidia.com​/​blog/​maximizing-unified-memory-performance-cuda/​ (accessed February 14, 2022).

NVIDIA Developer Blog. Unified Memory for CUDA Beginners | NVIDIA Developer Blog. https://​developer.nvidia.com​/​blog/​unified-memory-cuda-beginners/​ (accessed February 14, 2022).

Programming Guide : CUDA Toolkit Documentation. https://​docs.nvidia.com​/​cuda/​cuda-c-programming-guide/​index.html (accessed February 1, 2022).

std:execution:sequenced\_policy, std:execution:parallel\_policy, std:execution:parallel\_unsequenced\_policy, std:execution:unsequenced\_policy - cppreference.com. https://​en.cppreference.com​/​w/​cpp/​algorithm/​execution\_policy\_tag\_t (accessed February 7, 2022).

std:for\_each - cppreference.com. https://​en.cppreference.com​/​w/​cpp/​algorithm/​for\_each (accessed February 9, 2022).

std:for\_each\_n - cppreference.com. https://​en.cppreference.com​/​w/​cpp/​algorithm/​for\_each\_n (accessed February 9, 2022).

std:transform - cppreference.com. https://​en.cppreference.com​/​w/​cpp/​algorithm/​transform (accessed February 9, 2022).

std:transform\_reduce - cppreference.com. https://​en.cppreference.com​/​w/​cpp/​algorithm/​transform\_reduce (accessed February 11, 2022).

Understanding GPU caches – RasterGrid. https://​www.rastergrid.com​/​blog/​gpu-tech/​2021/​01/​understanding-gpu-caches/​ (accessed February 14, 2022).

Voss, M.; Asenjo, R.; Reinders, J. TBB and the Parallel Algorithms of the C++ Standard Template Library. In *Pro TBB*; Voss, M., Asenjo, R., Reinders, J., Eds.; Apress: Berkeley, CA, 2019; pp 109–136. DOI: 10.1007/978-1-4842-4398-5\_4.