



**THU**  
Technische  
Hochschule Ulm  
University of  
Applied Sciences

Bachelorarbeit

# Modernes paralleles C++ für Hardwarebeschleuniger

Technische Hochschule Ulm  
Fakultät Informatik  
Studiengang Informatik

vorgelegt von  
Erwin Kenner

April 2022

Erstgutachter: Prof. Dr. Georg Schied  
Zweitgutachter: Prof. Dr. Philipp Graf

Firmenbetreuer:  
Christian Aßfalg,  
Continental ADC Automotive Distance  
Control Systems GmbH

## Eigenständigkeitserklärung

Diese Abschlussarbeit wurde von mir selbständig verfasst. Es wurden nur die angegebenen Quellen und Hilfsmittel verwendet. Alle wörtlichen und sinngemäßen Zitate sind in dieser Arbeit als solche kenntlich gemacht.

Biberach an der Riß, den 08.04.2022

Erwin Kenner

## Abstract

In Automobilen werden Hardwarebeschleuniger verwendet, um Algorithmen zu beschleunigen. Damit neue Software für Fahrzeuge zugelassen werden kann, müssen diese Fahrzeuge Testfahrten absolvieren. Diese Testfahrten können auch simuliert werden. Ziel einer solchen Simulation ist es Algorithmen mit GPUs und nicht mit externen Hardwarebeschleunigern zu beschleunigen. Mithilfe der in C++17 eingeführten Parallel STL Algorithmen ist es möglich die Parallelisierung dem Compiler zu überlassen. Das Nvidia Toolkit unterstützt mit seinem eigenen C++ Compiler die Parallel STL Algorithmen und macht ihre Ausführung auf GPUs möglich. Anhand eines Beispiels aus der Automobilbranche wird in dieser Arbeit dargestellt, welche neuen Möglichkeiten sich hinsichtlich der Laufzeiten und Portabilität der Parallel STL Algorithmen in Verbindung mit dem Nvidia Toolkit ergeben. Die erhaltenen Ergebnisse zeigen Verschlechterungen dieser Laufzeiten, die durch Optimierung des Algorithmus reduziert werden können. Dennoch ist die parallelisierte Anwendung auf der GPU langsamer als auf der CPU. Schlussfolgernd zeigt sich, dass die Möglichkeiten zur Algorithmen Beschleunigung durch Parallel STL in Verbindung mit dem Nvidia Toolkit begrenzt sind.

# Inhaltsverzeichnis

1. Einführung	1
2. Grundlagen	2
2.1. Taskparallelität und Datenparallelität	3
2.2. Parallel STL	3
2.2.1. Standard Algorithmen	3
2.2.2. Execution Policies	4
2.2.3. <code>std::for_each</code> , <code>std::for_each_n</code>	5
2.2.4. <code>std::transform</code>	8
2.2.5. <code>std::transform_reduce</code>	10
2.3. GPU und die Nvidia HPC SDK	12
2.3.1. Graphics Processing Unit	12
2.3.2. CUDA	12
2.3.3. CUDA Unified Memory	14
2.3.4. Nvidia HPC Software Development Kit	16
2.4. Optischer Fluss mit OpenCV	17
3. Versuchsaufbau	18
3.1. Anforderungsanalyse	18
3.2. Lösungsansätze	24
3.3. Ausgewählter Ansatz und Durchführung	25
3.3.1. Detailanalyse der Polynomkoeffizienten Berechnung	25
3.3.2. Übersetzung in Per-Pixel-Berechnung	30
3.3.3. Parallelisierung der Per-Pixel-Berechnung.	33
4. Ergebnisse	38
4.1. Validierung der Ergebnisse und Richtigkeit der Implementierung	38
4.2. Übersicht der Ergebnisse	40
4.3. Ergebnisdiskussion und Erkenntnisse	41
4.3.1. Bewertung der Ergebnisse	41
4.3.2. Gesammelte Erfahrungen und Erkenntnisse	42
5. Zusammenfassung	43
6. Literaturverzeichnis	44
7. Anlagen	47

## 1. Einführung

Software für Anwendungen in der Automobilbranche muss eine Reihe von Tests und Kriterien erfüllen, um für den Straßenverkehr zugelassen zu werden. Eine dieser Kriterien ist, dass ein Testfahrzeug inklusive Software mehrere hunderttausend Kilometer gefahren sein muss. Fahrten dieser Testfahrzeuge können auch simuliert werden. Die in der Simulation zurückgelegten Kilometer werden ebenfalls gewertet. In Fahrzeugen werden Hardwarebeschleuniger verwendet, um die Performance von rechen- oder speicherintensiven Algorithmen zu verbessern. Diese Hardwarebeschleuniger sind speziell für die Art ihrer Aufgabe konstruiert. Allerdings ist die Verwendung von mehreren externen Hardwarebeschleunigern in Simulationen nicht wünschenswert, da die für die Simulation benötigte Hardware nicht portabel ist. Die Hardware ist eine spezifische Lösung, konzipiert für nur ein einzelnes Projekt. Sie wird dementsprechend nutzlos, sobald das Projekt beendet wurde. Um Hardware effizienter nutzen zu können, braucht es daher eine generalisierte Hardware, welche nicht auf eine Art von Problem spezifiziert ist. Generalisierte Hardware ist in den meisten Rechnern verbaut. Eine solche generalisierte Hardware ist beispielsweise die Graphics Processing Unit (GPU). Wenn Simulationen auf Rechenknoten mit eingebauten GPUs ausgeführt werden, können diese zur Beschleunigung derer Algorithmen verwendet werden. Wenn die Simulation der Testfahrten auf generalisierter Hardware läuft, kann sie, nachdem das Projekt beendet wurde, für neue Projekte weiterverwendet werden.

Dennoch können die Algorithmen der in der Simulation verwendeten Software nicht ohne weiteres auf GPUs ausgeführt werden. Der vorher auf den speziellen Hardwarebeschleunigern ausgeführte Code muss für die Ausführung auf GPUs angepasst werden. Dies kann je nach Art des Algorithmus äußerst aufwendig sein. Der C++17 Standard stellte Parallelisierungsmöglichkeiten vor, welche es dem Nutzer ermöglichen die Parallelisierung der Standard Library Algorithmen anzufordern.[1] Nvidia, einer der größten Entwickler von Grafikprozessoren und Chipsätzen, unterstützt diese „Parallel Algorithms“.[2] Mit den C++17 Algorithmen und den von Nvidia entwickelten Compilern soll die Kompilierung für GPUs vereinfacht werden.

Ziel dieser Arbeit ist es, Erfahrungswerte im Umgang mit den in C++17 neu eingeführten „Parallel Algorithms“ zu sammeln.[1] Um diese Erfahrungswerte sammeln zu können, soll mithilfe der C++17 „Parallel Algorithms“ ein im Bereich Automotive verwendeter Algorithmus parallelisiert werden. Der zu parallelisierende Algorithmus ist ein „dense optical flow“ Algorithmus und wird zur Berechnung des optischen Flusses einer Videosequenz verwendet. Der sich ergebende, parallelisierte Algorithmus soll zur weiteren Nutzung auf einer GPU ausführbar gemacht werden. Hierfür wird das Nvidia Toolkit verwendet, welches nötige C++ Compiler für die Generierung von GPU-Code zur Verfügung stellt. Nachdem der Algorithmus auf einer GPU ausführbar gemacht wurde, werden etwaige Performanceeinbrüche ermittelt. Während der Parallelisierung des Algorithmus und der Portierung auf eine GPU, sollen zudem mögliche Limitierungen in der Nutzung von C++17 „Parallel Algorithms“ oder des Nvidia Toolkits erfasst und bewertet werden.

Hierzu werden im ersten Kapitel dieser Arbeit zunächst die Grundlagen dargestellt. Unter anderem werden die Themen der Parallelität, GPUs, Nvidia HPC SDK und der verwendete Algorithmus behandelt. Im nächsten Kapitel werden die Gegebenheiten vor der Parallelisierung des Algorithmus analysiert. Resultierende Lösungsansätze zur Parallelisierung des Algorithmus werden vorgestellt, welche daraufhin bewertet werden, um den optimalen Ansatz auszuwählen. Dieser Ansatz wird im Detail verfolgt, woraufhin aufkommende Probleme diskutiert werden, um eine endgültige Lösung des Ansatzes zu präsentieren. Aufbauend darauf wird die erarbeitete Lösung im folgenden Kapitel validiert, sodass die erhaltenen Ergebnisse dargestellt und bewertet werden können. Anschließend folgt eine Zusammenfassung der erarbeiteten Erkenntnisse, Probleme und erreichten Ergebnisse, woraufhin ein Ausblick auf weitere mögliche Schritte folgt.

## **2. Grundlagen**

In den Grundlagen werden Themen und Begriffe eingeführt, welche für das Lösungskonzept relevant sind. Themen sind hierbei Parallelität, GPUs, das Nvidia High Performance Computing Software Development Kit und der verwendete Algorithmus.

## **2.1. Taskparallelität und Datenparallelität**

Beim parallelen Programmieren gibt es zwei Arten von Parallelität. Die Taskparallelität und die Datenparallelität. Parallelität kann erreicht werden, indem Ausführungsschritte eines Programms voneinander abgekapselt 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. Möglich wird dies durch häufige Aufteilung der Ausführungsschritte in kleinere Unterschritte, um sie besser auf die Anzahl an Threads und Prozessorkerne zu verteilen. Dies kann sich, je nach Natur des Programms, durchaus als schwierig oder unmöglich erweisen.[3]

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. [3]

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

## **2.2. Parallel STL**

### **2.2.1. Standard Algorithmen**

Die C++ Standard Template Library (STL) ist eine Menge von Template Klassen, welche häufig verwendete Programmier- und Datenstrukturen sowie Funktionen zur Verfügung stellt. Sie ist eine Bibliothek von Klassen, Algorithmen und Iteratoren. [4]

Die Algorithmen der C++ STL sind Funktionen, die von der „Algorithms Library“ definiert werden. Diese Funktionen agieren auf einem Bereich von Elementen. Beispiele hierfür sind das Sortieren, Suchen, Zählen oder Modifizieren von Elementen einer Liste oder eines Arrays.[1]

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

Um die Verwendung dieser Algorithmen darzustellen, werden Anwendungsbeispiele aufgezeigt. Diese beinhalten Lambdadausdrücke beziehungsweise Lambdafunktionen. Dies sind anonyme Funktionen welche nicht an einen Bezeichner gebunden sind. Sie eignen sich ideal als Funktionen für die „Parallel STL“ Algorithmen.

### **2.2.2. Execution Policies**

Mit C++17 wurden mehrere Algorithmen der Standard Library mit sogenannten „execution policies“ erweitert. Diese ermöglichen die Parallelisierung von Standard Algorithmen. Viele Algorithmen besitzen seitdem Überladungen, die Objekte dieser „execution policies“ akzeptieren.

Zu den „execution policies“ gehören:

- sequenced\_policy
- parallel\_policy
- parallel\_unsequenced\_policy

[4]

Wird das Objekt „seq“ der sequenced\_policy Klasse als Parameter übergeben, wird die Funktion in richtiger Reihenfolge (sequenziell) ausgeführt. Dennoch kann der Compiler den Algorithmus parallelisieren, wenn dies für das Programm nicht sichtbar ist.[5]



Beim Übergeben des „par“ Objekts der parallel\_policy Klasse wird dem Algorithmus signalisiert, dass dieser die Ausführung auf mehrere Threads aufteilen darf. Die Operationen innerhalb eines Threads werden jedoch sequenziell abgearbeitet.[5]

Wird das „par\_unseq“ Objekt der Klasse parallel\_unseq\_policy übergeben, wird die Ausführung auf mehrere Threads aufgeteilt und vektorisiert.[5]

### 2.2.3. **std::for\_each, std::for\_each\_n**

Der Algorithmus „std::for\_each“ wird verwendet, um eine Funktion auf alle Elemente eines Bereichs anzuwenden. Er besitzt mehrere Interfaces, wobei das für diese Arbeit relevante Interface in Codeausschnitt 2.1 abgebildet ist.

```
template< class ExecutionPolicy, class ForwardIt,
          class UnaryFunction2 >
void for_each( ExecutionPolicy&& policy, ForwardIt first,
              ForwardIt last, UnaryFunction2 f );
```

*Codeausschnitt 2.1, eines der Interfaces von „std::foreach“ mit „execution policy“.*

„std::for\_each“ akzeptiert als ersten Parameter ein Objekt der Klasse „ExecutionPolicy“ und kann somit parallelisiert werden. Weitere akzeptierte Parameter sind Objekte der Klasse „ForwardIt“. Hierbei handelt es sich um Iteratoren, die den Bereich bestimmen, der von „std::for\_each“ behandelt wird, 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. [6]

Neben dem „std::for\_each“ Algorithmus, gibt es noch den dem „std::for\_each“ ähnlichen Algorithmus „std::for\_each\_n“. Das für diese Arbeit wichtige Interface wird in Codeausschnitt 2.2 dargestellt.

```

template< class ExecutionPolicy, class ForwardIt,
          class Size, class UnaryFunction2 >
ForwardIt for_each_n( ExecutionPolicy&& policy,
                     ForwardIt first, Size n,
                     UnaryFunction2 f );

```

*Codeausschnitt 2.2, eines der Interfaces von „std::for\_each\_n“ mit „execution policy“.*

Dieses Interface unterscheidet sich nicht wesentlich 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. [7]

Als Anwendungsbeispiel, welches die Nutzung von „std::for\_each“ verdeutlicht, wird eine Stencil Operation mit „std::for\_each“ implementiert. Stencil Operationen brauchen meist Informationen zu benachbarten Werten in einem zweidimensionalen Array, um Berechnungen durchzuführen oder Arraywerte miteinander zu vergleichen.[8] Werden Stencil Operationen mit einem Algorithmus der Parallel STL umgesetzt, kann das im Folgenden beschriebene Hindernis entstehen. Die verwendete Lambdafunktion besitzt nur eine Referenz auf das benutzte Element, enthält jedoch keinerlei Informationen über die Position des Elements innerhalb des Bereichs. Somit können keine Informationen über benachbarte Elemente erhalten werden. Um dieses Hindernis zu umgehen, muss mithilfe von Pointer Arithmetik der Index des Elements berechnet werden, um benachbarte Elemente zu adressieren.[9]

Die Verwendung einer Stencil Operation und die Umgehung eines solchen Hindernisses wird mithilfe von Codeausschnitt 2.3 erläutert.

```
1.  std::for_each(std::execution::par, field.begin(),
2.      field.end(), [=](auto &e){
3.      size_t i = &e - field_ptr;
4.      size_t x = i % N;
5.      size_t y = i / N;
6.      int sum = 0;
7.
8.      bool on_x_bound = x == 0 || x == N-1;
9.      bool on_y_bound = y == 0 || y == N-1;
10.
11.     if(!on_x_bound && !on_y_bound){
12.         sum = field_ptr[i-1] + field_ptr[i+1] +
13.             field_ptr[i-N] + field_ptr[i+N] +
14.             field_ptr[i-N-1] + field_ptr[i+N-1] +
15.             field_ptr[i-N+1] + field_ptr[i+N+1];
16.     }
17.
18.     if(sum >= 3){
19.         field2_ptr[i] = 1;
20.     } else{
21.         field2_ptr[i] = 0;
22.     }
23. });
```

*Codeausschnitt 2.3, Beispielcode einer Stencil Operation mit Parallel STL Algorithmus.*

Bei diesem Beispiel handelt es sich um eine vereinfachte Implementierung von Conway's "Game of Life".[10]

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

In Zeile 3 des Beispiels wird der Index berechnet. Die Adresse des Bereichs „field“, welche gleichzeitig die Adresse des ersten Elements darstellt, wird von der Adresse des gerade verwendeten Elements „e“ subtrahiert. Als Ergebnis erhält man den Abstand des verwendeten Elements zum ersten Element des Bereichs und somit den Index des verwendeten Elements innerhalb des Bereichs. Die eigentliche Stencil Operation findet in den Zeilen 12 – 15 statt. Hier werden die Werte benachbarter Zellen mithilfe des zuvor berechneten Index aufsummiert.

#### 2.2.4. *std::transform*

Der Algorithmus „std::transform“ besitzt zwei Varianten. Die Erste appliziert eine einstellige Verknüpfung „unary\_op“ auf einen Bereich festgelegt durch die Iteratoren „first1“ und „last1“. Das Ergebnis wird danach in einem Bereich gespeichert, der vom Iterator „d\_first“ bestimmt wird. Grundsätzlich ähneln die akzeptierten Parameter von „std::transform“ denen von „std::for\_each“. Das wichtigste Interface von „std::transform“ ist in Codeausschnitt 2.4 abgebildet.

```
template< class ExecutionPolicy,
          class ForwardIt1,
          class ForwardIt2,
          class UnaryOperation >
ForwardIt2 transform( ExecutionPolicy&& policy,
                     ForwardIt1 first1,
                     ForwardIt1 last1,
                     ForwardIt2 d_first,
                     UnaryOperation unary_op );
```

*Codeausschnitt 2.4, ein Interface von „std::transform“ mit „execution policy“.*

Die zweite Variante ähnelt der ersten, ihr Interface ermöglicht es allerdings zwei Eingangsbereiche anzugeben. Möglich wird dies, indem das Interface einen weiteren 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. Dieses Interface ist in Codeausschnitt 2.5 zu sehen.

```

template< class ExecutionPolicy,
          class ForwardIt1,
          class ForwardIt2,
          class ForwardIt3,
          class BinaryOperation >
ForwardIt3 transform( ExecutionPolicy&& policy,
                    ForwardIt1 first1,
                    ForwardIt1 last1,
                    ForwardIt2 first2,
                    ForwardIt3 d_first,
                    BinaryOperation binary_op );

```

*Codeausschnitt 2.5, ein Interface von „std::transform“ mit „execution policy“ und einem weiteren Iterator.*

Bei „std::transform“ ist zu beachten, dass die Reihenfolge der Operationen nicht garantiert werden kann. Möchte man die Reihenfolge der Operationen garantieren, sollte „std::for\_each“ stattdessen verwendet werden. [11]

Um die Anwendung von „std::transform“ zu verdeutlichen, wird die Helligkeit eines Graustufenbildes mithilfe von „std::transform“ in einem Beispiel geändert. Soll die Helligkeit eines Bildes geändert werden, 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, da das Skalieren eines Werts eine einstellige Verknüpfung ist. Ein Beispiel hierfür wird in Codeausschnitt 2.6 dargestellt.

```

1. std::transform(std::execution::par, //execution policy
2.   image.begin(), //begin of source image vector
3.   image.end(), //end of source image vector
4.   image2.begin(), //begin of destination image vector
5.   [=](auto &a){return a * 3; } //lambda for scaling
6. );

```

*Codeausschnitt 2.6, Beispielcode zur Änderung der Helligkeit in einem Graustufenbild „image“.*

Das Graustufenbild wird im eindimensionalen Vector „image“ gespeichert. Das Ergebnis wird in einem, ebenfalls eindimensionalen, Vector „image2“ gespeichert. Durch die Lambdafunktion 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.

### 2.2.5. `std::transform_reduce`

„std::transform\_reduce“ fungiert prinzipiell genauso wie „std::transform“, allerdings werden die Ergebnisse der „transform“ Operation hierbei zusätzlich noch aufsummiert. Für den Wert der Summe wird ein Initialwert „init“ als Parameter akzeptiert, auf welchen die Ergebnisse addiert werden. Es kann jedoch nicht nur eine Summe produziert werden, sondern beispielsweise auch ein Produkt. Dies wird durch den Parameter „reduce“ der Klasse „BinaryReductionOp“ möglich gemacht. In den Codeausschnitten 2.7 und 2.8 sind die zwei wichtigsten Interfaces von „std::transform\_reduce“ dargestellt.

```
template< class ExecutionPolicy,
          class ForwardIt, class T,
          class BinaryReductionOp,
          class UnaryTransformOp >
T transform_reduce( ExecutionPolicy&& policy,
                  ForwardIt first, ForwardIt last,
                  T init,
                  BinaryReductionOp reduce,
                  UnaryTransformOp transform );
```

*Codeausschnitt 2.7, eines der Interfaces von „std::transform\_reduce“ mit „execution policy“.*

Wie zuvor bei „std::transform“, akzeptiert „std::transform\_reduce“ ebenfalls zwei Eingangsbereiche. [12]

```

template< class ExecutionPolicy,
          class ForwardIt1, class ForwardIt2, class T,
          class BinaryReductionOp,
          class BinaryTransformOp >
T transform_reduce( ExecutionPolicy&& policy,
                   ForwardIt1 first1,
                   ForwardIt1 last1,
                   ForwardIt2 first2,
                   T init,
                   BinaryReductionOp reduce,
                   BinaryTransformOp transform );

```

*Codeausschnitt 2.8, ein Interfaces von „std::transform\_reduce“ mit „execution policy“ und zweitem Eingangsbereich.*

Die Nutzung von „std::transform\_reduce“ wird durch die Berechnung des Skalarprodukts zweier Vektoren mit „std::transform\_reduce“ veranschaulicht. Bei der parallelen Berechnung des Skalarprodukts zweier Vektoren können Racing Conditions eingeführt werden, da für die Summierung der Produkte jeweils der aktuelle Wert der Summe gelesen werden muss, bevor die Summe aktualisiert werden kann. Um Racing Conditions zu vermeiden, wird deshalb „std::transform\_reduce“ verwendet. Als Beispiel wird in Codeausschnitt 2.9 das Skalarprodukt der beiden Vektoren „vec1“ und „vec2“ gebildet und in der Variablen „sum“ gespeichert.

```

1. auto sum = std::transform_reduce(
2.     std::execution::par, // execution policy
3.     vec1.begin(), // begin of first vector
4.     vec1.end(), // end of first vector
5.     vec2.begin(), // begin of second vector
6.     0, // initial value of sum
7.     [](auto a, auto b){ return a + b; } // reduce
8.     [](auto v, auto w){ return v * w; } // transform
9. );

```

*Codeausschnitt 2.9, „std::transform\_reduce“ Beispiel zur Berechnung des Skalarprodukts zweier Vektoren.*

## 2.3. GPU und die Nvidia HPC SDK

### 2.3.1. Graphics Processing Unit

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 deutlich von der einer CPU.[13]

Die Grundarchitektur einer GPU unterscheidet sich in mehreren Aspekten von der einer CPU. 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.[14] Abbildung 2.1 veranschaulicht den Aufbauunterschied.

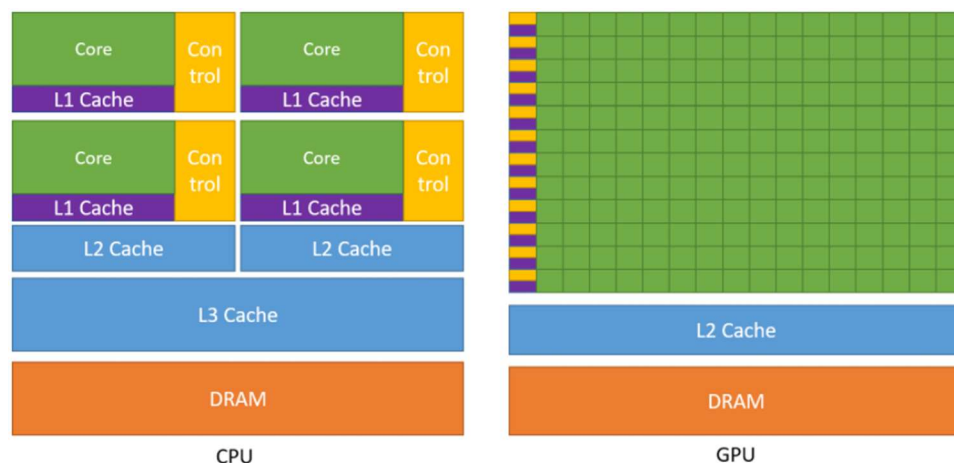


Abbildung 2.1, die GPU dediziert mehr Kerne für die Datenverarbeitung als eine CPU. [13]

### 2.3.2. CUDA

CUDA ist eine von Nvidia entwickelte Programmierschnittstelle. 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 die Aufteilung eines Problems in Teilprobleme ermöglichen. Abstrahiert werden unter anderem die zu Verfügung stehenden Threads. Threads werden dabei zu gleichgroßen Blöcken zusammengefasst. Teilprobleme können dann unabhängig voneinander parallel gelöst werden, indem jedem Teilproblem ein Block mit Threads zugeteilt wird. Die Anzahl an Threads in einem Block ist durch die



Anzahl der Threads, die sich auf einem Prozessorkern befinden, limitiert. Auf aktuellen GPUs kann ein Threadblock bis zu 1024 Threads beinhalten. Threadblöcke sind Teile eines sogenannten Gitters, welches die Threadblöcke organisiert.[13] Abbildung 2.2 zeigt die logische Unterteilung von Threads in Blöcke und Gitter.

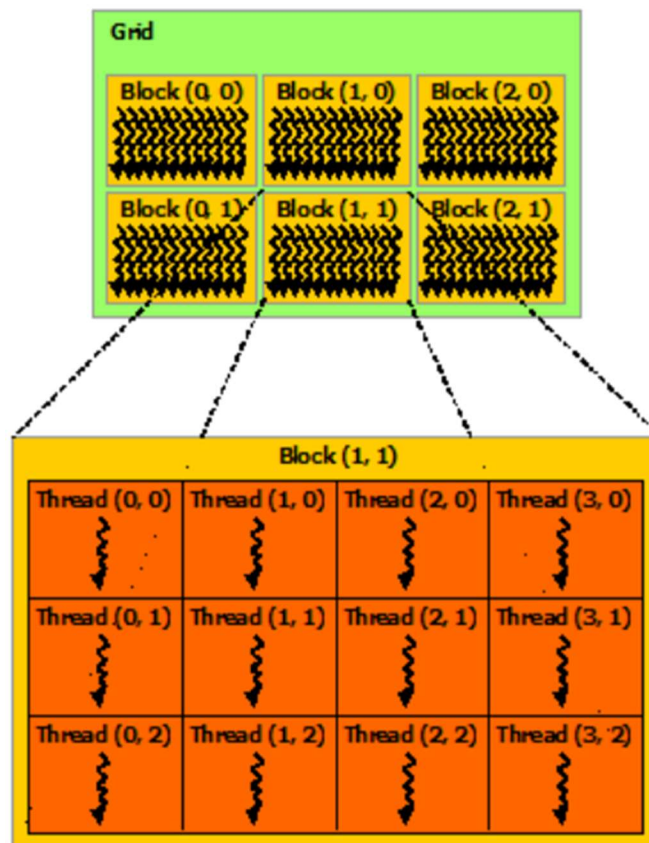


Abbildung 2.2, ein Gitter (engl. Grid) ist unterteilt in Blöcke (engl. Blocks) von Threads. [13]

CUDA erlaubt dem Programmierer bestimmte Funktionen zu definieren, welche parallel von CUDA Threads ausgeführt werden können. Diese Funktionen werden Kernel genannt. Wie viele CUDA Threads den Kernel ausführen, wird mit der Kerneldefinition vorgegeben. 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 an Blöcken pro Gitter angegeben.[13]

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. [13]

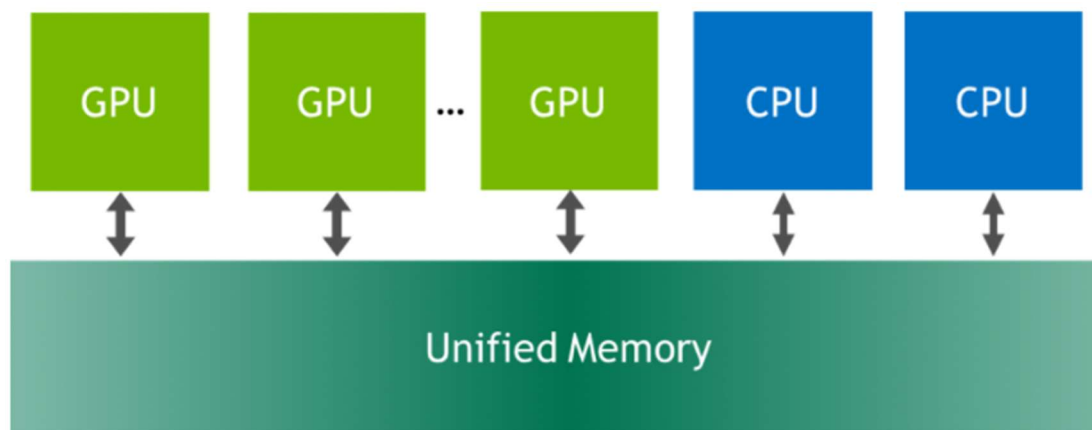
Dies wird durch Abbildung 2.3 veranschaulicht.



Abbildung 2.3, automatische Skalierung durch Aufteilung der Blöcke auf die verschiedenen Streaming Multiprozessoren. [13]

### 2.3.3. CUDA Unified Memory

Unified Memory bezeichnet einen Speicheradressbereich, auf welchen jeder Prozessor in einem System zugreifen kann. Abbildung 2.4 zeigt die Funktion dieses Adressbereichs.



*Abbildung 2.4, Unified Memory ist von allen Prozessoren abrufbar. [15]*

Diese Technologie ermöglicht es Speicher für Daten zu allokalieren, welche sowohl von CPU als auch von GPU gelesen und geschrieben werden können. Um Unified Memory anzulegen, werden Aufrufe zu `malloc()` oder `new` mit Aufrufen zu `cudaMallocManaged()` ersetzt. `cudaMallocManaged()` ist eine Funktion zum Allokieren von Speicher, 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 die Hardware um das Verschieben von Speicher Pages auf den Speicher des zugreifenden Prozessors.[16]

Greift die GPU auf Speicher zu, der von `cudaMallocManaged` angelegt wurde, werden folgende Schritte ausgeführt. Zuerst werden neue Pages auf der GPU angelegt. Danach werden die Verlinkungen der alten Pages auf der CPU gelöst. Daten werden dann von der CPU zur GPU kopiert und die neuen Pages auf der GPU werden gelinkt. Zuletzt werden die alten CPU-Pages freigegeben.

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 Verlinkungen 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.[15]

#### **2.3.4. Nvidia HPC Software Development Kit**

Das Nvidia HPC Software Development Kit (SDK) beinhaltet sowohl C, C++ und Fortran Compiler, als auch Bibliotheken und Tools für die Verbesserung von Leistung und Portabilität von High Performance Computing (HPC) Anwendungen. [17] Die Nvidia HPC SDK steht zum Zeitpunkt der Erstellung dieser Arbeit nur für Linux-Systeme zur Verfügung.

Der Nvidia HPC C++ Compiler NVC++ unterstützt den C++17 Standard. Dieser 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.[2]

GPU-Beschleunigung kann aktiviert werden, indem der Kommandozeilenbefehl „-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 Im Folgenden aufgezeigt werden.[2]

Der NVC++ Compiler erkennt automatisch, welche Funktionen für die Ausführung auf einer GPU kompiliert werden müssen. Hierfür geht der Compiler den Kontrollfluss jedes Source Files durch und schließt daraus, welche Funktionen für die Ausführung auf der GPU kompiliert werden müssen. Dies ist aber nur möglich, wenn die Funktionsdefinitionen im selben Source File liegen wie die Aufrufe der Funktionen.[2] Funktionen, die auf einer GPU kompiliert werden müssen, sind Funktionen, welche von dem Parallel STL Algorithmus aufgerufen werden. Da der Algorithmus auf der GPU ausgeführt werden soll, muss für jede Funktion, die er aufruft, GPU-Code existieren.

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 sowohl Objekte, als auch

Pointer, die von einem Parallel STL Algorithmus verwendet werden, auf Daten aus dem CPU-Heap referieren müssen. Außerdem müssen diese Daten von einer Anwendung, welche vom NVC++ Compiler kompiliert wurde, allokiert worden sein.[2]

Ebenso wie für verwendete Daten, gibt es Limitierungen bei der Verwendung von Funktionen im Parallel STL Aufruf. So können keine Funktionspointer an einen Parallel STL Aufruf als Parameter übergeben werden. Lediglich Funktionsobjekte oder Lambdas können übergeben werden. Die letzte wichtige Einschränkung ist, dass Exceptions im Code, welcher auf der GPU ausgeführt wird, nicht verwendet werden sollten. Der Exception-Code kompiliert zwar, hat jedoch nicht das erwartete Verhalten. So werden Catch-Bereiche ignoriert, sodass das Werfen einer Exception dazu führt, dass der GPU-Kernel abgebrochen wird. [2]

#### **2.4. Optischer Fluss mit OpenCV**

OpenCV ist eine open-source Library für Computer Vision.[18] Relevant für diese Arbeit ist das Modul zur Berechnung des optischen Flusses, welchen der zu parallelisierende Algorithmus beinhaltet. Als optischer Fluss wird das Schema einer offensichtlichen Bewegung eines beliebigen Objekts bezeichnet. Hierbei wird sich auf die Bewegung zwischen zwei aufeinanderfolgenden Bildern (Frames) einer Bild- oder Videosequenz, in der das Objekt bewegt wurde, beschränkt.[19] Um den optischen Fluss darzustellen, werden Vektoren verwendet, welche die Bewegung des Objekts beschreiben. Die Berechnung des optischen Flusses kann in zwei Kategorien aufgeteilt werden: Die Berechnung eines „sparse“ optischen Flusses und die eines „dense“ optischen Flusses. Zur Berechnung eines „sparse“ optischen Flusses werden Flussvektoren zu bestimmten Eigenschaften eines Frames bestimmt. Solche Eigenschaften können hier Kanten oder Ecken von Objekten sein. Beim „dense“ optischen Fluss werden Flussvektoren für jeden Pixel des Frames bestimmt. Der „dense“ optische Fluss ist dadurch zwar genauer, jedoch auch deutlich rechenintensiver.[20]

Der von OpenCV implementierte Algorithmus berechnet einen „dense“ optischen Fluss, basierend auf dem Paper „Two-Frame Motion Estimation Based on Polynomial Expansion“ von Gunnar Farneback.[20] In diesem Paper stellt Farneback einen Algorithmus zur Bewegungsabschätzung durch polynomiale Expansion vor. Die Idee hinter dieser polynomialen Expansion ist es, die Umgebung (Neighbourhood) eines Pixels mit einem quadratischen Polynom abzuschätzen. Der Mittelpunkt des Neighbourhoods wird am stärksten gewichtet. Die Gewichtung nimmt gleichmäßig nach außen hin ab. Somit hat der Mittelpunkt des Neighbourhoods mehr Einfluss auf das abzuschätzende Polynom. Das Wichtige beim Abschätzen der Polynome sind ihre resultierenden Koeffizienten, die zur Berechnung des optischen Flussfelds genutzt werden, da die Polynome aus den Koeffizienten gebildet werden können.[21]

### **3. Versuchsaufbau**

In diesem Kapitel wird zunächst die Entstehung der parallelisierten Implementierung schrittweise erläutert. Um den von OpenCV implementierten „dense optical flow“ Algorithmus zu parallelisieren, wird zuerst der zeitaufwendigste Bereich des Algorithmus identifiziert. Anschließend werden Lösungskonzepte aufgestellt und evaluiert, woraufhin der geeignetste Lösungsansatz ausgewählt und implementiert wird.

#### **3.1. Anforderungsanalyse**

Um entscheiden zu können welchen Teil der OpenCV Implementierung am meisten von einer Parallelisierung profitiert, ist es notwendig die zeitaufwendigsten Teile der Implementierung herauszufinden. Die 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 eine Videosequenz verwendet, welche in 300 einzelne Framebilder aufgeteilt ist. Zeiten werden mithilfe der „chrono::steady\_clock“ Klasse aus der Standard Library von C++ genommen. Diese ermöglicht es Zeitpunkte zu speichern und daraus Zeitintervalle zu berechnen.

Um das Input Video einzulesen und den daraus resultierenden optischen Fluss darzustellen, wird das Programm „denseFlow“ verwendet. Der vollständige Code von „denseFlow“ findet sich im Anhang 7.3 wieder. „denseFlow“ basiert auf dem Beispielcode zur Berechnung von optischem Fluss der OpenCV Dokumentation. Die Ausführungszeiten folgender Funktionen werden gemessen:

- `calcOpticalFlowFarneback()`
- `FarnebackPolyExp()`
- `FarnebackUpdateMatrices()`
- `FarnebackUpdateFlow_Blur()`

Diese Funktionen wurden aus den im Folgenden erklärten Gründen ausgewählt. `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`, `FarnebackUpdateMatrices` und `FarnebackUpdateFlow_Blur` werden jeweils von `calcOpticalFlowFarneback` aufgerufen und zur Flussberechnung verwendet.

Erwähnenswert ist, 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 sowie der Videosequenz, 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.

$$s = \sum_{i=1}^a t_i = t_1 + t_2 + \dots + t_a$$

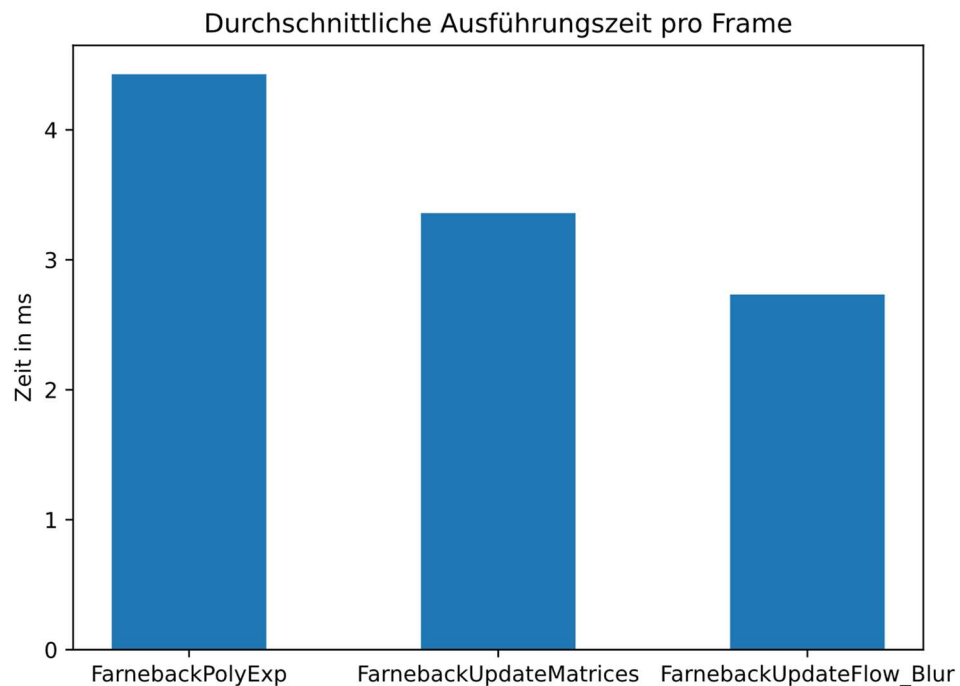
$$\bar{s} = \frac{1}{n * a} \sum_{k=1}^n s_k = \frac{s_1 + s_2 + \dots + s_n}{n * a}$$

$\bar{s}$  ist das arithmetische Mittel der Ausführungszeit eines Funktionsaufrufs.

Sind die Ausführungszeiten erfasst, sollte der Code der ausgewählten Funktion genauer betrachtet 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.

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 das Balkendiagramm in Abbildung 3.1 und der Tabelle 3.1 dargestellt.





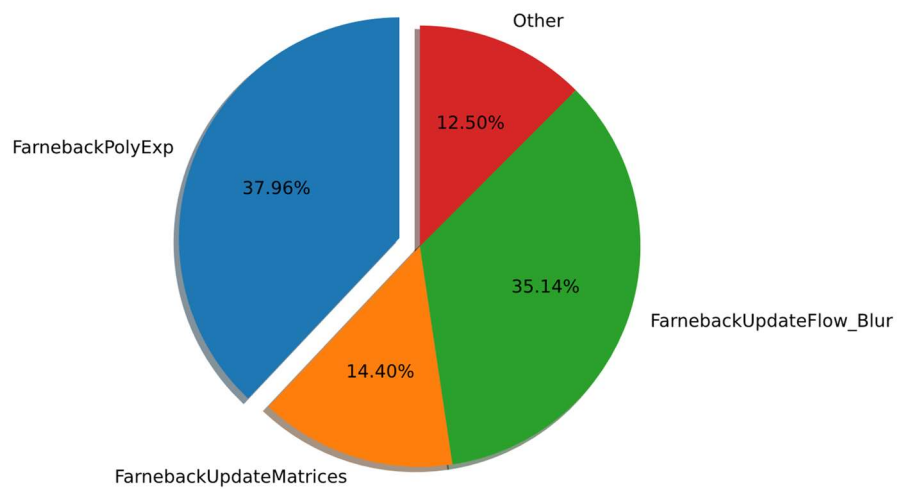
*Abbildung 3.1, durchschnittliche Ausführungszeiten pro Frame der Funktionen FarnebackPolyExp, FarnebackUpdateMatrices und FarnebackUpdateFlow\_Blur. Gemessen auf einer Intel(R) Core(TM) i7-9850H CPU.*

Funktion	berechnete Ausführungszeit
FarnebackPolyExp	4,428 ms
FarnebackUpdateMatrices	3,359 ms
FarnebackUpdateFlow_Blur	2,733 ms

*Tabelle 3.1, berechnete Ausführungszeiten von FarnebackPolyExp, FarnebackUpdateMatrices und FarnebackUpdateFlow\_Blur in Millisekunden.*

Auf den ersten Blick ist FarnebackPolyExp die Funktion mit der längsten durchschnittlichen Ausführungszeit. Dennoch sollte 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 und in Abbildung 3.2 festgehalten.



*Abbildung 3.2, Verteilung der Ausführungszeit von calcOpticalFlowFarneback auf die Funktionen FarnebackPolyExp, FarnebackUpdateMatrices und FarnebackUpdateFlow\_Blur.*

Other bezeichnet hier Code, welcher von calcOpticalFlowFarneback direkt ausgeführt wird und nicht in eine andere Funktion ausgelagert ist. Bemerkenswert 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 durch die unterschiedliche Anzahl der jeweiligen Funktionsaufrufe innerhalb eines Aufrufs von calcOpticalFlowFarneback begründet werden. Mit den in der Messung verwendeten Parametern von calcOpticalFlowFarneback wird FarnebackPolyExp acht Mal, FarnebackUpdateMatrices vier Mal und FarnebackUpdateFlow\_Blur zwölf Mal aufgerufen.

Auch nach mehreren Ausführungen von „densFlow“ sind keine nennenswerten Änderungen der Messergebnisse zu beobachten. Daraus lässt sich schließen, dass die Funktion FarnebackPolyExp im Durchschnitt 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 werden verwendete Schleifen analysiert. `FarnebackPolyExp` beinhaltet insgesamt sieben Schleifen, welche in Codeausschnitt 3.1 zusammengefasst werden. Die steuernden Parameter dieser Schleifen sind „height“, „width“ und `n`. Bei „height“ und „width“ handelt es sich um Höhe und Breite des Inputframes. Der Parameter `n` gibt den Radius des Neighbourhoods an, in welchem das Polynom angenähert wird. Die vollständige `FarnebackPolyExp` Funktion ist in Anhang 7.4 zu finden.

```
114. static void
115. FarnebackPolyExp((...)){
116.     (...)
132.     for( y = 0; y < height; y++ ){
133.         (...)
138.         for( x = 0; x < width; x++ ){ (...) }
143.         for( k = 1; k <= n; k++ ){
144.             (...)
150.             for( x = 0; x < width; x++ ){ (...) }
161.         }
164.         for( x = 0; x < n*3; x++ ){ (...) }
169.         for( x = 0; x < width; x++ ){
170.             (...)
177.             for( k = 1; k <= n; k++ ){ (...) }
186.             (...)
193.         }
194.     }
195.     (...)
197. }
```

*Codeausschnitt 3.1, Darstellung der genutzten Schleifen in der Funktion `FarnebackPolyExp`.*

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 den Radius des verwendeten Neighbourhoods und liegt in realistischen Anwendungen bei Ganzzahlwerten zwischen fünf und sieben. Wie in Kapitel 2.4 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. Aufgrund dieser Gegebenheit kann die Schleifenstruktur von `FarnebackPolyExp` so zusammengefasst werden, dass die Schleifen im Grunde über den ganzen Inputframe iterieren. So werden für jeden Pixel des Inputframes die zugehörigen Polynomkoeffizienten mithilfe des Neighbourhoods berechnet. Dies deutet auf mögliche Datenparallelität hin, da es sich hierbei um Berechnungen für jedes Datenelement, in diesem Fall Pixel, handelt.

### **3.2. Lösungsansätze**

Es werden zwei Lösungsansätze behandelt. 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, das Neighbourhood in einen horizontalen und vertikalen Teil aufzuteilen, beibehalten werden. Da es sich bei den Schleifen von `FarnebackPolyExp` um einfache for-Schleifen handelt, können diese in Befehle der „Algorithms Library“ übersetzt und parallelisiert werden. Vor allem Befehle wie „`std::for_each`“ oder „`std::transform`“ sind dafür geeignet.

Der zweite Lösungsansatz greift stärker in die Struktur des originalen Codes ein. Die originale Implementierung von `FarnebackPolyExp` 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 staten gehen. In dieser Weise wird dann auch von der Datenparallelität Gebrauch gemacht.

Ein Hauptproblem des ersten Ansatzes ist die Komplexität des originalen Codes. Wird der originale Code in Befehle der „Algorithms Library“ übersetzt, kann es zu Performance-Einbußen kommen. Für jeden Befehl der „Algorithms 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 der Start des Kernels zeitaufwendiger als dessen Ausführung ist. Dieses Problem 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 setzt dies um, indem nur ein „Algorithms Library“ Befehl aufgerufen wird. Da dieser Ansatz die einzelne Berechnung jedes Pixels als Ziel verfolgt, wird nur ein „Algorithms Library“ Befehl benötigt. Der zweite Lösungsansatz sorgt für einen höheren Aufwand, weil der originale Code signifikant geändert werden muss.

### **3.3. Ausgewählter Ansatz und Durchführung**

#### **3.3.1. Detailanalyse der Polynomkoeffizienten Berechnung**

Erste Versuche den ersten Lösungsansatz zu implementieren, zeigten sehr schlechte Performance. Deshalb wird der zweite Lösungsansatz, den originalen Code in eine Per-Pixel-Berechnung umzuschreiben und zu parallelisieren, ausgewählt. Hierfür ist die im Folgenden beschriebene Vorgehensweise geplant. Zuerst soll der originale Code in eine Per-Pixel-Berechnung übersetzt werden. Ist dies gelungen, wird mithilfe der Parallel STL und ihrer „execution policies“ parallelisiert. Danach soll der parallelisierte Code mit dem NVC++ Compiler der Nvidia HPC SDK kompiliert und gebaut werden, um ihn auf der GPU auszuführen.

Um die Berechnung in eine Per-Pixel-Berechnung umzuschreiben, soll im Detail analysiert werden, wie der originale Code die Polynomkoeffizienten berechnet. Codeausschnitt 3.2 zeigt den Beginn der Polynomkoeffizienten Berechnung. 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  $y$  verwendet, um Pointer auf korrespondierenden Zeilen des Inputframes zu erhalten. Der Inputframe ist in der Matrix „src“ gespeichert, welche als Parameter an die

Funktion übergeben wurde. 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 Polynomkoeffizienten.

```
132. for( y = 0; y < height; y++ )
133. {
134.     float g0 = g[0], g1, g2;
135.     const float *srow0 = src.ptr<float>(y), *srow1 = 0;
136.     float *drow = dst.ptr<float>(y);
```

*Codeausschnitt 3.2, Pointer werden verwendet, um Zeilen des Inputframes referenzieren zu können.*

Die Schleife in Zeile 138 des Codeausschnitts 3.3 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.

```
138. for( x = 0; x < width; x++ )
139. {
140.     row[x*3] = srow0[x]*g0;
141.     row[x*3+1] = row[x*3+2] = 0.f;
142. }
```

*Codeausschnitt 3.3, ein Buffer wird mit Initialwerten befüllt.*

Direkt im Anschluss folgt in Codeausschnitt 3.4 eine Schleife, welche mit „k = 1“ startet und bis zur Variable n inkrementiert. Innerhalb dieser Schleife befindet sich in Zeile 149 eine weitere 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.

```

143. for( k = 1; k <= n; k++ )
144. {
145.     g0 = g[k]; g1 = xg[k]; g2 = xxg[k];
146.     srow0 = src.ptr<float>(std::max(y-k,0));
147.     srow1 = src.ptr<float>(std::min(y+k,height-1));
148.
149.     for( x = 0; x < width; x++ )
150.     {
151.         float p = srow0[x] + srow1[x];
152.         float t0 = row[x*3] + g0*p;
153.         float t1 = row[x*3+1] + g1*(srow1[x]-srow0[x]);
154.         float t2 = row[x*3+2] + g2*p;
155.
156.         row[x*3] = t0;
157.         row[x*3+1] = t1;
158.         row[x*3+2] = t2;
159.     }
160. }

```

*Codeausschnitt 3.4, der vertikale Teil der Faltung wird für eine Zeile berechnet und im Buffer gespeichert.*

Wichtig festzuhalten ist, dass bis zu diesem Punkt Werte aus einem vertikalen Neighbourhood in „row“ zusammengefügt wurden. Außerdem werden die Ränder des Inputframes bei der Akquisition der Inputframezeilen im Neighbourhood beachtet.

„std::max“ und „std::min“ sorgen dafür, die Randzeile des Inputframes der eigentlichen Inputframezeile vorzuzuziehen, falls das Neighbourhood auf eine Zeile außerhalb des Inputframes zugreifen möchte. Somit wird die Randzeile effektiv kopiert und imaginär an den Rand des Inputframes angehängt. Zusammengefasst wurde in vertikaler Richtung das Neighbourhood für jeden x-Wert der Inputframezeile bearbeitet.

Um für die Randbehandlung die gleiche Funktionalität wie bisher bieten zu können, werden mit der Schleife in Codeausschnitt 3.5 die ersten drei Werte am Anfang des Buffers n-mal an den Anfang angefügt und die letzten drei Werte des Buffers n-mal an das Ende angefügt.

```
163. for( x = 0; x < n*3; x++ )
164. {
165.     row[-1-x] = row[2-x];
166.     row[width*3+x] = row[width*3+x-3];
167. }
```

*Codeausschnitt 3.5, Randwerte werden kopiert um „out-of-bounds“ Zugriffe zu vermeiden.*

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

Da der vertikale Teil der Berechnung für eine Inputframezeile abgearbeitet wurde, folgt nun in Codeausschnitt 3.6 der horizontale Teil. Zeilen 172 – 174 initialisieren Variablen für die zu berechnenden Polynomkoeffizienten. Die Schleife in Zeile 175 wird verwendet, um das horizontale Neighbourhood jedes x-Wertes zu bearbeiten. Die aus diesem Neighbourhood berechneten Werte werden dann in der jeweiligen Koeffizientenvariable gespeichert. Nachdem das Neighbourhood für einen x-Wert abgearbeitet ist, werden die endgültigen Polynomkoeffizienten 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.



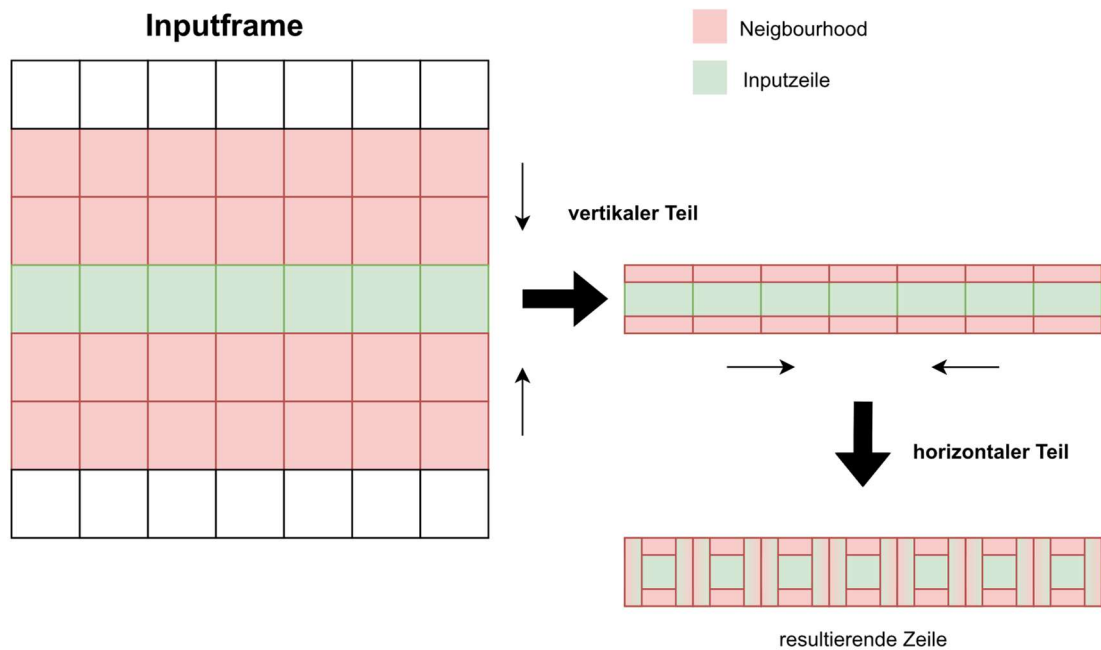
```

168. for( x = 0; x < width; x++ )
169. {
170.     g0 = g[0];
171.     //r1 ~ 1, r2 ~ x, r3 ~ y, r4 ~ x^2, r5 ~ y^2, r6~xy
172.     double b1 = row[x*3]*g0, b2 = 0,
173.            b3 = row[x*3+1]*g0, b4 = 0,
174.            b5 = row[x*3+2]*g0, b6 = 0;
175.
176.     for( k = 1; k <= n; k++ )
177.     {
178.         g0 = g[k];
179.         b1 += (row[(x+k)*3] + row[(x-k)*3])*g0;
180.         b2 += (row[(x+k)*3] - row[(x-k)*3])*xg[k];
181.         b4 += (row[(x+k)*3] + row[(x-k)*3])*xxg[k];
182.         b3 += (row[(x+k)*3+1] + row[(x-k)*3+1])*g0;
183.         b6 += (row[(x+k)*3+1] - row[(x-k)*3+1])*xg[k];
184.         b5 += (row[(x+k)*3+2] + row[(x-k)*3+2])*g0;
185.
186.     }
187.     // do not store r1
188.     drow[x*5+1] = (float) (b2*ig11);
189.     drow[x*5] = (float) (b3*ig11);
190.     drow[x*5+3] = (float) (b1*ig03 + b4*ig33);
191.     drow[x*5+2] = (float) (b1*ig03 + b5*ig33);
192.     drow[x*5+4] = (float) (b6*ig55);

```

**Codeausschnitt 3.6, der horizontale Teil der Faltung wird abgearbeitet und die Polynomkoeffizienten werden berechnet.**

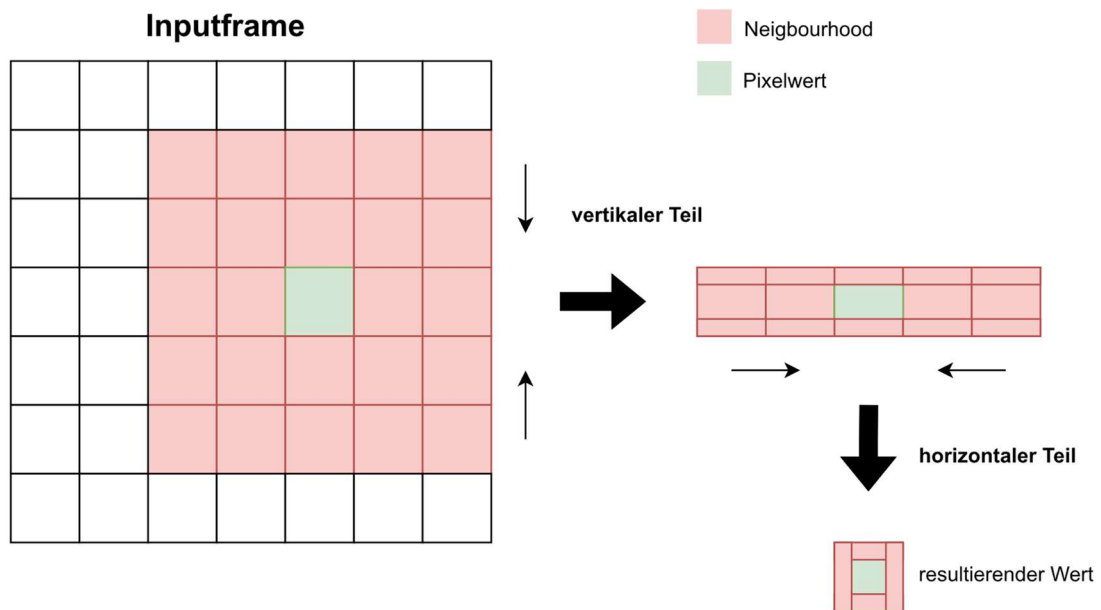
Zusammenfassend wird im originalen Code der Inputframe Zeile für Zeile eingelesen. Dann werden über ein Neighbourhood für jeden Pixel der Inputframezeile drei Werte berechnet. Mithilfe dieser drei Werte werden dann die Polynomkoeffizienten berechnet und in der Zielmatrix gespeichert. Das Schema der Zusammenführung des Neighbourhoods einer Inputframezeile wird durch Abbildung 3.3 veranschaulicht.



*Abbildung 3.3, zuerst wird das Neighbourhood der Inputzeile in vertikaler Richtung in eine Zeile zusammengefügt. Danach wird für jeden Wert der Zeile das Neighbourhood in horizontaler Richtung zusammengefügt.*

### **3.3.2. Übersetzung in Per-Pixel-Berechnung**

Das zuvor beschriebene Verhalten wird nun in eine Per-Pixel-Berechnung übersetzt. Die vollständige Per-Pixel-Implementierung ist in Anhang 7.4 zu finden. Die Eigenschaft der Aufteilung der Berechnung in einen vertikalen und horizontalen Teil soll dabei beibehalten werden. Abbildung 3.4 veranschaulicht das Schema, in dem das Neighbourhood bearbeitet werden soll.



*Abbildung 3.4, zuerst wird das Neighbourhood 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 Neighbourhood, um alle benötigten Werte zwischenspeichern zu können. Codeausschnitt 3.7 zeigt den Aufbau dieser Schleifen.

```
for( y = 0; y < height; y++ )
{
    for( x = 0; x < width; x++ )
    {
        float g0 = g[0];
        std::vector<float> rBuf((2 * n + 1)*3, 0.f);
        int offset = 2*n+1;
```

*Codeausschnitt 3.7, zwei verschachtelte Schleifen werden verwendet um über den Inputframe zu iterieren. Außerdem wird ein Buffer eingeführt, um benötigte Werte zwischenspeichern.*

Danach wird der vertikale Teil des Neighbourhoods, veranschaulicht durch Codeausschnitt 3.8, 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 Neighbourhoods abzuarbeiten. Zuerst werden neue variablen „neighY0“ und „neighY1“ 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 auf die Werte des Randes zugegriffen.

```
for(int a = 0; a < 2*n+1; a++){
    int neighX = std::max(x + a-n, 0);
    neighX = std::min(neighX, width-1);
    rBuf[a] = _src[neighX + y * width] * g0;

    for(int b = 1; b <= n; b++){
        int neighY0 = std::max((y-b)*width, 0);
        int neighY1 = std::min((y+b)*width,
                                (height-1)*width);

        rBuf[a] += (_src[neighX + neighY0] +
                    _src[neighX + neighY1]) * g[b];

        rBuf[a + offset] += (_src[neighX + neighY1] -
                              _src[neighX + neighY0]) * xg[b];

        rBuf[a + offset*2] += (_src[neighX + neighY0] +
                               _src[neighX + neighY1]) * xxg[b];
    }
}
```

***Codeausschnitt 3.8, der vertikale Teil der Polynomkoeffizienten  
Berechnung für einen Pixel.***

Nachdem das Neighbourhood eines Pixels im vertikalen Teil auf eine Zeile des Neighbourhoods zusammengefasst wurde, wird nun in Codeausschnitt 3.9 der horizontale Teil behandelt. Hierbei werden die Variablen „b1“ – „b6“ verwendet, um nach dem Zusammenfassen der Neighbourhoodzeile direkt die Polynomkoeffizienten berechnen zu können.

```
double b1 = rBuf[n]*g0, b2 = 0, b3 = rBuf[n + offset]*g0,
       b4 = 0, b5 = rBuf[n + offset*2]*g0, b6 = 0;

for( int a = 1; a <= n; a++){
    b1 += (rBuf[n+a] + rBuf[n-a]) * g[a];
    b2 += (rBuf[n+a] - rBuf[n-a]) * xg[a];
    b4 += (rBuf[n+a] + rBuf[n-a]) * xxg[a];
    b3 += (rBuf[n+a+offset] + rBuf[n-a+offset]) * g[a];
    b6 += (rBuf[n+a+offset] - rBuf[n-a+offset]) * xg[a];
    b5 += (rBuf[n+a+offset*2] + rBuf[n-a+offset*2]) * g[a];
}
```

*Codeausschnitt 3.9, der horizontale Teil der Polynomkoeffizienten Berechnung für einen Pixel.*

Die in den Variablen „b1“ – „b6“ gespeicherten Werte werden dann, wie im originalen Code zuvor, dazu verwendet die Polynomkoeffizienten für einen Pixel zu berechnen und in der Matrix „dst“ zu speichern. Dies wird in Codeausschnitt 3.10 dargestellt.

```
int pixel = x+y*width;
_dst[pixel*5+1] = (float) (b2*ig11);
_dst[pixel*5] = (float) (b3*ig11);
_dst[pixel*5+3] = (float) (b1*ig03 + b4*ig33);
_dst[pixel*5+2] = (float) (b1*ig03 + b5*ig33);
_dst[pixel*5+4] = (float) (b6*ig55)
```

*Codeausschnitt 3.10, die endgültigen Polynomkoeffizienten werden berechnet und in einer Matrix gespeichert.*

### 3.3.3. Parallelisierung der Per-Pixel-Berechnung.

Die Funktion FarnebackPolyExp ist nun auch als Per-Pixel-Berechnung implementiert. Unter Verwendung der „Algorithms Library“ kann dadurch mit geringem Aufwand paralleler Code entstehen. Hierfür werden in Codeausschnitt 3.11 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. Die vollständige, mit „std::for\_each“ parallelisierte Implementierung ist in Anhang 7.6 zu finden.

```
std::for_each(std::execution::par_unseq, //execution policy
             _src, // pointer to the source image
             _src + (width * height), //pointer to the last element
             [=](auto &pix){(...)}) // lambda for coefficients
);
```

*Codeausschnitt 3.11, „std::for\_each“ wird verwendet um über den Inputframe zu iterieren und macht Parallelisierung möglich.*

Innerhalb der Lambda-Funktion befindet sich nahezu 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 können ermittelt werden, indem der Index des Pixels mithilfe von Pointer-Arithmetik berechnet wird. Die Speicheradresse auf die der Pointer des Inputframes zeigt wird von der Speicheradresse des aktuellen Pixels subtrahiert. Mit dem dadurch berechneten Index werden x- und y-Koordinate bemessen. Codeausschnitt 3.12 zeigt diese Berechnungen.

```
auto index = &pix - src_ptr;
int x = index % width;
int y = index / width;
```

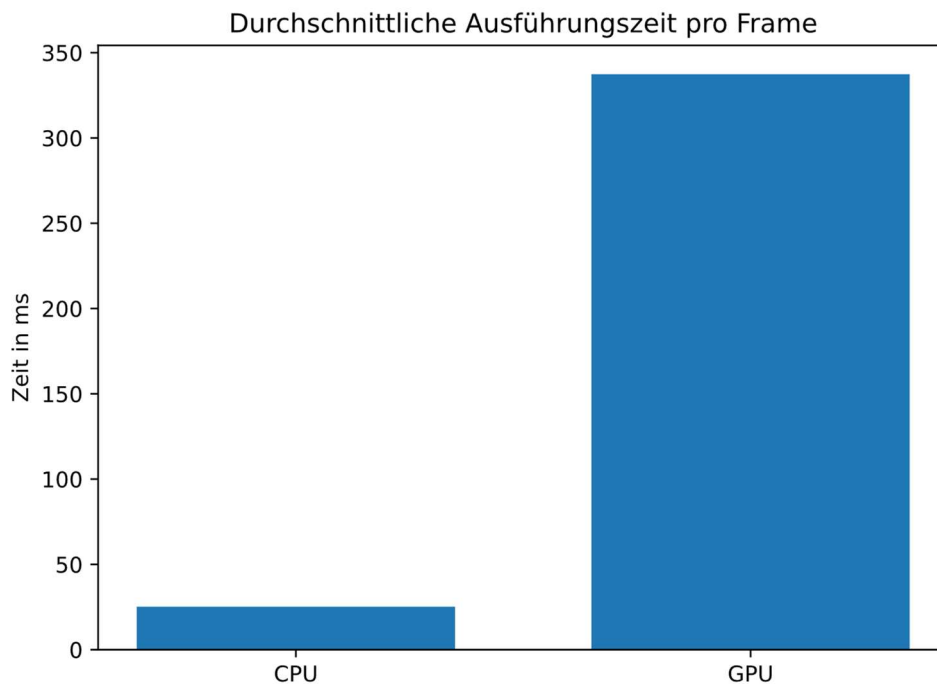
*Codeausschnitt 3.12, durch Pointer-Arithmetik wird der Index des verwendeten Pixels berechnet.*

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 mithilfe des NVC++ Compilers zu kompilieren, zeigen mehrere Probleme auf. Diese können auf die Verwendung von OpenCV zurückgeführt werden. Der NVC++ Compiler kann die Definitionen der OpenCV Klassen nicht auffindig machen, da die Klassen über die OpenCV Library inkludiert werden. Somit ist die Nutzung von CUDA Unified Memory nicht möglich. Aufgrund dessen kann der Code nicht erfolgreich ausgeführt werden.

Die Lösung der zuvor beschriebenen Probleme erfordern eine umfassende Änderung der originalen Implementierung von OpenCV. Da die Parallelisierung mit der Parallel STL und die dadurch resultierende Performance im Vordergrund steht, wird die Funktion `FarnebackPolyExp` von der OpenCV Implementierung abgekapselt. Hierfür wird ein neues Programm erstellt, welches Platzhalterdaten zur Berechnung verwendet. Diese Platzhalterdaten sollen einem Videoframe ähneln. Auch die ursprüngliche Hauptfunktionalität des Speicherns der berechneten Polynomkoeffizienten in einer Matrix soll beibehalten werden.

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 genutzt, da der Speicher für diesen Vektor auf dem CPU Heap allokiert wird. Dies ist Voraussetzung zur Nutzung der Daten mit CUDA Unified Memory. Die berechneten Polynomkoeffizienten werden ebenfalls in 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. Der vollständige Code von „`polyExp-stl`“ ist in Anhang 7.8 zu finden.

Erstmals kann nun die neue `FarnebackPolyExp` Implementierung auf der GPU ausgeführt werden. Erste Performance Messungen sind in Abbildung 3.5 dargestellt.



*Abbildung 3.5, 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. Zur Begründung der unerwartet langen Ausführungszeit, wird die Ausführung mithilfe von einem Profiling-Tool der Nvidia HPC SDK genauer analysiert. Allerdings muss die GPU-Implementierung hierfür auf einer anderen GPU ausgeführt werden, da das Profiling-Tool die zuvor genutzte Hardware nicht unterstützt. Zum Profiling wird eine GPU einer neueren Generation verwendet. Die Profiling Ergebnisse zeigen eine hohe Anzahl von Speicher- und Ladebefehlen, welche reduziert werden sollen.

Der letzte Schritt in der Parallelisierung der FarnebackPolyExp Funktion ist es die Anzahl von benötigten Speicherzugriffen zu reduzieren und somit die bisher erarbeitete Implementierung zu optimieren. Als konkreten Optimierungsschritt soll die Anzahl der verwendeten Zwischenspeicher innerhalb des Kernels reduziert werden. Um die Berechnung der Polynomkoeffizienten effizienter zu machen, wird in Codeausschnitt 3.13 auf den Zwischenspeicher für die horizontale Abarbeitung des Neighbourhoods verzichtet. Die Werte des

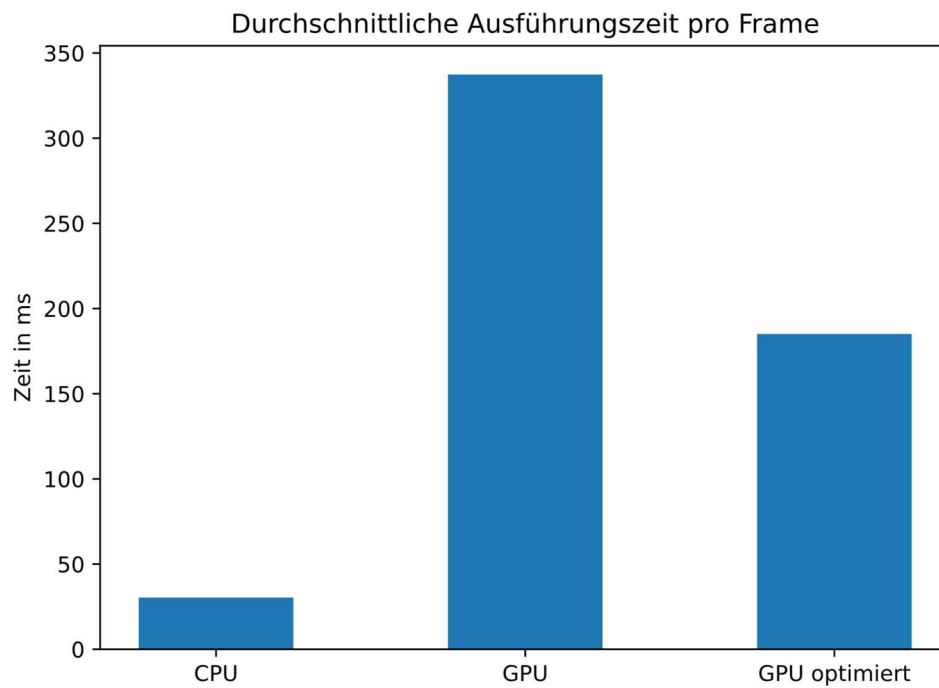


vertikalen Neighbourhoods werden nicht mehr zwischengespeichert. Stattdessen wird jeder erhaltene Wert mit dem richtigen Faktor multipliziert und je nach Koeffizienten und Position innerhalb des Neighbourhoods auf die Variablen der Polynomkoeffizienten addiert oder subtrahiert. Der vollständige Code dieser optimierten Per-Pixel-Implementierung ist in Anhang 7.7 zu finden.

```
for( int a = 0; a < 2*n+1; a++){  
    (...)  
  
    float g = kbuf[abs(a-n) + n];  
    float xg = kbuf[abs(a-n) + xgOff];  
    float xxg = kbuf[abs(a-n) + xxgOff];  
    if(a == n){  
        b1 += rBuf * g;  
        b3 += xrBuf * g;  
        b5 += xxrBuf * g;  
    } else if(a > n){  
        b1 += rBuf * g;  
        b2 += rBuf * xg;  
        b3 += xrBuf * g;  
        b4 += rBuf * xxg;  
        b5 += xxrBuf * g;  
        b6 += xrBuf * xg;  
    } else {  
        b1 += rBuf * g;  
        b2 -= rBuf * xg;  
        b3 += xrBuf * g;  
        b4 += rBuf * xxg;  
        b5 += xxrBuf * g;  
        b6 -= xrBuf * xg;  
    }  
}
```

*Codeausschnitt 3.13, Optimierung durch Verzicht auf Zwischenspeicher und Reduzierung der benötigten Schleifen.*

Die CPU-Implementierung, die ursprüngliche GPU-Implementierung und die optimierte GPU-Implementierung werden in Abbildung 3.6 verglichen.



*Abbildung 3.6, Vergleich der durchschnittlichen Ausführungszeit zwischen der CPU-Implementierung, der nicht optimierten GPU-Implementierung und der optimierten GPU-Implementierung.*

## 4. Ergebnisse

Die bei der Parallelisierung erhaltenen Ergebnisse und Erkenntnisse werden zusammengeführt und in den selben Kontext gebracht. Es wird die Richtigkeit der Implementierung validiert und eine Ergebnisdiskussion geführt.

### 4.1. Validierung der Ergebnisse und Richtigkeit der Implementierung

Zum korrekten Vergleich der Messergebnisse der verschiedenen Implementierung, müssen die Ergebnisse der Funktionen immer identisch sein. Da es sich bei den berechneten Polynomkoeffizienten um floating-point Werte handelt, können gewisse Abweichungen entstehen. Die Abweichung zwischen den Polynomkoeffizienten der Original-Implementierung und der Per-Pixel-Implementierung sollten möglichst gering sein.

Um die Richtigkeit der FarnebackPolyExp Implementierungen gewährleisten zu können, wird der durchschnittliche absolute Fehler der Polynomkoeffizienten aus der Original-Implementierung und den Per-Pixel-Implementierungen berechnet:

A: Polynomkoeffizienten der Original-Implementierung

B: Polynomkoeffizienten der Per-Pixel-Implementierung

b: Summe der absoluten Differenz

d: durchschnittliche absolute Fehler

$$b = \|A - B\|_1 = \sum_{i=1}^{M*N*5} |a_i - b_i|$$

$$d = \frac{b}{M * N * 5}$$

In Tabelle 4.1 sind die berechneten absoluten Fehler der jeweiligen Implementierungen festgehalten. Alle Implementierungen berechnen die Polynomkoeffizienten korrekt, da die durchschnittlichen absoluten Fehler äußerst gering sind.

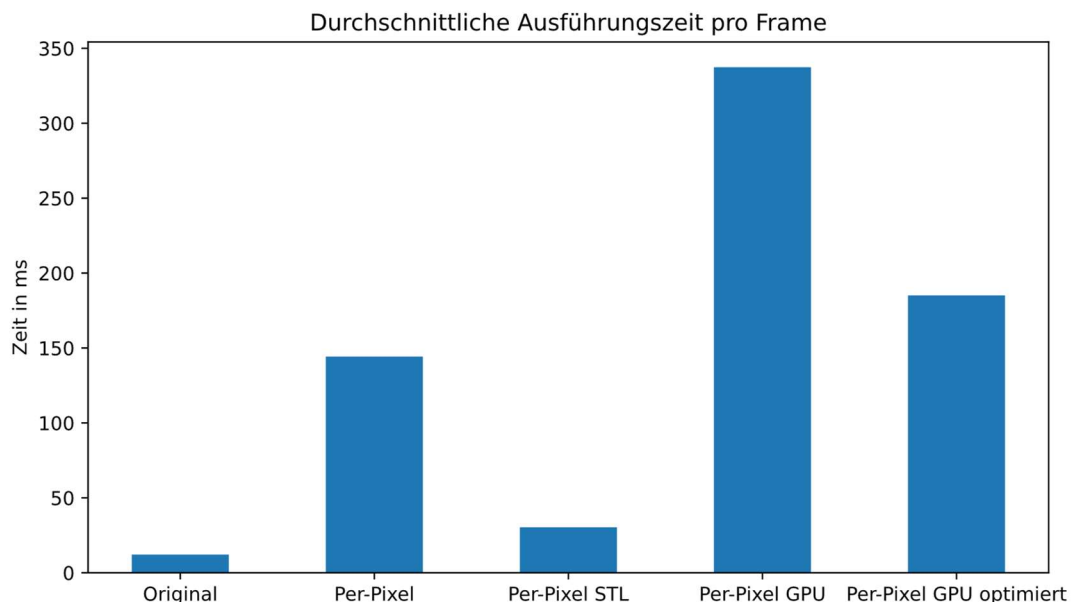
Implementierung	Durchschnittliche absolute Fehler
Per-Pixel auf CPU	0,0
Per-Pixel STL auf CPU	0,0
Per-Pixel STL auf GPU	0,0
Per-Pixel STL auf GPU optimiert	4,78885410879754e-07

*Tabelle 4.1, Auflistung der berechneten durchschnittlichen absoluten Fehler der erarbeiteten Implementierungen.*

## 4.2. Übersicht der Ergebnisse

Sowohl die originale 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 danach parallelisiert und somit auf mehreren CPU-Kernen ausgeführt. Die Per-Pixel Implementierung mit Parallel STL wurde mithilfe des Nvidia NVC++ Compilers für die Ausführung auf einer GPU kompiliert.

Um die Ausführungszeit zu reduzieren, wurde die Implementierung anschließend optimiert. Abbildung 4.1 zeigt die erreichten durchschnittlichen Ausführungszeiten der verschiedenen Implementierungen. In Tabelle 4.2 werden diese Ausführungszeiten als Zahlenwerte dargestellt.



*Abbildung 4.1, durchschnittliche Ausführungszeiten pro Frame für alle erarbeiteten Implementierungen*

Implementierung	Durchschnittliche Ausführungszeit
Original auf CPU	12,123 ms
Per-Pixel auf CPU	144,257 ms
Per-Pixel STL auf CPU parallelisiert	30,360 ms
Per-Pixel STL auf GPU	337,379 ms
Per-Pixel STL auf GPU optimiert	185,078 ms

*Tabelle 4.2, Auflistung der gemessenen durchschnittlichen Ausführungszeiten in Millisekunden.*

Die mit einer GPU parallelisierten Implementierungen haben beide eine höhere durchschnittliche Ausführungszeit als die nicht parallelisierten Implementierungen. Durch Optimierung der Speichernutzung wurde die Anzahl der benötigten Speicher- und Ladebefehle reduziert. In Anhang 7.2. ist ein Diagramm aus den Profiling Ergebnissen zu finden, welches die Anzahl der aufgerufenen Instruktionen darstellt. Die Instruktionen LDG (Load from Global Memory), LD (Load from generic Memory) und ST (Store to generic Memory) sind hierbei die verwendeten Speicher- und Ladebefehle.[22] Daran wird erkennbar, dass die Anzahl dieser Instruktionen bei der optimierten Implementierung viel geringer ist. Somit führt die Reduzierung der Speicher- und Ladebefehle zu einer Verbesserung der Ausführungszeit der parallelisierten GPU-Implementierung.

### **4.3. Ergebnisdiskussion und Erkenntnisse**

#### **4.3.1. Bewertung der Ergebnisse**

Der Algorithmus ist nicht vollständig parallelisiert. Nur ein Teil des Algorithmus, welcher den größten Performance Einfluss hat, ist parallelisiert. Die Ausführung auf einer GPU verschlechtert hier die allgemeine Performance des Algorithmus. Die ursprüngliche Implementierung des Algorithmus ist für die Ausführung auf einer CPU vorgesehen und dementsprechend optimiert. Mithilfe der Parallel STL und der Nvidia Compiler ist es möglich Teile dieses CPU-Codes auf einer GPU auszuführen. Die mangelhaften Ausführungszeiten lassen sich dadurch begründen, dass die GPU-Implementierung nicht für die genutzte GPU Hardware optimiert wurde. Deshalb konnte die Ausführungszeit

bereits durch geringe Optimierung reduziert werden. Wenn weitere Maßnahmen ergriffen werden um die GPU-Implementierung besser für GPU Hardware zu optimieren, kann sich die Ausführungszeit weiter verbessern. Jedoch sind diese Optimierungen auf Standard C++ Code begrenzt, da der NVC++ Compiler keinerlei Optionen dazu bietet, die Generierung des GPU-Codes zu beeinflussen.[23] Eine beispielhafte Maßnahme könnte die Einführung einer Art Pipelining sein. So könnten die Daten des nächsten Frames schon auf den GPU Speicher kopiert werden, während die GPU mit der Berechnung der Polynomkoeffizienten beschäftigt ist.

#### **4.3.2. Gesammelte Erfahrungen und Erkenntnisse**

Durch die Algorithmen der Parallel STL ist es möglich sequenziellen Code schnell und unkompliziert zu parallelisieren oder von Anfang an parallelen Code zu schreiben. Für die Ausführung auf einer GPU muss auf Limitierungen in Software und Hardware geachtet werden. So kann die Parallel STL beispielsweise nicht auf GPU Architekturen älter als Pascal verwendet werden. Bei der Auswahl der „execution policies“ muss ebenfalls auf die GPU Architektur geachtet werden, da die Pascal Architektur nur das Objekt der „parallel\_unsequenced\_policy“ unterstützt. Wird das Objekt der „parallel\_policy“ verwendet, kompiliert der NVC++ Compiler sequenziellen CPU-Code. Außerdem beinhaltet der NVC++ Compiler nicht alle Header der Standard Library, weshalb fehlende Header extra inkludiert werden müssen.

Werden Algorithmen der Parallel STL auf einer GPU ausgeführt, erstellt der NVC++ Compiler für jeden Algorithmus einen Kernel. Da dies alles automatisch passiert, hat der Entwickler keinen direkten Einfluss auf die Parameter mit denen der Kernel erstellt wird. Zu diesen Parametern gehören beispielsweise die Anzahl von Threads pro Block oder die Anzahl von Blöcken pro Grid. Der NVC++ Compiler entscheidet diese Anzahl und legt sie für den Kernel fest. Außerdem werden sämtliche Datentransfers und Synchronisationen von CUDA Unified Memory verwaltet. Somit sind auch hier keinerlei Anpassungsmöglichkeiten vorhanden.

Zusammenfassend lässt sich sagen, dass das Kompilieren mit dem NVC++ Compiler herausfordernd ist. Es kann durchaus vorkommen, dass die Kompilierung mit internen Compilererrors fehlschlägt. Problematisch ist zudem, dass

Fehlermeldungen des NVC++ Compilers oft nicht aussagekräftig genug sind, um die Fehlerquelle sofort identifizieren zu können.

## **5. Zusammenfassung**

Um einen Algorithmus zu parallelisieren, wird zunächst herausgefunden, welcher Teil des Algorithmus am zeitaufwendigsten ist. Hierzu sind Laufzeitmessungen und deren Evaluierung unvermeidlich. Anhand der erhaltenen Messergebnisse können Entscheidungen über die zu parallelisierenden Funktionen getroffen werden. Bei der Parallelisierung und darauf folgender Kompilierung zeigt sich, dass eine bereits optimierte Implementierung, wie beispielsweise die des in OpenCV implementierten „dense optical flow“ Algorithmus, umgeschrieben werden muss, um sie daraufhin mit der Parallel STL zu parallelisieren. Probleme sind hierbei nicht unumgänglich, weshalb entschieden werden muss, ob die zu parallelisierende Funktion in etwa von der eigentlichen Implementierung abgekapselt oder anderweitig modifiziert werden sollte. Eine mit der Parallel STL parallelisierte Implementierung wird auf einer GPU ausführbar. Deren Optimierung kann des Weiteren dazu führen, dass die Laufzeit verbessert wird. Allerdings resultiert die Ausführung auf einer GPU nicht unbedingt in einer vollständigen Beschleunigung des anfänglich verwendeten Algorithmus.

Schlussfolgernd kann festgestellt werden, dass es durch die Kombination der Parallel STL mit dem Nvidia Toolkit grundsätzlich möglich ist, C++ Standard Algorithmen auf GPUs auszuführen. Dies führt jedoch nicht zwangsläufig zu einer Beschleunigung der Algorithmen, wie es anhand der Parallelisierung des Beispiels zur Berechnung des optischen Flusses durch die Parallel STL Algorithmen und dem Nvidia NVC++ Compiler erkennbar wird.

Die fortlaufende Optimierung des parallelisierten Algorithmus bildet jedoch eine Perspektive zur weiteren Senkung der durchschnittlichen Ausführungszeit. Neue Erkenntnisse könnten durch weitere Schritte und Versuche, wie in etwa die Parallelisierung von anderen im Automobilbereich verwendeten Algorithmen, gewonnen werden. Um weitere Möglichkeiten von parallelem C++ auf Hardwarebeschleunigern auszuschöpfen, könnte der Algorithmus mit OpenACC anstatt der Parallel STL parallelisiert werden und ebenfalls mit dem Nvidia Toolkit auf GPU ausführbar gemacht werden.

## 6. Literaturverzeichnis

- [1] *Algorithms library - cppreference.com*. [Online]. Verfügbar unter: <https://en.cppreference.com/w/cpp/algorithm> (Zugriff am: 7. Dezember 2021).
- [2] *C++ Parallel Algorithms Version 22.2 for ARM, OpenPower, x86*. [Online]. Verfügbar unter: <https://docs.nvidia.com/hpc-sdk/compilers/c++-parallel-algorithms/index.html> (Zugriff am: 27. Februar 2022).
- [3] C. Boyd, „Data-parallel computing“ in *ACM SIGGRAPH 2008 classes*, Los Angeles, California, 2008, S. 1, doi: 10.1145/1401132.1401150.
- [4] *std::execution::sequenced\_policy, std::execution::parallel\_policy, std::execution::parallel\_unsequenced\_policy, std::execution::unsequenced\_policy - cppreference.com*. [Online]. Verfügbar unter: [https://en.cppreference.com/w/cpp/algorithm/execution\\_policy\\_tag\\_t](https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t) (Zugriff am: 7. Februar 2022).
- [5] M. Voss, R. Asenjo und J. Reinders, „TBB and the Parallel Algorithms of the C++ Standard Template Library“ in *Pro TBB*, M. Voss, R. Asenjo und J. Reinders, Hg., Berkeley, CA: Apress, 2019, S. 109–136, doi: 10.1007/978-1-4842-4398-5\_4.
- [6] *std::for\_each - cppreference.com*. [Online]. Verfügbar unter: [https://en.cppreference.com/w/cpp/algorithm/for\\_each](https://en.cppreference.com/w/cpp/algorithm/for_each) (Zugriff am: 9. Februar 2022).
- [7] *std::for\_each\_n - cppreference.com*. [Online]. Verfügbar unter: [https://en.cppreference.com/w/cpp/algorithm/for\\_each\\_n](https://en.cppreference.com/w/cpp/algorithm/for_each_n) (Zugriff am: 9. Februar 2022).
- [8] *Stencil Operations on a GPU - MATLAB & Simulink Example*. [Online]. Verfügbar unter: <https://www.mathworks.com/help/parallel-computing/stencil-operations-on-a-gpu.html> (Zugriff am: 31. März 2022).
- [9] Jonas Latt und Christophe Coreixas, „C++ Parallel Algorithms for GPU Programming: A Case Study with the Lattice Boltzmann Method“, 2021.
















- [10] *John Conway's Game of Life - Einführung in Zellulare Automaten*. [Online]. Verfügbar unter: [https://beltoforion.de/de/game\\_of\\_life/](https://beltoforion.de/de/game_of_life/) (Zugriff am: 31. März 2022).
- [11] *std::transform - cppreference.com*. [Online]. Verfügbar unter: <https://en.cppreference.com/w/cpp/algorithm/transform> (Zugriff am: 9. Februar 2022).
- [12] *std::transform\_reduce - cppreference.com*. [Online]. Verfügbar unter: [https://en.cppreference.com/w/cpp/algorithm/transform\\_reduce](https://en.cppreference.com/w/cpp/algorithm/transform_reduce) (Zugriff am: 11. Februar 2022).
- [13] *Programming Guide :: CUDA Toolkit Documentation*. [Online]. Verfügbar unter: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> (Zugriff am: 1. Februar 2022).
- [14] A. R. Brodtkorb, T. R. Hagen und M. L. Sætra, „Graphics processing unit (GPU) programming strategies and trends in GPU computing“, *Journal of Parallel and Distributed Computing*, Jg. 73, Nr. 1, S. 4–13, 2013, doi: 10.1016/j.jpdc.2012.04.003.
- [15] NVIDIA Developer Blog, *Maximizing Unified Memory Performance in CUDA* | *NVIDIA Developer Blog*. [Online]. Verfügbar unter: <https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/> (Zugriff am: 14. Februar 2022).
- [16] NVIDIA Developer Blog, *Unified Memory for CUDA Beginners* | *NVIDIA Developer Blog*. [Online]. Verfügbar unter: <https://developer.nvidia.com/blog/unified-memory-cuda-beginners/> (Zugriff am: 14. Februar 2022).
- [17] NVIDIA Developer, *HPC SDK* | *NVIDIA*. [Online]. Verfügbar unter: <https://developer.nvidia.com/hpc-sdk> (Zugriff am: 27. Februar 2022).
- [18] *OpenCV: Introduction*. [Online]. Verfügbar unter: <https://docs.opencv.org/4.5.5/d1/dfb/intro.html> (Zugriff am: 31. März 2022).

- [19] GeeksforGeeks, *OpenCV - The Gunnar-Farneback optical flow - GeeksforGeeks*. [Online]. Verfügbar unter: <https://www.geeksforgeeks.org/opencv-the-gunnar-farneback-optical-flow/> (Zugriff am: 28. Februar 2022).
- [20] C. Lin, „Introduction to Motion Estimation with Optical Flow“, *AI & Machine Learning Blog*, 24. Apr. 2019, 2019. [Online]. Verfügbar unter: <https://nanonets.com/blog/optical-flow/>. Zugriff am: 28. März 2022.
- [21] G. Farnebäck, „Two-Frame Motion Estimation Based on Polynomial Expansion“ in *Scandinavian Conference on Image Analysis*, 2003, S. 363–370, doi: 10.1007/3-540-45103-X\_50.
- [22] *CUDA Binary Utilities :: CUDA Toolkit Documentation*. [Online]. Verfügbar unter: [https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#maxwell-pascal\\_\\_maxwell-pascal-instruction-set](https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#maxwell-pascal__maxwell-pascal-instruction-set) (Zugriff am: 1. April 2022).
- [23] *HPC Compilers User's Guide Version 22.3 for ARM, OpenPower, x86*. [Online]. Verfügbar unter: <https://docs.nvidia.com/hpc-sdk/compilers/hpc-compilers-user-guide/index.html#cmdln-opt-overview> (Zugriff am: 31. März 2022).

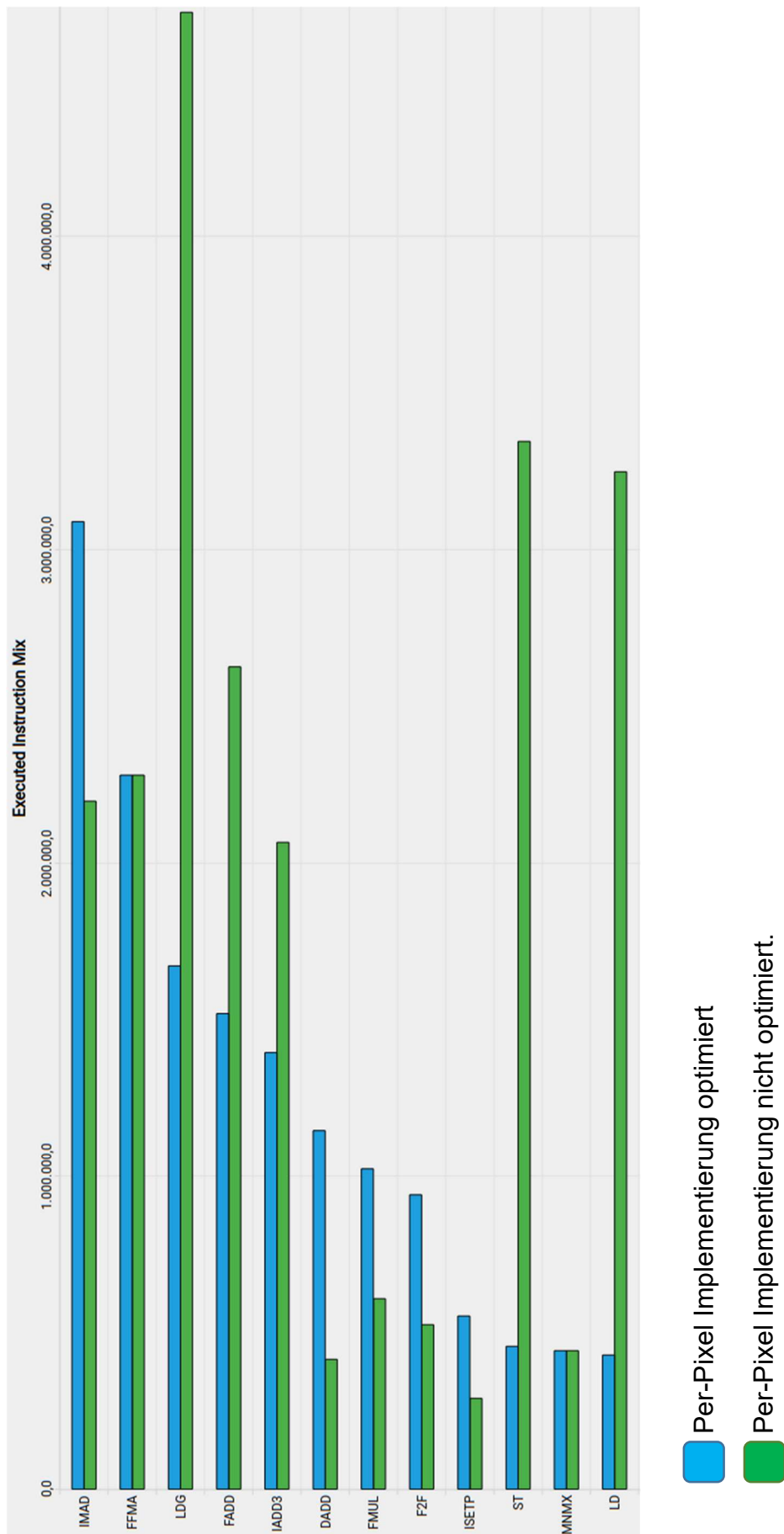
## 7. Anlagen

### 7.1. Ordnerstruktur der Beileg-CD

Der gedruckten Fassung liegt eine CD bei, welche sämtliche erarbeitete Programme und Diagramme enthält. Außerdem ist dort auch eine digitale Fassung dieser Arbeit zu finden. Die inhaltliche Struktur der CD sieht wie folgt aus:

 Python src	Beinhaltet alle Python Skripte.
 plots	Beinhaltet alle Plots und Diagramme
 docs	Beinhaltet diese Arbeit in PDF-Format
 measurements	Enthält alle Messergebnisse
 sample	Beinhaltet das verwendete Beispielvideo
 vtest_000	Enthält jeden Frame des Beispielvideos
 src	Enthält die angeführten Source Files
 CMakeList.txt	
 denseflow.cpp	
 optflow.cpp	
 polyExp-STL.cpp	
 CMakeList.txt	
 README.md	

## 7.2. Profiling Ergebnis



### 7.3. denseflow.cpp

```
#include <iostream>
#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include "optflowgf.cpp"
#include <filesystem>
#include <chrono>

using namespace cv;
using namespace std;
namespace fs = std::filesystem;

#if defined(_WIN32)
#define VIDEO "../sample/vtest_000/vtest_%03d.png"
#else
#define VIDEO "sample/vtest_000/vtest_%03d.png"
#endif

int main()
{
    cout << "start optflow" << endl;
    VideoCapture capture((fs::current_path() /
                          VIDEO).generic_string());

    if (!capture.isOpened()){
        //error in opening the video input
        cerr << "Unable to open file!" << endl;
        return 0;
    }
    //initialise first frame
    Mat frame1, prvs;
    capture >> frame1;
    //convert into Grayscale picture
    cvtColor(frame1, prvs, COLOR_BGR2GRAY);

    while(true){
        //initialize second frame
        Mat frame2, next;
        capture >> frame2;
        //check if sequence ended
        if (frame2.empty())
            break;
        //convert into Grayscale picture
        cvtColor(frame2, next, COLOR_BGR2GRAY);
        Mat flow(prvs.size(), CV_32FC2);

        calcOpticalFlowFarneback(prvs, next, flow, 0.5, 3, 15, 3, 5,
                                1.2, 0);

        // visualization
        Mat flow_parts[2];
        //split flow into multiple
        split(flow, flow_parts);
        Mat magnitude, angle, magn_norm;
```

```

        //calculate magnitude and angles
        cartToPolar(flow_parts[0], flow_parts[1], magnitude,
                    angle, true);
        normalize(magnitude, magn_norm, 0.0f, 1.0f, NORM_MINMAX);
        angle *= ((1.f / 360.f) * (180.f / 255.f));

        //build hsv image
        Mat _hsv[3], hsv, hsv8, bgr;
        _hsv[0] = angle;
        _hsv[1] = Mat::ones(angle.size(), CV_32F);
        _hsv[2] = magn_norm;
        merge(_hsv, 3, hsv);
        hsv.convertTo(hsv8, CV_8U, 255.0);
        cvtColor(hsv8, bgr, COLOR_HSV2BGR);
        imshow("frame2", bgr);
        int keyboard = waitKey(30);
        if (keyboard == 'q' || keyboard == 27)
            break;
        prvs = next;
    }
}

```

## 7.4. FarnebackPolyExp: OpenCV-Implementierung

```
static void
FarnebackPolyExp( const Mat& src, Mat& dst, int n, double sigma )
{
    int k, x, y;

    CV_Assert( src.type() == CV_32FC1 );
    int width = src.cols;
    int height = src.rows;
    AutoBuffer<float> kbuf( n*6 + 3 ), _row( (width + n*2)*3 );
    float* g = kbuf.data() + n;
    float* xg = g + n*2 + 1;
    float* xxg = xg + n*2 + 1;
    float *row = _row.data() + n*3;
    double igl1, ig03, ig33, ig55;

    FarnebackPrepareGaussian( n, sigma, g, xg, xxg, igl1, ig03,
                              ig33, ig55 );

    dst.create( height, width, CV_32FC(5) );
    for( y = 0; y < height; y++ )
    {
        float g0 = g[0], g1, g2;
        const float *srow0 = src.ptr<float>(y), *srow1 = 0;
        float *drow = dst.ptr<float>(y);
        // vertical part of convolution
        for( x = 0; x < width; x++ )
        {
            row[x*3] = srow0[x]*g0;
            row[x*3+1] = row[x*3+2] = 0.f;
        }
        for( k = 1; k <= n; k++ ) //k equals to Poly_n
        {
            g0 = g[k]; g1 = xg[k]; g2 = xxg[k];
            srow0 = src.ptr<float>(std::max(y-k,0));
            srow1 = src.ptr<float>(std::min(y+k,height-1));

            for( x = 0; x < width; x++ )
            {
                float p = srow0[x] + srow1[x];
                float t0 = row[x*3] + g0*p;
                float t1 = row[x*3+1] + g1*(srow1[x] - srow0[x]);
                float t2 = row[x*3+2] + g2*p;

                row[x*3] = t0;
                row[x*3+1] = t1;
                row[x*3+2] = t2;
            }
        }
    }
}
```

```

// horizontal part of convolution
// rowBuf padding left and right
for( x = 0; x < n*3; x++ )
{
    row[-1-x] = row[2-x];
    row[width*3+x] = row[width*3+x-3];
}
for( x = 0; x < width; x++ )
{
    g0 = g[0];
    // r1 ~ 1, r2 ~ x, r3 ~ y, r4 ~ x^2, r5 ~ y^2, r6 ~ xy
    double b1 = row[x*3]*g0, b2 = 0, b3 = row[x*3+1]*g0,
           b4 = 0, b5 = row[x*3+2]*g0, b6 = 0;

    for( k = 1; k <= n; k++ )
    {
        g0 = g[k];
        b1 += (row[(x+k)*3] + row[(x-k)*3])*g0;
        b2 += (row[(x+k)*3] - row[(x-k)*3])*xg[k];
        b4 += (row[(x+k)*3] + row[(x-k)*3])*xxg[k];
        b3 += (row[(x+k)*3+1] + row[(x-k)*3+1])*g0;
        b6 += (row[(x+k)*3+1] - row[(x-k)*3+1])*xg[k];
        b5 += (row[(x+k)*3+2] + row[(x-k)*3+2])*g0;

    }
    // do not store r1
    drow[x*5+1] = (float)(b2*ig11);
    drow[x*5] = (float)(b3*ig11);
    drow[x*5+3] = (float)(b1*ig03 + b4*ig33);
    drow[x*5+2] = (float)(b1*ig03 + b5*ig33);
    drow[x*5+4] = (float)(b6*ig55);
}
row -= n*3;
}

```



## 7.5. FarnebackPolyExp: Per-Pixel-Implementierung

```
static void
FarnebackPolyExpPP( const Mat& src, Mat& dst, int n, double sigma )
{
    int k, x, y;

    CV_Assert( src.type() == CV_32FC1 );
    int width = src.cols;
    int height = src.rows;
    AutoBuffer<float> kbuf(n*6 + 3);
    float* g = kbuf.data() + n;
    float* xg = g + n*2 + 1;
    float* xxg = xg + n*2 + 1;
    double igl1, ig03, ig33, ig55;

    FarnebackPrepareGaussian(n, sigma, g, xg, xxg, igl1, ig03,
                             ig33, ig55);

    dst.create( height, width, CV_32FC(5));
    auto _src = src.ptr<float>(0);
    auto _dst = dst.ptr<float>(0);
    for( y = 0; y < height; y++ )
    {
        for( x = 0; x < width; x++ )
        {
            float g0 = g[0];
            std::vector<float> rBuf((2 * n + 1)*3, 0.f);
            int offset = 2*n+1;

            for( int a = 0; a < 2*n+1; a++){
                int neighX = std::max(x + a-n, 0);
                neighX = std::min(neighX, width-1);
                rBuf[a] = _src[neighX + y * width] * g0;

                for(int b = 1; b <= n; b++){
                    int neighY0 = std::max((y-b)*width, 0);
                    int neighY1 = std::min((y+b)*width,
                                            (height-1)*width);

                    rBuf[a] += (_src[neighX + neighY0] +
                               _src[neighX + neighY1]) * g[b];

                    rBuf[a + offset] += (_src[neighX + neighY1] -
                                         _src[neighX + neighY0]) * xg[b];

                    rBuf[a + offset*2] += (_src[neighX + neighY0] +
                                           _src[neighX + neighY1]) * xxg[b];
                }
            }
            double b1 = rBuf[n]*g0, b2 = 0,
            b3 = rBuf[n + offset]*g0, b4 = 0,
            b5 = rBuf[n + offset*2]*g0, b6 = 0;
```

```

        for( int a = 1; a <= n; a++){
            b1 += (rBuf[n+a] + rBuf[n-a]) * g[a];
            b2 += (rBuf[n+a] - rBuf[n-a]) * xg[a];
            b4 += (rBuf[n+a] + rBuf[n-a]) * xxg[a];
            b3 += (rBuf[n+a+offset] + rBuf[n-a+offset]) * g[a];
            b6 += (rBuf[n+a+offset] - rBuf[n-a+offset]) * xg[a];
            b5 += (rBuf[n+a+offset*2] +
                    rBuf[n-a+offset*2]) * g[a];
        }

        int pixel = x+y*width;
        _dst[pixel*5+1] = (float) (b2*ig11);
        _dst[pixel*5] = (float) (b3*ig11);
        _dst[pixel*5+3] = (float) (b1*ig03 + b4*ig33);
        _dst[pixel*5+2] = (float) (b1*ig03 + b5*ig33);
        _dst[pixel*5+4] = (float) (b6*ig55);
    }
}

```

## 7.6. FarnebackPolyExp: Per-Pixel STL-Implementierung

```
static void
FarnebackPolyExpPPstl(const Mat& src, Mat& dst, int n, double sigma)
{
    CV_Assert( src.type() == CV_32FC1 );
    int width = src.cols;
    int height = src.rows;
    std::vector<float> kbuf(n*6 + 3);
    float* g = kbuf.data() + n;
    float* xg = g + n*2 + 1;
    float* xxg = xg + n*2 + 1;
    double ig11, ig03, ig33, ig55;

    FarnebackPrepareGaussian(n, sigma, g, xg, xxg, ig11, ig03,
                             ig33, ig55);

    dst.create( height, width, CV_32FC(5));
    auto _src = src.ptr<float>(0);
    auto src_ptr = src.ptr<float>(0);
    auto _dst = dst.ptr<float>(0);

    std::for_each(std::execution::par_unseq, _src, _src +
                  (width * height), [=](auto &pix){
        float g0 = kbuf[0+n];
        int xgOff = n + n*2 + 1;
        int xxgOff = xgOff + n*2 + 1;
        std::vector<float> rBuf((2 * n + 1)*3, 0.f);
        int offset = 2*n+1;

        auto index = &pix - src_ptr;
        int x = index % width;
        int y = index / width;

        for( int a = 0; a < 2*n+1; a++){
            int neighX = std::max(x + a-n, 0);
            neighX = std::min(neighX, width-1);
            rBuf[a] = _src[neighX + y * width] * g0;

            for(int b = 1; b <= n; b++) {
                int neighY0 = std::max((y - b) * width, 0);
                int neighY1 = std::min((y + b) * width,
                                         (height - 1) * width);

                rBuf[a] += (_src[neighX + neighY0] +
                           _src[neighX + neighY1]) * kbuf[b+n];

                rBuf[a + offset] += (_src[neighX + neighY1] -
                                     _src[neighX + neighY0]) * kbuf[b+xgOff];

                rBuf[a + 2 * offset] += (_src[neighX + neighY0] +
                                         _src[neighX + neighY1]) * kbuf[b+xxgOff];
            }
        }
    })
}
```

```

double b1 = rBuf[n]*g0, b2 = 0, b3 = rBuf[n + offset]*g0,
        b4 = 0, b5 = rBuf[n + 2*offset]*g0, b6 = 0;

for(int a = 1; a <= n; a++){
    b1 += (rBuf[n+a] + rBuf[n-a]) * kbuf[a+n];
    b2 += (rBuf[n+a] - rBuf[n-a]) * kbuf[a+xxgOff];
    b4 += (rBuf[n+a] + rBuf[n-a]) * kbuf[a+xxgOff];
    b3 += (rBuf[n+a+offset] + rBuf[n-a+offset]) * kbuf[a+n];
    b6 += (rBuf[n+a+offset] - rBuf[n-a+offset]) *
        kbuf[a+xxgOff];
    b5 += (rBuf[n+a+offset*2] + rBuf[n-a+offset*2]) *
        kbuf[a+n];
}

int pixel = x+y*width;
_dst[pixel*5+1] = (float) (b2*ig11);
_dst[pixel*5] = (float) (b3*ig11);
_dst[pixel*5+3] = (float) (b1*ig03 + b4*ig33);
_dst[pixel*5+2] = (float) (b1*ig03 + b5*ig33);
_dst[pixel*5+4] = (float) (b6*ig55);
});
}

```

## 7.7. FarnebackPolyExp: Per-Pixel STL-Implementierung optimiert

```
static void
FarnebackPolyExpPPstl2(const Mat& src, Mat& dst, int n,
                      double sigma)
{
    CV_Assert( src.type() == CV_32FC1 );
    int width = src.cols;
    int height = src.rows;
    std::vector<float> kbuf(n*6 + 3);
    float* _g = kbuf.data() + n;
    float* _xg = _g + n * 2 + 1;
    float* _xxg = _xg + n * 2 + 1;
    double igl1, ig03, ig33, ig55;

    FarnebackPrepareGaussian(n, sigma, _g, _xg, _xxg, igl1, ig03,
                             ig33, ig55);

    dst.create( height, width, CV_32FC(5));
    auto _src = src.ptr<float>(0);
    auto src_ptr = src.ptr<float>(0);
    auto _dst = dst.ptr<float>(0);

    std::for_each(std::execution::seq, _src, _src +
                  (width * height), [=](auto &pix){
        int xgOff = n + n*2 + 1;
        int xxgOff = xgOff + n*2 + 1;
        float g0 = kbuf[0+n];
        float rBuf = 0.f;
        float xrBuf = 0.f;
        float xxrBuf = 0.f;
        double b1 = 0, b2 = 0, b3 = 0, b4 = 0, b5 = 0, b6 = 0;
        auto index = &pix - src_ptr;
        int x = index % width;
        int y = index / width;

        for( int a = 0; a < 2*n+1; a++){
            int neighX = std::max(x + a-n, 0);
            neighX = std::min(neighX, width-1);
            rBuf = src_ptr[neighX + y * width] * g0;
            xrBuf = 0.f;
            xxrBuf = 0.f;

            for(int b = 1; b <= n; b++) {
                int neighY0 = std::max((y - b) * width, 0);
                int neighY1 = std::min((y + b) * width,
                                         (height - 1) * width);
                rBuf += (src_ptr[neighX + neighY0] +
                        src_ptr[neighX + neighY1]) * kbuf[b+n];
                xrBuf += (src_ptr[neighX + neighY1] -
                        src_ptr[neighX + neighY0]) * kbuf[b+xgOff];
                xxrBuf += (src_ptr[neighX + neighY0] +
                        src_ptr[neighX + neighY1]) * kbuf[b+xxgOff];
            }
        }
    });
}
```

```

float g = kbuf[abs(a-n) + n];
float xg = kbuf[abs(a-n) + xgOff];
float xxg = kbuf[abs(a-n) + xxgOff];
if(a == n){
    b1 += rBuf * g;
    b3 += xrBuf * g;
    b5 += xxrBuf * g;
} else if(a > n){
    b1 += rBuf * g;
    b2 += rBuf * xg;
    b3 += xrBuf * g;
    b4 += rBuf * xxg;
    b5 += xxrBuf * g;
    b6 += xrBuf * xg;
} else if(a < n){
    b1 += rBuf * g;
    b2 -= rBuf * xg;
    b3 += xrBuf * g;
    b4 += rBuf * xxg;
    b5 += xxrBuf * g;
    b6 -= xrBuf * xg;
}
}

int pixel = x+y*width;
_dst[pixel*5+1] = (float) (b2*ig11);
_dst[pixel*5] = (float) (b3*ig11);
_dst[pixel*5+3] = (float) (b1*ig03 + b4*ig33);
_dst[pixel*5+2] = (float) (b1*ig03 + b5*ig33);
_dst[pixel*5+4] = (float) (b6*ig55);
});
}

```

## 7.8. polyExp-stl.cpp

```
#include <iostream>
#include <chrono>
#include <algorithm>
#include <execution>
#include <fstream>

const size_t testWidth = 768;
const size_t testHeight = 576;
const size_t sampleSize = 300;
const int n = 5;

static void FarnebackPolyExpPPstl(const std::vector<float>& src,
                                  std::vector<float>& dst)
{
    int width = testWidth;
    int height = testHeight;
    std::vector<float> kbuf (n*6 + 3, 0.23);
    double ig11 = 0.12, ig03 = 0.14, ig33 = 0.13, ig55 = 1.23;

    auto src_ptr = &src[0];
    auto dst_ptr = dst.data();
    std::for_each(std::execution::par_unseq, src.begin(),
                  src.end(), [=](auto &pix) {
        int xgOff = n + n*2 + 1;
        int xxgOff = xgOff + n*2 + 1;
        float g0 = kbuf[0+n];
        std::vector<float> rBuf((2 * n + 1)*3, 0.f);
        int offset = 2*n+1;

        auto index = &pix - src_ptr;
        int x = index % width;
        int y = index / width;

        for( int a = 0; a < 2*n+1; a++){
            int neighX = std::max(x + a-n, 0);
            neighX = std::min(neighX, width-1);
            rBuf[a] = src_ptr[neighX + y * width] * g0;
            for(int b = 1; b <= n; b++) {
                int neighY0 = std::max((y - b) * width, 0);
                int neighY1 = std::min((y + b) * width,
                                         (height - 1) * width);
                rBuf[a] += (src_ptr[neighX + neighY0] +
                           src_ptr[neighX + neighY1]) * kbuf[b+n];
                rBuf[a + offset] += (src_ptr[neighX + neighY1] -
                                     src_ptr[neighX + neighY0]) * kbuf[b+xxgOff];
                rBuf[a + 2 * offset] += (src_ptr[neighX + neighY0] +
                                         src_ptr[neighX + neighY1]) * kbuf[b+xxgOff];
            }
        }
    })
}
```

```

double b1 = rBuf[n]*g0, b2 = 0, b3 = rBuf[n + offset]*g0,
        b4 = 0, b5 = rBuf[n + 2*offset]*g0, b6 = 0;
for(int a = 1; a <= n; a++){

    b1 += (rBuf[n+a] + rBuf[n-a]) * kbuf[a+n];
    b2 += (rBuf[n+a] - rBuf[n-a]) * kbuf[a+xcgOff];
    b4 += (rBuf[n+a] + rBuf[n-a]) * kbuf[a+xxcgOff];
    b3 += (rBuf[n+a+offset] + rBuf[n-a+offset]) * kbuf[a+n];
    b6 += (rBuf[n+a+offset] - rBuf[n-a+offset]) *
        kbuf[a+xcgOff];
    b5 += (rBuf[n+a+offset*2] + rBuf[n-a+offset*2]) *
        kbuf[a+n];
}

int pixel = x+y*width;
dst_ptr[pixel*5+1] = (float) (b2*ig11);
dst_ptr[pixel*5] = (float) (b3*ig11);
dst_ptr[pixel*5+3] = (float) (b1*ig03 + b4*ig33);
dst_ptr[pixel*5+2] = (float) (b1*ig03 + b5*ig33);
dst_ptr[pixel*5+4] = (float) (b6*ig55);

});
}

static void FarnebackPolyExpPPstl2(const std::vector<float>& src,
                                   std::vector<float>& dst)
{
    int width = testWidth;
    int height = testHeight;
    std::vector<float> kbuf (n*6 + 3, 0.23);
    double ig11 = 0.12, ig03 = 0.14, ig33 = 0.13, ig55 = 1.23;

    auto src_ptr = &src[0];
    auto dst_ptr = dst.data();
    std::for_each(std::execution::par_unseq, src.begin(),
                  src.end(), [=] (auto &pix){
        int xcgOff = n + n*2 +1;
        int xxcgOff = xcgOff + n*2 +1;
        float g0 = kbuf[0+n];
        float rBuf;
        float xrBuf;
        float xxrBuf;

        double b1 = 0, b2 = 0, b3 = 0, b4 = 0, b5 = 0, b6 = 0;

        auto index = &pix - src_ptr;
        int x = index % width;
        int y = index / width;

```



```

for( int a = 0; a < 2*n+1; a++){
    int neighX = std::max(x + a-n, 0);
    neighX = std::min(neighX, width-1);
    rBuf = src_ptr[neighX + y * width] * g0;
    xrBuf = 0;
    xxrBuf = 0;
    for(int b = 1; b <= n; b++) {
        int neighY0 = std::max((y - b) * width, 0);
        int neighY1 = std::min((y + b) * width,
                                (height - 1) * width);
        rBuf += (src_ptr[neighX + neighY0] +
                 src_ptr[neighX + neighY1]) * kbuf[b+n];
        xrBuf += (src_ptr[neighX + neighY1] -
                  src_ptr[neighX + neighY0]) * kbuf[b+xxgOff];
        xxrBuf += (src_ptr[neighX + neighY0] +
                   src_ptr[neighX + neighY1]) * kbuf[b+xxgOff];
    }
    float g = kbuf[abs(a-n) + n];
    float xg = kbuf[abs(a-n) + xgOff];
    float xxg = kbuf[abs(a-n) + xxgOff];
    if(a == n){
        b1 += rBuf * g;
        b3 += xrBuf * g;
        b5 += xxrBuf * g;
    } else if(a > n){
        b1 += rBuf * g;
        b2 += rBuf * xg;
        b3 += xrBuf * g;
        b4 += rBuf * xxg;
        b5 += xxrBuf * g;
        b6 += xrBuf * xg;
    } else {
        b1 += rBuf * g;
        b2 -= rBuf * xg;
        b3 += xrBuf * g;
        b4 += rBuf * xxg;
        b5 += xxrBuf * g;
        b6 -= xrBuf * xg;
    }
}
int pixel = x+y*width;
dst_ptr[pixel*5+1] = (float) (b2*ig11);
dst_ptr[pixel*5] = (float) (b3*ig11);
dst_ptr[pixel*5+3] = (float) (b1*ig03 + b4*ig33);
dst_ptr[pixel*5+2] = (float) (b1*ig03 + b5*ig33);
dst_ptr[pixel*5+4] = (float) (b6*ig55);
});
}

```

```

int main() {
    srand(43156844);
    /*
    std::ofstream myFile;
    myFile.open ("example.txt");
    myFile << "Timings\n";
    myFile.close();
    */
    std::cout << "calculating polynomial coefficients for "
                << sampleSize << " Frames..." << std::endl;
    for (int z = 0; z < sampleSize ; ++z) {
        //create src vector with random sample data
        std::vector<float> src (testHeight * testWidth);
        for(float & i : src){
            i = float(rand() % 255 + 30);
        }
        std::vector<float> dst ((testHeight*testWidth)*5);
        //auto start = std::chrono::steady_clock::now();
        FarnebackPolyExpPPstl2(src, dst);
        //auto end = std::chrono::steady_clock::now();
        //auto duration = std::chrono::duration_cast
                        <std::chrono::duration<double,std::milli>>
                        (end - start).count();

        /*
        myFile.open ("example.txt",  std::ios_base::app);
        myFile << duration << "\n";
        myFile.close();
        */
    }
    return 0;
}

```