# Einführung

## Motivation

## Zielsetzung

## Kapitelübersicht

# Grundlagen

## Paralleles Programmieren

### Daten Parallelität

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

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

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

Compute bound and memory bound

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

### Parallel STL

#### Standard Algorithms

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

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

Die für diese Arbeit nützlichsten Algorithmen der C++ STL sind „std::for\_each“, „std::for\_each\_n“, „std::transform“ und „std::transform\_reduce“.

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

Der Algorithmus „std::for\_each“ hat mehrere Interfaces. Das für diese Arbeit wichtige Interface sieht wie folgt aus:



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

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



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

1. std::transform

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



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

Das Interface sieht wie folgt aus:



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

(std:transform - cppreference.com, 2022)

1. std::transform\_reduce

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



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



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

#### Execution Policies

Seit C++17 wurden die Standard Algorithms mit sogennanten execution policies erweitert. Viele Algorithmen haben nun Überladungen, welche Objekte dieser execution policies akzeptieren.

Zu diesen execution policies gehören:

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

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

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

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

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

was ist vectorisieren?

#### Typische anwendung von Parallelem Code

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

Helligkeit in einem Bild ändern.

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



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

Transform\_reduce?

Stencil operationen

Stencil Operationen brauchen meist Informationen zu benachbarten Werten in einem zweidimensionalen Array um Berechnungen durchzuführen oder Arraywerte miteinander zu vergleichen.

Beispiel zweidimensionale Wärmeleitgleichung:



Erklären was einelne schritte machen oder nur den Stencil teil?

Foreach?

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

### GPU

##### Hardwarearchitektur

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

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

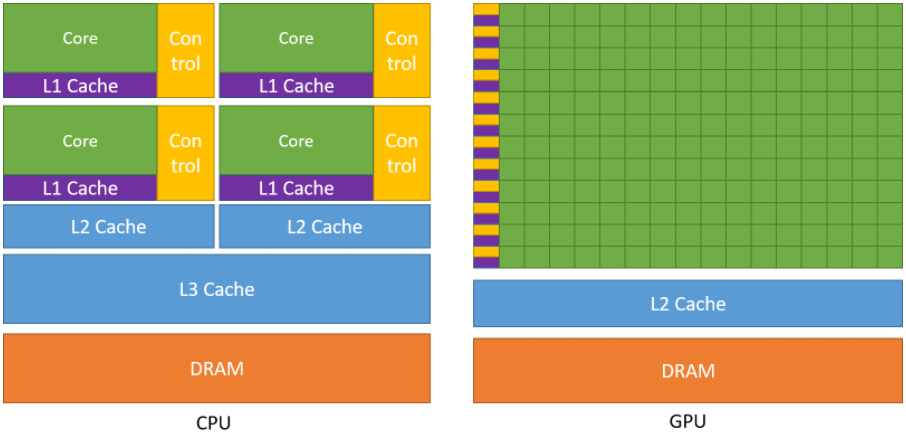


Abbildung . Die GPU dediziert mehr Kerne für die Datenverarbeitung. (Programming Guide : CUDA Toolkit Documentation, 2022)

##### Speicherverwaltung / Heterogene Systeme

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

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

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

### CUDA

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

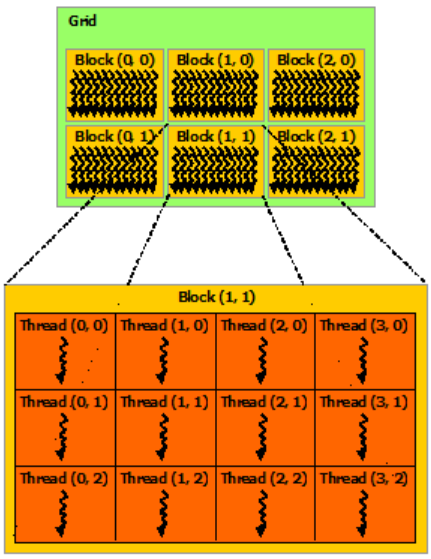


Abbildung . Ein Gitter (engl. Grid) ist unterteilt in Blöcke (engl. Blocks) von Threads. (Programming Guide : CUDA Toolkit Documentation, 2022)

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

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

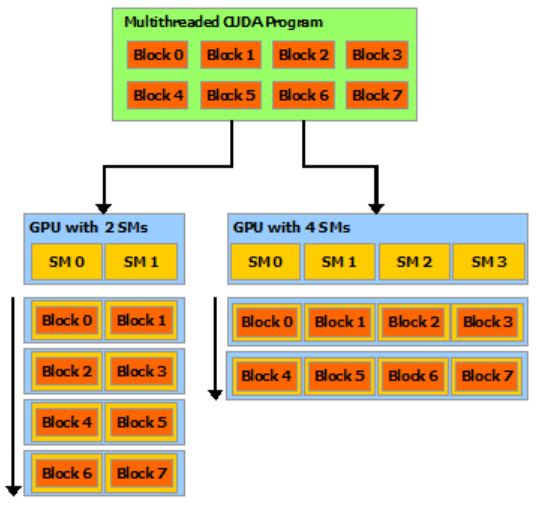


Abbildung 3. Automatische Skalierung durch Aufteilung der Blöcke auf die Verschiedenen Streaming Multiprozessoren. (Programming Guide : CUDA Toolkit Documentation, 2022)

#### CUDA Unified Memory

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

Veranschaulicht durch Abbildung 4:

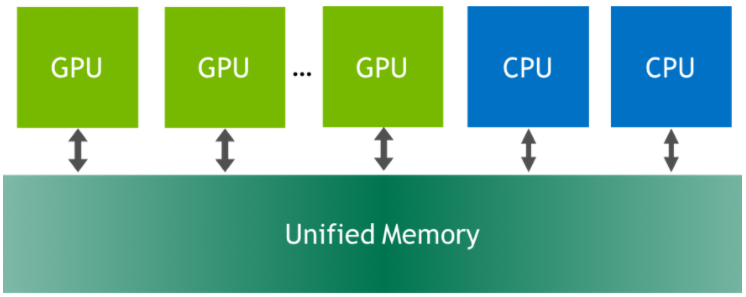


Abbildung . Unified Memory von allen Prozessoren abrufbar. (NVIDIA Developer Blog, 2017a)

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

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

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

### (Nvidia HPC)

Position in der Gliederung noch nicht sicher.

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

## **(Algorithmus**)

Kurz (halbe seite)! Was macht der Algorithmus

# Lösungsansatz / Versuchsaufbau

## Anforderungsanalyse / IST Zustand /

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

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

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

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

(Berechnung veranschaulichen?)

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

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

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

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

a : Anzahl der Funktionsaufrufe pro Aufruf von calcOpticalFlowFarneback.

n : Anzahl der Aufrufe von calcOpticalFlowFarneback Aufrufen.

t : Ausführungszeit eines Funktionsaufrufs.

ist das arithmetische Mittel der Ausführungszeit eines Funktionsaufrufs.

Ergebnisse der Messungen

Mesungen vom orginal code

Auch nach mehreren Ausführungen von „densFlow“ ist keine nennenswerte Änderung der Zeiten zu sehen. Daraus lässt sich schließen, dass im Durchschnitt eine Ausführung FarnebackPolyExp die meiste Zeit benötigt.

## Lösungsansätze / Probleme

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

Erster Lösungsansatz ->schleifen übersetzen in STL befehle  
Zweiter Ansatz -> Berechnung in pro Pixel umwandeln. -> dann pro pixel in STL.

Evaluierung der Ansätze ->

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

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

## Ausgewählter Ansatz und durchführung

Welcher ansatz wurde gewählt. -> beschreibung pro pixel STL code -> nutzung von Stencil operation wegen natur der funktion. -> warum dieser STL algo -> Probleme wegen der Nutzung von Opencv – klassen -> abstrahierung des Problems -> (Validierung dass mein code immernoch das gleiche macht wie im orginal?)

# Ergebnisse

(Validierung dass mein code immernoch das gleiche macht wie im orginal?) -> standard fehler berechnen

Darstellung der messergebnisse

Übersicht über die ergebnisse -> Bewertung

# Zusammenfassung

References

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

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

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

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

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

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

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

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

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

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

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

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

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