# 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 (…) genannt. Um (…) 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)

Ein paar der nützlichsten allgorithms:

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“. Grundsätzlich ähneln die akzeptierten Parameter von „std::transform“ denen von „std::for\_each“.

Das Interface für „std::transform“ sieht wie folgt aus:





(std:transform - cppreference.com, 2022)

Std::transform\_reduce

(Voss et al., 2019)

#### Execution Policies

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

#### Typische anwendung von Parallelem Code

Stencil operations, Bildverarbeitung,

## (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 Kernen besteht, ist eine GPU mit hunderten einfacheren Kernen ausgestattet. Diese Kerne besitzen jeweils tausende parallel laufende Hardware Threads.(Brodtkorb et al., 2013, 2013) Abbildung 1 veranschaulicht den Aufbauunterschied.

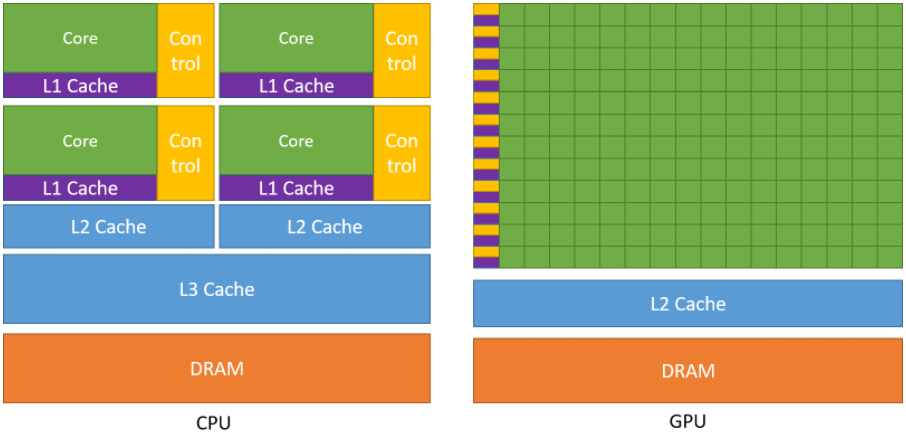


Abbildung . Die GPU dediziert mehr Kerne für die Datenverarbeitung. (Programming Guide : CUDA Toolkit Documentation, 2022)

##### Speicherverwaltung / Heterogene Systeme

Zusammenarbeit zwischen CPU und GPU in bezug auf Speicherverwaltung. Cache etc.

### 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.

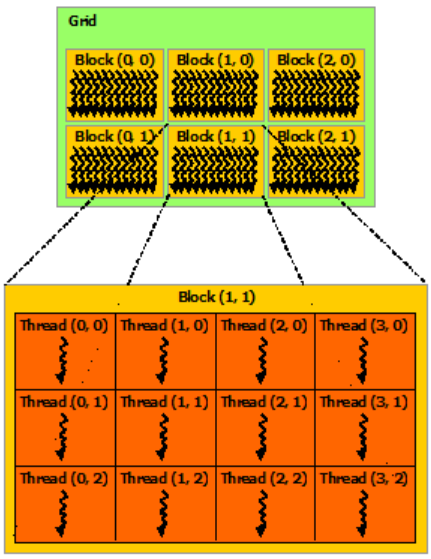


Abbildung . Ein Gitter (engl. Grid) ist unterteilt in Blöcke (engl. Blocks) von Threads. (Programming Guide : CUDA Toolkit Documentation, 2022)

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)

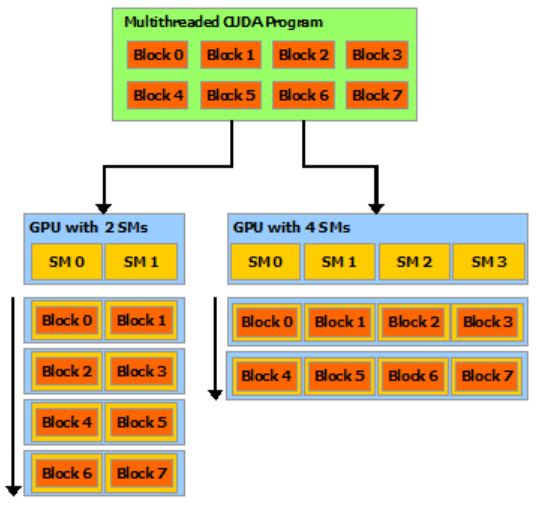


Abbildung 3. Automatische Skalierung durch Aufteilung der Blöcke auf die Verschiedenen Streaming Multiprozessoren. (Programming Guide : CUDA Toolkit Documentation, 2022)

#### CUDA Unified Memory

## **(Algorithmus**)

# Lösungsansatz / Versuchsaufbau

# Ergebnisse

# 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.

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).

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.