# GPU-Implementierung des Modells der strukturellen Plastizität

**Softwaredokumentation – Gruppe 2**

Alexander Heeg <alexander.heeg@stud.tu-darmstadt.de>
Finn Karpenstein <finn.karpenstein@stud.tu-darmstadt.de>
Julian Ewald <julian.ewald@stud.tu-darmstadt.de>
Marwin Kreuzig <marwin.kreuzig@stud.tu-darmstadt.de>
Maximilian Michael Steinbach <maximilian.steinbach@stud.tu-darmstadt.de>

Teambegleitung:
Roni Aleksanian <r.aleksanian@hotmail.de>

Auftraggeber:
Marvin Kaster <marvin.kaster@tu-darmstadt.de>
FG Parallele Programmierung
Fachbereich Informatik (FB 20)
15. März 2024

TECHNISCHE UNIVERSITÄT DARMSTADT

Bachelorpraktikum
Wintersemester 2023/24
Fachbereich Informatik

# Inhaltsverzeichnis

# 1 Grundlegendes

## 1.1 Kurze Einleitung

In diesem Bachelor-Praktikum haben wir damit begonnen, die effiziente Ausführung des RELeARN-Projekts auf der GPU zu ermöglichen.
Wir haben dabei nicht alle User-Storys, die wir uns vorgenommen haben, geschafft (siehe Abschnitt 3 „Zukünftige Weiterentwicklung“), obwohl wir die im Rahmen des BP erwartete Stundenanzahl (1350 h) sogar etwas überschritten haben. Das liegt vor allem an:

- der sehr langen Einarbeitungszeit in die Thematik und die gegebene Codebasis
- zahlreichen aufgetretenen Fehlern und nötigen Umstrukturierungen (von denen einige auch in diesem Dokument beschrieben werden)
- dem temporären Ausfall des Lichtenbergclusters aufgrund eines Brands und der deswegen notwendigen lokalen Einrichtung des Projekts

Bisher haben wir, wie vom AG vorgegeben, alle User-Storys nur für einen MPI-Rang umgesetzt.
In dieser Dokumentation möchten wir den geschriebenen Code sowie die vorgenommenen Umstrukturierungen erklären und aufzeigen, was noch zu tun ist, um die Migration auf die GPU abzuschließen.

## 1.2 Allgemeine Infos

Der größte Teil des von uns geschriebenen Codes befindet sich im Ordner `relearn/source/gpu`.
`./relearn` lässt sich für die GPU mit den gleichen Argumenten ausführen wie für die CPU. Um zwischen der GPU- und der CPU-Version zu wechseln, muss lediglich `ENABLE_CUDA` in der Datei `relearn/cmake/Cuda.cmake` auf ON bzw. OFF gesetzt werden.

Wie in C++ üblich, haben wir die Deklarationen für die meisten Methoden in Headerdateien geschrieben (für Cuda `.cuh`-Dateien) und die eigentlichen Implementationen in separate `.cu`-Dateien. Wir haben alle geschriebenen Methoden in den Headerdateien kommentiert mit der folgenden, auch im bisherigen Projekt verwendeten Notation:

```
/**
 * @brief Kurzbeschreibung der Methode
 * @pre Vorbedingungen, die gelten müssen, damit die Methode funktioniert wie
 ↪  gewünscht
 * @param Parameter
 * @exception mögliche geworfene Exceptions
 * @return Rückgabewert
 */
```

Zu allen implementierten User-Storys gibt es Tests und Benchmarks:

Tests: Die GPU-Tests, die mit dem nvcc kompiliert werden, liegen im Ordner `relearn/tests_cuda`, und lassen sich mit bzw. `./relearn_tests_cuda` ausführen.
Es gibt aber auch viele Tests, bei denen es bei einer Kompilierung mit dem nvcc zu Konflikten mit neueren C++-Funktionalitäten (z.B. der `concepts`-Library) kommt, mit denen Cuda nicht umgehen kann; diese Tests werden mit gcc kompiliert und liegen im Ordner `relearn/test/gpu`. Sie werden zusammen mit den CPU-Tests bei `./relearn_tests` ausgeführt.

Benchmarks: Die GPU-Benchmarks liegen im Ordner `relearn/benchmark_cuda` und werden bei `./relearn_benchmarks_cuda` ausgeführt. (Die CPU-Benchmarks lassen sich weiterhin mit `./relearn_benchmarks` ausführen.)

Bei der Struktur der Tests und Benchmarks für die GPU haben wir uns an der CPU-Struktur orientiert, außer beim Oktalbaum, weil es hier kein CPU-Äquivalent gibt.

## 1.3 Handles und die Czappa-Struktur

Bei vielen User-Storys haben wir auf Marvin Kasters Vorschlag hin sogenannte Handles verwendet. Der Zweck eines Handles ist es, in CPU-Code auf Daten zuzugreifen, die auf der GPU liegen.
Handles bestehen aus mehreren Komponenenten:

- einer virtuellen Klasse in der Datei `relearn/source/gpu/utils/Interface.h` (z.B. `OctreeHandle`)
- einer Klasse in einer Cudadatei, die von der virtuellen Klasse abgeleitet ist und deren Methoden implementiert (z.B. `OctreeHandleImpl` in `relearn/source/gpu/structure/Octree.cu`)
- einer create-Methode, die in `Interface.h` deklariert und in einer Cudadatei implementiert wird (z.B. `create_octree()`). Mit dieser Methode lassen sich Instanzen des Handles für die CPU erstellen.

Außerdem haben wir für eine valide Initialisierung der Kernels bei der Berechnung der Calciumkonzentration, beim Barnes-Hut und den Neuronenmodellen folgende von Fabian Czappa vorgeschlagene Struktur verwendet:

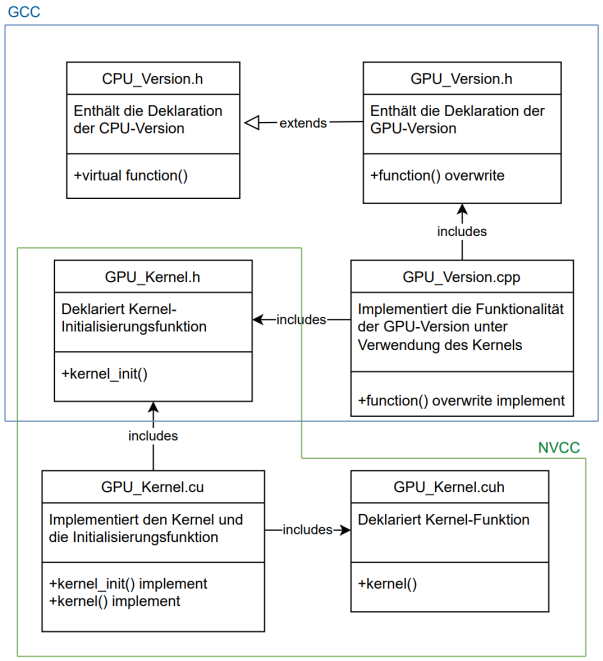


Abbildung 1.1: Darstellung des Zusammenhangs zwischen C++-Code ohne Cuda und Cuda-Code, um korrektes Kompilieren zu ermöglichen

# 2 Projektzustand

## 2.1 Oktalbaum

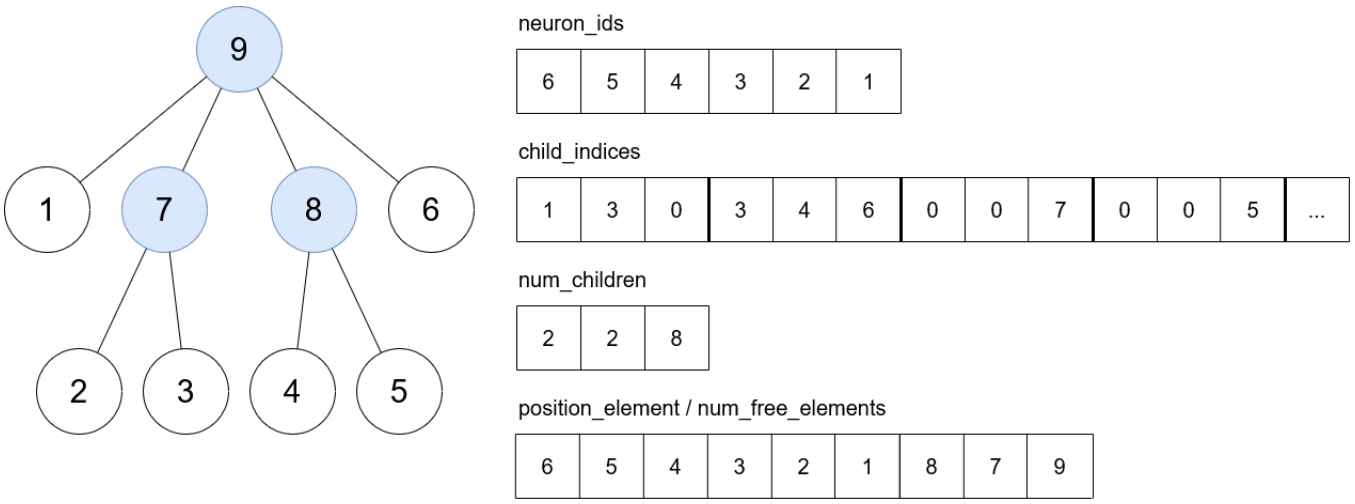


Abbildung 2.1: Octree-Datenstruktur auf der GPU

Ein wichtiger vorbereitender Schritt für den Barnes-Hut-Algorithmus war es, den Octree auf die GPU zu kopieren. Die hierfür relevanten Dateien befinden sich großteils im Ordner `relearn/source/gpu/structure`.

Auf der CPU ist der Octree als Pointerstruktur gespeichert, bei der jeder Knoten des Baums einen Pointer zu seinem Elternknoten hat. Die auf der CPU vorhandene Datenstruktur ist aber ungeeignet für effiziente Berechnungen auf der GPU, daher haben wir eine eigene Datenstruktur für die GPU-Version erstellt.

In Abbildung 2.1 ist der GPU Octree dargestellt. Weiße Kreise stellen (reale) Neuronen dar und blaue Kreise virtuelle Neuronen. Die Zahlen innerhalb der Kreise ist die NeuronID. Neben dem Octree sind die einzelnen Arrays der Datenstruktur. Die Länge von `neuron_ids` entspricht der Anzahl der Neuronen. Die Neuronen-IDs werden von rechts nach links in dem Array abgelegt.

Das Array `child_indices` hat die Länge „Anzahl virtueller Neuronen · 8“. In diesem Array werden die Indizes der Kinder der einzelnen virtuellen Neuronen abgelegt. Hierbei ist wieder zu beachten, dass die Neuronen von rechts nach links indiziert werden (Neuron mit ID 6 hat Index 0, Neuron mit ID 5 hat Index 1, ...). Der erste Block [1, 3, 0] repräsentiert alle ersten Kinder der virtuellen Neuronen. Das heißt, das virtuelle Neuron mit ID = 8 hat das Neuron mit Index 1 (hier ist es das Neuron mit ID 5) als erstes Kind.

Das Array `num_children` speichert die Anzahl der Kinder der einzelnen virtuellen Neuronen. Die Reihenfolge ist hierbei wieder von rechts nach links (ID 8, ID 7, ID 9).

Die Arrays für die Position der Elemente und Anzahl der freien Elemente hat die Länge „Anzahl Neuronen + Anzahl virtueller Neuronen“. Die Werte werden wieder von links nach rechts abgelegt.

Um die CPU-Datenstruktur in die GPU-Datenstruktur zu überführen, verwenden wir eine Zwischendarstellung namens `OctreeCPUCopy` in der Datei `OctreeStructure.h` erstellt. Diese Zwischendarstellung entspricht der finalen Datenstruktur auf der GPU, welche in `Octree.h` implementiert wurde.

Mittels der in `Octree.h` deklarierten Methode `octree_to_octree_cpu_copy` wird der CPU Octree in einen `OctreeCPUCopy` umgewandelt. Diese neue Datenstruktur besteht aus mehreren Arrays, welche die relevanten Daten des Octrees beinhalten. Danach wird ein Octree auf der GPU erstellt und die Arrays des `OctreeCPUCopy` kopiert.

Der Octree auf der GPU bleibt hierbei zugänglich mittels eines Handels der in `relearn/source/gpu/utils/Interface.h` deklariert wurde.

## 2.2 Barnes-Hut-Algorithmus

Für Barnes Hut wurde der Ansatz gewählt, die Czappa-Struktur zu verwenden, da sich der Barnes Hut hauptsächlich um zwei Funktionalitäten kümmert, die überschrieben werden können. Diese Funktionalitäten sind der Octree Update und der Connectivity Update. Alle Daten, die für den Barnes-Hut-Algorithmus auf der GPU gebraucht werden, werden durch einen Handle verwaltet. Der Barnes-Hut-Algorithmus selbst funktioniert auch sehr ähnlich wie auf der CPU. Für die GPU-Version lässt sich jedoch zusätzlich noch sowohl die Anzahl der Neuronen, die pro Thread bearbeitet werden sollen, manuell spezifizieren, als auch ein anderer Wert namens `num_nodes_gathered_before_pick`. Dieser Wert wurde eingeführt, da der benötigte Speicherplatz sehr groß werden kann, wenn man für jeden Thread auf der GPU alle nodes bei der prefix traversal erstmal sammelt, bevor man einen von ihnen als das Target aussucht. `Num_nodes_gathered_before_pick` sagt damit quasi, wie viele nodes maximal gesammelt werden können bevor man einen von ihnen auswählt. Danach wird weiter mit dem prefix traversal gemacht, wobei man nun die memory für die gathered nodes und das probability interval wieder von vorne verwenden kann.

Um das beste Verhältnis zwischen `num_nodes_gathered_before_pick` und `neurons_per_thread` zu bestimmen, müssen am besten einfach sehr viele Benchmarks gemacht werden. Eine gute Konfiguration, die wir zum Beispiel gefunden hatten, waren bei 800.000 Neuronen 4 Neuronen pro Thread und 10.000 `num_nodes_gathered_before_pick`. Bei dieser Konfiguration war die Runtime 1468 ms (bei dem Run wurden die meisten Runtime-Korrektheitschecks auskommentiert, da diese die Performance etwas verschlechtern) gegen eine einzelne MPI Prozess CPU Runtime von 1 405 455 ms . Damit ist die Runtime der GPU-Version 957 mal schneller als ein CPU core. Dies ist jedoch mit dem Vorbehalt, dass hier nur der Barnes-Hut- Algorithmus in Isolation gerechnet wurde. Andere Daten, die nicht direkt für den Algorithmus auf der GPU gebraucht werden, wie zum Beispiel der Netzwerkgraph, waren nicht auf der GPU gespeichert und würden normalerweise die vorhandene Memory für den Barnes Hut noch etwas drücken, was auch die Performance beeinflussen würde.

Auf dem Lichtenbergcluster befinden sich in der MPI Sektion der ersten Ausbaustufe insgesamt 630 Knoten mit je 96 cores. In der Accelerator Section befinden sich insgesamt 32 GPUs. Wenn wir davon ausgehen, das

wir alle GPUs auslasten, haben wir damit eine Leistung von ungefähr 319 CPU Knoten erreicht, was noch einiges weniger ist, als die 630 Knoten, die zu Verfügung stehen, für die maximale Leistung. Wir würden also vermuten, dass dies noch nicht ganz der gewünschten Performance entspricht.

Es wurden verschiedene Methoden probiert, um den Barnes-Hut-Algorithmus auf der GPU noch mehr zu beschleunigen, jedoch scheiterten die meisten davon. Viele der Dinge, die in dem Paper von Burtscher et. al. verwendet wurden, in dem Barnes Hut auch auf der GPU implementiert wurde, konnten hier nicht wiederverwendet werden, aufgrund des quasi rekursiven Aufbaus des Algorithmus auf Neuronen, bei dem nach der Auswahl eines virtuellen Neurons auf diesem weiter gesucht werden muss. Der größte Verbesserungsversuch, der an das Paper angelehnt war, war der Versuch, Threads, die das prefix des geichen root nodes traversieren, zu bündeln, so dass sie das shared memory für ihr prefix gemeinsam nutzen können. Leider hatte dieser Ansatz keinen Erfolg und hat die Laufzeit sogar etwas verschlechtert (vermutlich ist der match_any_sync overhead zu groß gewesen). Ein anderer Versuch, den größten prefix des root nodes in shared memory zu behalten war auch nicht erfolgreich. Im Allgemeinen erschien es uns so, als wäre der Algorithmus selbst nicht sehr gut geeignet für die GPU, zumindest braucht es vermutlich eine schlauere Person, um das ganze noch effizienter hin zu bekommen. Geachtet wurde aber immer darauf, dass das global memory möglichst coalesced gelesen und geschrieben wird, und dies machte auch große Unterschiede bei der Runtime.

Implementiert wurde hier nur der Barnes Hut, in dem Axone Dendriten suchen (also nicht `BarnesHutverted`).

## 2.3 Neuronenmodelle

Achtung: Wir haben keine Tests zu dieser User Story geschrieben, weil wir diese User Story als eine der letzten begonnen haben und die Zeit nicht gereicht hat, um neue zu schreiben. Die CPU-Tests waren leider unbrauchbar für diesen Zweck.

Die bisherige Implementation der Neuronenmodelle nutzte virtuelle Funktionen auf der GPU, um zwischen den verschiedenen Modellen zu wechseln. Aufgrund der generellen Ineffizienz dieser Lösung folgten wir Fabian Czappas Vorschlag und überarbeiteten die Implementation. Unsere neue Lösung verwendet die Czappa-Struktur. Wir überschreiben die relevanten Methoden der Klasse `NeuronModel` in `NeuronModelGPU` und den davon abstammenden Klassen für die einzelnen Modelle.

Für die relevanten Daten nutzten wir zwei Handles, die beide in `NeuronModelGPU` gespeichert sind. Der `NeuronModelDataHandle` verwaltet die Daten, die alle Neuronenmodelle teilen, während der `models::ModelDataHandle` die Daten für die spezifischen Modelle hält. Für jedes Neuronenmodell gibt es daher eine Klasse, die von `models::ModelDataHandle` abstammt.

## 2.4 Calciumberechnung

### 2.4.1 Allgemein

Hier verwendeten wir die Czappa-Struktur. Um die Berechnung der neuen Calciumkonzentrationen in jedem Schritt zu beschleunigen, überschrieben wir die Methoden `update_current_calcium` und `update_target_calcium` in der Klasse `CalciumCalculatorGPU`.

Um Dataraces zu vermeiden, brauchten wir atomic-Methoden zur Ermittlung des Minimums bzw. Maximums mehrerer Calciumwerte. Da `atomicMin()` und `atomicMax()` aber nicht für Dezimalzahlen funktionieren, schrieben wir in der Datei `relearn/source/gpu/Commons.cuh` eigene Methoden `atomicMinimum()` und `atomicMaximum()`, die das berechnen. Diese funktionieren wie folgt:

1. Der aktuelle Wert aus der Adresse des Minimal-/Maximalwerts wird geladen
2. Der eigene Wert wird mit diesem Wert verglichen, wenn nicht erfüllt, wird abgebrochen
3. Sollte der Wert kleiner bzw. größer sein, so wird ein `atomicCAS` ausgeführt. Sei nun der Wert der Adresse, den wir in Punkt 1 gelesen haben, `assumed`. Die Funktion führt nun in einem atomaren Schritt folgendes aus:
   a) Lade den aktuellen Wert an der Adresse.
   b) Vergleiche diesen Wert mit `assumed`.
   c) Wenn der Wert gleich ist, dann ersetze den Wert der Adresse mit dem übergebenen Wert (der Calciumwert des Threads). Ansonsten ändere nichts (bzw. ersetze den Wert mit dem gleichen Wert). In beiden Fällen gebe den aktuellen Wert der Adresse vor dem Ersetzen zurück.
   d) Solange die Rückgabe gleich `assumed` ist wiederhole ab Schritt 1.

### 2.4.2 Datenoptimierung

Um die hierfür relevanten Daten nicht in jedem Schritt zwischen CPU und GPU hin- und herkopieren zu müssen, wurden die Daten zu Beginn auf die GPU kopiert und die Adressen über Handles erfasst und an die Kernels übergeben. Hierbei wurden die folgenden drei Datenstrukturen mittels Handles abgerufen:

1. `calcium` bzw. `target_calcium` durch `CalciumCalculatorDataHandle`
2. `fired_status` durch `NeuronModelDataHandle`
3. `disabled_flags` durch `NeuronsExtraInfos`

Wobei `CalciumCalculatorData` auf eine eigens hierfür erstellte Datenstruktur zugreift, die folgende Werte speichert:

- `calcium`: Array, welcher die Calciumwerte für jedes Neuron speichert
- `target_calcium`: Array, welcher die zukünftigen Calciumwerte für jedes Neuron speichert
- `minimum_ca`: Threadübergreifender minimaler Calciumwert
- `maximum_ca`: Threadübergreifender maximaler Calciumwert
- `minimum_id`: Neuronenidentifikationsnummer des Neurons mit minimalem Calciumwertes
- `maximum_id`: Neuronenidentifikationsnummer des Neurons mit maximalem Calciumwertes

## 2.5 Netzwerkgraph (noch unvollständig)

In dieser User Story sollte der Netzwerkgraph der CPU, beschrieben in `NetworkGraph.h`, auf die GPU migriert werden. Hierbei musste beachtet werden, dass der Netzwerkgraph in folgender Struktur vorliegt:

```
NetworkGraph
├── class NetworkGraphBase
│   ├── vector NeuronLocalInNeighborhood
│   └── vector NeuronLocalOutNeighborhood
├── NetworkGraphBase plastic_network_graph
└── NetworkGraphBase static_network_graph
```

Hierbei ist `NetworkGraphData` das GPU-Äquivalent von `NetworkGraphBase` und `NetworkGraphGPU` von `NetworkGraph`. Da die meisten Funktionalitäten auf die Distant-Arrays und nicht ausschließlich die Local-Informationen verwenden, wurden viele Funktionen noch nicht implementiert. Um die Funktionalität fertigzustellen, muss noch ein Handle für die Rückführung der Daten und der einen Device Pointer enthält. Weiterhin muss eine `create`-Funktion erstellt werden, ähnlich der der Czappa-Struktur. Zuletzt gibt es Datenstrukturen, welche bisher nur für die CPU-Version vorliegen bspw. `Synapse.h`, welche für die GPU mit `__device__`-Funktionen erstellt werden müssen.

# 3 Zukünftige Weiterentwicklung

Dinge, die noch gemacht werden sollten:

- Die Neuronenmodelle wurden noch nicht in den Master gemergt. Sie sollten funktionieren, allerdings gibt es bisher noch keine Tests, um das zu verifizieren (siehe Abschnitt 2.3), weshalb wir diese noch nicht gemergt haben.
- Beim Aufräumen der allokierten Daten auf der GPU enststehen teilweise noch Fehler. Dies ist manchmal der Fall beim destroyen der CudaArrays und trat zuerst in während der Calcium Calculator Implementierung auf und ist dort immernoch ein Problem. Wenn in free_content versucht wird, nach der Deallokation des data members des structs einen leeren struct zurück zu kopieren, meint er, der pointer zu dem struct auf der GPU wäre nicht mehr valide, obwohl er selbst nicht deallokiert wurde. Dann hatten wir teilweise noch versucht, in den Destruktoren die nicht CudaArray members die auf der GPU allokiert wurden zu cudaFreen, jedoch gab es hier auch den Fehler, das diese pointer wieder invalid für ihn sind, obwohl die Daten davor nie deallokiert wurden.
- Die GPU-Version des Netzwerkgraphen ist bisher nicht vollständig implementiert (siehe Abschnitt 2.5) und deshalb auch noch nicht gemergt.
- Die strukturelle Plastizität und die Neuronenmonitore bzw. Areamonitore wurden noch nicht auf die GPU portiert.
- Marvins Implementierung der BackgroundActivity-Klasse haben wir nicht optimiert/korrigiert.
- ./relearn nutzt noch nicht alle unsere GPU-Implementierungen.
- Bisher wurde die GPU-Portierung nur für einen einzelnen MPI-Rang vorgenommen.
- Die Tests müssten wahrscheinlich noch genauer und umfangreicher sein, um dem jetzigen Test-Qualitätsstandard des Projektes voll zu entsprechen. Bei den Neuronenmodellen gibt es im Moment noch gar keine Tests (siehe Abschnitt 2.3). Dazu könnte noch auf mehr nicht valide Eingaben in den Funktionen als runtime checks getestet werden.
- Als Teil des Barnes Hut wurde in `relearn/source/gpu/utils/RandomNew.cuh` auch eine neue Random-Implementation für die GPU kreiert, die sehr viel schneller ist als die gerade existierende. Die existierenden Kernel benutzen jedoch noch die alte Implementation und sollten auf die neue gewechselt werden.
- Einzelne Variablen, die sich nicht viel ändern und gerade in verschiedenen Structs auf der GPU stehen, wie zum Beispiel number_neurons im Octree und neurons_per_thread im BarnesHutData, können in

die constant memory geschoben werden für schnelleren Zugriff.

- Um die Funktionalität des Netzwerkgraphen fertigzustellen, muss noch ein Handle für die Rückführung der Daten und der einen Device Pointer enthält. Weiterhin muss eine `create`-Funktion erstellt werden, ähnlich der der Czappa-Struktur. Zuletzt gibt es Datenstrukturen, welche bisher nur für die CPU-Version vorliegen bspw. `Synapse.h`, welche für die GPU mit `__device__`-Funktionen erstellt werden müssen.