N-Body Simuation mit Cuda - Technische Dokumentation

Daniel Noske, Theodor Wübker, Leonard Franke

31. März 2022

1 Einleitung

Bei einer N-Body Simulation wird das Verhalten von Körpern durch gegenseitige Interaktionen simuliert. Diese Art von Simulationen ist vor allem in der Astrophysik gebräuchlich, um anhand der Gravitationskräfte von Körpern im Raum eine Umlaufbahn herzuleiten. In relevanten Simulationen ist die Anzahl an betrachteten Körper extrem hoch und es müssen konstant Berechnungen für alle Paare an Körpern getätigt werden. Es sind folglich Berechnungen mit einem Zeitaufwand von $\mathcal{O}(n^2)$ in sehr kurzen Zeitabschnitten nötig. Eine effiziente Parallelisierung der Aktionen in jedem Zeitschritt ist notwendig.

Im Zuge dessen wurde sich im Rahmen der Verantaltung *Programmierpraktikum GPU-Programmierung* mit der Implementierung einer N-Body Simulation unter Nutzung des *NVIDIA Jetson Nano* beschäftigt. Die konkrete Simulation ermöglicht die Platzierung von Asteroiden mit gegebenen Massen in einer Benutzeroberfläche. Kraftfelder und Gravitationskräfte wirken auf die gegebenen Körper im zweidimensionalen Raum.

Die Graphics Processing Unit (GPU) des Jetson Nano ermöglicht die effiziente parallele Ausführung von Funktionen. Der Renderingaufwand der Simulation wird zweitrangig betrachtet und primär soll die Dauer der Berechnung von Beschleunigungen für jeden Körper optimiert und auf die GPU ausgelagert werden.

Neben der Implementierung einer effizienten Simulation umfasst das Projekt zusätzlich mehrere Versionen in denen die Rechenleistung der GPU unterschiedlich genutzt wird und Programme, welche die Dauer der Berechnungen ermitteln.

Im Folgenden werden in Abschnitt 2 die genutzen Bibliotheken und Sprachen und in Abschnitt 3 die Funktionalitäten des Simulationsprogramms vorgestellt. Abschließend werden in Abschnitt 4 die Ideen für unsere verschiedenen Versionen erklärt und im Abschnitt 5 analysiert.

2 Tools

Um die N-Body Simulation umzusetzen, waren einige Werkzeuge notwendig. Hierzu zählen beispielsweise die verwendete Programmiersprache, ein Build-Tool und eine Rendering Bibliothek. Auf diese Werkzeuge und den Entscheidungsprozess wird im Folgenden eingegangen.

2.1 CUDA C und C++

NVIDIA sieht CUDA C, einen C++ Dialekt, für die Steuerung der GPUs vor. Während des Semesters wurde hiermit bereits programmiert, weshalb es nahe liegt, auch für das Praktikum CUDA C zu nutzen. Des Weiteren verwenden die meisten Quellen im NVIDIA Developer Forum CUDA C und nicht beispielsweise einen Python Wrapper. So ebenfalls die *Fast N-Body Simulation with CUDA* aus dem Magazin GPU Gems 3 [2], die als Orientierung diente.

Weil CUDA C eine Erweiterung für C++ ist, lässt es sich problemlos mit C++ kombinieren. Zusammenfassend lässt sich sagen, dass CUDA C und C++ als Sprachen für die Implementierung anhand des Materials eine leichte Wahl waren.

2.2 Rendering Bibliothek

Anfangs war es geplant, die aus der Veranstaltung *Die Programmiersprache C++* bekannte Bibliothek SDL2 [3] zu verwenden. Dies war aufgrund von Kompatibilitätsproblemen zwischen CUDA und SDL2 in Kombination mit dem Build-Tool CMake [1] nicht möglich. Quellen zu dieser Kombination waren auch nur sehr wenige vorhanden, weshalb die Wahl letztendlich auf Qt5 [4] fiel. Qt ist eine Rendering Bibliothek, die es ermöglicht, intuitive graphische Benutzeroberflächen zu erstellen. Sie basiert stark auf Events, die beispielsweise beim Klick auf einen Knopf ausgelöst werden können.

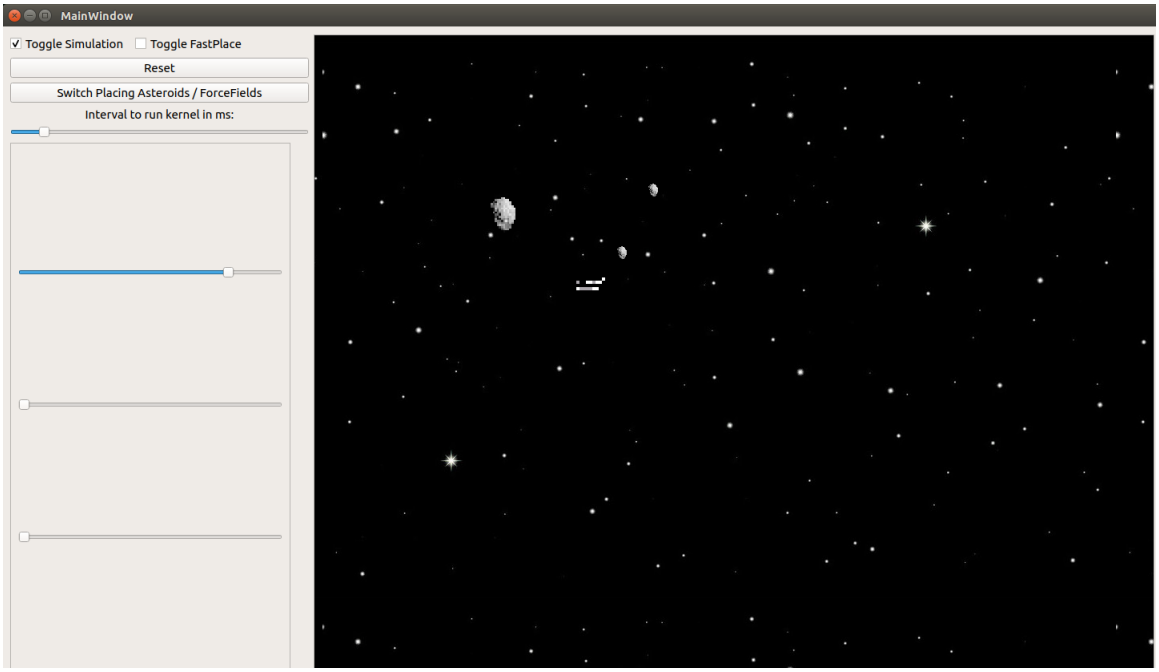

Abbildung 1: Das Fenster der Anwendung bei der Ausführung.

3 Die Anwendung

Dieser Abschnitt gibt einen Überblick über die Funktionen der Anwendung sowie einige Besonderheiten.

3.1 Architektur

Die Anwendung ist in Zwei Teile gegliedert. Auf der einen Seite steht der vollständig in C++ geschriebene Teil, der das Rendering mittels Qt übernimmt. Auf der anderen Seite gibt es die CUDA Kernels, welche die Physiken auf der Grafikkarte berechnen. Die beiden Teile kommunizieren über eine Schnittstelle der CUDA Datei. Diese strikte Trennung ermöglicht es, leicht neue Versionen der Kernels einzubinden, um Verbesserungen zu testen.

3.2 Funktionen

Das Hauptfenster (siehe Abbildung 1), welches mit dem Qt-Designer erstellt wurde, besteht aus einer Menüleiste links und dem Fenster für die Asteroi-

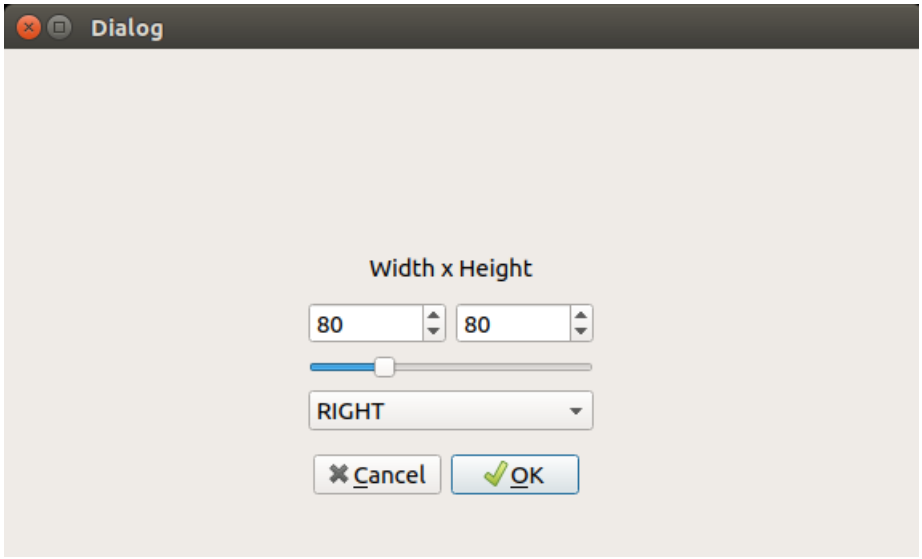

Abbildung 2: Der Dialog zum Platzieren von Kraftfeldern.

den rechts. Alle der Objekte in der Menüleiste, wie beispielsweise die Checkbox *Toggle Simulation* oder der Knopf *Reset* feuern Events, wenn mit ihnen interagiert wird. Für diese Events kann man sogenannte Event-Listener implementieren, die bestimmen, was passieren soll, wenn das Event ausgelöst wird.

In der Szene sind Zwei Arten von Objekten platzierbar: Asteroiden und Kraftfelder. Asteroiden sind die Körper in der N-Body Simulation, Kraftfelder lediglich Rechtecke, welche die Asteroiden bei Eintritt in das Kraftfeld in eine Richtung drücken. Mit dem Knopf *Switch placing Asteroids / ForceFields* kann man zwischen den beiden Objekten wechseln.

Hat man die Asteroiden ausgewählt, so wird beim Klick auf die Szene sofort ein Asteroid an der entsprechenden Stelle platziert. Außerdem wird links in der Leiste ein Slider ergänzt, mit dem man die Masse des Asteroiden anpassen kann. Je höher die Masse, desto größer der Asteroid, was in Abbildung 1 zu erkennen ist.

Hat man widerum die Kraftfelder ausgewählt, öffnet sich ein Dialog, wie er in Abbildung 2 dargestellt ist. Dort ist es möglich die Größe des Kraftfelds, seine Kraft (per Slider) und die Richtung, in der die Asteroiden gedrückt werden sollen, einzustellen. Ist der Dialog ausgefüllt, wird in der Szene ei-

ne Wind-ähnliche Animation an der entsprechenden Stelle platziert, die das Kraftfeld symbolisieren soll. Ein Frame dieser Wind Animation ist in Abbildung 1 zu erkennen.

Um eine große Anzahl an Asteroiden schnell zu platzieren, ist es möglich eine *FastPlace*-Funktion zu aktivieren. Dafür muss die Checkbox *Toggle FastPlace* aktiviert sein, welche in Abbildung 1 ganz links oben zu erkennen ist. Anschließend können durch gedrückthalten und ziehen der Maus in der Szene viele Asteroiden auf einmal gesetzt werden.

Sobald alle gewünschten Kraftfelder und Asteroiden platziert sind, kann die Animation über *Toggle Simulation* gestartet werden. An dieser Stelle kommt der CUDA Kernel ins Spiel, der die Kräfte berechnet, die auf die Asteroiden wirken und so ihre neuen Positionen nach jedem Durchlauf festlegt. Die Simulationsgeschwindigkeit kann per Slider angepasst werden, wie in Abbildung 1 zu erkennen ist. Die Maximale Geschwindigkeit ist, den CUDA Kernel jede Millisekunde auszuführen.

Schlussendlich kann die Szene durch betätigen des *Reset* Knopfes oben links zurückgesetzt werden (siehe Abbildung 1).

3.3 Starten der Anwendung

Um die Anwendung zu starten, sind Qt5 und CUDA erforderlich. Qt5 kann über apt-get installiert werden, während CUDA bereits auf dem Jetson Nano vorhanden sind. Sind die Abhängigkeiten installiert, so lässt sich das Programm wie folgt ausführen:

An erster Stelle steht die Navigation in den bereits vorhandenen Build Ordner, anschließend muss der Befehl *make* ausgeführt werden. Danach kann die Anwendung mit ./GPU gestartet werden.

4 Kernelversionen

Das Ziel war es die Berechnungen auf der GPU zu optimieren. Beginnend bei der ersten GPU Vesion entstanden durch stückweise Veränderungen zwei weitere Varianten. Um besonders den generellen Vorteil von Parallelisierungen

zeigen zu können, wurde zusätzlich ein Algorithmus für die CPU geschrieben. In den folgenden Abschnitten werden die insgesamt 4 Versionen vorgestellt und die Überlegungen zwischen diesen erläutert.

4.1 CPU

```
for (int i = 0; i < N_ASTEROIDS; i++)
{
    Vector3 a_g = { 0,0,0 };

    for (int j = 0; j < N_ASTEROIDS; j++)
    {
        if (j == i) continue;

        Vector3 r_vector;

        r_vector.x = bodys[i].posX - bodys[j].posX;
        r_vector.y = bodys[i].posY - bodys[j].posY;
        r_vector.z = bodys[i].posZ - bodys[j].posZ;

        double r_mag = len(r_vector);

        double acceleration = -1.0 * BIG_G * (bodys[j].mass)
   / pow(r_mag, 2.0);

        Vector3 r_unit_vector = r_vector / r_mag;

        a_g.x += acceleration * r_unit_vector.x;
        a_g.y += acceleration * r_unit_vector.y;
        a_g.z += acceleration * r_unit_vector.z;

    }

    bodys[i].velX += a_g.x * dt;
    bodys[i].velY += a_g.y * dt;
    bodys[i].velZ += a_g.z * dt;
}

for (int i = 0; i < N_ASTEROIDS; i++)
```

```
{
    bodys[i].posX += bodys[i].velX * dt;
    bodys[i].posY += bodys[i].velY * dt;
    bodys[i].posZ += bodys[i].velZ * dt;
}
```

Listing 1: Pseudocode einer N-Body Simulation [6]

Die erste CPU Version hielt isch an einen sehr simplen Algorithmus (siehe Listing 1), welcher mithilfe zweier *For-Schleifen* für jeden Asteroiden die Beschleunigungen aufsummiert, die durch alle anderen Asteroiden verursacht werden. Am Ende wird über die Beschleunigung die Geschwindigkeit aktualisiert und damit dann die Position des Asteroiden. Das Aktualisieren der Positionen kann erst nach sämtlichen Berechnungen durchgeführt werden, da die Kräfte unter Anderem von der Position abhängt und ein vorzeitiges Verändern keine korrekte Simulation erzeugen würde. Diese Vorgang kann beliebig oft wiederholt werden, wobei jede Iteration einen Zeitschritt in der Simulation darstellt. Insgesamt befindet sich dieser Algorithmus in der Laufzeitklasse $\mathcal{O}(n^2)$.

4.2 GPU V1

Die erste Version der GPU orientiert sich wie die CPU Version am Code aus Listing 1. Da nun die Möglichkeit zur Parallelisierung bestand, wurde der Code der äußeren For-Schleife in einen CUDA Kernel ausgelagert. Dadurch kann jeder Asteroid seine Beschleunigung parallel zu den anderen Asteroiden berechnen. Hierbei ist zu beachten, dass die Berechnungen der Physik und das Aufsummieren der Beschnleunigungen aus Sicht eines konkreten Asteroiden immernoch sequentiell durchgeführt werden. Wichtig zu nennen ist, dass es natürlich ein Thread Limit gibt, welches die Anzahl an zeitgleich laufenden Threads limitiert. Dieses liegt bei der Grafikkarte des Jetson Nanos bei 1024. Da somit schnell diese Grenze erreicht wird, müssen die Asteroiden auf mehrere Blöcke der Größe 1024 aufgeteilt werden. Diese Blöcke werden jedoch nur sequentiell abgearbeitet. Wie in der CPU Version müssen, nachdem die neuen Beschleunigungen berechnet wurden die Positionen verändert werden. Da dies ebenfalls erst durchgeführt werden kann, nachdem alle Blöcke

abgearbeitet wurden, wurde dies in einen eigenen Kernel ausgelagert.

4.3 GPU V2

Damit die GPU mit den Daten arbeiten kann, müssen diese vorher von der CPU zur GPU übertragen werden und anschließende wieder zurück. Dieser Vorgang wurde in GPU V1 jedes mal vor und nach einer Iteration durchgeführt. Jedoch ist das Hinkopieren der Daten nicht nötig, sofern sich die Daten auf der CPU, zum Beispiel durch Hinzufügen oder Verändern der Asteroiden, nicht verändern.

Version 2 spart sich somit diesen Kopiervorgang um so möglichst viel Zeit einzusparen. Das Programm schickt nur noch neue Daten zur GPU, falls sich diese CPU-seitig verändert haben. Andernfall werden die Berechnungen mit den vorhandenen Daten auf der GPU fortgeführt und im Anschluss fürs Rendering an die CPU zurückgeschickt.

4.4 GPU V3

Im letzten Ansatz sollte versucht werden sämtliche Interaktionen zwischen zwei Asteroiden zu parallelisieren. Konkret bedeutet das, für jedes Paar von Asteroiden zwei Threads zu erzeugen, die genau die gegenseitige Beschleunigung für diese beiden Asteroiden berechnet. Somit wurde nun auch die zweite innere For-Schleife des Algorithmus (1) seperat in einen Kernel ausgelagert.

In den vorherigen Versionen war es noch möglich die Beschleunigungen lokal innerhalb des Kernels aufzusummieren. Da jetzt alle Berechnungen für einen bestimmten Asteroiden in unterschiedlichen Threads und durch das Threadlimit sogar in unterschiedlichen Blöcken stattfindet, musste ein Weg gefunden werden möglichst effizient die Werte aufzusummieren. Würde hier jeden Thread beliebig auf eine für den Asteroiden vorgesehene Speicherstelle schreiben, so würde es durch die Parallelisieren zu Race Conditions kommen. Deshalb wird einen Algorithmus verwendet der Race Conditions verhindert, jedoch dafür in der Laufzeitklasse $\mathcal{O}(logn)$ liegt.

Abbildung 3 visualisiert das Vorgehen der Summierung. Die Idee ist das Abarbeiten immer größer werdender Bereiche. Zunächst addiert jeder zweite

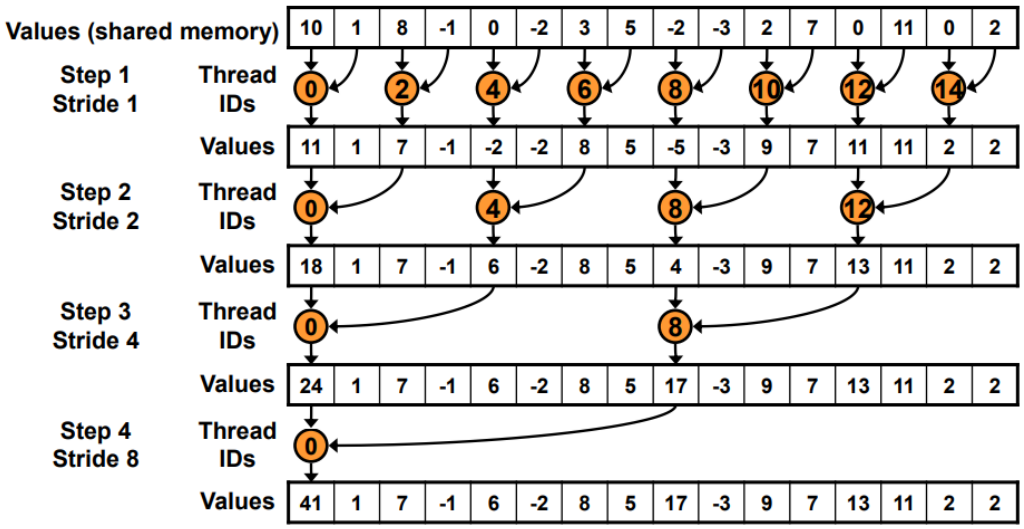

Abbildung 3: Reduction eines Arrays im geteilten Speicher [5]

Thread seine eigene Zahl und die des rechten Nachbarn. Im nächsten Schritt rechnet nur noch jeder Vierte und addiert seine eigene Zahl mit der des zweiten Nachbarn. Die Abstände werden dabei exponentiell größer und die endgültige Summe steht am Ende an der ersten Stelle. Ein Durchgang im Algorithmus kann hier von der GPU parallelisiert werden, wodurch die Laufzeit geringer als in einer rein serielle Variante ist.

5 Messergebnisse

Um den Nutzen der Änderungen zu ermitteln, wurden die Laufzeiten der Algorithmen für verschiedene Asteroidenanzahlen gemessen. Abbildung 4 zeigt zunächst den deutlichen Vorteil der GPU gegenüber der CPU. Bei der CPU ist der quadratische Zusammenhang zwischen Asteroidenanzahl und Laufzeit gut zu erkennen, was sich auf die doppelte For-Schleife im Algorithmus zurückführen lässt.

Abbildung 5 zeigt nur noch die GPU Versionen. Zum Einen ist zu sehen, dass V2 konstant besser abschneidet als V1. Dies ist eine direkte Folge aus dem reduzierten Datentransfer in der zweiten Version. Ebenfalls lässt sich folgern, dass V3 trotz der Änderungen kein besseres Ergebnis erzieht als V2.

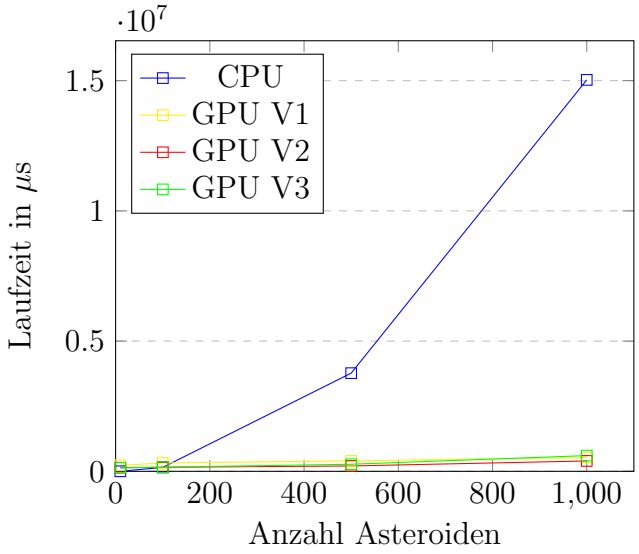

Abbildung 4: Messung der Laufzeiten für alle Versionen

Vermutlich fällt Synchronisationsaufwand bei GPU V3 stärker ins Gewicht, als das sequentielle Aufsummieren von GPU V2.

6 Fazit

Das Ziel des behandelten Projekts war es, eine N-Body Simulation mit Cuda C zu implementieren und die Berechnungen zu optimieren. Die Benutzeroberfläche konnte sich leicht mit SDL2 und C++ implementieren lassen und es wurde eine hinreichende gute Schnittstelle geboten, um die verschiedenen Cuda-Kernelversionen zu optimieren.

Man hat sehr schnell festgestellt, dass eine einfache GPU-Version bereits deutlich besser abschneidet als eine reine CPU-Version. Zur Optimierung der GPU-Version gab es verschiedene Ansätze. Besonders auffällige Unterschiede wurden ermittelt nach der Minimierung von Kopiervorgängen und einem veränderten Grad an Parallelität. Das Kopieren der Daten zwischen dem CPU- und GPU-Speicher hatte den größten Laufzeitaufwand und das Vermeiden von unnötigem Kopieren hatte die Leistung am deutlichsten verbessert. Es war zudem zu sehen, dass eine maximale Parallelisierung der

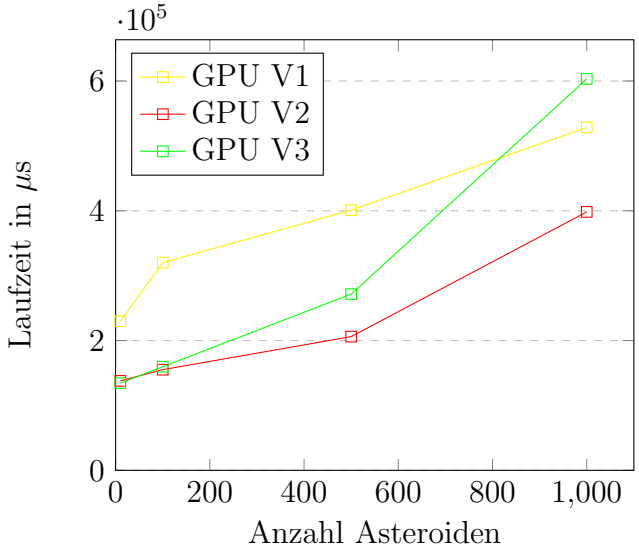

Abbildung 5: Messung der Laufzeiten der GPU Versionen

Vorgänge nicht optimal ist und wegen einem erhöhten Synchronisationsaufwand schlechtere Laufzeiten erzielt.

Die zweite implementierte GPU-Version war folglich am geeignetsten. Man konnte erkennen, dass selbst bei neuer Berechnung der Beschleunigungen in sehr kurzen Zeitintervallen für eine hohe Anzahl an Asteroiden eine reibungslose Simulation möglich war. Folglich ist die GPU des Jetson Nano unter Nutzung von Cuda sehr gut für die Realisierung von N-Body Simulationen geeignet.

Literatur

[1] *CMake*. URL: https://cmake.org/ (besucht am 27. 03. 2022).

[2] Jan Prins Mark Harris Lars Nyland. „Fast N-Body Simulation with CUDA“. In: *NVIDIA GPU Gems 3* (). URL: https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda.

[3] *SDL2*. URL: https://www.libsdl.org/ (besucht am 27. 03. 2022).

[4] *SDL2*. URL: https://doc.qt.io/qt-5/qt5-intro.html (besucht am 27. 03. 2022).

[5] Mark Harris - NVIDIA Developer Technology. *Optimizing Parallel Reduction in CUDA*. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[6] Wikipedia contributors. *N-body simulation — Wikipedia, The Free Encyclopedia*. 2022. URL: https://en.wikipedia.org/w/index.php?title=N-body_simulation&oldid=1065789661 (besucht am 31. 03. 2022).