GPU UNTERSTUETZTE MULTI-AGENTEN SIMULATION

Philipp Kayser,
Hamburg University of Applied Sciences, Dept. Computer Science,
Berliner Tor 7
20099 Hamburg, Germany
philipp.kayser@haw-hamburg.de

1. EINLEITUNG

Die Grundlage der Arbeit stellen Multi-Agenten Systeme (MAS) und insbesondere Mu MAS werden genutzt um Szenarien zu modellieren, bei denen das Verhalten des globalen Systems nicht mit partiellen Differentialgleichungen oder linearen Systemen dargestellt werden kann. Ein solches System besteht aus vielen Elementen, welche ein eigenes unabhängiges Verhalten haben. Ein Beispiel für ein solches Szenario wäre der Verkehr in einer Großstadt, bei dem in einer MAS das Straßennetz die Umgebung darstellt und jeder Verkehrsteilnehmer wäre ein Agent. MAS ermöglichen es nahezu beliebige Szenarien zu Simulieren, wobei hier die benötigte Rechenleistung oft den Flaschenhals darstellt. Dies zeigt sich zum einen wenn in einer sehr feinen Granularität simuliert wird, ein Beispiel hierfür wäre eine Simulation des Wachstums eines Baumes auf Atomebene. Das andere Extrem stellt zum Beispiel die Simulation des Verkehrsaufkommens in Europa dar. Ein Weiterer Punkt der bedacht werden muss ist das viele Simulationen sehr oft durchgeführt werden müssen, um vergleichbare Ergebnisse zu liefern und die Parameter entsprechend anzupassen. Als Beispiel kann ebenfalls die Verkehrssimulation genutzt werden, um herauszufinden welche Ampelschaltung die besten Ergebnisse liefert, müssen unzählige Simulationen durchgeführt werden, für die Optimierung der Parameter für die Ampelschaltung. In einen solchen Fall muss die Simulation nicht das gesamte Europäische Verkehrsnetz umfassen, damit eine Berechnung auf der CPU nicht mehr sinnvoll ist. Aufgrund der Vielzahl von Agenten sind MAS jedoch massiv parallelisierbar, wodurch eine verteilte parallele Berechnung möglich ist.

Graphics processing units (GPU) haben ursprünglich die Aufgabe die Bildschirmausgabe zu berechnen, dafür besitzen GPUs eine Struktur die viele parallele Berechnungen ermöglicht. Diese Rechenleistung kann und wird heutzutage auch für allgemeine Berechnungen genutzt, sogenanntes General Purpose Graphics processing unit (GPGPU) Computing. GPGPU Computing kann bei parallelen Szenarien einen großen Leistungsschub bringen. Das GPUs große Rechenreserven bieten wird dadurch deutlich, dass seit 2011 bei Supercomputern zur Leistungssteigerung auf eine Kombination von GPU und CPUs gesetzt wird. 2011 wurde in China der Tianhe-1A vorgestellt, zu der Zeit der schnellste Computer der Welt war und erstmals mit GPUs arbeitet. Dadurch konnte er bei deutlich geringerer Leistungsaufnahme eine größere Rechenleistung erreichen. In den folgenden Jahren wurde bei den Supercomputern immer mit einer Kombination aus GPUs und CPUs gearbeitet.

Ziel der Arbeit ist es die MAS durch die massive parallele Rechenleistung der GPUs zu unterstützen. Frameworks für die Erstellung von MAS gibt es einige zum Beispiel MASON [6] oder repast [10] , jedoch unterstützen diese bisher keine GPGPU Programmierung. FlameGPU ist das einzige Framework, welches Multiagentensimulation auf GPUs unterstützt, jedoch baut FlameGPU auf CUDA auf [9], der GPGPU Schnittstelle von Nvidia und ist somit an Hardware des Herstellers Nvidia gebunden. Die Basis für die Entwicklung ist Life System von der MARS (Multi Agent Research and Simulation) Group, für die eine Schnittstelle zur GPU Unterstützung entworfen werden soll.

2. MULTI- AGENTEN SIMULATION (MAS)

Der top down Ansatz für Simulationen sind die Equation-based models (EBMs), diese werden verbreitet genutzt um systemseitiges Verhalten in der Biologie zu modellieren. Es wird das Gesamtsystem betrachtet und versucht das Verhalten des Systems durch mehrere Gleichungen darzustellen. Bei komplexen Problemen mit mehreren Individuen stoßen EBMs an ihre Grenzen, die Gleichungen können das komplexe Verhalten der Individuen nicht repräsentieren. Für diese Art von Problemen wird die Multi-Agenten Simulation (MAS) verwendet. Im Gegensatz zu EBMs sind MAS der bottom up Ansatz für eine Simulation. Bei MAS werden alle Elemente der Simulation als einzelne Individuen modelliert. Eine MAS basiert auf einem darunterliegenden Agent Based Model (ABM). Das ABM beschreibt die Umgebung in der sich die Agenten bewegen, sowie die Art in der sie miteinander interagieren.

Ein Beispiel für ein Szenario, für das sich die MAS gut eignet, ist die Simulation eines Feuers bei einem Konzert. Die fliehenden Menschen sowie das Feuer stellen Agenten dar und das Konzertgebäude ist die Umgebung in der die Agenten sich bewegen.

Mit MAS können Ereignisse in der Vergangenheit nachgestellt werden und mit den daraus ermittelten Parametern bis in die Zukunft weitersimuliert werden. Ein Beispiel dafür ist das ARS Africae Projekt der MARS Group, dort wird auf Basis von Messwerten die über die letzten 10 Jahre erhoben wurden, eine Simulation erstellt, die das Verhalten der Natur über die Jahre darstellen soll. Simuliert wird hier vor allem eine spezielle Baumart, sowie Elefanten und Nomaden, die durch die Landschaft ziehen. Im ersten Schritt wird versucht den Verlauf der letzten 10 Jahre durch die Simulation nachzuvollziehen, indem anhand der genauen Messdaten simuliert wird. Nachdem das vollbracht ist kann, basierend auf den ermittelten Parametern, auch in die Zukunft simuliert werden und es im möglich zu bestimmen, was getan werden muss um die Natur so zu erhalten, wie sie zur Zeit ist.

Rückschlüsse werden bei MAS durch das beobachten der Agenten sowie über die Parameter der Simulationsumgebung gewonnen.

MAS werden eingesetzt für komplexe nicht lineare Probleme, mögliche Einsatzgebiete sind:

- Organismen
- Ökologie
- Massenverhalten
- Verkehr

3. GPU

Die Graphic Processing Unit (GPU) ist im allgemeinen auch als Grafikkarte bekannt. Dieser Abschnitt beschreibt im ersten Teil, wie eine GPU aufgebaut ist und wie die einzelnen Komponenten miteinander interagieren. Im zweiten Teil wird erläutert, wie die Rechenleistung der GPU für Berechnungen genutzt werden kann, die unabhängig von der Grafikverarbeitung sind.

3.1. Aufbau und Funktionsweise

Hauptaufgabe der GPU ist es, wie ihr Name bereits andeutet, Grafikberechnungen durchzuführen und die Bildschirmausgabe zu erzeugen. Am meisten belastet wird sie beim Spielen vom Computerspielen das folgt daraus, dass dort viele dynamische Berechnungen durchgeführt werden müssen. Ein großer Teil der Aufgaben welche die GPU bearbeiten muss sind aus dem Bereich der Bildverarbeitung, welche größtenteils aus Matrixoperationen besteht. Diese haben die Eigenschaft, dass sie auf der einen Seite eine hohe Komplexität haben, jedoch auf der anderen Seite sich sehr gut parallelisieren lassen. Es kann nahezu jeder Pixel parallel berechnet werden. Speziell ist die GPU auf die Berechnung von Fließkommazahlen ausgelegt. Das führt dazu dass die GPU, anders als die CPU, darauf ausgelegt ist, möglichst viele Berechnungen gleichzeitig durchzuführen. Die Geschwindigkeit in dem die GPU die einzelnen Elemente berechnet ist im Vergleich zur CPU geringer, dafür hat sie deutlich mehr Kerne und kann mehr Operationen parallel ausführen. In diesem Abschnitt wird etwas genauer auf den Aufbau und die Funktionsweise einer Grafikkarte eingegangen.

Eine GPU beinhaltet eine Vielzahl an Elementen. Der Cache stellt einen Bereich davon dar, er wird genutzt um häufig verwendete Daten vorzuhalten, jedoch ist der Cache einer GPU im Vergleich zur CPU deutlich kleiner. Neben dem Cache besitzt die GPU einen eigenen Arbeitsspeicher, dort werden die Zwischenergebnisse der Berechnungen gespeichert. Das Hauptelemente stellen die Compute Units (CU) dar, sie sind die eigentliche Recheneinheit, welche die Berechnungen durchführt. Jede GPU enthält mehrere CUs die parallel angesprochen werden. Auf der Abbildung 1 ist eine CU einer Grafikkarte des Herstellers AMD dargestellt. Jede der CUs besitzt mehrere Singe Instruction Multiple Data (SIMD) Elemente, die in der Lage sind, parallel die selbe Operation auf mehreren Datenblöcken durchzuführen. Das erlaubt der CU, zum Beispiel, pro Takt jeden einzelnen Wert in einem Array quadrieren. Damit ist sie in der Lage viele parallele Operationen durchzuführen, bei identischen Instruktionen. Pro SIMD Einheit gibt es einen entsprechenden Buffer, der die Daten vorhält. Zusätzlich hat jede CU ein gemeinsamen lokalen Speicher (Local Data Share), auf den alle Elemente der CU zugreifen können. Ein weiteres Element, welches die GPU von einer CPU unterscheidet ist der Scheduler, der bei der GPU ebenfalls Teil der Hardware ist. Bei der CPU wird das Scheduling softwareseitig gelöst.

3.2. GPGPU Programmierung

Die große Parallelperformance der GPU wird mittlerweile nicht ausschließlich zur Erstellung der Bildschirmausgabe verwendet.

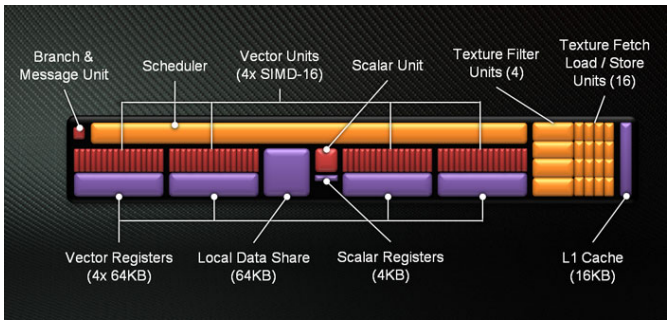

Figure 1: Compute Unit von AMD (GCU) [4]

Auch Anwendungen, die viele parallele Berechnungen benötigen setzten heutzutage auf die GPU. Das General Purpose GPU (GPGPU) Computing ist der Begriff der für die Nutzung der GPU für allgemeine Aufgaben steht. Computer-aided design CAD stellt einen Bereich dar, in dem die professionellen Tools größtenteils für ihre Berechnungen die GPU nutzen.

Um eine Grafikkarte für GPGPU Berechnungen zu nutzen, gibt es aktuell zwei Möglichkeiten. Das ist zum einen das CUDA Framework von Nvidia [9]. CUDA ist eine in 2011 vorgestellte Schnittstelle, die GPGPU Computing ermöglicht. Der Nachteil bei CUDA ist, dass es eine herstellerspezifische Schnittstelle ist und somit nur für Nvidia GPUs verwendet werden kann. Alternativ gibt es OpenCL [13] welches von Apple entworfen wurde und von der Khronos Group verwaltet wird. OpenCL ist ein herstellerunabhängiger Standard, der von den meisten großen GPU und CPU Anbietern unterstützt wird. Anders als CUDA besteht bei OpenCL zusätzlich auch die Möglichkeit den Code auf der CPU auszuführen. Die Struktur von CUDA und OpenCL ist sehr ähnlich. Um flexibel bei der Wahl des GPU Herstellers zu bleiben ist bei diesem Projekt die Wahl auf OpenCL gefallen.

Beim Programmieren mit OpenCL gibt es im Vergleich zu herkömmlichen Programmiersprachen einige Besonderheiten, die im folgenden Abschnitt erläutert werden. Ein OpenCL Programm besteht immer aus zwei Teilen. Einem Host, der auf der CPU ausgeführt wird und den organisatorischen Teil übernimmt und einem Kernel welcher direkt auf der GPU ausgeführt wird. Der Host hat folgende Aufgaben:

- Compilieren der Kernelprogramme
- Datenübertragung zur und von der GPU
- Erstellen von Command Queues zum ausführen von Kerneln oder Speicheroperationen

Ein Minimalbeispiel einer Hostapplikation ist auf Abbildung 2 dargestellt. Der abgebildete Ausschnitt gibt eine gute Übersicht, welche Schritte im Host vorgenommen werden müssen, um ein Programm als Kernel auszuführen.

Eine Hostapplikation ist in mehrere Abschnitte aufgeteilt und arbeitet mit einer Reihe von Structs . OpenCL hat einen vergleichsweise hohen Initialisierungsaufwand. Dieser ist eine Folge von der Feingranularität mit der im Host gearbeitet wird. Der Programmierer hat bei OpenCL volle Kontrolle darüber wann und wohin Daten übertragen werden. Daraus folgt, dass zum Beispiel auch die Argumente für die Kernelmethoden eigenhändig vom Host übertragen werden müssen. Die Structs, die die Hostapplikation nutzt, haben folgenden Hintergrund:

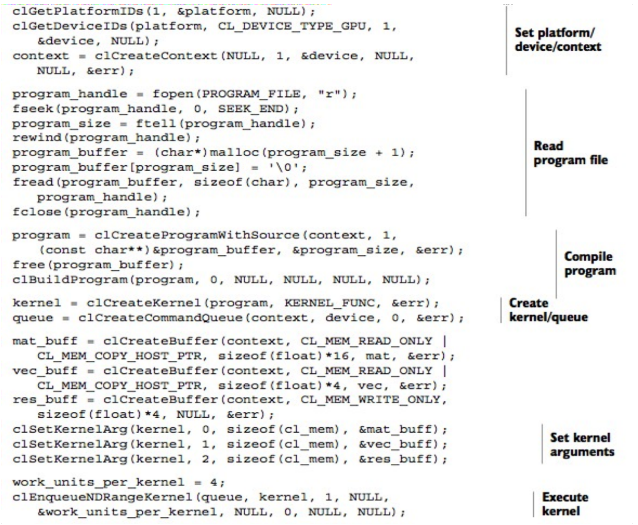

Figure 2: Minimalbeispiel einer OpenCL Hostapplication [12]

- Platform : Repräsentativ für die Plattform ist zum Beispiel der PC
- Device : Als Device gelten CPUs, GPUs oder andere Hardwarebeschleunigungselemente
- Context : Eine Untergliederung der Devices
- Commandqueue : Verbindungskanal zwischen Host und Device, wird dafür genutzt Anweisungen, wie die Ausführung eines Kernels oder das Übertragen eines Buffers an das Devices zu senden
- Kernel : Die Kernelmethode, die vom Device ausgeführt werden kann
- Programm : Eine Sammlung von Kerneln

Der Aufbau dieser Structs ist hierarchisch, als erstes wird die Plattform generiert, über die Angabe der Plattform erhält man die Devices, für ein Device kann ein Context generiert werden u.s.w..

Der Kernel ist der eigentliche Programmcode der von der GPU ausgeführt wird. Er wird in einer Abwandlung von C Programmiert: OpenCL C. OpenCL C ist nahezu identisch mit C Programmierung hat jedoch einige Sonderfunktionen für die Matrizenrechnung und ein paar Einschränkungen, wie zum Beispiel, dass keine Rekursion erlaubt ist.

```
// Quadrieren der Arraywerte in C
for(int i = 0; i < size ; i++)
{
     result[i] = input[i] * input[i];
}

// Quadrieren im Kernel mit OpenCL C
__kernel void square(__global float* input,
                     __global float* output,
                       const unsigned int count)
{
// Ermitteln des zu bearbeitenden
    Arrayslots
   int i = get_global_id(0);
// Abbruchbedingung pruefen und Operation
    durchfuehren
   if(i < count)
     output[i] = input[i] * input[i];
}
```

Der Codeausschnitt zeigt zum einen eine reguläre C Forschleife, in der alle Werte eines Arrays quadriert werden. Im unteren Teil wird eine Kernelmethode gezeigt, die den selben Effekt hat, wie die Forschleife. Bei Betrachtung der beiden Codeabschnitte fällt auf, dass bei dem Kernelcode keine Schleife existiert. Das hat den Hintergrund, dass die Operationen in OpenCL parallel ausgeführt werden, innerhalb der Kernelmethode wird nur der Body der Schleife ausgeführt, in diesem Fall das Quadrieren. Anstatt in jedem Schleifendurchlauf die Operation durchzuführen wird die Kernelmethode von *size* Threads parallel ausgeführt. Die Information, welche Arrayposition bearbeitet werden soll liefert die get_global_id Methode, die für jeden Kernel den zu bearbeiteten ArrayIndex zurückliefert. Weiterhin ist es so, dass Kernelmethoden immer mit *__kernel* beginnen und *void* als Rückgabewert liefern. Zusätzlich muss zu jedem Argument der Speicherbereich angegebene werden, in dem die Variable abgelegt werden soll. In diesem Fall *__global* für den Global Memory und *const* für den Read Only Memory.

In OpenCL besteht das Device aus mehreren Speicher und Arbeitsgruppen. Bei den ausführenden Elementen wird zwischen Workgroups und Work Items unterschieden. Ein Work Item ist vergleichbar mit einem Thread und führt aktiv Code aus. Die Workgroup bezeichnet eine Gruppe von Work Items die gemeinsam die selbe Aufgabe bearbeiten. In Rückblick auf den GPU Aufbau entspricht die Workgroup einer CU. Der Speicher wird bei OpenCL in folgende Bereiche unterteilt:

- Global Memory: Zentraler Speicher, von allen Elementen erreichbar
- Local Memory: Lokaler Speicher, gemeinsamer Zugriff innerhalb einer Workgroup
- Private Memory: Speicher eines Work Items, nur vom Work Item erreichbar

Der Global Memory ist von allen Elementen erreichbar und stellt auch die Schnittstelle der GPU dar, auf die vom Host aus zugegriffen werden kann. Im Vergleich zu den anderen Speicherbereichen ist der Global Memory der Langsamste. Der Local Memory ist ca. um Faktor 10 schneller als der Global Memory. Es kann keine direkte Datenübertragung vom Host zum Local Memory durchgeführt werden, die Daten müssen immer erst in den Global Memory geschrieben werden und das Device hat denn die Möglichkeit Zwischenergebnisse auf dem Local Memory zu speichern.

Um die Zusammenhänge der Speicherbereiche zu verdeutlichen wird in dem Buch OpenCL in Action eine Analogie verwendet [12], die auf Abbildung 3 dargestellt ist. Dort wird ein OpenCL Device mit einer Mathematikschule verglichen. In der Schule gibt es mehrere Klassenräume in denen die Schüler Mathematikaufgaben berechnen. Jeder Schüler hat ein eigenes Notebook und jede Schulklasse hat eine gemeinsame Tafel. Nur Schüler die in die selbe Klasse gehen, können gemeinsam arbeiten. Sobald die Schüler die Klasse betreten, wissen sofort welche Aufgabe sie berechnen sollen, kennen jedoch die Zahlenwerte zu der Aufgabe nicht. Diese müssen die Schüler zu beginn des Unterrichts von der zentralen Tafel ablesen, die von der gesamten Schule erreichbar ist. Die um die zentrale Tafel zu erreichen müssen die Schüler jedoch

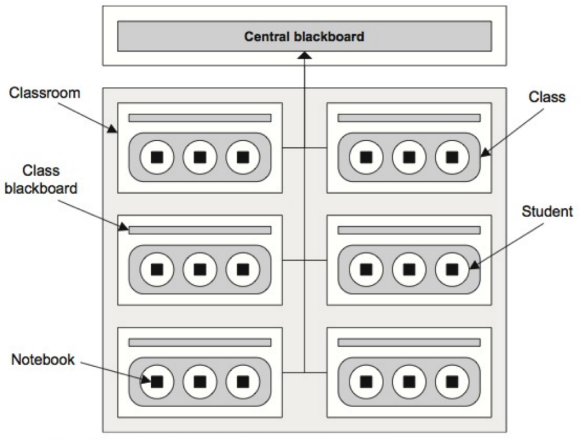

Figure 3: Schulanalogie aus OpenCL in Action [12]

einen langen Weg zurücklegen, weswegen sie versuchen möglichst selten zur zentralen Tafel zu gehen. Eine mögliche Aufgabe wäre das Summieren aller Zahlen bis 1000. In dem Fall rechnet die erste Klasse das Ergebnis der Zahlen von 1 bis 100, die zweite Klasse rechnet die 101-200 usw.. Die Schüler der Klassen teilen sich die Aufgaben auf, so das ein Schüler zu Beispiel 1+2 rechnet und der nächste 3+4. Die Ergebnisse werden von an der Klassentafel zusammengetragen und sobald alle Schüler fertig sind an die zentrale Tafel geschrieben. Um möglichst wenig laufen zu müssen gehen die Schüler idealerweise nur zweimal zur zentralen Tafel. Zum Schulbeginn um die Aufgaben abzuschreiben und am Ende zum Eintragen der Ergebnisse.

Bei der Rückführung auf das OpenCL Device Modell stellt die zentrale Tafel den Global Memory dar. Auf den möglichst wenig zugegriffen wird, aufgrund der langsamen Zugriffsgeschwindigkeit. Die Klassen sind repräsentativ für die Workgroups und die Klassentafel entspricht dem Local Memory. Jeder Schüler steht für ein Work Item und die Notebooks der Schüler entsprechen dem Private Memory.

4. RELATED WORK

Das Umsetzen von MAS mit OpenCL ist ein relativ neues Thema. Jedoch wurde es bereits in einigen Projekten durchgeführt, einige davon werden hier genauer vorgestellt.

SAMPO wird im Rahmen eines Papers von Klaus Kofler et al. vorgestellt [5]. Es simuliert die Population des Anopheles gambiae Mosquitos, welcher die Krankheit Malaria verbreitet. Die Basis für SAMPO stellt AGiLESim dar, welches die Mosiquitopopulation in Java simuliert. In dem Projekt wurde die gesamte Simulation in OpenCL umgesetzt und stark auf die Strukturen einer GPU angepasst. Zur Veranschaulichung der Komplexität dieser Simulation wird auf Abbildung 4 das Statemodell eines Mosquitos dargestellt. Mit insgesamt acht verschiedenen States ist es keine triviale Simulation.

Es wurde bei diesem Projekt viel Arbeit in die Optimierung der Algorithmen auf die OpenCL Strukturen investiert. Im Rahmen dessen wurde die Kommunikation zwischen Host und Device so weit reduziert, dass nach der Initialisierung maximal 100 Bytes pro Zyklus zwischen Host und Device ausgetauscht werden. Dieser Wert ist unabhängig von der Anzahl der Agenten. Die

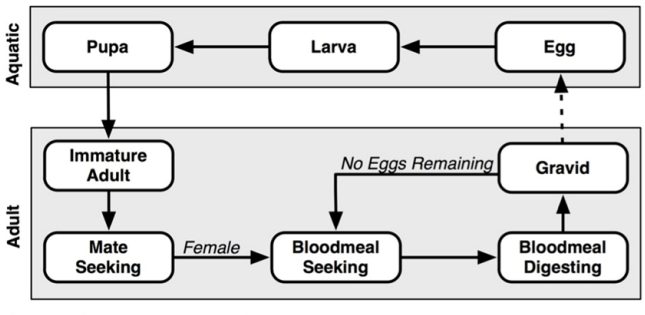

Figure 4: Statediagramm des SAMPO Mosquito life cycles

Performance und die geringe Kommunikation wird unter anderem dadurch erreicht, dass zu Beginn der Simulation angegeben werden muss, wieviele Agenten maximal entstehen können und auf Basis dieses Wertes werden zum Simulationsbeginn die Strukturen für alle Agenten auf dem Device angelegt.

Ein Problem welches in diesem Projekt elegant gelöst wurde ist die Generierung von Zufallszahlen. OpenCL bietet keine direkte Möglichkeit Zufallszahlen zu generieren, anders wie zum Beispiel die CPU, welche meistens einen Zufallsgenerator integriert hat. Eine mögliche Lösung die oft angewandt wird, ist die Zufallszahlen auf dem Host zu generieren und bei Bedarf immer aktuelle Werte zum Device übertragen. Das ist jedoch nicht performant, denn die Datenübertragung kostet viel Zeit und es werden viele Zufallszahlen benötigt. Bei SAMPO werden pro Zyklus vier Zufallszahlen vom Host generiert und zum Device übertragen. Aus diesen werden mit Hilfe des Tausworthe/LCG Generators [14][7] bis zu vier Zufallszahlen mit einer Periodenlänge von 2^{121} generiert. Um ein für jeden Agenten eigene Zufallszahlen zu erhalten, werden die Zahlen mit der ID des berechnenden Work Items verwoben. Dadurch erreichen sie es mit nur 4 übertragenen Zufallszahlen vom Host, beliebig viele Agenten mit unterschiedlichen Zufallswerten zu generieren.

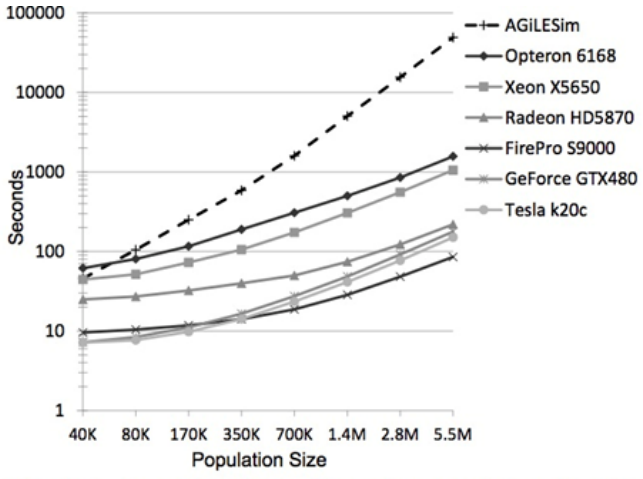

Figure 5: Ergebnissdiagramm der SAMPO Simulation

Abbildung 5 zeigt die Ergebnisse der Simulation. Es ist ihnen gelungen durch die Optimierung mit OpenCL und der Simulationsdurchführung auf der Grafikkarte eine Leistungssteigerung

von Faktor 576 gegenüber der Simulation mit AGiLESim zu erreichen. Jedoch muss hierbei beachtet werden, dass die Agenten in SAMPO größtenteils unabhängig voneinander sind und sich optimal parallel ausführen lassen. Das ist bei MAS die Ausnahme, viele Simulationen haben höhere Kommunikationsanteile, die die Ausführungsgeschwindigkeit mit OpenCL deutlich ausbremsen.

Eine Verkehrssimulation ist ein Szenario mit vermehrtem Kommunikationsaufwand. Die Verkehrsteilnehmer müssen untereinander Kommunizieren, genauso wie mit der Umgebung, wie zum Beispiel Ampeln. Das Paper von Wang aus dem Jahre 2013 [16], behandelt eine genau solche Verkehrssimulation und wird als nächstes genauer betrachet. Dort wird speziell die Simulation auf heterogenen Systemen evaluiert und einige interessante Ansätze des Clusterings im Rahmen des Papers umgesetzt. Als heterogenes System werden CPUs mit integrierter GPU bezeichnet, wie zum Beispiel die Sandy Bridge CPUs von Intel oder die AMD Fusion APUs [3]. Der große Vorteil eines heterogenen Systems ist, dass GPU und CPU auf dem selben Speicher arbeiten, wodurch es möglich ist direkt vom Device auf Teile des Hostspeichers zuzugreifen. Ein weitere Punkt der in diesem Paper gut gelöst wird ist das Clustering. Die simulierte Straße wird in mehrere Abschnitte unterteilt, für die anschließend die äußeren Einflüsse bestimmt werden .

Abbildung 6 zeigt die Aufteilung und wie die Abhängigkeiten ermittelt werden. Es wird bestimmt wird welche Fahrzeuge außerhalb des Clusters Auswirkung auf den Cluster haben und diese fließen als Parameter in die Clusterberechnung ein. Bei Betrachtung des Beispiels auf Abbildung 6 sind Fahrzeug 12, 15 sowie 24 und 27 die äußeren Einwirkungen auf das Cluster welches in der Mitte als Block dargestellt wird. Die Berechnung eines Blockes erfolgt daraufhin unabhängig vom Rest der Simulation. Damit kann die Berechnung eines Blockes in einer OpenCL Workgroup durchgeführt werden, wodurch die Performance deutlich verbessert wird. Der Nachteil ist, dass die Cluster oft neu Sortiert werden müssen, da sich die Agenten durch die Cluster bewegen. Das hat sich in diesem Paper auch als Flaschenhals herausgestellt. Bei einer hohen Anzahl von Agenten ist die Zeitaufwand für das Sortieren deutlich höher, als der für die eigentliche Berechnung. Unabhängig davon wurde auch hier durch den Einsatz von GPU und OpenCL ein großer Performanceschub erreicht.

Im Rahmen einer Veröffentlichung von Dominik Moser et al. in 2011 [8], wurde ein Performancevergleich zwischen CPUs mit Cilk [1] und GPUs mit OpenCL durchgeführt. Es wurden Tests mit Agenten mit einem unterschiedlichen Synchronisationsanteil besitzen durchgeführt, sowohl wie Performancetests mit verschiedenen Clustergrößen. Die Tests haben gezeigt, das die GPU bei vielen Agenten deutlich schneller ist als die CPU. Auch im Vergleich zu einem Cluster mit 100 CPUs erreicht eine GPU eine Leistungssteigerung von Faktor 2.5 bei relativ hohem Synchronisationsaufwand, mit wenig Synchronisation ist sogar eine Leistungssteigerung von Faktor 50 erreichbar. Diese Werte hängen von der Anzahl der Agenten ab, sowie von dem Ausmaß an Synchronisationsaufwand. Die Tests wurden mit $10e^6$ und $10e^7$ Agenten durchgeführt, dort hat sich herausgestellt, dass die GPU am effektivsten mit möglichst vielen Agenten arbeitet. Des Weiteren hat sich herausgestellt, das die GPU am besten mit Clustergrößen von 16 und 32 Einheiten arbeitet.

Zusammengenommen zeigen die Paper, dass die GPU viel Potential für die MAS bietet, wobei auf Clustering und Synchronisation geachtet werden muss.

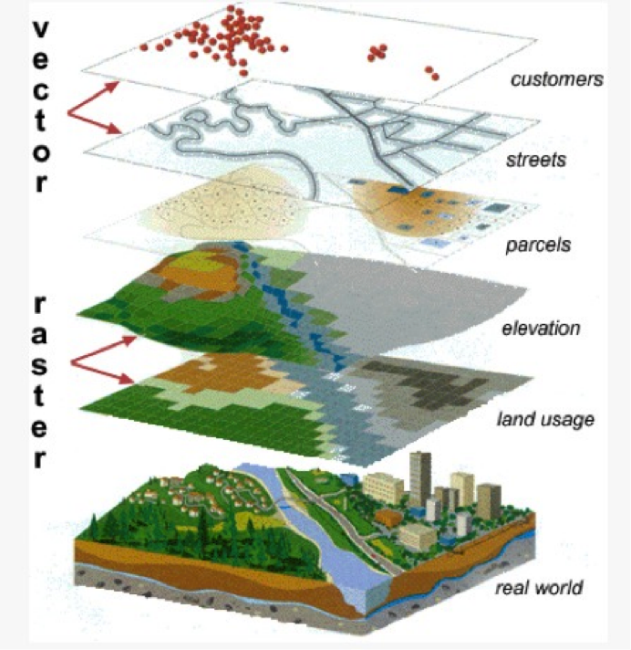

Figure 7: Die GIS Layer Struktur wird für Geodaten verwendet [15]. Der Aufbau ist vergleichbar mit den MARS Life Layern. Agenten die auf der selben Ebene interagieren werden in einem Layer zusammengefasst

5. ZIEL

Die Masterthesis wird im Rahmen der MARS Group durchgeführt. MARS Life ist eine von der MARS Group entwickelte Plattform für MAS. Sie ist in C# geschrieben und nutzt eine Layerarchitektur für die Verwaltung der Agenten. Es werden die Agenten nach Gruppen zugeordnet und für jede Gruppe wird ein eigener Layer verwendet. Abbildung 7 zeigt eine GIS Layerveranschaulichung. GIS Daten werden als Umgebungswerte für die Simulationen bei MARS Life verwendet. Jeder Layer ist einer Darstellungsebene zugeordnet, wie zum Beispiel dem Straßennetz oder den Fahrzeugen. Das Konzept für die MARS Life Agentenlayer ist identisch. So werden Agenten die auf der selben Ebene arbeiten in einem Layer zusammengefasst. Erklärung am Beispiel des ARS AfricaE [2] Projekts: Das ARS AfricaE ist ein Projekt was aktuell von der MARS Group durchgeführt wird und simuliert einen Abschnitt eines Savannen-Ökosystems. Im Rahmen dessen werden alle Baumarten in einem eigenen Layer zusammengefasst, Elefanten bilden einen weiteren Layer und die durch die Savanne streifenden Nomaden stellen ebenfalls einen Layer dar. Die MARS Life Umgebung ermöglicht es, dass jeder Layer auf einem anderen Node instanziiert werden kann, wodurch breite Verteilung der gesamten Simulation ermöglicht wird.

5.1. Erstellung eines OpenCL Layers

Ziel im Projekt eins ist es einen OpenCL Agenten Layer für das MARS Life System zu entwickeln. Der OpenCL Layer soll ein eigenständiger Teil der MARS Life Projekts sein und mit anderen Layern, die nicht auf OpenCL basieren, interagieren können. Umgesetzt werden soll der Baum-Agentenlayer aus dem ARS AfricaE

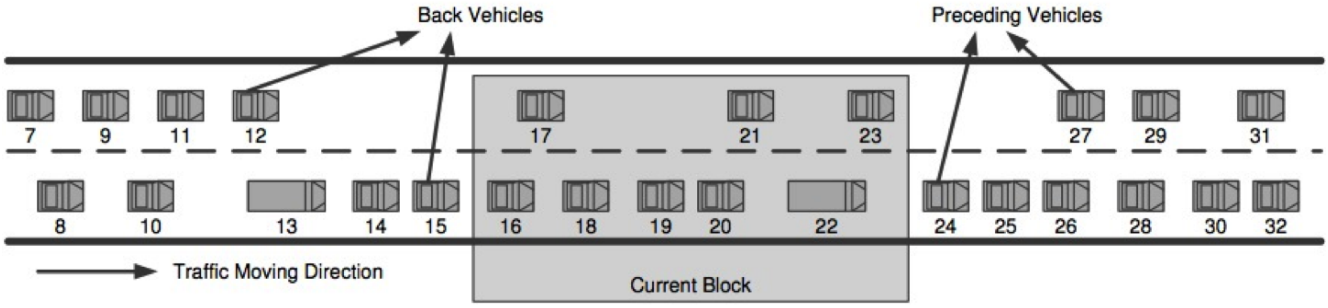

Figure 6: Der Current Block stellt das eigenständige Cluster dar, die Back / Front Vehicles sind die Elemente, die das Cluster von außen beeinflussen [16].

Projekt. Hierzu muss im ersten Schritt evaluiert werden, wie die Kommunikation zwischen dem OpenCL Layer und den anderen Layern des MARS Systems umgesetzt werden kann. Als Basis wird ein Active Layer aus MARS Life verwendet. Dieser stellt einen aktiven Teil der Simulation dar und erhält wie die Agenten regelmäßig Ticks vom System, wenn der nächste Simulationsschritt ausgeführt werden soll. Bei jedem dieser Ticks wird ein Schritt der OpenCL Agenten durchgeführt. Der Active Layer stellt die OpenCL Hostapplikation dar und es werden bei den Ticks die entsprechenden Kernelmethoden ausgeführt. Erster Ansatz für den Datenaustausch ist ein Mapping des Global Memorys vom OpenCL Device, auf einen Speicherbereich des Nodes, der wiederum mit der Datenbank synchronisiert wird. Um mit der Simulationsumgebung konform zu bleiben wird für jeden OpenCL Agenten, der in dem Aktive Layer erstellt wird, ein Dummyagent angelegt und im System eingetragen, da dieses ein Agentenobjekt benötigt.

Durch das Entwickeln des Layers soll deutlich werden, welche Schritte vorgenommen werden müssen, um einen MARS Life Layer in OpenCL umzusetzen.

5.2. Abstraktion des OpenCL Layers

Ansatz für das zweite Projekt ist es, für den im ersten Projekt entwickelten OpenCL Layer, eine Abstraktionsschicht zu entwickeln.

Die MARS Group besteht aus 10 - 20 Studenten der HAW Hamburg, mit diesem Projekt soll OpenCL für die gesamte MARS Group verfügbar gemacht werden. Um dies zu ermöglichen soll ein Framework erstellt werden, welches eine einfache Nutzung von OpenCL Agenten ermöglicht.

Ähnliches wurde bereits im Rahmen von FLAMEGPU [11] entwickelt. Es ist einer Erweiterung der FLAME Frameworks, welches eine templatebasierte MAS ermöglicht. Für die Umsetzung nutzt FAMEGPU das CUDA Framework, die Modellierung der Agenten erfolgt in XML.

Die Abstraktion für das MARS Life System soll speziell auf das Projekt angepasst werden, um eine möglichst leicht zu bedienende Schnittstelle zu erstellen. Der Ansatz von FLAMEGPU, XML für die Modellierung der Agenten zu verwenden, könnte hierfür übernommen werden. Es wird der Hauptteil daraus bestehen zu evaluieren, welche funktionalitäten von OpenCL gekapselt werden sollen, welche für die Entwickler angeboten werden sollen und wie diese herausgeführt werden können.

6. REFERENCES

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[2] C. Bruemmer. Ars africae - adaptive resilienz in südafrikanischen savannengebieten. Available at https://www.ti.bund.de/de/ak/projekte/ars-africae/, accessed February 26, 2015.

[3] S.R. Gutta, D. Foley, A. Naini, R. Wasmuth, and D. Cherepacha. A low-power integrated x86 x2013;64 and graphics processor for mobile computing devices. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 270–272, Feb 2011.

[4] H. Hagedoorn. AMD Radeon HD 7970 review - Graphics engine architecture . Available at http://www.guru3d.com/articles-pages/amd-radeon-hd-7970-review,5.html, accessed February 26, 2015.

[5] Klaus Kofler, Gregory Davis, and Sandra Gesing. SAMPO: an agent-based mosquito point model in OpenCL. In *ADS '14: Proceedings of the 2014 Symposium on Agent Directed Simulation*. Society for Computer Simulation International, April 2014.

[6] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, July 2005.

[7] P. L'Ecuyer. Maximally equidistributed combined tausworthe generators. *Mathematics of Computation*, 65:203–213, 1996.

[8] Dominik Moser, Andreas Riener, Kashif Zia, and Alois Ferscha. Comparing Parallel Simulation of Social Agents Using Cilk and OpenCL. In *DS-RT '11: Proceedings of the 2011 IEEE/ACM 15th International Symposium on Distributed Simulation and Real Time Applications*, pages 88–97. IEEE Computer Society, September 2011.

[9] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[10] Michael J. North, Nicholson T. Collier, and Jerry R. Vos. Experiences creating three implementations of the repast

agent modeling toolkit. *ACM Trans. Model. Comput. Simul.*, 16(1):1–25, January 2006.

[11] Paul Richmond, Dawn Walker, Simon Coakley, and Daniela Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in Bioinformatics*, 11(3):334–347, May 2010.

[12] Matthew Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computations*. Manning Publications, November 2011.

[13] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[14] R. C. Tausworthe. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation*, 19:201–209, 1965.

[15] Unknown. Supplement - geographic information systems (gis). Available at http://www.seos-project.eu/modules/agriculture/agriculture-c03-s01.de.html, accessed February 26, 2015.

[16] Jin Wang, Norman Rubin, Haicheng Wu, and Sudhakar Yalamanchili. Accelerating simulation of agent-based models on heterogeneous architectures. In *GPGPU-6: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 108–119, New York, New York, USA, March 2013. ACM Request Permissions.