HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Bachelorarbeit

GPU-Parallelisierung der Flachwassergleichungen in einer Evakuierungssimulation

Hochschule für Technik Rapperswil

Frühjahrssemester 2015

Erstellt: 12. Juni 2015

Autoren Robin Bader
Philipp Meier

Betreuer Prof. Dr. Farhad D. Mehta
Experte Dr. Simon Meier
Gegenleser Prof. Dr. Luc Bläser

Abstract

Diese Arbeit befasst sich mit der GPU-Parallelisierung der Flachwassergleichungen in einer Evakuierungssimulation von Gebäuden. Die zweidimensionalen Flachwassergleichungen werden in der bestehenden Java-basierten Anwendung (Siemens Crowd Control) zur Berechnung von Wasserflutungsszenarien verwendet. Um die Problemstellung zu lösen, folgt in einem ersten Schritt eine Analyse der bestehenden, seriellen und auf hexagonalen Zellen basierende Implementierung der Wassersimulation. Darauf folgt die GPU-parallelisierte Umsetzung mittels NVIDIA CUDA. Zur Sicherstellung der Kommunikation zwischen Java-Applikation und CUDA-Implementierung wird eine JNI-Schnittstelle eingesetzt, die den Datenaustausch zwischen den beiden Plattformen ermöglicht. Die Umsetzung hat gezeigt, dass bereits durch Anpassungen an der Softwarearchitektur und der Datenstruktur eine erhebliche Optimierung bei der seriellen Verarbeitung auf der CPU erreicht werden konnte. Zudem hat sich herausgestellt, dass durch eine parallelisierte Implementierung auf der GPU eine Verbesserung der Berechnungsgeschwindigkeit eines Referenzszenarios um den Faktor 58 gegenüber der Ausgangslage erreicht werden kann.

Management Summary

Ausgangslage

Die Siemens Building Technologies ist eine Division der Siemens AG und bietet diverse Dienstleistungen für Gebäudeplanung- und management an. Mit Siemens Crowd Control wurde ein Werkzeug entwickelt, das die Simulation des Evakuierungsverhaltens von Personen während Katastrophen wie Feuer, Erdbeben oder Wasserüberflutungen ermöglicht. Die für die Simulation von Wasserüberflutungen notwendigen mathematisch sehr aufwendigen Berechnungen verlangsamen die Arbeitsgeschwindigkeit der Anwendung dermassen, dass Bedienkomfort und Nutzen in Frage gestellt werden muss. Es wird vermutet, dass durch die parallele Ausführung der Berechnungen mittels GPU (Graphics Processing Unit) eine signifikante Verbesserung der Berechnungsgeschwindigkeit der Anwendung realisiert werden kann. Diese Arbeit hat zum Ziel, die Machbarkeit der GPU-Parallelisierung bei der Berechnung der Wassersimulation zu zeigen.

Vorgehen und Technologien

In dieser Arbeit wird die bestehende serielle CPU-basierte Implementierung mittels NVIDIA CUDA auf der GPU parallelisiert. Die GPU verfügt gegenüber der CPU über ein Vielfaches an Prozessor-Cores und erlaubt somit einen hohen Parallelisierungsgrad für Berechnungen. Folgende Anpassungen und Veränderungen des Programmcodes waren notwendig:

- Anpassung der Architektur für Zugriff von Java auf GPU

- Implementierung von threadsicheren Berechnungen
- Optimierungen der CUDA-Implementierung hinsichtlich Ausführungsgeschwindigkeit auf der GPU

Ergebnis

Mittels Geschwindigkeitsmessungen wurde ein Vergleich der verschiedenen Implementierungen gegenüber der ursprünglichen Implementierung durchgeführt. Die Berechnungszeit der GPU-parallelisierten Applikation konnte innerhalb eines Beispielszenarios um den Faktor 58 erhöht werden. Bereits durch Änderungen und Optimierungen an der Softwarearchitektur konnte eine vierfache Performance erreicht werden. Die restliche Beschleunigung wurde durch die Parallelisierung auf der GPU erreicht. Nach der Integration der GPU-parallelisierten Implementierung in Siemens Crowd Control wird den Anwendern der Software wieder effiziente und zuverlässige Simulationsergebnisse ermöglicht.

Eigenständigkeitserklärung

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt sind oder mit dem Betreuer schriftlich vereinbart wurden,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben,
- dass wir keine durch Copyright geschützten Materialien (z. B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Rapperswil-Jona, 12. Juni 2015

Robin Bader

Philipp Meier

Danksagungen

Wir danken folgenden Personen für Ihre Unterstützung während der Bachelorarbeit:

- *Prof. Dr. Farhad D. Mehta* für die Betreuung unserer Bachelorarbeit.
- *Dr. Hermann Georg Mayer* und *Dr. Meinhard Paffrath* für viele hilfreiche Erklärungen während der gesamten Arbeit.
- Unseren Partnerinnen und Familien für Rat und motivierende Worte.
- Unseren Korrekturlesern und Korrekturleserinnen.

Inhaltsverzeichnis

Kapitel 1 Einleitung

Mit Siemens Crowd Control wurde ein Werkzeug entwickelt, das die Simulation von Evakuationsszenarien in Zusammenspiel mit dem Fluchtverhalten von Personen während Katastrophen wie Feuer, Erdbeben oder Überflutungen erlaubt. Zusammen mit den Gebäudeplanern können so bereits im Entwurfsprozess mögliche Risikozonen während einer Evakuation erkannt werden. Um den für eine Evakuation optimalen Gebäudegrundriss mit den notwendigen Fluchtwegen zu finden, müssen viele verschiedene Gebäudegrundrisse jeweils neu simuliert und miteinander verglichen werden.

Im Rahmen eines Forschungsprojekts der Europäischen Union wird Siemens Crowd Control fokussiert auf Umweltkatastrophen weiterentwickelt. Besonders relevant sind Szenarien wie Überflutungen, Tsunamis oder Dammbrüche. Die Simulation des Wasserflusses innerhalb des Simulationsmodelles ist eine der Basiskomponenten der Anwendung. Als mathematische Grundlage zur Berechnung der Wasserströme dienen die sogenannten Flachwassergleichungen. Aufgrund der aufwendigen, komplexen mathematischen Berechnungen bei der Simulation von Wasserströmen und der daraus folgenden langen Programmlaufzeiten während der Simulation, sind Bedienkomfort und Anwenderfreundlichkeit der Software stark beeinträchtigt, im weitesten Sinne der Nutzen sogar in Frage gestellt. Diese Arbeit beschäftigt sich mit der Optimierung der Wasserflussberechnungen innerhalb von Siemens Crowd Control.

1.1 Optimierungsansätze

Aus den verschiedenen Optimierungsansätzen wie

- Anpassungen von Algorithmen,
- Einsatz schnellerer und teurerer Hardware
- Parallelisierung des Algorithmus

wird in Abstimmung mit der Auftraggeberin den vielversprechenden Ansatz, die Berechnungen der Flachwassergleichungen mittels einer GPU-Parallelisierung zu analysieren, verfolgt. Argumente für diesen Lösungsansatz bei den Flachwassergleichungen sind:

Explizite Berechnung

Die Berechnung eines Zeitschrittes der Flachwassergleichungen ist nur abhängig von den Werten aus dem letzten Zeitschritt.

Deterministische Berechnung in zufälliger Reihenfolge möglich

Für jede Zelle der Wassersimulation können die neuen Werte aus den Werten der umliegenden Zellen berechnet werden. Durch dieses Argument und die gegebene Explizität folgt, dass jeder Zellwert auch in einer zufälligen Reihenfolge berechnet werden kann und die Lösung immer noch deterministisch bleibt.

Massive Parallelisierbarkeit der Berechnung[1]

Aufgrund des zugrundeliegenden hexagonalen Zellmodells kann die Berechnung, dieser grossen Anzahl Zellen, parallel ausgeführt werden.

Gleiche mathematische Operation auf vielen Zellen

Single instruction, multiple data (SIMD)-Prozessoren wie auf der GPU vorhanden, sind für diese Problemstellung besonders gut geeignet. [20]

Günstige Hardware

Die benötigte Technologie ist in gängigen GPUs enthalten oder lässt sich

1 Bezeichnet, im Gegenteil zu Mehrprozessorsystemen eine Architektur, bei denen die Komplexität des einzelnen Prozessors reduziert ist, um eine höhere Anzahl parallel rechnender Einheiten zu ermöglichen.

einfach nachrüsten.

Aufgrund der beschriebenen Argumente wird eine Beschleunigung der Simulation bei der Parallelisierung auf der GPU, gegenüber einer reinen Central Processing Unit (CPU)-Implementation, erwartet.

1.2 Projektziele

Nachfolgend sind die detaillierten Projektziele aufgelistet.

Möglichkeit für Parallelisierung der Flachwassergleichungen analysieren
Für die Analyse der Parallelisierbarkeit wird die bisherige Implementierung, die auf den Flachwassergleichungen basiert, berücksichtigt. Es muss herausgefunden werden, ob die Berechnung der Flachwassergleichungen überhaupt parallelisierbar ist.

Potential für Parallelisierung zwischen CPU und GPU abschätzen
Die Auftraggeberin hat den Fokus für die Arbeit auf die Parallelisierung mittels GPU gelegt. Das Potential zwischen CPU- und GPU-Parallelisierung soll dabei abgeschätzt werden. Dabei wird auf die Vorteile der beiden Architekturen eingegangen.

Implementierung der GPU-Parallelisierung
Im Rahmen des Projekts wird die Parallelisierung mit geeigneter Technologie umgesetzt. Dazu entsteht eine Integration in die bestehende Simulationssoftware, sodass die Machbarkeit für die Auftraggeberin gezeigt wird.

Empirische Messung der Resultate
Zum Abschluss werden die Ergebnisse empirisch verglichen und bewertet. Für die Bewertung werden neben Zeitmessungen auch Programmanalysen mit geeigneter Software durchgeführt.

Richtigkeit der Implementierung wahren
Während des gesamten Projekts wird die Richtigkeit des Algorithmus gewährleistet.

1.3 Struktur des Dokumentes

Folgend eine Einführung in den Aufbau des Dokumentes:

Hintergrundinformationen
: Erklärt die wichtigsten Begriffe dieser Arbeit und gibt eine kurze Einführung in die GPU-Parallelisierung und die Flachwassergleichungen.

Analyse & Design
: Der Aufbau von Siemens Crowd Control und die Einbindung des Überflutungsalgorithmus werden aufgezeigt. Ebenfalls wird darauf eingegangen, wie die Architektur der Applikation geändert werden muss, um eine Parallelisierung auf der GPU überhaupt zu ermöglichen.

Implementation
: Es wird auf die Funktionsweise des Algorithmus eingegangen. Dazu werden die Schwierigkeiten während der GPU-Parallelisierung erörtert und die Konzepte zur Parallelisierung und Optimierung auf der GPU werden erklärt.

Erreichtes Ergebnis
: Mittels Messungen werden die Ergebnisse der entstandenen Implementierungen ausgewertet und miteinander verglichen.

Schlussfolgerung
: Zum Schluss folgt ein Ausblick inklusive Fazit des Projektes.

Kapitel 2 Hintergrundinformationen

2.1 Siemens Building Technologies

Die Building Technologies ist eine Division der Siemens AG mit internationalen Hauptsitz in Zug, Schweiz. Building Technologies ist weltweit führend auf dem Markt für sichere, energieeffiziente und umweltfreundliche Gebäude und Infrastrukturen. Als Technologiepartner, Dienstleister, Systemintegrator und Produktlieferant verfügt Building Technologies über Angebote für Brandschutz und Sicherheit sowie Gebäudeautomation, Heizungs-, Lüftungs- und Klimatechnik sowie ein breites Angebot zum Energiemanagement. [14]

Die Forschungsfrage für die Arbeit ist von Siemens Building Technologies gestellt worden.

2.2 Siemens Crowd Control

Siemens Crowd Control ist eine Simulationssoftware, die das Verhalten von Menschen in Notfallsituationen simuliert [16]. Diese Software wird von Siemens Building Technologies beim Gebäudeplanungsprozess als Dienstleistung angeboten. Hiermit können kritische Situationen bereits im Voraus erkannt und in der Gebäudearchitektur berücksichtigt werden [15]. Diese Software unterstützt die Simulation verschiedener Katastrophenszenarien wie Erdbeben, Explosionen, Feuer und Wasser. Während des Simulationsverlaufs breiten sich die Katastrophen unterschiedlich aus und die Fluchtwege können eingeschränkt werden. Die simulierten Katastrophen beeinflussen dabei das Verhalten der Personen.

2.2.1 Simulator Aufbau

Der Simulator wurde in der Programmiersprache Java als Desktopapplikation entwickelt. Das Simulationsverfahren basiert auf einem Zellulären Automaten (siehe Glossar Seite 64) mit hexagonalen Zellen. Eine Simulation in Siemens Crowd Control besteht aus folgenden Komponenten:

- Szenario, welche unter anderem folgende Elemente enthält:
  - 1 bis n Stockwerke
  - n Treppen
  - n Wände
  - 1 bis n Personenquellen
  - 1 bis n Evakuationsziele
- Parameter für Szenario
  - Simulationszeit
  - Stressverhalten

2.2.2 ELASSTIC Szenario

Das »Enhanced Large scale Architecture with Safety and Security Technologies and special Information Capabilities[7] (ELASSTIC)«-Szenario ist ein von der Europäischen Union mitfinanziertes, drei-jähriges Forschungsprojekt, das 2013 mit einem Budget von gut fünf Millionen Euro gestartet wurde. Bei diesem Projekt ist das Ziel, die Gebäudesicherheit in Bezug auf Katastrophen bereits beim Planungsprozess zu verbessern[7]. Die Siemens AG ist eine der acht Projektpartner und forscht auf dem Gebiet der Gebäudeevakuation. Dazu entwickelt sie Siemens Crowd Control. Im Rahmen von ELASSTIC entsteht mit »ELASSTIC complex« ein Modellgebäude, das zur Verifizierung der Forschungsresultate dient. Wie in Abbildung 2.1 aus [4] dargestellt, handelt es sich um einen U-förmigen Gebäudekomplex, der im Inneren Platz für Büros, Wohnungen, Einkaufsmöglichkeiten und ein Kino bietet. Das ELASSTIC Szenario wird innerhalb dieser Arbeit als Grundlage für die Messungen und die Verifikation der Resultate verwendet.

Abbildung 2.1: *Visualisierte Abbildung von »ELASSTIC complex«*

2.3 Flachwassergleichungen

Siemens Crowd Control berechnet die Wasserausbreitung anhand den zweidimensionalen Flachwassergleichungen. Das Gleichungssystem beschreibt die Änderung der Wasserhöhe- und geschwindigkeit und berücksichtigt die Strömungen in zwei Dimensionen. Die dritte Dimension (die Höhe) wird gemittelt (tiefengemittelt). Wie in Abbildung 2.2 aus [10] gezeigt, wird pro Zelle die Wasserhöhe- und geschwindigkeit mithilfe der Finite-Volumen-Methode [6] berechnet. Für diese Berechnungen werden die Wassergeschwindigkeit- und Höhe sowie die Reibungsterme vom Zellmittelpunkt und die Flüsse der angrenzenden Zellen berücksichtigt. Siemens Crowd Control unterstützt eine zweidimensionale oder dreidimensionale Ansicht zur Visualisierung der Simulation. In Abbildung 2.3 ist die zweidimensionale Darstellung einer Wasserkatastrophe in Siemens Crowd Control gezeigt.

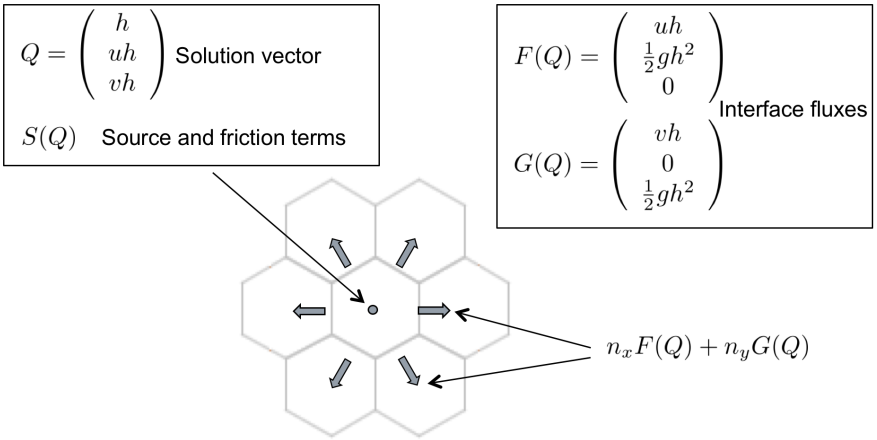


Abbildung 2.2: *Die wichtigsten Einflussgrössen der Flachwassergleichungen*

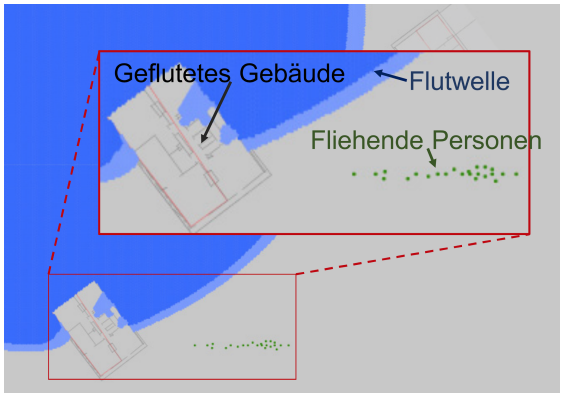
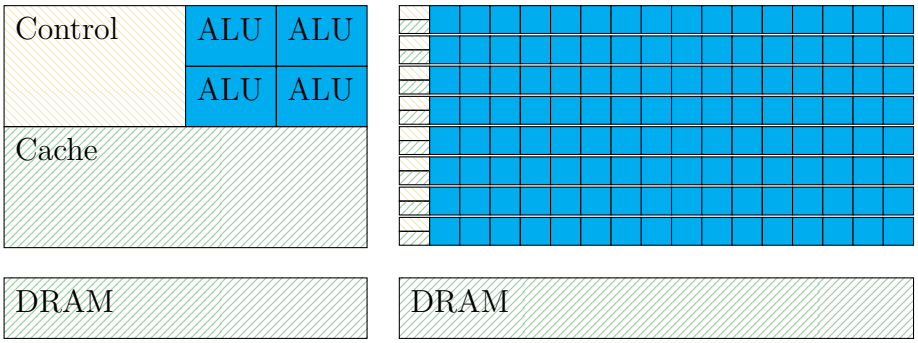


Abbildung 2.3: *Darstellung der Flut in Siemens Crowd Control*

2.4 GPU Parallelisierung

Die Graphics Processing Unit (GPU) ist die Prozessoreinheit der Grafikkarte. Diese unterscheidet sich gegenüber der Central Processing Unit (CPU) durch eine andere Prozessorarchitektur. Eine CPU basiert auf wenigen Cores, die auf eine schnelle, serielle Ausführung ausgelegt sind. Die GPU hingegen basiert auf einer sogenannten »Massively Parallel Architecture«, welche tausende kleinere Cores besitzt, die für die gleichzeitige Berechnung von parallelen Tasks ausgelegt wurden. Auf modernen GPUs befinden sich mehrere Streaming Multiprozessoren (SM). Diese können unabhängig voneinander Berechnungen innerhalb der GPU ausführen. Der Unterschied der beiden Architekturen wird in Abbildung 2.4 aus [8] gezeigt. Wie in Tabelle 2.2 sichtbar ist, unterscheidet sich die GPU durch eine viel grössere



(a) CPU (b) GPU

Abbildung 2.4: *GPU mit sehr vielen Cores (Arithmetic Logic Unit (ALU)) um parallele Aufgaben effizient zu berechnen.*

Anzahl parallel ansprechbaren Cores von der CPU. Auf der GPU werden Operationen nach dem »Single instruction, multiple data (SIMD)« Prinzip ausgeführt. Dies ermöglicht die gleichzeitige Ausführung von gleichartigen Rechenoperationen auf vielen Daten[21]. Bei einer hoch parallelisierbaren Aufgabe kann die GPU so ein Vielfaches der Rechenleistung gegenüber einer CPU erreichen.

	CPU Intel i7-4470 Haswell (Juni 2013)	***GPU NVIDIA Tesla K40 Kepler (Nov 2013)***
Cores	4 Cores	15 SM mit 192 Cores = 2880 Cores
Taktrate	3.5 GHz	745 MHz
Logische Cores / Logische Threads	8 Logische Cores	15 SM mit je 2048 Resident Threads = 30'720
Memory Bandbreite	25.6 GB/s	288 GB/s
L1 Cache Grösse	64 KB/Core	16 KB/SM
L2 Cache Grösse	256 KB/Core	48 KB/SM shared
L3 Cache Grösse	8 MB	–

Tabelle 2.2: *Vergleich CPU und GPU (Quelle [2, Vl. »GPU Parallelisierung«])*

2.4.1 NVIDIA CUDA

CUDA (Compute Unified Device Architecture) ist eine von NVIDIA Corporation entwickelte Programmiertechnik. Mittels dieser Technik können Programmteile auf die GPU ausgelagert werden [19]. CUDA steht unter anderem für die Programmiersprachen C und C++ zur Verfügung.

Kapitel 3 Analyse & Design

Dieses Kapitel beschreibt die Analyse des bestehenden Simulators in Bezug auf die Berechnung der Flachwassergleichungen. In einem zweiten Teil wird aufgezeigt, welche Änderungen an der Architektur vorgenommen werden, um die Berechnungen der Wassersimulation auf der GPU auszuführen.

3.1 Aufbau Siemens Crowd Control

Der Simulator basiert auf einem Zellulären Automaten mit einer hexagonalen Zellstruktur. Aus der architektonischen Struktur des Szenarios wird vom Simulator die Zellstruktur berechnet. Eine symmetrische Anordnung der Hexagone wie in Abbildung 3.1 dargestellt, ist in der Realität selten gegeben. Die einzelnen Zellen werden durch eine Identifikationsnummer im Zellen-Grid identifiziert. Jede Zelle besitzt Informationen, mit welchen Zellen sie benachbart ist. Während der Simulation be-

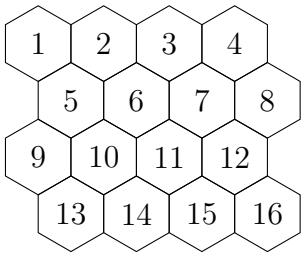


Abbildung 3.1: *Hexagonale Zellstruktur in Siemens Crowd Control*

wegen sich die Personen von Zelle zu Zelle, sofern die Zelle nicht durch statische Objekte, wie beispielsweise Wände oder Tische, blockiert ist. Pro Zeitschritt darf

sich jeweils nur eine Person in der Zelle befinden. Die hexagonale Zellstruktur ermöglicht, gegenüber einer quadratischen Struktur, eine diagonale Bewegung der Personen.

3.1.1 Ablauf einer Simulation

Zur Berechnung wird die Simulation in einzelne Zeitschritte unterteilt. Die Dauer eines Zeitschritts lässt sich im Simulator konfigurieren. Der Ablauf einer Simulation ist wie folgt aufgebaut: Der *Simulations-Kernel* ist der zentrale Bestandteil von

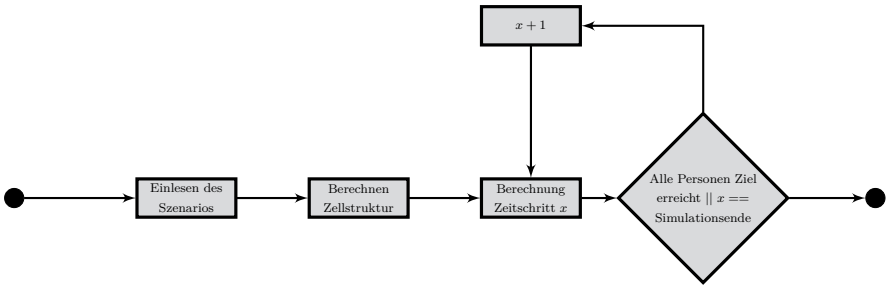


Abbildung 3.2: *Beschreibung des Simulationsablaufs*

Siemens Crowd Control. Er ist für das Berechnen des Zeitschrittes verantwortlich, welcher in der Abbildung 3.2 dargestellt wird. Durch ihn wird das Observer Pattern[1] implementiert, um in jedem Zeitschritt die einzelnen Schritte der Simulation auszuführen. Zur Verdeutlichung der Funktionsweise nachfolgend eine Auflistung der wichtigsten Observer der Simulation:

- *CongestionObserver* steuert den Personenstrom während der Simulation.
- *RepulsionManager* führt die Simulation aller dynamischen Objekte[2] durch.
- *TripwireObserver* erstellt zu allen Personen, die diesen Messpunkt durchlaufen, eine Statistik.

Die folgende Grafik 3.3 visualisiert den Ablauf der Aufrufe auf die Observer der Simulation.

1 Ein Pattern der Softwareentwicklung um Änderungen an einem Objekt an andere, sog. Subscriber weiter zu propagieren

2 Alle beweglichen Elemente eines Szenarios wie beispielsweise Personen, Feuer, Wasser etc. werden als dynamische Objekte bezeichnet

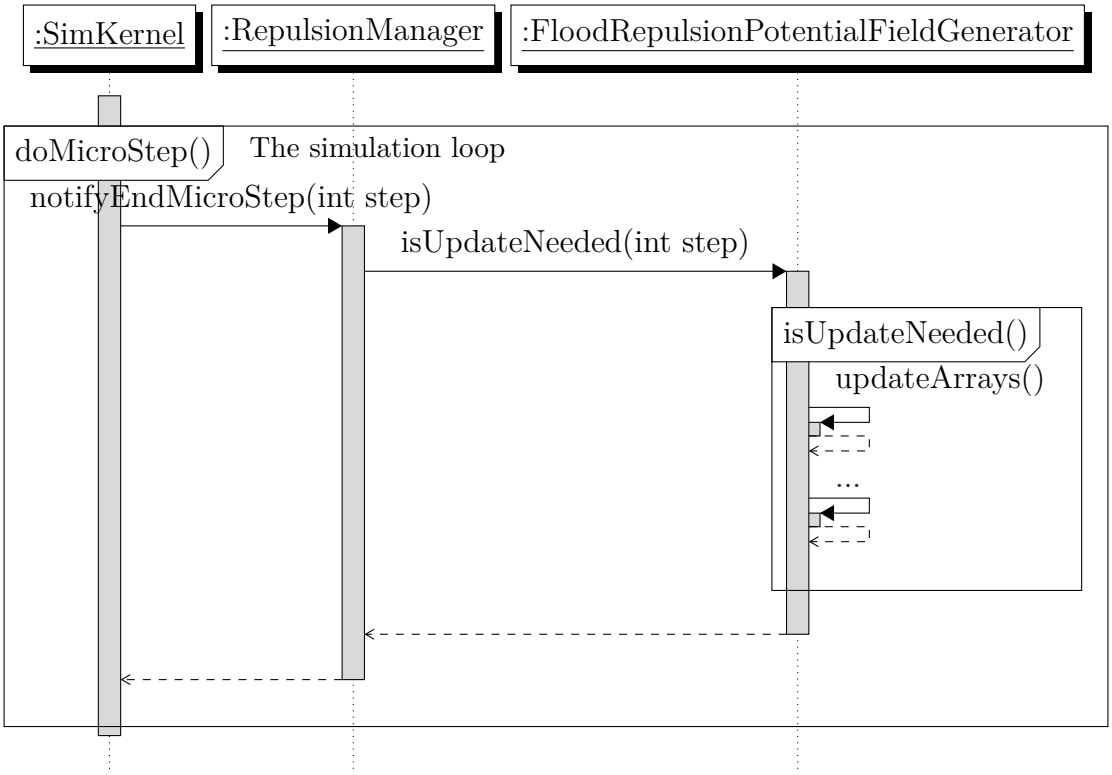


Abbildung 3.3: *Funktionsweise Simulations-Kernel*

Die Klasse *FloodRepulsionPotentialFieldGenerator* implementiert das Interface *RepulsionGenerator* wodurch sie vom *RepulsionManager* nach jedem Zeitschritt aufgerufen wird. Die Klasse *FloodRepulsionPotentialFieldGenerator* implementiert die Flachwassergleichungen und berechnet die Wasserausbreitung der Zellen während der Simulation. Somit bildet der *FloodRepulsionPotentialFieldGenerator* die Grundlage für die zu parallelisierenden Flachwassergleichungen dieser Arbeit. Die Methode *isUpdateNeeded* gibt einen Boolean zurück. Der Rückgabewert gibt an, ob in diesem Zeitschritt eine neue Zelle geflutet wurde, wodurch die Visualisierung aktualisiert wird.

3.1.2 Struktur *FloodRepulsionPotentialFieldGenerator*

Für die Wassersimulation wird ein Objekt der Klasse *FloodRepulsionPotentialFieldGenerator* erstellt. Dabei hat die Klasse folgende Funktionen:

Konstruktion

Erstellung eines neuen Flut-Zellmodells für die Wassersimulation basierend auf dem ursprünglichen Personen-Zellmodell. Projektion des Flut-Zellmodells in mehrere statische Arrays.

Berechnung

Ausführung des Algorithmus der Flachwassergleichungen auf das Flut-Zellmodell.

Datentransfer

Schnittstelle zum Auslesen der berechneten Daten und Umrechnung des Flut-Zellmodells zum Personen-Zellmodells.

3.2 Architekturänderungen

Um diese beschriebene Implementierung in Siemens Crowd Control auf der GPU auszuführen, müssen Änderungen an der Architektur vorgenommen werden. Diese Änderungen können wie folgt zusammengefasst werden:

- Aufteilen der Struktur von *FloodRepulsionManager* in eigene Klasse pro Funktion
- Abschaffung des Flut-Zellmodells für den effizienten und einfachen Zugriff auf das Personen-Zellmodell
- Entkopplung von Referenzen auf Objekte für effizienten, zusammenhängenden Zugriff auf der GPU
- Implementierung einer Bridge zum Transfer der Daten auf die GPU

Die genauen Architekturänderungen werden in den folgenden Unterkapiteln beschrieben.

3.2.1 Aufteilung *FloodRepulsionPotentialFieldGenerator*

Die in Kapitel 3.1.2 dargestellte Struktur zeigt die verschiedenen Funktionen der Klasse *FloodRepulsionPotentialFieldGenerator* auf. Anhand dieser wurde die Klasse in drei einzelne aufgeteilt.

Facade-Klasse

Die Funktion von Siemens Crowd Control soll nicht verändert werden. Damit

das Interface des *FloodRepulsionManagers* gleich bleibt, wird das Facade-Pattern angewendet. Beim Aufruf der einzelnen Funktionen innerhalb der Klasse werden die Aufrufe an die zuständigen Programmteile weitergeleitet. Ausserdem bietet diese Klasse den direkten Zugriff auf die berechneten Werte an.

Factory-Klasse

Die Parameter, die für den Simulator gesetzt werden, sind in der Factory-Klasse zwischengespeichert. Sobald der erste Aufruf der Berechnung stattfindet, erstellt die Factory-Klasse die für die Simulation notwendigen Komponenten (Lazy-Loading). Dazu gehören die Abbildung des Zellmodels auf die jeweiligen Arrays und die Initialisierung der Java Native Interfaces (JNI)-Bridge auf C++.

Datentransfer Java und C++/CUDA

Durch die JNI-Bridge auf C++ wird die Berechnung nicht mehr in Java implementiert, sondern kann nun in C++ beziehungsweise CUDA implementiert werden.

Die Architekturänderungen werden in Abbildung 3.4 nochmals zur Übersicht dargestellt.

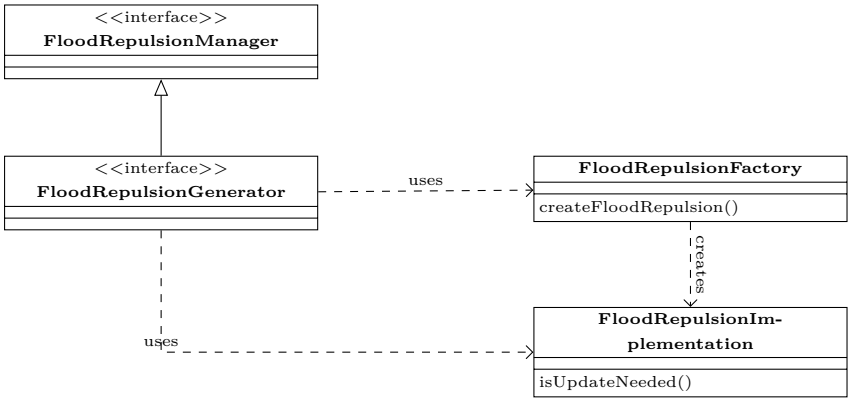


Abbildung 3.4: *Visualisierung der Architekturänderungen*

3.2.2 Abschaffung des Flut-Zellmodells

Neben des Personen-Zellmodells wird ein dediziertes Flut-Zellmodell für die Wassersimulation verwendet. Dadurch ergeben sich folgende Vorteile:

- Erstellen eines Zellmodells mit grösseren Zellen zur schnelleren Berechnung der Flut
- Verwendung von anderen Zellstrukturen wie ein quadratisches Zellmodell

Dieses dedizierte Zellmodell wird innerhalb dieser Arbeit nicht weiter verwendet. Folgendes sind die Gründe dafür:

Vereinfachung des Algorithmus
: Die Projektion zwischen den Zellmodellen sind mit der Berechnung der Flachwassergleichungen in der Siemens Crowd Control Implementierung verknüpft. Um den Algorithmus zu isolieren, musste diese Funktionalität entfernt werden.

Schlechte Performance zur Projektion zwischen Zellmodellen
: Die Projektion wurde durch den Einsatz eines HashSets pro Stockwerk implementiert. Somit musste für jede Zelle zuerst das HashSet des Stockwerks gesucht werden und erst dann konnte auf die Zelle innerhalb des HashSets zugegriffen werden.

Durch die Abschaffung des Flut-Zellmodells konnte der Algorithmus einfacher portiert werden und es konnte ein erheblicher Performance-Gewinn erreicht werden (siehe Kapitel 5).

3.2.3 Entkopplung von Referenzen auf Objekte

Die Werte für die Wasserhöhe und für die Ausbreitung des Wassers müssen pro Zelle zwischengespeichert werden. Pro zu speicherndem Wert wurde ein Array mit der Grösse »Anzahl Zellen« erstellt. Mithilfe der *ID* der jeweiligen Zelle ist definiert, an welchem Index die jeweiligen Daten der Zelle liegen. Diese Datenstruktur ist für den Einsatz von JNI und CUDA aus folgenden Gründen optimal:

Nur primitive Datentypen
: Der Datentransfer von Java auf die GPU ist mit primitiven Datentypen am

einfachsten

Lokale Memory-Zugriffe

Array des jeweiligen Attributs ist im Memory zusammenhängend abgelegt. Dies ist eine wichtige Voraussetzung für die Effizienz der GPU (siehe Kapitel 4.3.2)

Anstatt Referenzen zu verwenden, werden diese durch eine alternative Implementierung mittels eines Nachbar-Arrays ersetzt. Die Abbildung 3.5 zeigt die Zelle mit der ID 4 und deren Nachbarn. Die Abbildung 3.6 zeigt eine Datenstruktur

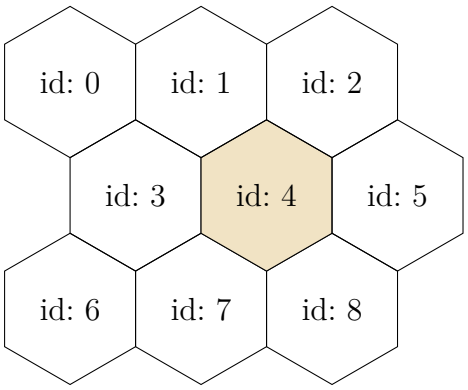


Abbildung 3.5: *Beispiel Zellmodell zur Visualisierung des NeighborId Arrays*

der Wasserhöhen aus Abbildung 3.5. Der Zellwert der einzelnen Wasserhöhen, ist an der jeweiligen Index-Position abgelegt. Beim Zugriff auf die Nachbarn wurden

x	0	1	2	3	4	5	6	7	8
	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Abbildung 3.6: *Wasserhöhen von verschiedenen Zellen mit markiertem Zellwert an Index-Position 4*

für die GPU-basierte Implementierung jegliche Referenzen auf die anderen Zellen abgeschafft. Daher wird anstelle der Liste mit Datenelementen vom Typ *Cell* neu ein Array vom Typ *int* mit der Grösse $Anz\_Zellen \times 6$ erstellt. So können die jeweiligen Identifikationsnummern der Nachbarn gefunden und auf deren Daten

zugegriffen werden. Dieses Array wird von der Factory-Klasse vorberechnet. Um

	0	1	2	3	4	5	6	7	8
0 →	1	2	-1	4	5	-1	7	8	-1
1 ↗	-1	-1	-1	1	2	-1	3	4	5
2 ↖	-1	-1	-1	0	1	2	-1	3	4
3 ←	-1	0	1	-1	3	4	-1	6	7
4 ↙	-1	3	4	6	7	8	-1	-1	-1
5 ↘	3	4	5	7	8	-1	-1	-1	-1

Abbildung 3.7: *Anzeige der Nachbar-IDs im Gegenuhrzeigersinn der Zelle mit der Index-Position 4*

den Zugriff auf die Nachbarn zu verdeutlichen, folgt das Code-Beispiel 3.1. Es zeigt, wie auf die Nachbarn zugegriffen werden kann.

```
1  for(int i = 0; i < numberOfCells; i++) {
2    float accumulatedHeight;
3
4    for(int j = 0; j < nEdges; j++) {
5      int indexNeighbor = i * nEdges + j;
6      int neighborId = neighborIds[indexNeighbor];
7      if(neighborId >= 0) {
8        accumulatedHeight += height[neighborId];
9      }
10   }
11 }
```

Code 3.1: *Vereinfachtes Code-Beispiel für den Zugriff auf Nachbarn*

Weitere Motivationsgründe für diese Datenstruktur werden im Detail in Kapitel 4.3.6 Array of Struct vs. Struct of Array aufgeführt.

3.2.4 Datentransfer auf die GPU

Um den Informationsaustausch zwischen Java-Applikation und der Algorithmusimplementierung mit CUDA zu ermöglichen, wird eine gemeinsame Schnittstelle benötigt. Im Rahmen dieser Arbeit haben wir uns für die Architektur in Abbildung 3.8 entschieden:

- Von *Java* zu *C++* mit JNI
- *C++* auf *GPU* mit NVIDIA CUDA

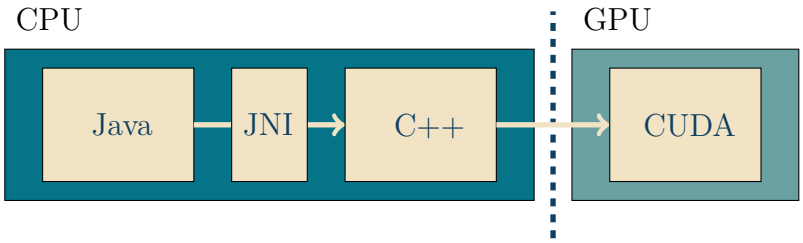


Abbildung 3.8: *Architektur zum Informationsaustausch zwischen Java und CUDA*

Diese Architekturentscheidung basiert auf dem Artikel von Strnad und Konfršt [17], welche die Verwendung von JNI oder JCuda[3] als beste Lösung für die Anbindung von CUDA mittels Java bewerten.

3.2.5 Entstandene Implementierungen

Aus den Architekturänderungen und Vorbereitungen für die GPU sind verschiedene Implementierungen entstanden. Um diese Versionen einfach miteinander zu vergleichen, werden diese wie folgt benannt:

SiemensJava
: Dies ist die ursprüngliche Implementierung der Flachwassergleichungen in Siemens Crowd Control.

SerialJava
: Repräsentiert die angepassten Architekturänderungen und die Entfernung eines eigenen Zellmodells implementiert in Java.

3 Java Library um CUDA-Kernels aus Java einzubinden (siehe `http://www.jcuda.org/`)

ParallelJava

Steht für die Implementierung der Java CPU-Parallelisierung mit dem Einsatz eines Thread-Pools.

SerialCpp

Bezeichnet die Implementierung von SerialJava in C++ mit der Verwendung der JNI.

ParallelGpu

Dies ist die Bezeichnung für die finale Implementierung des parallelisierten Algorithmus auf der GPU.

Kapitel 4 Implementierung

Die vorangegangenen Architekturänderungen ermöglichen die Implementierung der Flachwassergleichungen auf der GPU. Dieses Kapitel beschreibt die Umsetzung der Berechnungen. Dabei werden folgende Implementierungsschritte ausgeführt:

- Erreichen von Threadsicherheit der Implementation
- GPU Optimierungstechniken

Zusätzlich wird die im Kapitel 5 eingeführte Umsetzung der *ParallelJava* ausgeführt.

Im Rahmen dieses Projekts haben sich die Autoren und Projektpartner für den Einsatz der NVIDIA CUDA Plattform für die Parallelisierung auf der GPU entschieden. Diese Entscheidung basiert auf folgenden Faktoren:

- Verbreitung von CUDA-fähigen Grafikchips
- Offizielle Dokumentation
- Keine neue Hardware Anschaffungen nötig

Im ersten Abschnitt wird der Aufbau des Flachwasseralgorithmus erklärt.

4.1 Beschreibung des Algorithmus

Der Algorithmus zur Berechnung der Flachwassergleichungen wird in Abbildung 4.1 dargestellt. Gestartet wird der Algorithmus mit einem Parameter, der angibt wie viele Simulationssekunden simuliert werden sollen. Nach der Berechnung werden die Ergebnisse persistiert und an den Simulations-Kernel zurückgegeben.

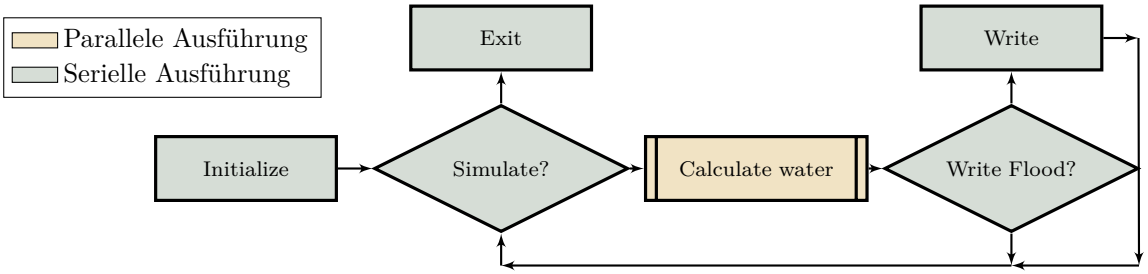


Abbildung 4.1: *Programmfluss der Applikation*

In Abbildung 4.2 ist detaillierter erkennbar, was im Schritt »Calculate Water« aus Abbildung 4.1 berechnet wird. Dabei erfüllen die visualisierten Programmab-

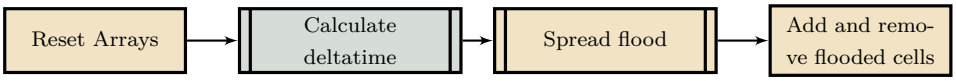


Abbildung 4.2: *Programmfluss von Calculate Water*

schnitte folgende Funktionen:

Reset Arrays

Jede Berechnung eines neuen Zeitschrittes basiert einzig auf den Werten des letzten Zeitschrittes. Diese Explizität wird durch das Zurücksetzen der Zellwerte in diesem Programmabschnitt erreicht.

Calculate deltatime

Die Zeit im Verhältnis zum letzten berechneten Zeitschritt ist bei den Flachwassergleichungen dynamisch. Der Vorteil dabei ist, dass in Zeitschritten in denen sich das Wasser nicht ausbreitet, weniger Zeitschritte berechnet werden müssen. Falls die Zeitschritte detaillierter werden müssen, verkleinert sich die *Deltatime* entsprechend. Die *Deltatime* wird für jeden Zeitschritt neu berechnet.

Spread flood

In diesem Programmabschnitt werden die Wasserausbreitung, -höhen und geschwindigkeiten mithilfe der Flachwassergleichungen berechnet.

Add and remove flooded cells

Neu geflutete oder nun trockene Zelle werden hier hinzugefügt und entfernt um die Berechnung möglichst effizient zu halten.

Das dynamische Zeitintervall *Deltatime* wird wie in Abbildung 4.3 gezeigt, aus der maximalen Wasserhöhe und -geschwindigkeit berechnet. Die Berechnung der *Deltatime* ist somit ein rein serieller Prozess. Die Funktion »Spread flood« aus

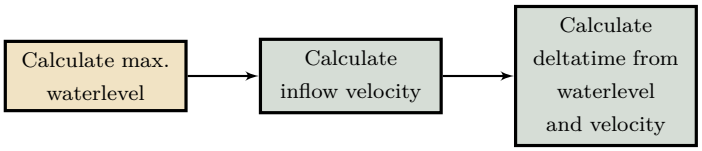


Abbildung 4.3: *Berechnung der Deltatime*

Abbildung 4.4 verwendet die *Deltatime* um den Zeitintervall zwischen den Berechnungen zu bestimmen. In Abbildung 4.4 sind die Teilschritte für die Berechnung

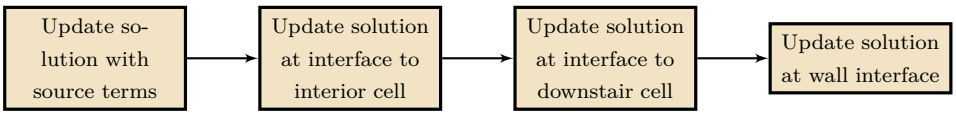


Abbildung 4.4: *Berechnung der Wasserausbreitung*

der Wasserausbreitung visualisiert. Dabei wird jede mit Wasser geflutete Zelle durchlaufen und folgende Funktionen ausgeführt:

- Zellwert bei Zellen mit Wasserquellen erhöhen
- Wasserausbreitung auf jeweils alle Nachbarn der Zelle berechnen
- Wasserausbreitung über mehrere Stockwerke berechnen
- Wasserausbreitung an Zellen, die an Wände grenzen, berechnen

4.2 Threadsicherheit des Algorithmus

Um den Algorithmus zu parallelisieren, ist es wichtig, die Programmstruktur threadsicher zu implementieren. Threadsicherheit heisst, dass sich die einzelnen Programmbereiche bei einer mehrfachen, gleichzeitigen Ausführung nicht gegenseitig

behindern. Die Flachwassergleichungen sind gemäss der Theorie explizit. Die Berechnung eines Zeitschrittes ist also nur abhängig von den Werten aus dem letzten Zeitschritt. Jede Zelle kann dabei seine neuen Werte aus den Werten der umliegenden Zellen berechnen. Daraus folgt, dass jeder Zellwert in einer zufälligen Reihenfolge berechnet werden kann und die Lösung immer noch deterministisch bleibt. Diese Zellen-Unabhängigkeit wird dann wichtig, wenn es um die parallelisierte Lösung des Algorithmus geht. Denn bei einem parallelen Programmablauf gibt es keine Garantie, dass die Zelle N vor der Zelle $N+1$ berechnet wird [6]. Durch diese vom Algorithmus gegebene Unabhängigkeit können dieselben Berechnungen auf vielen verschieden Zellen ausgeführt werden. Bei der Analyse des Siemens gegebenen Algorithmus wurde aber klar, dass nicht alle Bereiche so implementiert wurden, dass diese einfach parallelisiert werden können. Folgend wird beschrieben welche Änderungen an der Implementierung vorgenommen werden mussten, um die Threadsicherheit zu erreichen.

4.2.1 Synchronisationszeitpunkte

Zur zeitlichen Koordinierung der Datenzugriffe von parallelen Threads werden verschiedene Synchronisationszeitpunkte definiert. Dabei müssen die einzelnen Threads auf Zwischenergebnisse zugreifen, welche von anderen Threads berechnet wurden. Diese Synchronisationspunkte werden möglichst minimal gehalten. Es wird versucht einem Thread nur Tasks zu geben, die keine Daten anderer Threads benötigen. Trotzdem braucht es aber noch koordinierte Zugriffe. Diese Synchronisationspunkte sind jeweils nach den in der Abbildung 4.1 – 4.4 visualisierten Teilschritten festgelegt.

4.2.2 Berechnung der Wasserausbreitung

Die Berechnung der Wasserausbreitung benötigt die Werte der jeweiligen Nachbarn. Dadurch unterscheidet sie sich von den anderen Berechnungen innerhalb von Siemens Crowd Control. Das Code-Beispiel 4.1 des Algorithmus von Siemens Crowd Control zeigt, in einer vereinfachten Form, die SiemensJava-Implementierung der Berechnung der Wasserausbreitung.

```
1  void updateSolutionAtInterfaceToInteriorCell() {
2    for (int indexCell = 0; indexCell < numberOfCells; indexCell++) {
3
4      for (int interfaceId = 0; interfaceId < nEdges; interfaceId++) {
5        const int indexNeighbor = indexCell * nEdges + interfaceId;
6        const int neighborId = neighborIds[indexNeighbor];
7        const int idIntfNeighbor = getInterfaceNeighbor(interfaceId);
8        if (neighborId >= 0) {
9          if(!flagInterface[indexCell* nEdges + interfaceId]) {
10           double fluxH = (h[indexCell] - h[indexNeighbor]) / 2;
11
12           h[indexCell] -= fluxH;
13           h[indexNeighbor] += fluxH;
14
15           flagInterface[indexCell* nEdges + interfaceId] = true;
16           flagInterface[neighborId * nEdges + idIntfNeighbor] = true;
17         }
18       }
19     }
20   }
21 }
```

Code 4.1: *Vereinfachtes Code-Beispiel der Wasserausbreitung an benachbarten Zellen*

Dabei kann der Code auch folgend beschrieben werden:

- *(2)* Iteration durch alle Zellen des Zellen-Grids
- *(4)* Iteration durch alle sechs Kanten jeder Zelle
- *(6)* Suche im Array neighborIds nach der ID der jeweiligen Nachbarszelle
- *(9)* Überprüfe in Bitmap FlagInterface, ob Werte von Nachbarn bereits berechnet wurden
- *(10)* Berechne die Hälfte der Differenz der Wasserhöhe von zwei Zellen
- *(12 – 13)* Neuberechnung der Wasserhöhe mit berechneten Werten
- *(15 – 16)* Markieren der berechneten Zellen in Bitmap

Bei zwei benachbarten Zellen sollte, gemäss der gegebenen Explizität der Flachwassergleichungen[3], das in Abbildung 4.5 dargestellte Ergebnis berechnet werden.

Das Code-Beispiel 4.1 enthält konkret drei Probleme, die diesen Code nicht threadsicher machen und nicht das in Abbildung 4.5 erwartete Resultat berechnen.

Abbildung 4.5: *Erwartetes Ergebnis der Berechnung von Nachbarn*

Nachfolgend werden die Lösungen der drei Probleme aufgezeigt.

4.2.2.1 Veränderung voneinander abhängigen Werten

Im gegebenen Code-Beispiel wird die Wasserhöhe der einzelnen Zellen zur Berechnung der Änderung verwendet und anschliessend geändert. Es werden voneinander abhängige Werte verändert. Konkret findet sich das Problem im Code 4.2.

```
10 double fluxH = (h[indexCell] - h[indexNeighbor]) / 2;
11
12 h[indexCell] -= fluxH;
13 h[indexNeighbor] += fluxH;
```

Code 4.2: *Vereinfachtes Code-Beispiel der Veränderung voneinander abhängigen Werten*

Sichtbar wird das Problem in der Abbildung 4.6, welche die Ausführung des Codes 4.2 visualisiert. Durch diese Abhängigkeit ist die Voraussetzung der Explizität

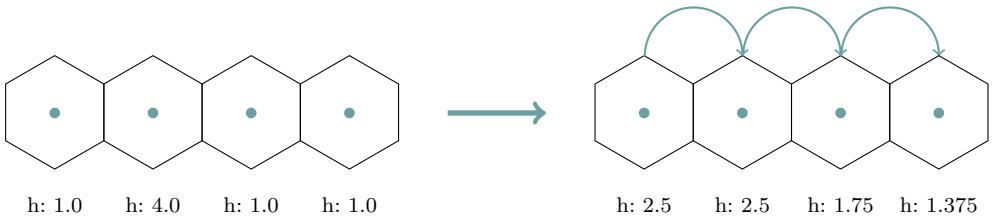


Abbildung 4.6: *Ergebnis bei sequentieller Ausführung mit voneinander abhängigen Werten*

des Algorithmus nicht mehr gegeben. Bei der seriellen Ausführung erhält man nach jedem Durchlauf mit gleichen Parametern das identische Ergebnis. Dies kann bei der Parallelisierung wegen der zufälligen Reihenfolge der Ausführungen nicht

mehr garantiert werden. Die Lösung für dieses Problem ist die Verwendung von unveränderlichen Höhenangaben basierend auf der Ausgangslage des Zeitschritts. Die Einführung dieses Immutable[1] Arrays *h0* sieht folgendermassen aus:

```
10 double fluxH = (h0[indexCell] + h0[indexNeighbor]) / 2;
11
12 h[indexCell] -= fluxH;
13 h[indexNeighbor] += fluxH;
```

Code 4.3: *Vereinfachtes Code-Beispiel der Änderung zu unabängigen Variablen*

Die in Code 4.3 ausgeführten Änderungen haben eine Auswirkung auf das Endergebnis der Berechnung. Nach dieser Code-Änderung unterscheiden sich die berechneten Werte des Algorithmus zur Ausgangslage.

4.2.2.2 Abschaffung des Bitmaps flagInterface

Bei einer parallelen Ausführung der Wasserberechnung kann das in Code 4.4 gezeigte Bitmap *flagInterface* seine ursprüngliche Funktion so nicht mehr erfüllen.

```
9 if(!flagInterface[indexCell* nEdges + interfaceId]) {}
```

Code 4.4: *Vereinfachtes Code-Beispiel des Bitmap FlagInterface*

Falls die Threads zur Berechnung der benachbarten Zellen gleichzeitig an diese Verzweigung kommen, wäre bei beiden Threads die Bedingung *true*. Wie sich diese Race Condition auf das Resultat auswirkt, ist in Abbildung 4.7 dargestellt. Um

Abbildung 4.7: *Parallele Ausführung mit Bitmap*

das Problem zu lösen wird das Bitmap entfernt. Als Alternative wird jeder Zellwert von beiden Zellen berechnet, aber jeweils nur die Hälfte des berechneten Wertes wird addiert. Diese Berechnung ist in Code 4.5 gezeigt.

1 Ein nicht veränderbares Objekt eines Programms wird als immutable bezeichnet.

```
8  if (neighborId >= 0) {
9    double fluxH = (h0[indexCell] – h0[indexNeighbor]) / 2;
10
11   h[indexCell] -= fluxH / 2;
12   h[indexNeighbor] += fluxH / 2;
13 }
```

Code 4.5: *Vereinfachtes Code-Beispiel der Berechnung pro Zelle*

Der Nachteil an dieser Änderung ist, dass der Prozessor die doppelte Anzahl Zellen berechnen muss. Dieser Kompromiss wurde eingegangen. Durch die Branch Divergence[2] der GPU und der SIMD-Architektur ist der Performance-Unterschied aber nicht genau doppelt so hoch. Mit dieser Lösung entsteht ein weiteres Problem. An Zellen deren Nachbar noch nicht geflutet ist, sollte dieser Code eigentlich nur einmal ausgeführt werden. Dies ist zwar weiterhin der Fall, allerdings wird er jetzt auch in diesem Fall durch zwei geteilt. Dieses Problem lässt sich mit der Benutzung des Bitmaps *floodedCells*, gezeigt in Code 4.6, beheben.

```
8  if (neighborId >= 0) {
9    double fluxH = (h0[indexCell] - h0[indexNeighbor]) / 2;
10
11   if(floodedCells[neighborId]) {
12     fluxH = fluxH / 2;
13   }
14
15   h[indexCell] += fluxH;
16   h[indexNeighbor] -= fluxH;
17 }
```

Code 4.6: *Vereinfachtes Code-Beispiel mit Ausnahme von nicht gefluteten Zellen*

Somit wird das gleiche Resultat berechnet ohne die Gefahr von Race Conditions zu haben.

2 Unterschiedliche Verzweigung im selben Warp. Streaming Multiprozessoren (SM) führt Instruktion der einen Verzweigung durch, während die anderen Threads warten müssen.

4.2.2.3 Einführung von atomaren Funktionen

Das letzte Problem im Zusammenhang mit der Threadsicherheit tritt durch die nicht Atomarität folgender Funktionen auf:

```
12 h[indexCell] += fluxH;
13 h[indexNeighbor] -= fluxH;
```

Code 4.7: *Vereinfachtes Code-Beispiel von Nicht-Atomarer Addition*

Das Problem in Code 4.7 ist, dass der Wert gelesen, neu berechnet und dann wieder geschrieben wird. In diesem Zyklus kann der Wert aber schon von einem anderen Thread modifiziert worden sein, ohne dass die Änderung propagiert wurde. Beim Schreiben des anderen Threads, wird die Änderung einfach überschrieben. Durch die SIMD-Architektur der GPU tritt dieses Problem umso häufiger auf. Es braucht also eine nicht unterbrechbare »Read-Modify-Write Operation«. Um dieses Problem zu lösen, wird die Funktion *atomicAdd* der CUDA-Library verwendet. Code 4.8 zeigt die Umsetzung mit *atomicAdd*.

```
12 atomicAdd(&h[indexCell], fluxH);
13 atomicAdd(&h[indexNeighbor], -fluxH);
```

Code 4.8: *Vereinfachtes Code-Beispiel von Atomarer Addition*

Dabei ist zu erwähnen, dass CUDA diese Funktion nur mit Single-Precision-Variablen Hardware-seitig unterstützt. Bei Variablen mit Double-Precision muss eine Software-Lösung implementiert werden, die um einiges langsamer ist als die Hardware-Lösung [9] (siehe Kapitel 4.3.5).

4.3 Angewandte Optimierungstechniken der GPU-Parallelisierung

Für die GPU-Parallelisierung wurden verschiedene Optimierungen vorgenommen, um den Algorithmus möglichst gut auf die GPU-Architektur abzustimmen. In diesem Kapitel wird beschrieben wie diese Techniken angewendet wurden.

4.3.1 Analyse mit NVIDIA Visual Profiler

Für die Optimierung des Algorithmus wurde der NVIDIA Visual Profiler[3] verwendet. Nachfolgend wurde mit dem ELASSTIC-Szenario ein Profil für die Auslastung der GPU aufgezeichnet. Die untere Abbildung 4.8 zeigt einen Durchgang von »Calculate Water« aus Abbildung 4.1, welcher insgesamt 19ms dauerte. In Ta-

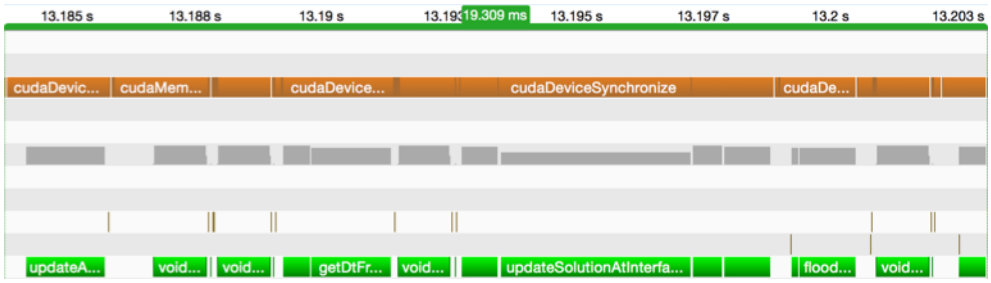


Abbildung 4.8: *Visual Profiler Aufzeichnung eines Durchgangs von Calculate Water*

belle 4.2 sind die detaillierten Ergebnisse zu den einzelnen Programmabschnitten aufgeführt. Besonders auffallend ist der Thread *updateSolutionAtInterfaceToInteriorCell*, der als einziger keine 100% Occupancy (Nutzung) aufweist. Er beansprucht 25% der gesamten Laufzeit innerhalb von »Calculate Water«. Die Occupancy wird berechnet durch die Anzahl aktiver Warp dividiert durch die maximale Anzahl Warps.

Jeder Thread braucht 40 Register. Das bedeutet, dass 10240 Register pro Block benötigt werden. Die eingesetzte NVIDIA GeForce GT 650M bietet pro Block 65536 Register. Das beschränkt die GPU auf 6 Blöcke pro SM die gleichzeitig ausgeführt werden können, obwohl auf der GPU 8 Blöcke pro SM verfügbar wären. Die Anzahl Register in dieser Methode des Threads konnte nicht minimiert werden. Der Grund dafür ist, dass viele Zwischenergebnisse berechnet und gespeichert werden müssen. Der Versuch der Auslagerung von Memory ins »Constant Memory«[4] der GPU hat die Performance verschlechtert und die Anzahl Register nicht

3 https://developer.nvidia.com/nvidia-visual-profiler

4 Schnell verfügbarer Memory-Bereich für Daten, die während der Ausführung unverändert bleiben.

Name	Duration	Registers/Thread	Shared M./ Block	Occupancy Achieved	Occpuancy Theoretical
updateArrays	8.5%	23	0 B	92.4%	100%
thrust::max_element(inflow)	6.3%	27	24 KB	98.7%	100%
thrust::max_element(h)	6.3%	27	24 KB	98.5%	100%
getDtFromWaterLevel	10.1%	28	0 B	86.4%	100%
updateSolutionWithSourceTerms	4.6%	21	0 B	92.4%	100%
updateSolutionAtInterfaceToInteriorCell	25.5%	40	0 B	62.6%	75 %
updateSolutionAtInterfaceToDownstairCell	3.8%	14	0 B	94.6%	100%
updateSolutionAtWallInterface	5.9%	32	0 B	89.2%	100%
floodNewCells	7.1%	18	0 B	88%	100%
removeCells	3.4%	14	0 B	94%	100%

Tabelle 4.2: *Auswertung der einzelnen Programmabschnitten im Algorithmus*

minimiert.

4.3.2 Zusammenhängender Zugriff auf Global Memory

Eine der wichtigsten Performance-Bedingungen auf der GPU ist, dass der Zugriff auf das Global Memory zusammenhängend (coalesced) ausgeführt werden kann. Ein Warp, also eine Gruppe von 32 Threads, sollte so wenig Transaktionen auf das Memory vornehmen wie möglich [8]. Falls Threads innerhalb eines Warps auf zusammenhängende Daten zugreifen, können diese Daten innerhalb einer Transaktion (Memory Burst) aus dem Global Memory geladen werden. Ist dies nicht der Fall, muss die GPU viele zeitaufwendige Zugriffe, z.B. 400 Zyklen pro Zugriff, auf das Global Memory machen [2, Vl. »GPU Parallelisierung«]. Maximal kann die GPU 128 Byte gleichzeitig auslesen, ähnlich zu CPUs die auch volle Cache Lines laden. Für eine maximale Performance sollte jeder Lesezugriff auf das Global Memory zusammenhängende 128 Byte sein [3]. Bei Single-Precision-Variablen sind dies je 32 Variablen, also genau die Grösse eines Warps.

Die Datenstrukturen der Datenwerte der Flachwassergleichungen enthalten die Zellen basierend auf der Reihenfolge des Simulations-Kernel. Dabei werden die Zellen von oben links nach unten rechts durchnummeriert. Somit ergibt sich automatisch ein durch die Datenstruktur zusammenhängender Memory-Zugriff wie in Abbildung 4.9 gezeigt. Der grösste Teil im Algorithmus befolgt dieses zusam-

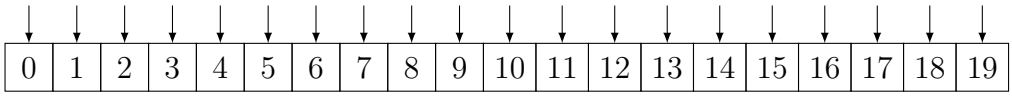


Abbildung 4.9: *Zusammenhängender Zugriff auf Memory*

menhängende Zugriffsmuster. Alle geladenen Datenelemente können also von allen Threads benutzt werden. Einzige Ausnahme ist der Zugriff auf die Werte der Nachbarzellen. In dem visualisierten Beispiel der Abbildung 4.10 müssen die Datenwerte der jeweiligen Nachbarzellen ausgelesen werden. Die Zelle mit der *ID 9* hat als Nachbarn die Zellen mit den IDs *2, 3, 8, 10, 16, 17*. Der Memory-Zugriff ist in Abbildung 4.11 veranschaulicht. In Abbildung 4.11 ist der Zugriff auf die Daten im Global Memory nicht mehr zusammenhängend. Viele der Datenwerte,

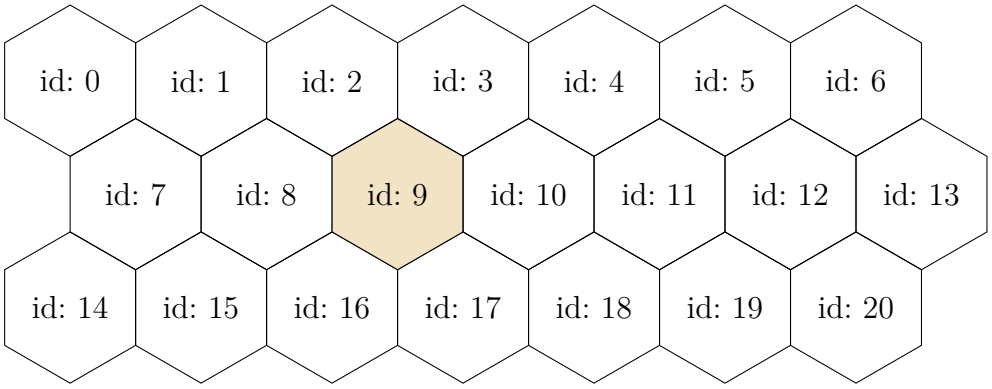


Abbildung 4.10: *Zellengitter zur Veranschaulichung von Coalescing*

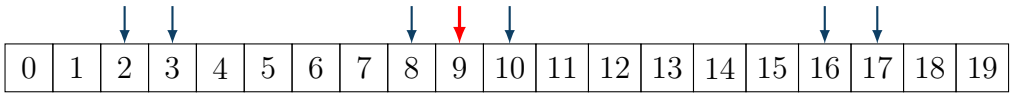


Abbildung 4.11: *Zugriff auf Nachbarzellen*

die im gleichen Schritt geladen werden könnten, werden nicht benutzt. Um die restlichen Daten zu laden, müssen diese einzeln geladen werden, was wiederum zu einer schlechteren Performance führt.

4.3.2.1 Einsatz von Shared Memory

Eine Lösung für das Problem mit nicht zusammenhängendem Memory-Zugriff ist der Einsatz von Shared Memory [6]. Da sich das Shared Memory direkt auf dem Chip befindet, verfügt es über wesentlich schnellere Zugriffszeiten und kleinere Latenz als das Global Memory [8]. Im Artikel von Seitz, Kennedy und Ransom [6] wurde eine Implementierung der Flachwassergleichungen mit dem Einsatz von Shared Memory und eine Implementierung ohne Shared Memory vorgenommen. Bei der Implementierung mittels Shared Memory konnte eine Performance-Steigerung von 10% bis 15% erreicht werden. Die Umsetzung aus dem Artikel [6] unterscheidet sich von der Umsetzung der Wassersimulation in Siemens Crowd Control durch folgende Punkte:

- Quadratische Zellen anstatt Hexagonale Zellen

- Berechnung der Nachbar-IDs möglich
- Grid lässt sich in Zweidimensionaler Datenstruktur zusammenhängend speichern

Innerhalb eines Blockes der GPU konnten aufgrund dieser Voraussetzung alle nötigen Daten innerhalb des Blockes in das Shared Memory der GPU geladen werden. Die *Halo-Zellen* aus der Abbildung 4.12 sind dabei die zusätzlichen Datenwerte, die für die Nachbarn übernommen werden müssen. Diese werden also mehrfach in

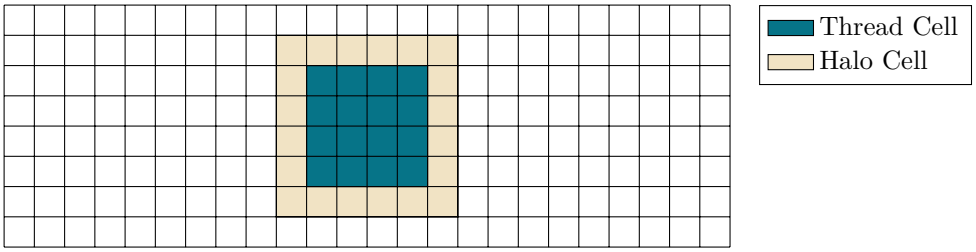


Abbildung 4.12: *Visualisierung der Umsetzung mit Shared Memory [6]*

das Shared Memory geladen (pro Block, der die Nachbarzelle berührt). Durch das Shared Memory konnten Wartezeiten vermieden werden, die auf den Zugriff im Global Memory entstanden wären. Die Grösse des Shared Memory limitiert dabei die Anzahl auszuführender Blöcke pro SM. Dies ist ein Nachteil beim Einsatz von Shared Memory [6].

4.3.2.2 Angewandte Lösung

Da das Hexagonales Grid dieser Arbeit die Anforderungen für die Shared Memory Optimierung nicht unterstützt, konnte diese Optimierungstechnik nicht angewendet werden. Als Alternative bietet die CUDA Plattform an, den nicht genutzten Shared Memory als Cache zu verwenden. In 4.9 ist die Umnutzung des Shared Memory zum Cache gezeigt.

```
1 cudaDeviceSetCacheConfig(cudaFuncCachePreferL1);
```

Code 4.9: *Cache Einstellungen*

Dies führt dazu, dass 48KB der verfügbaren 64KB für Cache verwendet werden. Durch diese Optimierung konnte die Performance bei einer Messung vom ELASS-TIC Szenario um 6% erhöht werden.

4.3.3 Anzahl Threads pro SM

In CUDA sollten folgende Konzepte angewandt werden:

- Auslastung möglichst vieler Threads, um Latenzen zu kaschieren
- Anwendung der gleichen Logik auf möglichst viele Threads
- Optimierung der Memory-Zugriffe

Dieses Konzept kann durch die Ausführung einer Zelle pro Thread erreicht werden. Dies ist im Code 4.10 dargestellt.

```
1  __global__
2  void resetNegativeValues(const unsigned int numberOfCells, float * p_h) {
3    int cellId = blockIdx.x * blockDim.x + threadIdx.x;
4    if(numberOfCells >= cellId)
5      return;
6
7    p_h[cellId] = fmax(p_h[cellId], 0.0f);
8  }
9
10 void resetNegativeValues(const unsigned int numberOfCells, float * p_h) {
11   int numSMs;
12   cudaDeviceGetAttribute(&numSMs, cudaDevAttrMultiProcessorCount, devId);
13   int numberOfThreads = 1024;
14   int numberOfBlocks = numberOfCells / numberOfThreads + 1;
15
16   resetNegativeValues<<<numberOfBlocks ,numberOfThreads>>>(numberOfCells, p_h);
17 }
```

Code 4.10: *Standard CUDA Thread Modell*

So wird auf die Zellwerte zusammenhängend zugegriffen (Coalesced). Allerdings wird eine Lösung, die besser mit der grossen Anzahl Zellen auskommt, bevorzugt. Diese sogenannte *Grid Stride Loop*[5] wird im Code 4.11 gezeigt.

```
1  __global__
2  void resetNegativeValues(const unsigned int numberOfCells, float * p_h) {
3    for(int cellId = blockIdx.x * blockDim.x + threadIdx.x;
4      cellId < numberOfCells;
5      cellId += blockDim.x * gridDim.x) {
```

```
6
7     p_h[cellId] = fmax(p_h[cellId], 0.0f);
8   }
9 }
10
11 void resetNegativeValues(const unsigned int numberOfCells, float * p_h) {
12   int numSMs;
13   cudaDeviceGetAttribute(&numSMs, cudaDevAttrMultiProcessorCount, devId);
14
15   resetNegativeValues<<<32 * numSMs, 256>>>(numberOfCells, p_h);
16 }
```

Code 4.11: *Grid Stride Loop*

Die in Code 4.11 gezeigte *Grid Stride Loop* bietet folgende Vorteile:

Skalierbarkeit und Thread Wiederverwendung

Die Threads werden wiederverwendet.

Debugging

Durch die Anwendung dieser Loops kann man den Code leicht Debuggen indem die Ausführung auf seriell umgeschaltet wird. Dies ist in Code 4.12 gezeigt.

```
1   resetNegativeValues<<<1, 1>>>(numberOfCells, p_h);
```

Code 4.12: *Debugging in Grid Stride Loop*

Automatische Berechnung von effizienten Werten für jeweilige GPU

Diese Technik berechnet automatisch gute Werte für die Blockgrösse und die jeweilige Anzahl SMs indem die Anzahl SMs mit 32 multipliziert wird (was der Anzahl Warps entspricht). Dabei sollten möglichst viele dieser Warps aktiv sein, da eine GPU bis zu 48 aktive Warps pro SM haben kann. Wenn aber zu viele Warps verfügbar sind, gibt es andere Probleme, da die Anzahl verfügbaren Register zu hoch ist. Bei zu wenigen hat die GPU wiederum zu viele Leerlaufzeiten [3].

4.3.4 Thrust Library

Thrust[5] ist eine Library mit parallelen Algorithmen, welche der C++ Standard Template Library (STL) gleichen. Durch den Einsatz von verbreiteten Funktionen ist es mit Thrust möglich einfache, lesbare und effiziente Applikationen zu schreiben [1]. Die Verwendung der Thrust Library in diesem Projekt wird in den nachfolgenden Kapiteln beschrieben.

4.3.4.1 Thrust-Vektor

Der Code konnte durch den Einsatz von Thrust-Vektoren vereinfacht werden. Die Code intensiven Memory-Operationen zum Kopieren der Daten auf und von der GPU können von der Library übernommen werden. Ein Thrust-Vektor ist eine Alternative zu C-Arrays. Durch Zeiger auf die Datenstruktur kann innerhalb eines CUDA-Threads direkt auf die Elemente zugegriffen werden. In der Library enthalten sind der *thrust::host_vector* und der *thrust::device_vector*, die funktional ähnlich sind, wie der *std::vector*. Der einzige Unterschied ist, dass die beiden Thrust-Vektoren um die Kopierfunktion zur/von der GPU erweitert sind. So lassen sich die Daten einfach vom *RAM* in das Global Memory der GPU kopieren. Um in CUDA-Threads mit Zeigern zu arbeiten, bietet der *thrust::device_vector* einen »Raw Pointer Cast« an.

4.3.4.2 Reduktion

Für die Berechnung der maximalen Wasserhöhe in einem Array ist auf der CPU ein sequentielles Iterieren des Arrays nötig. Das Finden des grössten Elements in einem Array kann auch mit einem parallelen Ansatz gelöst werden. Auf der GPU wäre ein Lösungsweg der Einsatz von atomaren Funktionen. Mit diesem Ansatz ist die Performance nicht optimal. Besser geeignet ist dafür ein Reduktions-Ansatz wie in Abbildung 4.13 gezeigt. Die Thrust-Library bietet diese Algorithmen bereits implementiert an. Die Wasserhöhe wird dabei, wie in Code 4.13 gezeigt, berechnet:

5 Webseite: https://thrust.github.io/

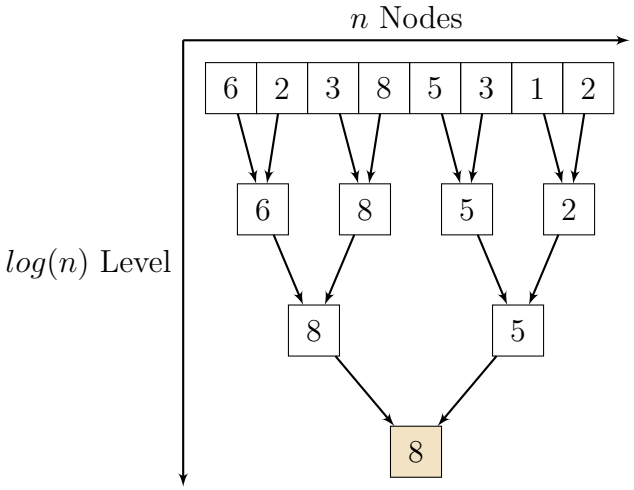


Abbildung 4.13: *Reduktion GPU*

```
1 float getMaximumWaterLevel(thrust::device_vector<float> &d_h) {
2   return *thrust::max_element(d_h.begin(), d_h.end());
3 }
```

Code 4.13: *Berechnung maximale Wasserhöhe*

Diese Vorgehensweise in Code 4.13 ist um einiges effektiver als die serielle Iteration durch das Array. Besonders innerhalb der ParallelGpu-Implementation, da die dafür benötigten Daten bereits im Memory der GPU geladen sind.

4.3.5 Double-Precision zu Single-Precision

Ursprünglich wurde mit CUDA nur Ganzzahl- und Floating-Point-Variablen mit Single-Precision unterstützt. Dies hat sich mittlerweile geändert. Siemens Crowd Control verwendete für die Berechnung der Wasserausbreitung Double-Precision-Variablen. Um eine möglichst hohe Effizienz bei der Programmierung mit der GPU zu erreichen, ist es sinnvoll Single-Precision-Variablen mit tieferer Genauigkeit zu verwenden. Dies liegt an den effizienten Memory-Optionen und an Optimierungsmöglichkeiten wie die Verwendung von *Fast Math* [18]. Weiter wird *atomicAdd* von der der GPU nur für Single-Precision-Variablen unterstützt. Für Double-Precision-Variablen müsste eine, von NVIDIA bereitgestellte, Software Lösung

implementiert werden. Der Einsatz von Single-Precision-Variablen birgt aber den Nachteil, dass die Präzision der Werte eingeschränkt wird. Die Tabelle 4.4 zeigt die Unterschiede zwischen Double-Precision und Single-Precision mit einer Berechnung im ELASSTIC-Szenario. Die Messung der Genauigkeit basiert auf der

	Übereinstimmung der Wasserhöhe mit Original Simulator	***Zeit***
Double-Precision	99.9999999999999%	17098 ms (100%)
Single-Precision	99.9999999999995%	12417 ms (63%)

Tabelle 4.4: *Vergleich Double-Precision gegenüber Single-Precision*

Ausgabedatei der Wasserberechnung, indem jeder Datenwert auf zwei Stellen nach dem Komma vom Simulator gerundet wird[6]. Die gesamten Werte der Wasserhöhen eines Zeitschrittes werden dann addiert und miteinander verglichen. In Tabelle 4.4 ist der Unterschied der Wasserhöhen mit den beiden unterschiedlichen Datentypen dargestellt. Ausserdem werden die unterschiedlichen Berechnungszeiten für das Szenario mit den jeweiligen Datentypen angegeben. Aufgrund dieser geringen Abweichung wurde entschieden, die Implementierung mit Single-Precision Floating-Point-Variablen abzuschliessen.

4.3.6 Array of Struct vs. Struct of Array

Die SiemensJava-Implementierung basiert auf Structure of Arrays (SoA). Während der Umsetzung zur Parallelisierung wurde klar, dass die entstandenen Implementierungen *ParallelJava* und *SerialCpp* aus Kapitel 3.2.5 nicht die optimale Performance beim Einsatz von SoA erreichen. *SerialCpp* ist beim Einsatz von Array of Structures (AoS) 1/3 schneller als die SoA Implementation. Die genauen Vergleichswerte für die beiden Varianten ist in Tabelle 4.6 dargestellt. Auf der GPU ist das Verhalten der Performance zwischen CUDA und C++ genau umge-

6 Die Rundung der Ausgabe wurde für Siemens Crowd Control entsprechend spezifiziert.

	Array of Structures (AoS)	***Structure of Arrays (SoA)***

Beispiel	`struct Cell {`
	`  float h;`
	`  float h0;`
	`};`
	`struct Cell cells[N];`
		`struct Cell {`
		`  float h[N];`
		`  float h0[N];`
		`};`
		`struct Cell cells;`

C++ Performance	9347 ms	16763 ms
CUDA Performance	3690 ms	2634 ms

Tabelle 4.6: *Array of Struct vs. Struct of Array*

kehrt. Dies liegt daran, dass ein AoS ein Coalesced Reading der Daten durch die Anordnung im GPU-Memory verhindert. Dieser Umstand ist in Abbildung 4.14 gezeigt. Bei SoA führt es in CUDA automatisch zu zusammenhängenden Leseope-

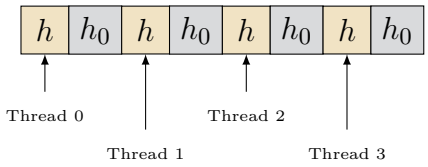


Abbildung 4.14: *Coalescing bei Array of Structures*

rationen. Durch das SIMD-Modell wird pro Array ein zusammenhängender Read erreicht. Aufgrund dessen wurde die komplette Implementierung der Wassersimulation mithilfe des SoA-Prinzips umgesetzt, sodass sich eine Memory-Anordnung wie in Abbildung 4.15 ergibt.

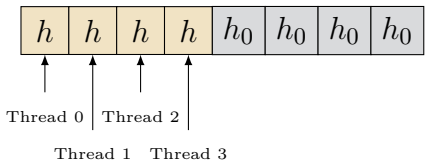


Abbildung 4.15: *Coalescing bei Structure of Arrays*

4.4 Umsetzung der CPU-Parallelisierung

Neben der GPU-Parallelisierung wurde in Java die Version *ParallelJava* umgesetzt. Dabei wurde die Umsetzung des GPU-optimierten und threadsicherem Code zurück nach Java portiert und dann parallel auf der CPU ausgeführt. Die Parallelisierung auf der CPU verwendet als Grundlage einen Thread-Pool mit einer fix definierten Anzahl Threads. Diese Anzahl wurde auf Anzahl Cores der jeweiligen CPU festgelegt. Dabei wurde wie auf der GPU mit einem *Grid Stride Loop* durch die Zellen iteriert. Dies ist im Kapitel 4.3.3 genauer erklärt.

```
1  List<Future<?>> futures = new ArrayList<>();
2  for(int i = 0; i < numberOfThreads; i++) {
3    final int thread = i;
4    futures.add(executorService.submit((Runnable)() -> {
5      for(int cellId = thread; cellId < numberOfCells; cellId += numberOfThreads) {
6        ...
7      }
8    }));
9  }
10 for(Future<?> future : futures) {
11   future.get();
12 }
```

Code 4.14: *CPU-Parallelisierung mit ThreadPool*

Code 4.14 zeigt die Umsetzung von *ParallelJava* mittels »FixedThreadPool«.

4.5 Testing

Um die Richtigkeit der Änderungen am Algorithmus zu überprüfen, wurde die Applikation ständigen Tests unterzogen. Dazu wurde ein Framework in JUnit[7] gebaut. Folgende Anforderungen wurden an das Framework gestellt:

- Einfaches Einlesen von verschiedenen Szenarien
- Variable Simulationszeit
- Möglichkeit für komplette oder schnelle Tests

7 Ein Framwork für automatisierte Unit-Tests in Java

Die Ausgabedatei, welche für die Visualisierung der Flutwelle verwendet wird, enthält alle mit Wasser bedeckten Zellen mit dessen Wasserhöhe. Dabei wird die Wasserhöhe auf cm genau gerundet. Diese Ausgabedatei bietet aus folgenden Gründen eine optimale Voraussetzung für das Testing:

- Ausgabedateien lassen sich leicht persistieren
- Zuordnung zu einzelnen Szenarien ist einfach
- Variable Simulationszeiten sind einfach zu überprüfen

Aus diesen Gründen wurden Unit-Tests verwendet, die diese Textfiles miteinander vergleichen, um Auswertungen über die Richtigkeit der Ergebnisse zu erhalten. Für den Vergleich der Ausgabedateien mussten von den Tests folgende Funktionen unterstützt werden:

Reihenfolge der Ausgabe

Die Zellen werden bei Parallelisierung in unterschiedlichen Reihenfolgen ausgegeben. Dies bedeutet, dass die Reihenfolge der Zellen variabel sein muss.

Toleranz für Abweichungen

Durch Änderungen am Code und durch den Einsatz von Single-Precision-Variablen kann nicht das exakt gleiche Resultat der Berechnungen erreicht werden. Es benötigt bei diesem Vergleich eine Möglichkeit für das Erlauben von Rundungsfehlern. Dabei sollte ausgegeben werden um wie viel Prozent die Ergebnisse voneinander abweichen.

Diese Anforderungen wurden durch das von den Autoren entwickelte Framework abgedeckt.

Kapitel 5 Erreichtes Ergebnis

Die Implementierungen werden in verschiedenen Messungen miteinander verglichen. Diese Messungen haben das Ziel das erreichte Ergebnis der Implementierungen aufzuzeigen.

5.1 Beschreibung der Messumgebung

Die Messungen 1-3 messen die Geschwindigkeit der Wassersimulation in einer isolierten Umgebung. Die Zeit wird für folgende Aufgaben gemessen:

- Berechnung der Wassersimulation
- Erstellen des *FloodRepulsionGenerators* (Konstruktionszeit)
- Datentransfer der Ergebnisse (Kopierzeit)

Alle Zugriffe auf die Peripheriegeräte werden deaktiviert, um eine Analyse der GPU-Beschleunigung, in einem isolierten Umfeld, festzustellen. Es werden also weder Zwischenergebnisse auf die Festplatte geschrieben, noch ist die Visualisierung der Simulation aktiviert. Das Sequenzdiagramm aus Abbildung 5.1 zeigt den Ablauf der Messungen. Die Funktion *updateCache* transferiert die Daten von der GPU über die JNI-Bridge in die Java-Applikation. Das Testsystem der Messumgebung ist in Tabelle 5.2 aufgeführt. Für die Messungen der GPU wird nur die zweite Grafikkarte verwendet, die dediziert für Berechnungen zur Verfügung steht.

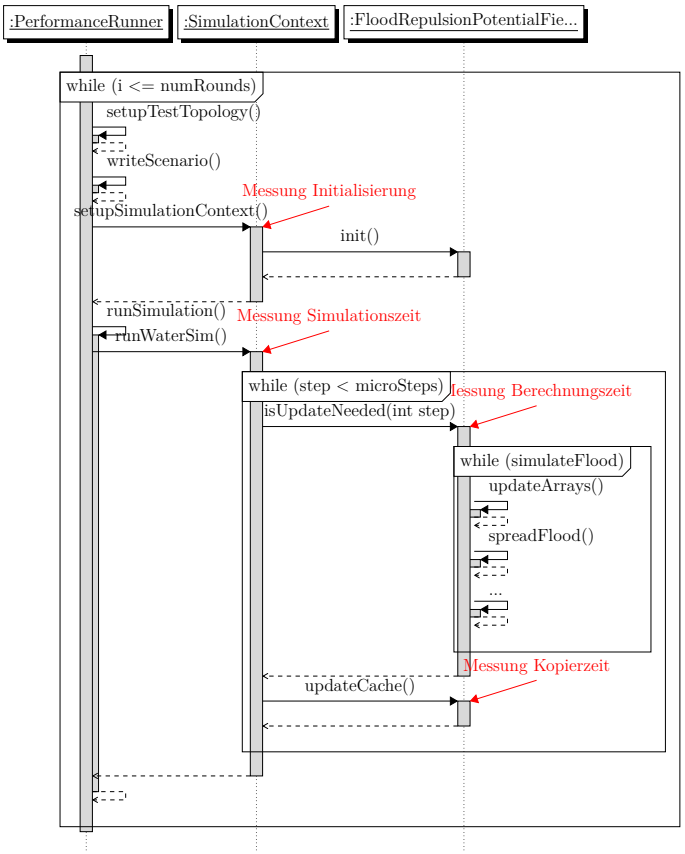


Abbildung 5.1: *Sequenzdiagramm des PerformanceRunner inklusive Markierung der einzelnen Messpunkte*

Bezeichnung	***Typ***
CPU	Intel Xeon E5-2609 @ 2.50GHz
RAM	2.0 GB
GPU	2 x NVIDIA Geforce GTX TITAN Black - 2880 CUDA-Cores - 889 MHz Taktrate - 6.0 GB GDDR5 RAM

Tabelle 5.2: *Testsystem*

5.2 Messaufbau 1: ELASSTIC-Szenario

Diese erste Messung dient dazu, die Berechnungszeiten aller in Kapitel 3.2.5 aufgeführten Implementierungen miteinander zu vergleichen. Dafür wird das Referenzszenario ELASSTIC für alle Implementierungen mit Konfiguration aus Tabelle 5.4 simuliert.

Parameter	***Wert***
Simulationsfläche	60'000 m^2
Anzahl Zellen	1'147'423 Zellen
Simulationszeit	600 Sekunden

Tabelle 5.4: *Konfigurationen für die Testsimulation*

5.2.1 Messergebnis

Die Abbildung 5.2 zeigt die Messergebnisse für die verschiedenen Implementierungen.

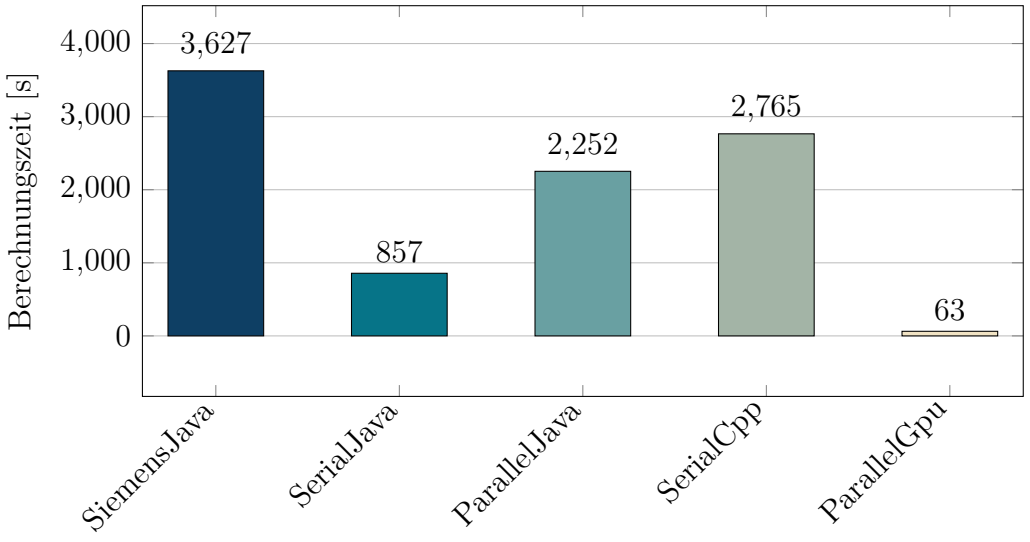


Abbildung 5.2: *Vergleich Berechnungszeit des ELASSTIC-Szenarios*

Nachfolgend werden die Implementierungen mit SiemensJava verglichen.

SiemensJava verglichen mit SerialJava

Auffallend ist der starke Unterschied zwischen SiemensJava und SerialJava. Es ist ein Geschwindigkeitsunterschied von einem Faktor 4.5 messbar. Begründen lässt sich dieser Unterschied in den beiden seriell ausgeführten Implementierungen folgendermassen:

Reduzierte Code-Ausführungen

In SiemensJava oft ausgeführte Berechnungen werden in SerialJava in der Factory-Methode einmal ausgeführt.

Entfernen des dedizierten Zellmodells

Durch die Abschaffung des dedizierten Zellmodells für die Wassersimulation in SerialJava, wird die Komplexität aus SiemensJava zur Umrechnung zwischen den beiden Zellmodellen entfernt.

Diese Messung zeigt, dass die ausgeführten Architekturänderungen einen grossen Beschleunigungsfaktor erzielt haben.

SerialJava verglichen mit ParallelJava

Das Ergebnis von ParallelJava ist sehr erstaunlich. Trotz der 8 logischen Cores auf dem Testgerät ist das erreichte Ergebnis der Parallelisierung um den Faktor 2.5 langsamer als SerialJava. Diese Verschlechterung der Performance im Vergleich zu SerialJava lässt sich folgendermassen begründen:

Threadsicherheit

Um die Threadsicherheit zu gewährleisten sind Synchronisationspunkte nötig, die sich auf die Geschwindigkeit negativ auswirken.

Thread-Aufbau

Die Zeit für das Starten und Stoppen von Threads ist eine weitere Grösse, die sich auf die Performance auswirkt.

Cache-Line Mismatches

Da der Cache durch die Zugriffe auf benachbarte Datenelemente invalidiert wird, kann es zu sogenannten »Cache-Line Mismatches« kommen. Die Autoren vermuten, dass dies der Hauptgrund für den enormen Ge-

schwindigkeitseinbruch ist. Für die Parallelisierung auf Java wurde die GPU optimierte Version ParallelGpu als Grundlage verwendet.

SerialJava verglichen mit SerialCpp

Die C++ Implementation ist im Vergleich zur SerialJava-Version um den Faktor 3 langsamer. Das Problem liegt weder an der Datentransfer-Zeit noch an der längeren Konstruktionszeit der C++ Version (siehe Abbildung 5.3 & 5.4). Die JIT der Java Virutal Machine (JVM) kann dass das Caching-Problem, dass durch weit verstreute Datenzugriffe entsteht, erkennen. Dieser transformiert den Code in eine logisch ähnliche, aber effizientere Reihenfolge. Dadurch lässt sich der Unterschied begründen.

ParallelGpu

ParallelGpu misst die Berechnungszeit des Algorithmus auf der GPU. Diese Messung zeigt das schlussendlich erzielte Resultat der GPU-Parallelisierung der Flachwassergleichungen. Der Vergleich von ParallelGpu zu der schnellsten seriellen Version (SerialJava) zeigt einen zusätzlichen Geschwindigkeitsgewinn um den Faktor 13.6. Das Projekt konnte somit im Vergleich zur ursprünglichen Version SiemensJava einen Performance-Anstieg um den Faktor 57.6 im ELASSTIC-Szenario erreichen.

5.2.2 Weitere Ergebnisse

Neben der gesamten Simulationszeit wurde auch die Zeit zur Konstruktion der Zellstruktur sowie die Zeit für Datentransfer gemessen. Die Abbildung 5.3 zeigt, dass die Konstruktionszeit durch die eingeführte Factory-Klasse bei allen Neuimplementierungen schneller ist als in der ursprünglichen Version. Wie in Abbildung 5.4 gezeigt, ist trotz der JNI-Bridge kein nennenswerter Performance-Unterschied zwischen den C++/CUDA und Java Versionen entstanden. Die Datentransferzeiten aus Abbildung 5.4 sind nur in den Versionen mit JNI-Bridge vorhanden.

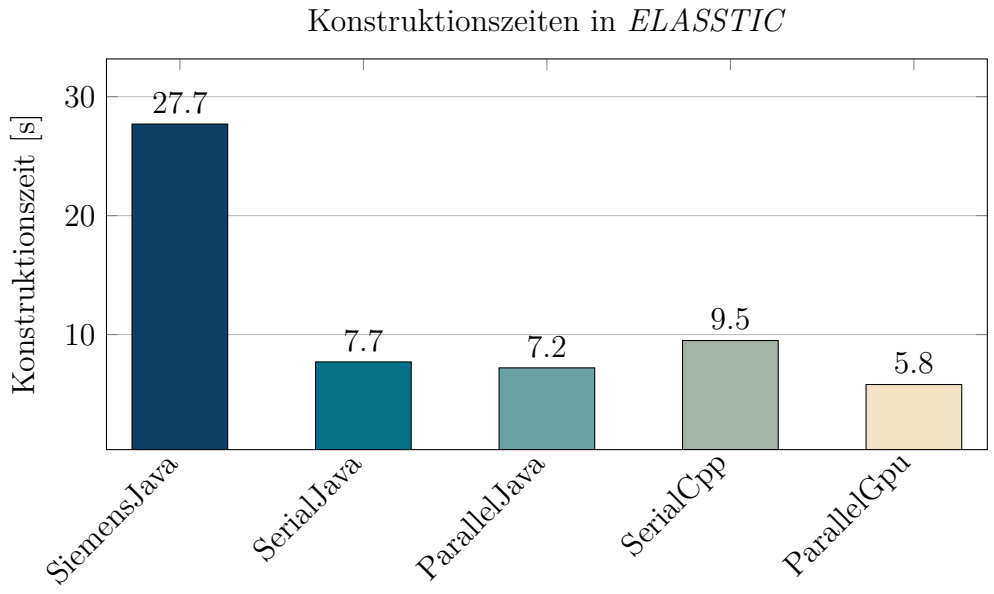


Abbildung 5.3: *Vergleich Konstruktionszeit während ELASSTIC-Szenarios*

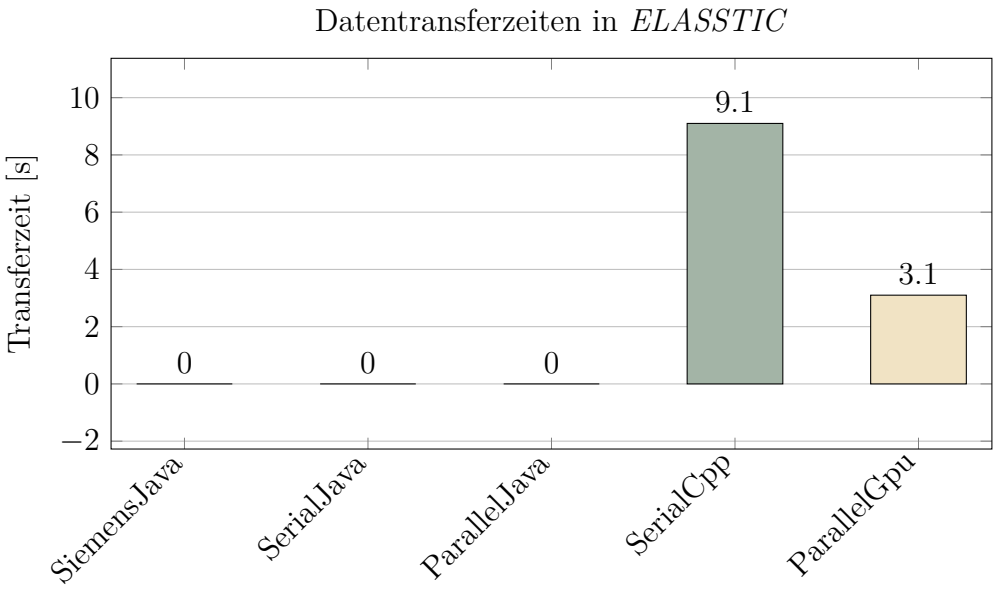


Abbildung 5.4: *Vergleich Datentransferzeit während ELASSTIC-Szenarios*

5.3 Messaufbau 2: Messung mit variabler Szenariogrösse

Das Ziel dieses Messaufbaus ist die Analyse, die aufzeigt, wie sich ParallelGpu im Verhältnis zu SerialJava mit variablen Szenariogrössen verhält. Das dazu erstellte Szenario besteht aus einer quadratischen Fläche mit einer einzigen Wasserquelle in der Mitte des Szenarios. Die Simulation wird mehrfach gestartet und die Fläche wird linear erhöht. Dabei wird die Menge des Wassers aus der Wasserquelle nicht skaliert. Die Simulationszeit bei allen Messungen beträgt 600 Sekunden. Das

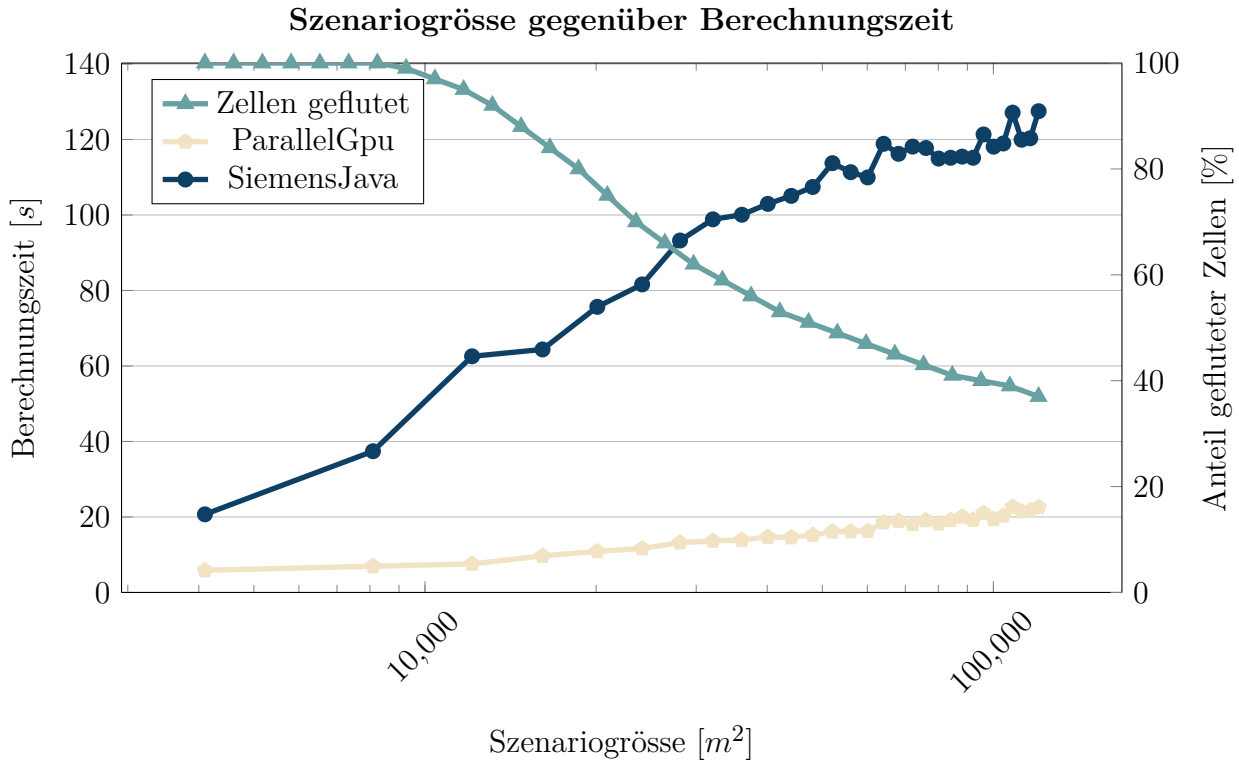


Abbildung 5.5: *Vergleich der gesamten Berechnungszeit des Messaufbau 2*

Ergebnis aus Abbildung 5.5 zeigt, dass ParallelGpu bereits bei einem sehr kleinen Szenario von nur 4'100m^2 schneller ist als SerialJava.

In der sekundären Achse ist die prozentuale Anzahl der mit Wasser gefluteten

Zellen dargestellt. Durch die statische Austrittsmenge von Wasser aus der Wasserquelle, wurde nach 600 Sekunden Simulationszeit bei grossen Szenarien noch nicht das ganze Szenario mit Wasser bedeckt. Hier ist erkennbar, dass SerialJava mit abnehmender gefluteten Wasserzellen schneller wird. Die Auswirkung der Anzahl gefluteten Zellen hat bei ParallelGpu nicht den gleich hohen Einfluss auf die Berechnungszeit. Erkennbar ist dies an der anfangs starken Steigung von SerialJava. Dies liegt an der durch die Branch-Divergence[1] entstehende Wartezeit der GPU-basierten Implementierung. Innerhalb eines Warp müssen alle nicht gefluteten Zellen auf die Berechnung der Wassersimulation warten. Diese Wartezeit fällt auf der CPU weg.

5.4 Messaufbau 3: Messung mit variabler Simulationszeit

Die Messung mit einer variablen Simulationszeit aber einer fixen Fläche soll zeigen, wie sich SerialJava und ParallelGpu mit längeren Laufzeiten verhalten. Das dazu erstellte Szenario besteht aus einer quadratischen Fläche von 120'000 m^2. Dabei wurden mehrere verschiedene Simulationslaufzeiten simuliert. In Abbildung 5.6 ist sichtbar, dass sich ParallelGpu bei steigender Simulationszeit weitaus besser verhält als SerialJava.

1 Durch unterschiedliche Verzweigungen im Code entstehen für Threads Wartezeiten innerhalb des Warps.

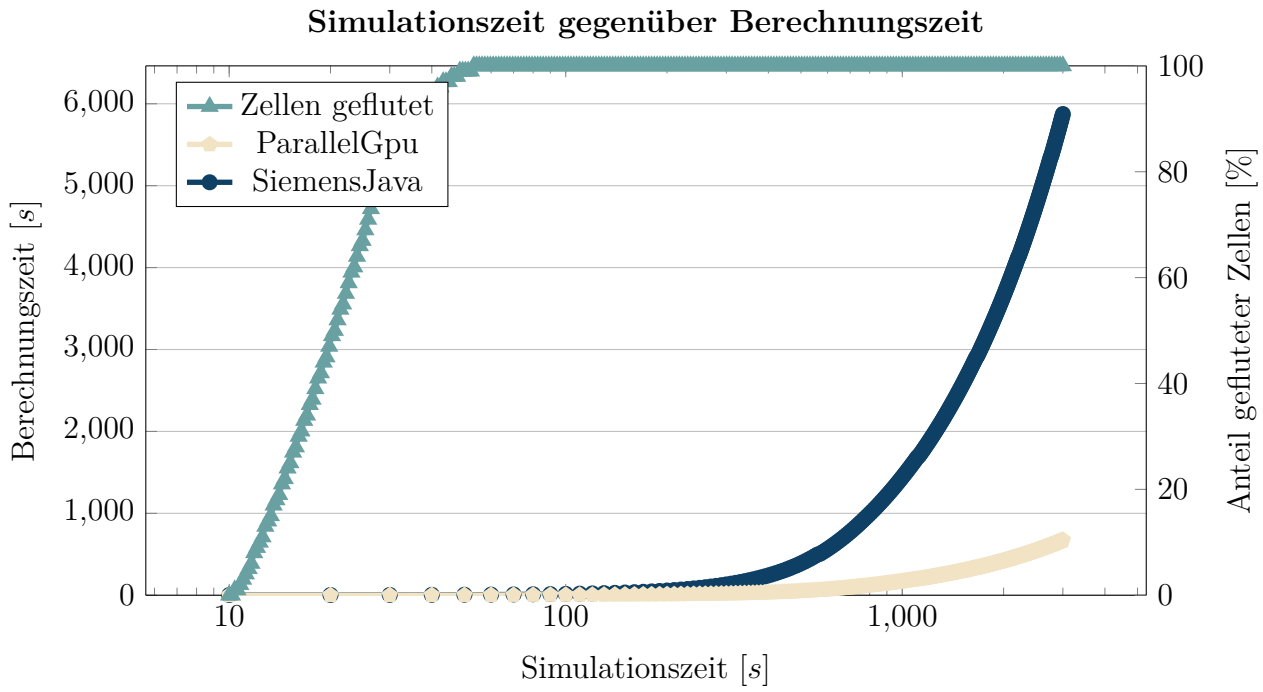


Abbildung 5.6: *Vergleich der Berechnungszeit mit variabler Simulationszeit*

5.5 Messaufbau 4: Messung mit Integration in Siemens Crowd Control

Dieser Messaufbau zeigt, im Gegensatz zu den Messaufbauten 1 - 3, wie sich die verminderte Berechnungszeit auf die Applikation Siemens Crowd Control verhält. Dazu wird das ELASSTIC-Szenario mit SiemensJava und ParallelGpu simuliert. Bei dieser Messung werden weiterhin keine Daten persistiert, jedoch wurde die Visualisierung der Wasserausbreitung aktiviert. Dabei werden beide Berechnungsdurchläufe mit dem Java-Profiler YourKit[2] aufgezeichnet. Die Tabelle 5.6 zeigt die

#	Programmteil	SiemensJava	ParallelGpu
1	*RepulsionManager.notifyEndMicrostep()*	84%	12%
2	*getPedestriansOfNextMicrostep()*	12%	68%
3	*ShowSimulationInGUI.notifyEndMicroStep()*	4%	20%

Tabelle 5.6: *Prozessorauslastung der Siemens Crowd Control Funktionen*

Berechnungszeiten der jeweiligen Implementierung prozentual an. Der Programmteil #1 steht für die Berechnung der Wassersimulation. Auf der CPU wird mit 84% der grösste Teil der Gesamtberechnungszeit für diese Berechnung verwendet. Auf der GPU benötigt dieser Programmteil nur noch einen Anteil von 12% für die Berechnung. Somit hat eine Verschiebung des rechenintensiven Teils stattgefunden.

5.6 Erreichte Ziele

Die in Kapitel 5 aufgeführten Ziele werden folgend erörtert:

Möglichkeit für Parallelisierung der Flachwassergleichungen analysieren
Die bestehende Implementierung der Flachwassergleichungen wurde analy-

2 YourKit nutzt die Profiler-Funktionalitäten der Oracle JVM (siehe `https://www.yourkit.com`)

siert. Die Implementierung von ParallelGpu zeigt, dass die Parallelisierung auf der GPU gewinnbringend möglich ist.

Potential für Parallelisierung zwischen CPU und GPU abschätzen

Es wurden parallele Implementierungen der Flachwassergleichungen sowohl auf der CPU als auch auf der GPU gemacht. Das erreichte Ergebnis aus Kapitel 5.2 vergleicht diese Implementierungen miteinander. Es ist erkennbar dass das Potential zur Parallelisierung auf der GPU sehr hoch ist. Die Parallelisierung der Flachwassergleichungen auf der CPU konnte nicht das gleiche Potential zeigen. Im Rahmen dieser Arbeit wurde die Entscheidung getroffen, sich auf die GPU-Parallelisierung zu konzentrieren, da ein besseres Ergebnis zu erwarten war. Zur Verbesserung der ParallelJava-Implementierung müsste ein zusätzlicher Aufwand investiert werden.

Implementierung der GPU-Parallelisierung

Die bestehende Software Siemens Crowd Control wurde, wie in Kapitel 3.2.4 beschrieben, um eine Bridge erweitert. So können die Daten aus der bestehenden Java-Umgebung auf die GPU transferiert werden. Die Integration konnte durch das entstandene Ergebnis gezeigt werden. Als Technologie zum Zugriff auf die GPU wurde NVIDIA CUDA eingesetzt. Diese Technologie hat sich als stabile Umgebung gezeigt und überzeugte durch das Angebot an Libraries und Werkzeugen. Die Bindung an NVIDIA-GPUs ist bei der eingesetzten Technologie als Nachteil anzusehen.

Empirische Messung der Resultate

Das Projekt wurde durch vier verschiedene Messungen (Kapitel 5.2 bis Kapitel 5.5) verglichen und bewertet. Dabei wurde im Messaufbau 4 (Kapitel 5.5) eine Programmanalyse mit einem Java Profiler durchgeführt. In Kapitel 4.3.1 wurde diese Analyse auch mit einem GPU-Profiler durchgeführt.

Richtigkeit der Implementierung wahren

Das Berechnungsresultat der GPU-Parallelisierung hat als Ziel das gleiche Ergebnis wie die Ausgangslage zu errechnen. Das schlussendliche Resultat weicht, wie in Kapitel 4.3.5 sichtbar, minimal von der Ausgangslage ab. Der Grund für diese Änderung entstand durch eine Änderung am Algorithmus

für die Threadsicherheit (siehe Kapitel 4.2.2.1) und die Änderung der Genauigkeit der Berechnung durch den Einsatz von Single-Precision- anstatt Double-Precision-Variablen (siehe Kapitel 4.3.5).

Kapitel 6 Schlussfolgerungen

Ziel der vorliegenden Arbeit war es, die Machbarkeit der GPU-Parallelisierung bei der Berechnung der Wassersimulation innerhalb von Siemens Crowd Control zu zeigen. Die Machbarkeit konnte mit einer realisierten Umsetzung der GPU-Parallelisierung bewiesen werden. Es stellt sich heraus, dass die Flachwassergleichungen parallelisierbar sind und dass ein erheblicher Geschwindigkeitsanstieg bei der Ausführung auf der GPU im Gegensatz zur Ausführung auf einer CPU messbar ist.

6.1 Vorgehen

- Isolierung der Berechnung der Flachwassergleichungen von den übrigen Berechnungen
- Notwendige Änderung der Softwarearchitektur zur Parallelisierung mit der GPU implementiert
- Implementierung einer JNI-Bridge zum Datenaustausch zwischen CPU und GPU
- Umstrukturierung des Algorithmus für die Threadsicherheit
- Optimierung des Algorithmus für die Ausführung auf der GPU

6.2 Erreichtes Ergebnis

Realisiert wurde eine lauffähige Implementierung der Flachwassergleichungen auf der GPU. Es wurde ein Weg gefunden, um von einer Java Applikation den Datenaustausch GPU zur GPU zu ermöglichen. Weiter wurde die parallelisierte Wasser-

flussberechnung in einem hexagonalen Zellmodell mittels entsprechenden Datenstrukturen ermöglicht. Entstanden ist eine Lösung, die besser mit einer grösseren Anzahl mit Wasser gefluteten Zellen skaliert und die Berechnung schneller ausführt. Die neue Lösung wurde während der Entwicklung stets auf ihre Richtigkeit überprüft. Trotz einiger Änderungen am Algorithmus und dem Rückschritt von Double-Precision zu Single-Precision von Floating-Point-Variablen konnte das Ergebnis seine Richtigkeit im Sinne der Auftraggeberin wahren.

Die durch die GPU-Parallelisierung erreichte Performance-Verbesserung macht die Simulationsanwendung wieder bedienbar und hilft Siemens ihre Zukunftsvision für eine Gebäudesteuerung in Echtzeit weiterzuentwickeln.

6.3 Erfahrungen

Der Einstieg in die Thematik der GPU-Parallelisierung ist eine Herausforderung. Es wird ein breites Grundwissen zur Architektur und dem Ausführungsmodell der GPU vorausgesetzt. Die hardwarenahe Programmierung fordert eine vertiefte Auseinandersetzung mit den technischen Machbarkeiten. Weiter muss mit verschiedenen Optimierungstechniken experimentiert werden um ein möglichst schnelle GPU-Berechnung zu erhalten.

Im Vergleich zu einer reinen Implementierung mit Java werden bei der GPU-Parallelisierung folgende Punkte als Nachteil gesehen:

Einstiegshürde
: Das GPU-Programmiermodell benötigt ein anderes Wissen als die Programmierung mit Java. Dies zeigen auch die oben genannten Erfahrungen.

Bruch im Programmfluss
: Durch die Verwendung von CUDA entsteht ein Bruch im Programmfluss. Die Programmlogik ist nicht mehr zentral in einem Java-Quellcode verfügbar.

Debugging erschwert
: Das Debuggen einer CUDA-Applikation ist je nach Systemumgebung sehr umständlich. Der Grafikkartentreiber lässt zum Teil kein Debugging zu, sofern die Grafikkarte für die Bildschirmausgabe verwendet wird.

Schwierige Fehleranalyse

Die Isolation des Codes für die Fehlersuche in der GPU-basierten Implementierung ist umständlich und komplex.

Eingeschränkte Portabilität

Durch den direkten Zugriff auf die GPU-Hardware entstehen Produkteabhängigkeiten, die einerseits die gesetzten Performance-Anforderungen ermöglichten, andererseits aber die Portabilität der Anwendung auf verschiedene Hardwareplattformen einschränkt.

6.4 Empfehlung

Wir empfehlen die Umsetzung der parallelisierten Berechnung der Flachwassergleichungen auf der GPU innerhalb von Siemens Crowd Control. Dadurch konnte ein signifikanter Performance-Anstieg gemessen werden und die Skalierbarkeit von Siemens Crowd Control konnte erhöht werden. Nachdem das Programmiermodell der GPU verstanden ist, sind Änderungen am Algorithmus und weitere Verbesserungen einfach möglich.

6.5 Ausblick

Nachfolgend Empfehlungen für weitere Forschung, die während des Projekts erkannt, aber aufgrund der beschränkten Zeitdauer des Projekts nicht umgesetzt wurden.

Abstraktion Zellmodell für Wasserberechnung

Wie in Kapitel Analyse & Design (siehe Kapitel 3.2.2) dokumentiert, wird kein dediziertes Flut-Zellmodell mehr für die Berechnung der Überflutung verwendet. Die Gründe für die Abschaffung dieser Projektion werden im Kapitel 4.2.3 beschrieben. Durch diese Abschaffung der Möglichkeit für ein dediziertes Zellmodell gehen aber auch die genannten Vorteile verloren. Durch den Einsatz eines alternativen Zellmodells könnten weitere Optimierungen erreicht werden:

Einsatz eines quadratischen Zellmodells für die Wasserberechnung

Beim Einsatz eines quadratischen Zellmodells mit berechenbaren Nachbarn könnte das im Kapitel 4.3.2.1 erklärte Modell mit Einsatz von Shared Memory verwendet werden. Ausserdem könnte durch die kleinere Anzahl der Nachbarn die Berechnung weiter beschleunigt werden.

Einsatz eines adaptiven Zellmodells

Durch den Einsatz eines Grids, dass sich dynamisch verfeinert, kann die Effizienz der Flachwassergleichungen weiter beschleunigt werden.[12] Deshalb ist die Empfehlung der Autoren die Simulation wieder um diese Abstraktionsstufe zu erweitern. Durch das geänderte Architekturmodell mit Aufteilung in Klassen nach Funktionalität, wäre dies nun einfach in einer dafür spezialisierten Klasse möglich. Um die Vorteile der Performance zu erhalten müsste nur eine Umrechnung in eine Richtung möglich sein, nämlich vom Flut-Zellmodell auf das Personen-Zellmodell. Diese Umrechnung könnte einmalig zwischengespeichert werden bis weitere Änderungen auf dem Flut-Zellmodell ausgeführt werden. Als Umrechnungsgrundlage wird ein statisches Array in Grösse des Flut-Zellmodells, das angibt in welche Zelle des Personen-Zellmodells die Daten geschrieben werden müssen, empfohlen. Dies könnte unter Umständen auch auf der GPU ausgeführt werden.

Berechnung der Nachbarn

Die Nachbarn für eine Zelle im Hexagongrid sind mithilfe eines Arrays, der die Indexe für alle Nachbarn enthält, umgesetzt. Anhand von Algorithmen [11] ist die Berechnung der Nachbarn relativ zum eigenen Index möglich. Dies würde einerseits zu einer Reduktion des Memory-Bedarfs führen, andererseits muss neu geprüft werden, ob der Algorithmus auf der GPU dadurch, bspw. wegen Coalescing nicht verlangsamt wird.

Mehrere GPUs

Anstatt die Applikation nur auf einer GPU zu parallelisieren, könnte der Einsatz von mehreren geclusterten GPUs in Betracht gezogen werden. Dies könnte die Performance der Berechnung weiter beschleunigen und würde

noch grössere Szenarien ermöglichen.

6.6 Fazit

Das Ziel dieser Arbeit war es eine GPU-Parallelisierung der Flachwassergleichungen in einer Evakuierungssimulation zu erreichen. Innerhalb der zur Verfügung stehenden Zeit war eine real einsetzbare Implementierung einer GPU-Parallelisierung der Flachwassergleichungen entstanden. Durch die Geschwindigkeitssteigerung der Berechnung der Wasserausbreitung um den Faktor 58 konnte ein elementarer Beitrag zur Vision der Auftraggeberin geleistet werden. Das Arbeitsresultat ermöglicht der Auftraggeberin Siemens Building Technologies, die auf Umweltkatastrophen fokussierte Weiterentwicklung von Siemens Crowd Control. Zudem kann die erzielte Beschleunigung im Rahmen des Forschungsprojekts der Europäischen Union einen Mehrwert zur Berechnung des ELASSTIC-Szenarios liefern.

Kapitel 7 **Verzeichnisse**

7.1 Abbildungsverzeichnis

7.2 Tabellenverzeichnis

7.3 Quellcodeverzeichnis

7.4 Literatur

[1] N. Bell und J. Hoberock. »Thrust: A Productivity-Oriented Library for CUDA«. In: *GPU Computing Gems Jade Edition* (2012), S. 359–371. URL: https://cloud.github.com/downloads/thrust/thrust/Thrust:%20A%20Productivity-Oriented%20Library%20for%20CUDA.pdf.

[2] L. Bläser. »Parallele Programmierung«. Vorlesungsfolien. 2014.

[3] A. R. Brodtkorb, M. L. Sætra und M. Altinakar. »Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation«. In: *Computers and Fluids* 55 (2012), S. 1–12.

[4] ELASSTIC. *ELASSTIC - About ELASSTIC*. 2013. URL: http://www.elasstic.eu/articles/test.html (besucht am 17. 05. 2015).

[5] Haris, Mark. *CUDA Pro Tip : Write Flexible Kernels with Grid-Stride Loops*. 2013. URL: http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (besucht am 11. 05. 2015).

[6] K. S. Jr, A. Kennedy und O. Ransom. »A GPU Implementation for Two-Dimensional Shallow Water Modeling«. In: *arXiv preprint* (2013), S. 9. eprint: 1309.1230. URL: http://arxiv.org/abs/1309.1230.

[7] WP-LEADERS und V. d. Z. Cor. »ELASSTIC – PERIODIC REPORTING M1-18«. In: (2014), S. 1–4. URL: http://www.elasstic.eu/userdata/file/Public%20deliverables/ELASSTIC%5C_M1-18%5C_Public%20Summary-2014.12.30%5C_FINAL.pdf (besucht am 17. 05. 2015).

[8] NVIDIA Corporation. »CUDA C Best Practices Guide«. In: v7.0.March (2015), S. 75. URL: http://docs.nvidia.com/cuda/pdf/CUDA%5C_C%5C_Best%5C_Practices%5C_Guide.pdf.

[9] L. Nyland und S. Jones. »Understanding and Using Atomic Memory Operations«. In: (2013). URL: http://on-demand.gputechconf.com/gtc/2013/presentations/S3101-Atomic-Memory-Operations.pdf.

[10] M. Paffrath. »Siemens Crowd Control - Flooding of buildings«. Präsentation. 2014.

[11] A. Patel. *Hexagonal Grids*. 2015. URL: http://www.redblobgames.com/grids/hexagons/ (besucht am 06. 06. 2015).

[12] M. Sætra, A. Brodtkorb und K.-A. Lie. »Efficient GPU-Implementation of Adaptive Mesh Refinement for the Shallow-Water Equations«. English. In: *Journal of Scientific Computing* 63.1 (2015), S. 23–48. URL: http://dx.doi.org/10.1007/s10915-014-9883-4.

[13] D. Shiffman. *The Nature of Code*. 2012, Chapter 10.

[14] Siemens AG. *Building technology - Building Technologies - Siemens*. 2015. URL: http://www.buildingtechnologies.siemens.com (besucht am 08. 06. 2015).

[15] Siemens AG. »Siemens Crowd Control - Hilfe«. Softwarehilfe. 2010.

[16] Siemens AG. *Von Big Data zu Smart Data: Simulationssoftware Crowd Control*. 2015. URL: http://www.siemens.com/innovation/de/home/pictures-of-the-future/digitalisierung-und-software/von-big-data-zu-smart-data-evakuierung-von-gebaueden-besser-planen.html (besucht am 11. 05. 2015).

[17] J. Strnad und K. Zdeněk. *Java on CUDA architecture*. 2012. URL: http://wscg.zcu.cz/WSCG2013/!_2013-WSCG-Poster-Proceedings.pdf.

[18] N. Whitehead und A. Fit-florea. »Precision & Performance : Floating Point and IEEE 754 Compliance for NVIDIA GPUs«. In: *NVIDIA white paper* 21.10 (2011), S. 767–75. URL: https://developer.nvidia.com/sites/default/files/akamai/cuda/files/NVIDIA-CUDA-Floating-Point.pdf.

[19] Wikipedia. *CUDA*. 2015. URL: http://www.wikipedia.com/en/CUDA (besucht am 11. 05. 2015).

[20] Wikipedia. *Generalpurpose computing on graphics processing units*. 2015. URL: http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units?oldformat=true#GPU_vs._CPU (besucht am 11. 05. 2015).

[21] Wikipedia. *Stream Processing*. 2015. URL: http://www.wikipedia.com/en/Stream_processing (besucht am 11. 05. 2015).

7.5 Abkürzungen

ALU Arithmetic Logic Unit, S. 9, 60

AoS Array of Structures, S. 39, 40

CPU Central Processing Unit, S. 3, 9, 20, 37, 41, 50, 52, 53, 55

ELASSTIC Enhanced Large scale Architecture with Safety and Security Technologies and special Information Capabilities[7], S. 6, 7, 30, 35, 39, 45, 47, 59, 60

GPU Graphics Processing Unit, S. VII, 2–4, 9–11, 14, 16, 17, 19–21, 28–30, 32, 34, 36–39, 41, 43, 47, 53, 55–58, 60, 61

JNI Java Native Interfaces, S. 15, 16, 19, 20, 43, 47, 55

JVM Java Virutal Machine, S. 47

SIMD Single instruction, multiple data, S. 2, 9, 28, 29, 40

SM Streaming Multiprozessoren, S. 9, 28, 30, 34, 36

SoA Structure of Arrays, S. 39

7.6 Glossar

Atomarität

Von einer atomaren Operation spricht man, wenn eine Sequenz von Daten-Operationen nicht von anderen Operationen unterbrochen werden kann. S. 29, 37

Double-Precision

Bezeichnung für eine Floating-Point-Zahl die zwei Memory-Einheiten im Rechner belegt, also 8 Byte (64 Bit). S. 29, 38, 39, 54, 56

Explizität

Explizität bezeichnet ein Mehrschrittverfahren für die Berechnung, bei dem die Informationen aus den zuvor bereits errechneten Ergebnissen genutzt werden. S. 2, 22, 24–26

Global Memory

Daten die im Global Memory der GPU gespeichert sind, ist sichtbar für alle Threads innerhalb der Applikation (inklusive der CPU). Das Memory bleibt solange alloziert wie das Host-Memory (CPU) den Speicher reserviert. S. 32–34, 37

NVIDIA CUDA

CUDA (Compute Unified Device Architecture) ist eine von Nvidia entwickelte Programmier-Technik, mit der Programmteile durch die GPU abgearbeitet werden können. S. 15, 16, 19, 21, 29, 34, 35, 38–40, 47, 53, 56, 60

Race Condition

Unbeabschtigte Konstellation von gleichzeitigen Codeausführungen, in der das Ergebnis einer Operation vom zeitlichen Verhalten einer anderen Operation abhängt. S. 27, 28

Shared Memory

Daten die im Shared Memory gespeichert sind, sind sichtbar für alle Thrads innerhalb des ausführenden Blocks. Die Daten sind für die Laufzeit des Blocks alloziert. S. 33, 34, 58

Single-Precision

Bezeichnung für eine Floating-Point-Zahl die eine Memory-Einheit im Rechner belegt, also 4 Byte (32 Bit). S. 29, 32, 38, 39, 42, 54, 56

Synchronisationspunkt

Ein Synchronisationszeitpunkt ist ein Steuerungspunkt im Programmfluss, an dem bestimmte Ergebnisse vorliegen müssen. S. 24, 46

Threadsicherheit

Threadsicherheit ist eine Eigenschaft die besagt, dass eine Komponente gleichzeitig von verschiedenen Programmbereichen mehrfach ausgeführt werden kann, ohne dass sich diese gegenseitig behindern. S. 21, 23, 24, 29, 46, 54

Warp

Der Kernel-Code wird in Gruppen von 32 Threads paralell ausgeführt. Diese Gruppe wird Warp genannt. S. 28, 30, 32, 36, 50

Zelle

Eine Zelle ist ein Quadrat oder Hexagon im Gitter des Simulators. S. 2, 6, 7, 11–13, 16–18, 23–28, 32, 33, 35, 41, 42, 45, 50, 56, 58, 60, 62, 67

Zellulärer Automat

Die Defintion von Shiffman [13] beschreibt ein Zellulärer Automat als ein Modell mit n Zellen, wobei das Modell folgende Charakteristik aufweist:

- Die Zellen sind in einem Gitter angeordnet,
- jede Zelle hat ein Status, wobei die Stati finit sind,
- jede Zelle verfügt über mindestens ein Nachbar.

S. 6, 11, 66