universität wien

# MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Parallele Verkehrsflusssimulation von Fußgängern basierend auf physikalischen Kräftemodellen“


verfasst von / submitted by

Jan Siever, BSc


angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2016 / Vienna 2016

| | |
|---|---|
| Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet: | A 066 910 |
| Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet: | Computational Science |
| Betreut von / Supervisor: | Ao. Univ.-Prof. Dipl.-Ing. Dr. Eduard Mehofer |

# Kurzfassung

Gegenstand dieser Arbeit ist die Entwicklung eines massiv parallelen Programms zur Simulation der Bewegung interagierender Fußgänger. Die möglichste exakte Voraussage der einzelnen Bewegungspfade hunderter Personen durch ein physikalisches Kräftemodell ermöglicht die Analyse potentieller Gefahrensituationen und eine entsprechende Planung von sicheren Räumlichkeiten und Verkehrswegen. Eine weitere Anwendung stellt die Darstellung und Animation von Menschenmassen in Filmen und Videospielen dar. Aus den voraussichtlichen Kollisionszeiten der simulierten Personen werden paarweise Interaktionskräfte berechnet, um iterativ die Dynamik jedes einzelnen Verkehrsteilnehmers zu bestimmen. Somit ist das Programm in der Lage Umgebungen und sich darin bewegende Menschenmengen unter Berücksichtigung der individuellen Ziele jedes Verkehrsteilnehmers zu simulieren und die resultierenden Szenarien vorauszusagen. Der Simulationsaufwand erfordert den Einsatz von leistungsstarken Rechnern. In dieser Arbeit werden moderne Grafikprozessoren als Acceleratoren eingesetzt, um die Ausführungszeiten zu minimieren. Die hochoptimierte parallele Programmversion basiert auf OpenCL und gewährleistet Portabilität zwischen GPUs unterschiedlicher Hersteller. Die Ergebnisse der Simulation werden anhand von repräsentativen Beispielszenarien überprüft. Die erreichten Laufzeiten und Speedups gegenüber einer seriellen Implementierung machen die in dieser Arbeit entwickelte Simulationssoftware zu einer attraktiven Möglichkeit zur Verkehrsflusssimulation von Fußgängern.

# Abstract

The goal of this thesis is the development of a massively parallel program simulating pedestrian dynamics. Predicting the trajectories as precisely as possible for hundreds of persons via physical force-models enables the analysis of dangerous traffic situations and corresponding construction of buildings and traffic infrastructure in a safe way. The representation and animation of large crowds in movies and video games is another attractive field of application.

The expected times of collision for the simulated pedestrians are calculated via pairwise interaction forces, and the dynamics of each person are determined iteratively. Therefore, the program is able to simulate moving crowds and their surroundings with respect to their individual goals and predict the resulting scenarios.

The complexity of the simulation requires the use of powerful computers. Modern GPUs are used as accelerators to minimize the running times. The highly optimized parallel version of the program based on OpenCL guarantees portability with respect to different GPU vendors. Representative simulation scenarios are used to test and compare the results. The running times and speedups compared with a serial implementation show the simulation software to be an appealing method for simulating large crowds.

# Inhaltsverzeichnis

# 1 Motivation und Hintergrund von Fußgänger-Verkehrsflusssimulationen

## 1.1 Motivation

Die Dynamik und Interaktion von Menschenansammlungen bei großen Veranstaltungen, im öffentlichen Straßenverkehr, aber auch in engen Räumlichkeiten in Gebäuden ist Gegenstand zahlreicher Studien unterschiedlichster Fachgebiete aus Sozial- und Naturwissenschaften. Die Voraussage von gruppendynamischen Bewegungsmustern sowie den möglichst exakten Trajektorien der einzelnen Personen kann als Grundlage zur entsprechenden Planung von Räumlichkeiten oder Gelände dienen.

Lebensgefährliche Situationen wie Massenpaniken aufgrund zu dichter Menschenansammlungen und unvorteilhaft träge Verkehrswege, sollen im Vorhinein berechnet und somit im besten Fall vermieden werden. Selbst auf streng überwachten Freiraumkonzerten in ganz Europa gibt es regelmäßig Verletzungen und sogar Todesfälle bedingt durch zu hohe Personendichte und dadurch hervorgerufene Massenpaniken.

Die Simulation entsprechender Szenarien stellt somit ein Gebiet von höchstem Interesse für Architektur, Landschaft- und Verkehrsplanung sowie Organisatoren von Massenveranstaltungen dar. Besonders in der Verkehrsforschung ist die Dynamik interagierender Personen und die individuelle Planung der Pfade der einzelnen Verkehrsteilnehmer maßgeblich für die Sicherheit des entstehenden Verkehrsbildes.

Das Interaktionsverhalten der Verkehrsteilnehmer untereinander stellt die größte Aufgabe zur Voraussage des Bewegungsverhaltens dar. Das Aufeinandertreffen zweier Personen auf einem engen Gehweg und das daraus resultierende Ausweichverhalten unterliegt individuell komplexen psychologischen Aspekten, die formal nur in statistischer Hinsicht sinnvoll erscheinen. Die Entwicklung zutreffender Modelle ist keiner eindeutigen wissenschaftlichen Fachrichtung zuzuordnen und bedarf je nach Anwendungsgebiet psychologische, mathematische und physikalische Methoden. Das dynamische Verhalten von Fußgängern spielt auch in der Film- und Spielindustrie eine wichtige Rolle. Entsprechende simple Modelle werden zur Darstellung von Menschenansammlungen im Aktionshintergrund verwendet, wobei hier das Hauptaugenmerk weniger auf Genauigkeit als auf möglichst effizienter Durchführung der Voraussage liegt.

## 1.2 Methoden zur Simulation interagierender Fußgänger

Den meisten gängigen Modellen zur Simulation von Fußgängern liegen einfache aber fundamentale Beobachtungen zugrunde. Menschen bewegen sich unbeeinflusst mit unterschiedlichen bevorzugten Geschwindigkeiten abhängig von Geschlecht, Alter, Gewicht, Ziel und persönlicher Befindlichkeit. Die Verteilung des Geschwindigkeitsprofils ist Grundlage von Simulationen und je nach Szenario aus empirischen Messungen zu bestimmen. Als Richtwert sich im Alltag entspannt bewegender Fußgänger gilt eine normalverteilte Geschwindigkeit mit Erwartungswert von 1.3-1.4 m/s **[1]**. Falls ein Fußgänger ein bestimmtes Ziel ansteuert bewegt er sich intuitiv auf dem kürzesten Weg dorthin. Diese Annahme trifft für einzelne "schlendernde" Personen wie es auf z.B. Einkaufsstraßen zu beobachten ist nur bedingt zu. Die Interaktion mit anderen Verkehrsteilnehmern hängt von Bewegungsrichtung, Bewegungsgeschwindigkeit sowie Aufenthaltsort aller umgebenden Personen und Hindernisse ab und beruht auf einer möglichst kollisionsfreien Fortbewegung. Dieses Bestreben ist bei besonders dicht gepackten Szenarien natürlich kaum erfüllbar, da Personen zwangsläufig kollidieren. Falls aufgrund der Interaktion mit anderen Verkehrsteilnehmern oder Hindernissen eine Veränderung der Bewegungsgeschwindigkeit zu beobachten ist, sind Personen bestrebt ihre bevorzugte Geschwindigkeit möglichst rasch wiederaufzunehmen. Neben qualitativer Aussagen gruppendynamischer Verhaltensmuster aus Psychologie und Sozialwissenschaften existieren eine Vielzahl an anerkannten Methoden zur computergestützten Simulation großer Menschenmengen.

Folgende Beispiele für erfolgreiche Ansätze sind weitverbreitet in Verwendung:

**Zelluläre Automaten:** Bestehend aus einer Menge von Zuständen in denen sich eine Person befindet, einer zellulären Einteilung des Simulationsgebiets und eines entsprechenden Gesetzes, welches diese Zustände ineinander überführt, werden Personen zu diskreten Zeitpunkten von einer Zelle zur nächsten versetzt.

**Flocken- und Flussmodelle:** Bewegungsgeschwindigkeiten und Richtungen sind abhängig von Dichte bzw. Geschwindigkeiten der Personen in nächster Umgebung. Verkehrsteilnehmer mit ähnlichen Zielen und Pfaden nehmen auch ähnliche Geschwindigkeiten an, ein Prinzip aus der Beobachtung massendynamischer Systeme wird hier also als Grundlage der Berechnung herangezogen.

**Soziales Kräftemodell:** Seinen Ursprung hat diese Methode in der Simulation von Teilchen-Bewegungen in der Molekulardynamik. Zugrunde liegt die Annahme das zwischen interagierenden

Personen, Zielen und auch Hindernissen paarweise Kräfte wirken. Durch ein entsprechendes Kraftgesetz werden Beschleunigung und Positionsveränderung jedes einzelnen Teilnehmers möglichst exakt über den vollen Simulationszeitraum berechnet. Diese rechenintensive Methode steht und fällt mit der Güte des verwendeten Kraftgesetzes und der Integrationsmethode der sich daraus ergebenden Bewegungsgleichungen.

In dieser Arbeit werden Simulationen mithilfe eines später vorgestellten Kräftemodell realisiert welches abhängig von voraussichtlicher Kollisionszeit mit der Umgebung und anderen Fußgängern sowie aufgrund derzeitiger Position und Geschwindigkeit die paarweisen Wechselwirkungen berechnet und aufsummiert.

## 1.3 Grenzen, Genauigkeit und statistischer Charakter der Simulationsdaten

Die möglichst exakte Vorhersage und Rekonstruktion der Bewegung eines einzelnen Individuums ist aufgrund der komplexen psychologischen, physikalischen und biologischen Zusammenhänge mit vertretbaren Aufwand derzeit kaum realisierbar. Selbst nur zwei einander ausweichende Personen verhalten sich bei Aufeinandertreffen mit exakt gleichen Umweltbedingungen für gewöhnlich unterschiedlich. Bei der Planung von Gebäuden und Gelände ist im Vorhinein oft nicht im Detail bekannt, welche Personen mit welchen Eigenschaften einzuplanen sind. Es bleibt nichts übrig, als eine gewisse Verteilung der Simulationsparameter aus realen Messungen zu verwenden. Dennoch lässt sich über eine zeitliche und räumliche Mittelung eine gewisse Charakteristik, zumindest Geschwindigkeitsprofil und Personendichte betreffend, beobachten.

Ein Dynamikmodell welches die einzelnen Pfade mit zugehörigen Geschwindigkeiten zu jedem Zeitpunkt bestimmt soll, obwohl für den tatsächlichen Pfad einer einzelnen Person eventuell als zu ungenau empfunden, die korrekten gemittelten Menschendichten und Problemgebiete korrekt voraussagen. Es ist nicht möglich bzw. zu aufwendig alle Eigenschaften jeder einzelnen Person in Betracht zu ziehen, bei größeren Menschenansammlungen geht man jedoch davon aus das kleine Unterschiede im Bewegungsmotiv nicht ins Gewicht fallen. Die Vorhersage für Gebiete mit gefährlich hoher Menschendichte ohne Ausweg, stockender Bewegung und panikartiger Beschleunigung von Personen sollte in jedem Fall möglichst zutreffende Ergebnisse liefern, um derartige Situationen in den entsprechenden realen Umgebungen vermeiden zu können.

## 1.4 Umfang und Szenarien

Die Berechnung der Trajektorien einzelner Fußgänger durch ein Kräftemodell ist durch nicht berücksichtigte psychologische und physiologische Details der einzelnen Personen unausweichlich Fehlern unterworfen. Gemittelte Größen wie Menschendichte und Evakuierungszeit eines Gebäudes müssen jedoch möglichst korrekt von der Simulation bestimmt werden. Abhängig vom räumlichen Umfeld kann die Simulation von 50-100 Personen, beispielsweise in einem Restaurant, aber auch 1000 oder mehr wie auf großen Messen, Konzerten oder Sportereignissen von Interesse sein. Die möglichst exakte Berechnung der Pfade sämtlicher Besucher solcher Veranstaltungen erfordert die Rechenkapazität modernster Hardware und entsprechender Algorithmen. Die in dieser Arbeit vorgestellten Methoden und Software sind in der Lage derartige Menschenmassen und Räumlichkeiten zu simulieren und zu untersuchen. Kleinere Szenarien welche in Räumlichkeiten beschränkter Kapazität, beispielsweise bei Evakuierung eines Gebäudes und entsprechenden Fluchtwegen, stattfinden können auf größere Menschenmengen bei gleicher Raumgeometrie skaliert werden.

### 1.4.1 Korridorszenario

Eine große Anzahl Menschen bewegen sich, wie in Abb. 1 dargestellt, aus gegenüberliegenden Richtungen durch einen Korridor, laufen also aneinander vorbei, ähnlich der Bewegung von Menschenmassen auf dem Gehsteig einer größeren Einkaufsstraße.


**Abbildung 1** *Korridor Szenario*

### 1.4.2 Fluchtszenario

Eine große Anzahl Menschen versucht gleichzeitig aus einem offenen Bereich durch einen verhältnismäßig schmalen Ausgang in einen kleineren Fluchtweg einzuströmen um den Bereich zu verlassen. Dieser Vorgang ist in Abb. 2 dargestellt.

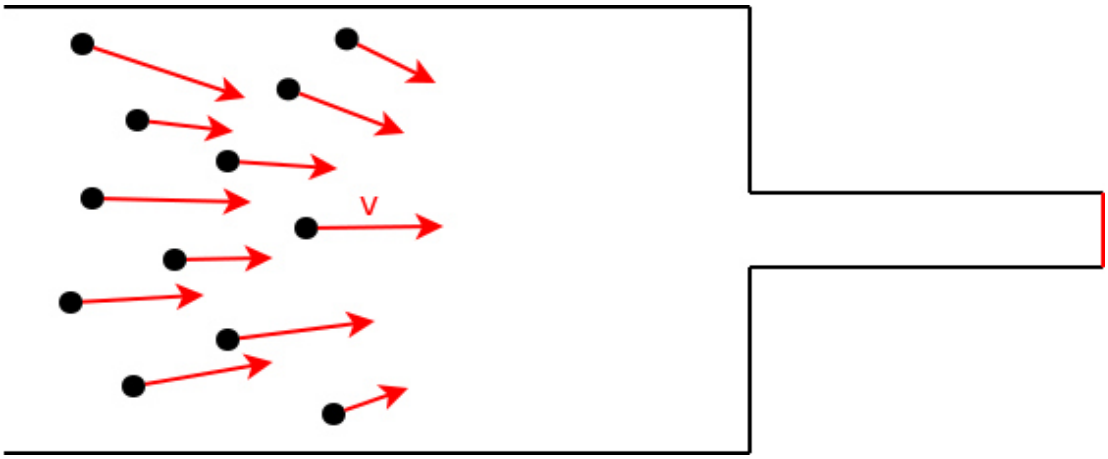


**Abbildung 2** *Flucht Szenario*

### 1.4.3 Kreisszenario

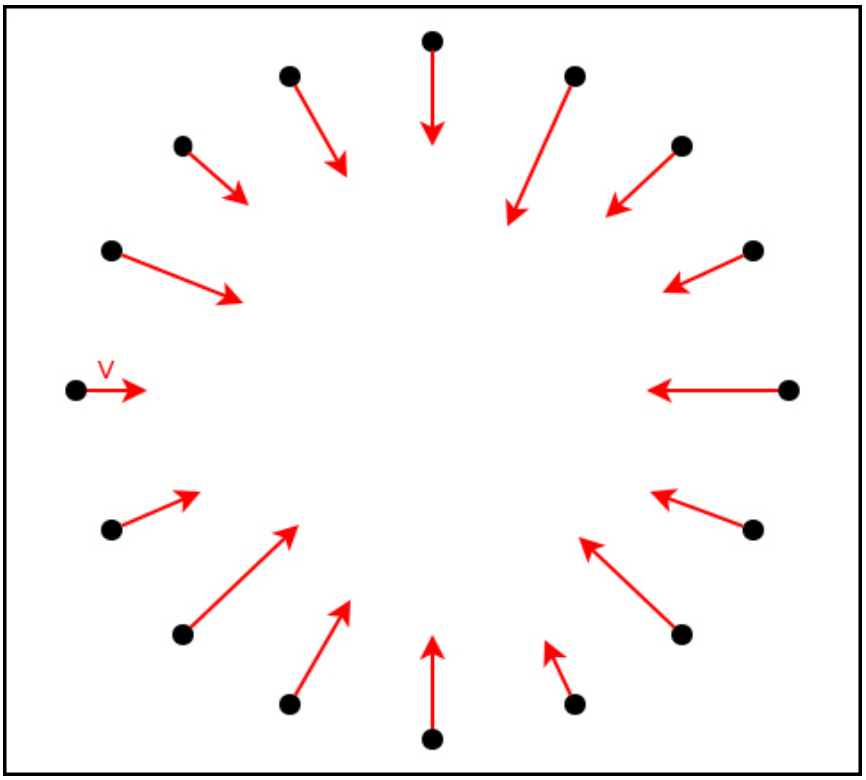


**Abbildung 3** *Kreis Szenario*

Abb. 3 zeigt einen offenen Bereich, in welchem Fußgänger mit kreisförmig angeordnete Startpositionen sich durch das Zentrum auf Ziele, welche den Startpositionen gegenüberliegen, zubewegen.

### 1.4.4 Bühnenszenario

Menschen strömen, wie in Abb. 4 ersichtlich, auf einem offenen Gelände auf eine Bühne zu um möglichst nahe zum Mittelpunkt der Bühne zu gelangen.

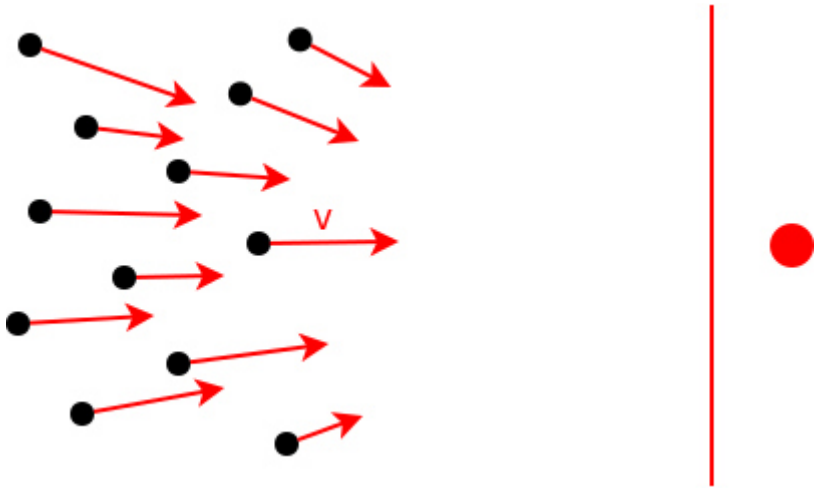


**Abbildung 4** *Bühnen Szenario*

## 1.5 Aufbau der Arbeit

Die Arbeit ist aufgebaut wie folgt. In Kapitel 2 wird der physikalische Hintergrund und das kollisionszeitabhängige Kraftgesetz zur Berechnung der Bewegungspfade der Fußgänger vorgestellt. Kapitel 3 beschäftigt sich mit den Anforderungen hinsichtlich Prallelisierbarkeit, der verwendeten Hardware sowie dem OpenCl Standard. Anschließend wird der verwendete Simulationsalgorithmus in Kapitel 4 motiviert und vorgestellt.
Das Simulationsprogramm selbt und dessen Struktur wird in Kapitel 5 im Detail behandelt. In Kapitel 6 werden die angestellten Überlegungen zur Optimierung des Simulationsprogramms beschrieben. Die Ergebnisse der simulierten Beispielszenarien werden in Kapitel 7 diskutiert, gefolgt von der Zusammenfassung dieser Arbeit in Kapitel 8.

# 2 Energie und Kraftfeld in Abhängigkeit voraussichtlicher Kollisionszeit

## 2.1 Quantifizierung der Interaktion

Das zweidimensionale Simulationsgebiet wird durch einen affinem Raum über einem euklidischen Vektorraum beschrieben, in welchem die Objekte und Simulationsteilnehmer durch entsprechende Raumpunkte und Vektoren repräsentiert sind. Jeder Fußgänger wird durch eine sich bewegende Sphäre mit individueller Ausdehnung dargestellt, die Kollisionen auf ihrem Bewegungspfad zu vermeiden sucht. Die Interaktion mit anderen Sphären sowie Objekten unterliegt einem später ausführlich behandelten Abstoßungsgesetz. Aus Performancegründen sind hier kreisförmige Körpergeometrien gewählt welche die Dauer von Kollisionsberechnungen stark reduzieren.

Zur eindeutigen Beschreibung der Zustände der Fußgänger sind eine Reihe Variablen und Parameter in Verwendung.

- **Identifikationsnummer:** über den Simulationszeitraum gleichbleibende ganze positive Zahl zur Unterscheidung und Adressierung der individuellen Teilnehmer
- **Position:** derzeitige Position im Simulationsgebiet als 2D-Vektor.
- **Momentangeschwindigkeit:** derzeitige Geschwindigkeit als 2D-Vektor.
- **Bevorzugte Geschwindigkeit:** Geschwindigkeitsbetrag bei ungestörter Bewegung ohne Interaktion mit anderen Simulationsteilnehmern. Fußgänger sind bestrebt ihre bevorzugte Geschwindigkeit nach Interaktionen wiederaufzunehmen.
- **Beschleunigung:** Geschwindigkeitsänderung pro Zeit welche im Zuge der der Interaktion zur Kollisionsvermeidung auftritt als 2D-Vektor.
- **Maximale Beschleunigung:** maximal mögliche Beschleunigung um Kollisionen zu vermeiden bzw. die jeweils bevorzugte Geschwindigkeit wiederaufzunehmen.
- **Maximale Interaktionsreichweite:** maximale Distanz bei der im Zuge einer Interaktion Geschwindigkeitsänderungen auftreten werden, bei genügendem Abstand berücksichtigen sich Fußgänger nicht gegenseitig bei der Planung ihres Bewegungspfads.
- **Masse:** Widerstand/ Trägheit der einer Geschwindigkeitsänderung im Zuge einer Interaktion entgegengesetzt wird.

## 2.2 Physikalische Motivation

Bei bekanntem Abstoßungspotential und sich daraus ergebenden Kräften kann nach den Gesetzen der klassischen Mechanik die resultierende Geschwindigkeitsänderung eines Körpers durch numerische Integration der entsprechenden Bewegungsgleichungen angenähert werden. Dieser Ansatz wird im Prinzip, ähnlich der Simulation von Trajektorien in der Molekulardynamik, auf die vereinfachte Darstellung der Fußgänger angewendet. Newtons 2. Gesetz beschreibt die bei bekanntem Kraftfeld einer Interaktion resultierende Beschleunigung:

$$F = m\ddot{\vec{v}}$$

Nach Superposition der paarweisen Kräfte $\overrightarrow{F_{ij}}$ der mit dem Körper $i$ interagierenden $N$ Simulationsteilnehmer ergibt sich die resultierende Gesamtkraft $\vec{F}_i$ auf den Körper als:

$$\vec{F}_i = \sum_{j=0}^{N} \vec{F}_{ij}$$

Durch numerische Lösung der sich nach Newton ergebenden Differentialgleichung für die Geschwindigkeit als zweite Ableitung der Beschleunigung erhält man die entsprechende Geschwindigkeit und Position des Körpers zum neuen Simulationszeitpunkt. Führt man diese Berechnungen mit ausreichender Genauigkeit nacheinander für sämtliche Teilnehmer und Objekte paarweise aus erhält man die Positionen zum nachfolgenden Zeitschritt. Nach Berechnung aller Zeitschritte der Simulation sind die Positionen aller Fußgänger zu jedem Zeitpunkt und somit der Ablauf des gewünschten Szenarios bekannt. Nach entsprechender Analyse und Visualisierung der erhaltenen Daten lassen sich Rückschlüsse auf bedrohliche Situationen, Verkehrsengpässe und Evakuierungszeiten ziehen.

## 2.3 Das kollisionszeitabhängige Kraftgesetz

Bei der beschriebenen Vorgangsweise stellt sich die entscheidende Frage nach einem geeigneten Abstoßungspotential und Kraftgesetz für den die Lösung der Bewegungsgleichung tatsächlich die auftretende Dynamik vorhersagen kann. Es existieren unzählige Kräftemodelle für Interaktion im

Verkehr, das hier verwendete Modell nach Karamouzas, Skinner und Guy **[2]** bietet einen aktuellen und vielversprechenden Ansatz. Unabhängig aller Simulationsdetails ist dieses Kraftgesetz jedoch durch alternative Interaktionsgesetze austauschbar. Die paarweise Kraft $F_{ij}$ zwischen den Fußgängern $i$ und $j$ mit vorrausichtlicher Kollisionszeit $\tau_{ij}$ als negativer Gradient des Interaktionspotentials $E(\tau_{ij})$ ergibt sich als:

$$F_{ij} = -\nabla_{ij}\, E(\tau_{ij})$$

Wobei das Interaktionspotential **[2]** gegeben ist durch:

$$E(\tau_{ij}) = k\, {\tau_{ij}}^{-2}\, e^{{-\tau_{ij}}/{\tau_0}}$$

$k$ entspricht hier einer Konstante die zur richtigen Dimensionierung der Energieeinheit dient während $\tau_0$ die maximal berücksichtigte vorrausichtliche Kollisionszeit einer potentiellen Interaktion bezeichnet.

## 2.4 Kollisionsberechnung

Die voraussichtliche Kollisionszeit $\tau_{ij}$ zweier sich mit Relativgeschwindigkeit $v_{ij}$ bewegenden kreisförmigen Körper mit Radius $R$ im Relativabstand $x_{ij}$ ist bestimmt durch:

$$\tau_{ij} = \frac{-x_{ij}v_{ij} - \sqrt{(x_{ij}v_{ij})^2 - \|v_{ij}\|^2(\|x_{ij}\|^2 - 4R^2)}}{\|v_{ij}\|^2}$$

Sowohl $x_{ij}$, $v_{ij}$, als auch $F_{ij}$ bezeichnen zweidimensionale Vektoren.
Es ergibt sich durch Bildung des Gradienten, also die $k$-te Komponente der paarweisen Kraft zwischen den Körper $i$ und $j$ als:

$$F_{ij} = \mathrm{k}\, e^{{-\tau_{ij}}/{\tau_0}} {\tau_{ij}}^{-2} \left(\frac{2}{{\tau_{ij}}^2} - \frac{1}{\tau_0}\right) \frac{\partial}{\partial (x_{ij})_k}\, \tau_{ij}$$

Nach Ableitung der voraussichtlichen Kollisionszeit erhält man die geschlossene Darstellung der paarweisen Kraft $F_{ij}$ als:

$$F_{ij} = g(\tau_{ij}) \left[ v_{ij} - \frac{\|v_{ij}\|^2 x_{ij} - (x_{ij} v_{ij}) v_{ij}}{\sqrt{(x_{ij} v_{ij})^2 - \|v_{ij}\|^2 \left(\|x_{ij}\|^2 - 4R^2\right)}} \right]$$

mit

$$g(\tau_{ij}) = -\mathrm{k}\, e^{-\tau_{ij}/\tau_0} \left( \tau_{ij} \|v_{ij}\| \right)^{-2} \left( \frac{2}{\tau_{ij}^2} + \frac{1}{\tau_0} \right)$$

Zusätzlich wird eine treibende Kraft $F_t$ **[3]** benötigt, die einen Fußgänger auf das Ziel seines Pfades zusteuern lässt. Diese Kraft wird nach folgender Vereinfachung bestimmt:

$$F_t = \frac{v_{pref} - v}{c}$$

wobei $v_{pref}$ die bevorzugte Geschwindigkeit in Richtung des Ziels und $v$ die Momentan-geschwindigkeit eines Fußgängers ist. Die Rückstellkonstante für die bevorzugte Geschwindigkeit $c$ bestimmt die Zeit zur Wiederaufnahme der bevorzugten Geschwindigkeit.

Zur numerischen Lösung der Bewegungsgleichung an Zeitschritt $i$ um Positionen $x_{i+1}$ und Geschwindigkeiten $v_{i+1}$ am nächsten Zeitschritt $i + 1$ aus der errechneten Kraft $F_i$ zu erhalten, wird das semi-implizite Euler-Verfahren angewandt:

$$v_{i+1} = v_i + F_i \Delta t$$
$$x_{i+1} = x_i + v_{i+1} \Delta t$$

# 3 Verwendete Hardware und OpenCL

## 3.1 Charakterisierung der benötigten Berechnungen

In jedem Zeitschritt der Simulation müssen für jeden einzelnen Körper die Kräfte alle anderen Körper in Wirkungsreichweite berechnet und aufsummiert werden. Je nach für die Berechnung der Kraft maximal berücksichtigter voraussichtlicher Kollisionszeit kommen somit für die Berechnung eines einzelnen Körpers die Wirkung von anderen Körpern in einer kreisförmigen Umgebung in Frage. Der Radius dieser Umgebung entspricht der doppelten Strecke, die ein Fußgänger in einem Zeitschritt der Simulation zurücklegen kann. Körper außerhalb dieses Radius bräuchten auch bei Maximalgeschwindigkeit länger als einen Zeitschritt zur Kollision und werden somit im aktuellen Zeitschritt nicht berücksichtigt.

## 3.2 Aufwand und Komplexität

Die maximale Anzahl $N$ von Berechnungen von paarweisen Kräften für einen Körper übersteigt also die Anzahl der in den berücksichtigten Umgebungskreis theoretisch bei dichtester Packung passenden Körper nie. Bei Idealisierung des Fußgängers als kreisförmiges Objekt, entspricht diese Anzahl also der Lösung der dichtesten Packung von Kreisen mit Fläche $A_K$ in einem größeren Umgebungskreis mit Fläche $A_I$, wobei nur die Mittelpunkte der nicht überlappenden Körper in dem größeren Kreis enthalten sein müssen. Es existieren mehrere Algorithmen zur näherungsweisen Berechnung dieses Problems, jedenfalls übersteigt sie niemals das Verhältnis der Flächen:

$$N = \frac{A_I}{A_K}$$

Bei insgesamt $N_{ges}$ simulierten Fußgängern ergeben sich bei fixiertem Interaktionsradius und maximal $N$ Kraftberechnungen pro Fußgänger $N_{ges}N$ paarweise Interaktionsberechnungen. Da also bei fixiertem Interaktionsradius die Anzahl der maximal pro Fußgänger durchgeführten Berechnungen $N$ konstant ist und nicht von $N_{ges}$ abhängt, ergibt sich eine Zeitkomplexität von $O(N_{ges})$.

## 3.3 Abhängigkeit und Parallelisierbarkeit der Berechnungen

Zur Bestimmung der beschriebenen Kräfte auf einen Fußgänger benötigt man ausschließlich die Positionen und Geschwindigkeiten der sich nähernden Fußgänger in der fest vorgegebenen Umgebung Die Reihenfolge in der die diese Kräfte aufsummiert werden spielt dabei physikalisch keine Rolle.

Die Berechnungen unterschiedlicher Körper in einem Zeitschritt sind ebenfalls an keine bestimmte Abfolge gebunden und bestimmen die erst im nächsten Zeitschritt notwendigen Ausgangspositionen und Geschwindigkeiten aller Fußgänger. Dies ist die Grundlage für die parallele Ausführung der benötigten Operationen, wobei die erhaltenen Ergebnisse nach jedem Zeitschritt synchronisiert werden müssen, um die neuen Positionen und Geschwindigkeiten für alle weiteren parallelen Berechnungen bereitzustellen. Motiviert durch die Unabhängigkeit der Kräfteberechnung wäre, bei Verwendung massiv paralleler Hardware für große Simulationen bei idealisierter Synchronisation und Speichermanagment, die Laufzeit proportional zur Anzahl der Recheneinheiten verringert. Die tatsächliche Beschleunigung des Programms im Vergleich zur seriellen Berechnung hängt vom Algorithmus und dessen Ausnutzung der vorhandenen Speicherhierarchie und parallelen Architektur ab und wird später behandelt.

Zu beachten ist, dass Relativabstände zwischen den einzelnen Positionen nicht erhalten bleiben und Gebiete, sowie die darin enthaltenen Positionen daher nicht unabhängig voneinander gespeichert werden können bzw. periodisch synchronisiert werden müssen. Diese Synchronisation ergibt das Hauptproblem bei der Parallelisierung. Zusätzlich zu ausreichend vorhandener Unabhängigkeit der arithmetischen Berechnungen muss die Anzahl der Speicheroperationen $OP_S$, im Verhältnis zur Anzahl davon abhängiger arithmetischer Rechenoperationen $OP_A$, minimal ausfallen.

Der optimale Wert $R_{opt}$ ergibt sich also bei minimalem Verhältnis:

$$R_{opt} = min(\frac{OP_S}{OP_A})$$

## 3.4 Anforderungen an die Hardware

Massiv parallele Prozessoren wie Grafikprozessoren arbeiten nach dem SIMD/SIMT-Prinzip (Single Instruction Multiple Data/Single Instruction Multiple Threads). Für einen einzelnen Körper bringt die parallele Ausführung der für eine einzelne Kraftberechnung und dazugehörigem Positionsupdate auftretenden Schritte aufgrund der Abhängigkeit und Unterschiedlichkeit dieser Operationen in einem SIMD-Kontext keinen nennenswerten Vorteil. Daher sollen dieselben Berechnungen für unterschiedliche Körper parallel auf den Rechenkernen ausgeführt werden, um zu einem Zeitpunkt dieselbe Operation der jeweiligen Kraftberechnung auf unterschiedliche Fußgängerpositionen anzuwenden. Die Anzahl der parallel arbeitenden Recheneinheiten sollte somit in derselben Größenordnung der Zahl der simulierten Körper liegen.

Jeder einzelne Fußgänger wird, unabhängig von für den Algorithmus benötigten Datenstrukturen durch Identifikationsnummer, Startposition, aktuelle Position, Zielposition, Geschwindigkeit und bevorzugte Geschwindigkeit beschrieben. Die Positionen und Geschwindigkeit entsprechen zweidimensionalen Vektoren. Damit ergeben sich für eben genannte 6 Parameter (32 Bit Ganz- bzw. Gleitkommazahlen, Single Precision) insgesamt 352 Bit. Diese Werte müssen permanent im Arbeitsspeicher vorgehalten und aktualisiert werden. Die bei den tatsächlichen Berechnungen auftretenden Variablen und Parameter pro Zeitschritt sind für die einzelnen Recheneinheiten temporär und lokal vorzuhalten. Bei einem Szenario mit 10.000 simulierten Fußgängern ergeben sich für die Speicherung der Fußgängerdaten eines Zeitpunkts weniger als 440 kB an Speicherplatz. Für die Speicherung der einzelnen Interaktionen in einem Zeitschritt muss schlimmstenfalls die Anzahl aller Körper in der maximalen Interaktionsumgebung aufgeführt werden was bei 10.000 Körpern mit einer überdimensionierten Anzahl von 100 Interaktionen lediglich 4 MB ausmacht.

Nach jedem Zeitschritt müssen die berechneten Positionen aller Fußgänger unabhängig der Reihenfolge der vorherigen Berechnungen synchronisiert vorliegen, um zu bestimmen welche paarweisen Wechselwirkungen tatsächlich zu berechnen sind. Es ist also zumindest nötig eine globale Synchronisationsbarriere nach jedem Zeitschritt der Simulation einzuführen.

## 3.5 Der Grafikprozessor als massiv parallele Lösung

Moderne Grafikkarten werden spätestens nach Erscheinen der CUDA Programmierumgebung von NVIDIA und dem Erscheinen der 8800GT GPU im Jahr 2006, als attraktive Möglichkeit zur Programmierung von Problemen mit Anforderungen für feinen Parallelismus und hohen Datendurchsatz, erfolgreich eingesetzt. Da arithmetische Operationen von den einzelnen Rechenkernen von kommerziell verfügbaren Grafikprozessoren mit SIMD/SIMT Technik ausgeführt werden können, sind diese seit Jahren in den Naturwissenschaften zur Simulation in verschiedensten Disziplinen im Einsatz. Neuere Grafikkarten verfügen über große Arbeitsspeicher von mehreren GB und erlauben Lade- und Schreiboperationen mit hoher Bandbreite (GDDR5). Der Grafikprozessor ist in mehrere Prozessoreinheiten eingeteilt, die ihrerseits in eine gewisse Anzahl von Recheneinheiten, auch als ALUs oder Shader bezeichnet, unterteilt werden. Jede Prozessoreinheit steuert seine Recheneinheiten für gewöhnlich nach dem SIMD Prinzip, sämtliche Recheneinheiten führen also zu einem Zeitpunkt die gleiche Rechenoperation aus. Der Hersteller AMD fasst parallele Threads zu Gruppen zusammen, die gemeinsam ausgeführt werden. Diese Gruppen werden als Wavefronts bezeichnet. Eine Wavefront besteht aus 64 Threads, die auf vorhandenen Recheneinheiten dieselbe Rechenoperation vollziehen. Jedoch müssen nicht alle 64 Threads einer Wavefront zeitgleich vom Prozessor ausgeführt werden. Ebenso können z.B. in zwei Befehlszyklen nacheinander jeweils 32 Threads laufen. Jede Prozessoreinheit verfügt über einen im Verhältnis zum Arbeitsspeicher der Karte um Größenordnungen schnelleren lokalen Speicher und lokalen/privaten Registersatz, auf welchen seine Recheneinheiten operieren können. Die Zugriffszeiten auf den lokalen Speicher der einzelnen Prozessoreinheiten übertreffen die des GPU-Arbeitsspeicher um Größenordnungen.

Der Wechsel zwischen Wavefronts ist bei intelligentem Speichermanagement mit keinem Zusatzaufwand verbunden, somit können auf andere Ereignisse, wie das Beenden einer Ladeoperation wartende Wavefronts schnell durch Wavefronts mit gerade lauffähigen Threads ersetzt werden. Voraussetzung für den Kontextwechsel ohne Verzögerung zwischen Wavefronts ist, dass sämtliche benötigte Daten der entsprechenden auszuführenden Operationen auch im entsprechenden lokalen Speicherbereich der Prozessoreinheit verfügbar sind. Die richtige Dimensionierung, bzw. wie viele Threads einer Prozessoreinheit gleichzeitig zugeordnet sind bzw. wie viele Speicherplatz im lokalen Speicher benutzt wird ist daher einer der wichtigsten Faktoren zur Optimierung der Laufzeit. Die direkte Datenkommunikation zwischen den Prozessoreinheiten ist nicht vorgesehen. Sämtliche Daten mit Synchronisationsbedarf können jedoch über den globalen Arbeitsspeicher der Karte, mit entsprechendem Aufwand, verwaltet werden.

## 3.6 OpenCL zur plattformunabhängigen Implementierung

OpenCL ist ein plattformübergreifender Standard, welcher auf einer Untermenge von C99 basiert. Die C++ spezifischen Erweiterungen der Sprache werden mittels Wrapper in OpenCL C Funktionsaufrufe übersetzt. Heterogene Systeme die aus mehreren CPUs, Grafikprozessoren sowie FPGAs bestehen, können durch die OpenCL Laufzeitumgebung zentral durch die CPU gesteuert und individuell programmiert werden. Die Sprache ist um zusätzliche Funktionen und Datentypen zur parallelen Berechnung auf massiv paralleler Hardware erweitert und ermöglicht die Programmierung von Grafikprozessoren zur Lösung beliebiger Probleme. Grafikprozessoren der letzten Jahre und zugehörige Treiber der Firmen AMD und NVIDIA unterstützen den Standard derzeit jedenfalls bis zur Version 1.2. Der auf dem Grafikprozessor ausgeführte Code kann zur Laufzeit kompiliert werden, um die Ausführung eines OpenCL Programms auf verschiedenen Grafikprozessoren und Plattformen zu ermöglichen, wie in Abb. 5 dargestellt wird. Der verwendete Programmcode wird in sogenannte OpenCL Kernel gegliedert. Der Kernel besteht aus den Instruktionen die auf dem jeweilig zugewiesenen Prozessor auszuführen sind.

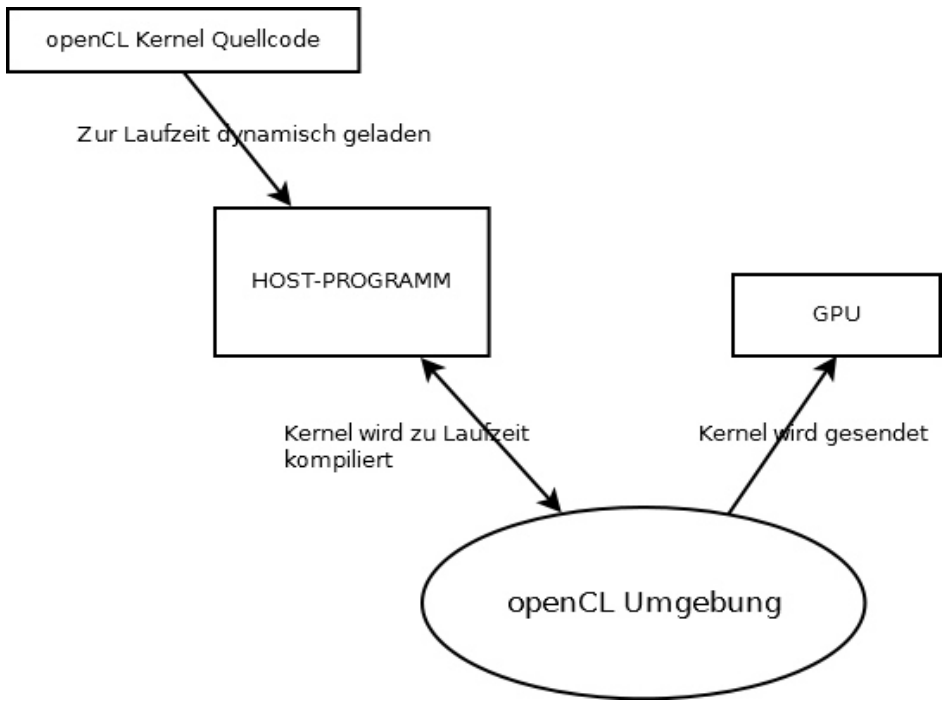


**Abbildung 5** *Kernel Erstellung zur Laufzeit*

Auf jeder einzelnen Recheneinheit (Processing Element) wird eine Instanz dieses Kernels mit dem der Einheit zugewiesenem Datensatz ausgeführt. Diese Instanzen bezeichnet man als Work-Items. Die jeweilige Recheneinheit einer Prozessoreinheit (Compute Unit) und dass ihr zu einem Zeitpunkt zugeordnete Work-Item werden in dieser Arbeit teilweise begrifflich gleichgesetzt. Eine festgelegte Anzahl von Work-Items bildet nach OpenCL eine Work-Group fester Größe. Jede Recheneinheit kann jedoch zu einem Zeitpunkt genau ein Work-Item, wie in Abb. 6 dargestellt. Innerhalb einer Work-Group ist die Synchronisation von Daten im Speicher gestattet, Work-Group übergreifende Synchronisation eines Kernels ist nicht vorgesehen. Die Festlegung der optimalen Anzahl von Work-Items und Work-Groups hängt von der tatsächlich verwendeten Hardware ab.

Durch Unterteilung des Grafikprozessors in einzelne Prozessoreinheiten hängt die maximale Anzahl von Work-Items und Work-Groups pro Prozessoreinheit von verfügbaren lokalen Speicherbereichen und Recheneinheiten ab. Der OpenCL Standard unterscheidet zwischen konstantem, globalem und lokalen Speicher.

Grafikprozessoren von AMD selbst unterteilen die Work-Items in Gruppen von Wavefronts. Diese Wavefronts bestehen aus einer Anzahl von 64 Threads, die gemeinsam auf den Rechenkernen des Grafikprozessors ausgeführt werden. Diese Unterteilung in Wavefronts wird im OpenCL Standard nicht explizit vom Programmierer berücksichtigt, für Optimierungsüberlegungen ist die Anzahl der Threads in einer Wavefront jedoch unter Umständen zu berücksichtigen

Bei Deklaration von Variablen in einer OpenCL Kernelfunktion handelt es sich um privaten Speicher der jedem Work-Item exklusiv zugeordnet ist und dessen Inhalt von anderen Work-Items aus nicht sichtbar ist. Lokale Speicher einer einzelnen OpenCL Work-Group ist exklusiv für alle Work-Items dieser Work-Group verfügbar und entspricht dem schnellen lokalen Speicher auf den individuellen Prozessoreinheiten.

Die Zugriffszeiten auf den lokalen Speicherbereich sind mit denen des auf der Prozessoreinheit verfügbaren Cache vergleichbar. Zugriffe auf den globalen Speicherbereich sind durch alle Work-Items eines Kernels möglich, unabhängig davon welcher Work-Group diese zugeordnet sind. Der globale Speicher entspricht dem Arbeitsspeicher der Grafikkarte.

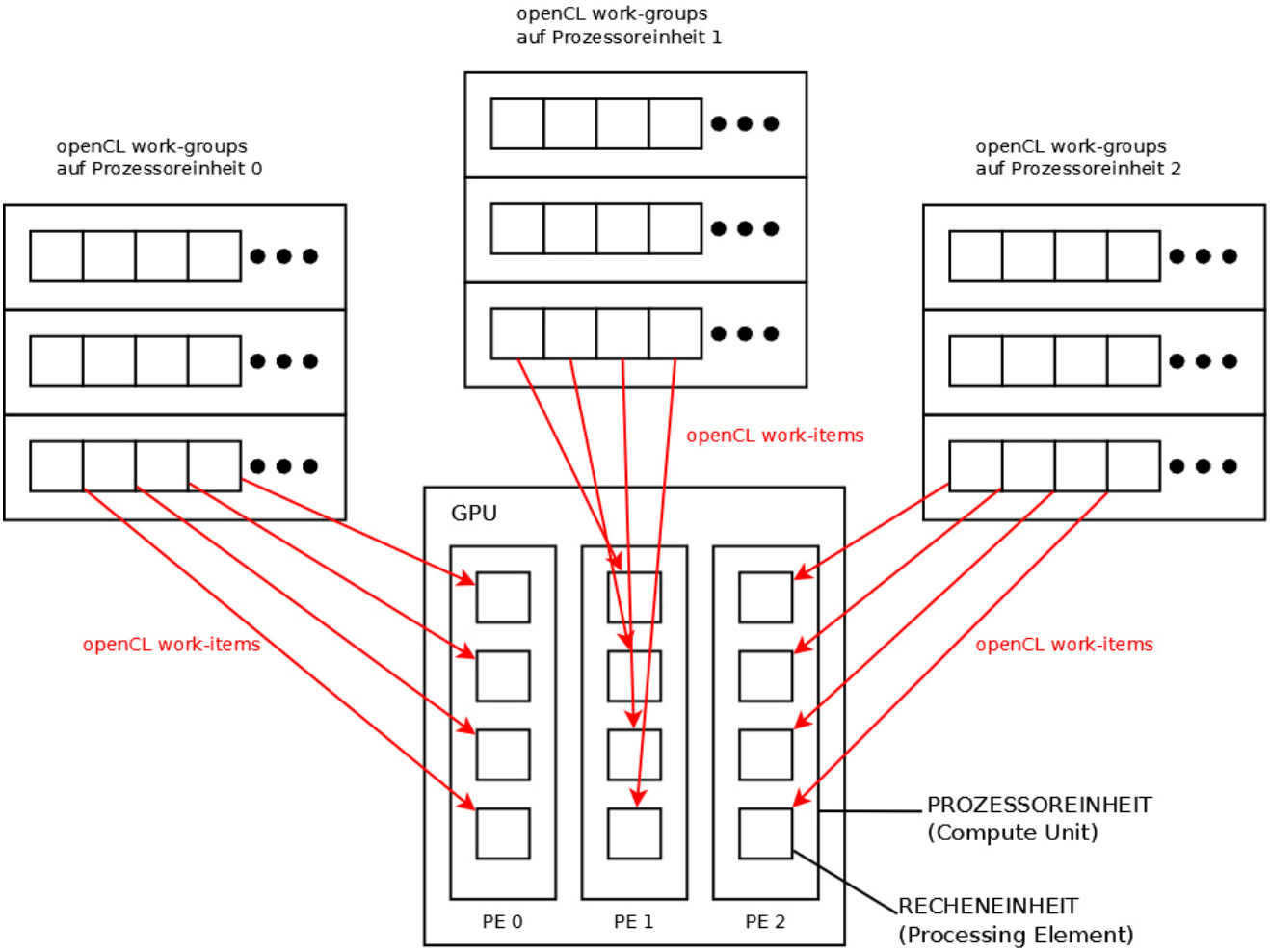


**Abbildung 6** *Zuordnung der work-items*

Die Ausführung der Kernel auf der Grafikkarte und die Verwaltung der dazugehörigen Speicherbereiche werden durch ein OpenCL Host-Programm gesteuert welches auf der CPU des heterogenen Computer ausgeführt wird.

Folgende Schritte sind zur Steuerung der Grafikkarte nötig:

- Auswahl der Zielplattform und des zu verwendeten Grafikprozessors
- Erstellung eines Ausführungskontextes für jeweilige Plattform und Prozessor
- Erstellung einer Warteschlange in die zur Ausführung bereite Kernel gereiht werden
- Deklaration von Speicherobjekten
- Reservierung der entsprechenden Speicherbereiche auf der Grafikkarte
- Befüllung der Speicherbereiche mit benötigten Daten
- Kompilieren der OpenCL Kernel
- Erstellung der Kernel

- Kernel-Funktionsargumente setzen und Kernel in Warteschleife aufnehmen
- Auslesen der Daten aus dem Speicher der Grafikkarte
- Verwendete Ressourcen freigeben

## 3.7 Verwendete Hardware:

**Grafikprozessor:**

Die für die Simulation verwendete Grafikkarte AMD Radeon R7 260X verfügt über 2048 MB GDDR5 Arbeitsspeicher, die Kommunikation von Daten wird über den PCI Express 2.0 Bus mit einer maximalen Busrate von bis zu 8000 MB/s ermöglicht. Der Prozessor basiert auf der von AMD 2013 entwickelten Bonaire Architektur, AMDs GCN Mikroarchitektur in der Version 1.1. GCN ist eine RISC SIMD/SIMT Architektur und ermöglicht die Programmierung des Grafikprozessors für generelle Probleme (GPGPU). Vom Hersteller wird eine theoretische Single Precision Gleitkomma-Performance von bis zu 2 TFLOPs angegeben. Der Chip taktet mit 1 GHz und ist mit einer Bandbreite von 96 GB/s bei einer Busbreite von 128 Bit an den Arbeitsspeicher der Karte angebunden. Berechnungen können durch 896 Shadern (ALUs) die in 14 Prozessoreinheiten unterteilt sind durchgeführt werden. Jede Prozessoreinheit umfasst somit 64 Recheneinheiten (ALUs).

Jede der 14 Prozessoreinheiten des Bonaire Prozessors verfügt über einen eigenen lokalen Speicherbereich mit geringer Latenz. Auf einer Prozessoreinheit können Work-Items einer Wavefront gemeinsam auf bis zu 32 kB lokalen Speicher pro SIMD zugreifen. Weiters ist ein 64 kB Speicherbereich für die gemeinsame Nutzung aller Work-Items der Prozessoreinheit vorhanden. Daten die einmalig in den lokalen Speicher geladen werden sollten möglichst oft wiederverwendet werden, somit bleibt der mehrmalige und um Größenordnungen langsamere Zugriff auf den Off-Chip Arbeitsspeicher erspart. Jede Prozessoreinheit verfügt über 4 SIMD Vektoreinheiten mit jeweils 16 Elementen (64 pro Prozessoreinheit). Die Vektoreinheiten besitzen jeweils 64 kB Speicherbereich für ihre allgemeinen Register. Alle Prozessoreinheiten haben zusätzlich Zugriff auf einen globalen 64 kB Speicherbereich der unabhängig von Wavefront verwendet werden kann. Der maximal verfügbare (L2) Cache beträgt 512 kB. OpenCL wird von der Bonaire Architektur bis zum Standard 2.1 unterstützt **[4]**. Abb. 7 zeigt den schematischen Aufbau einer Prozessoreinheit.

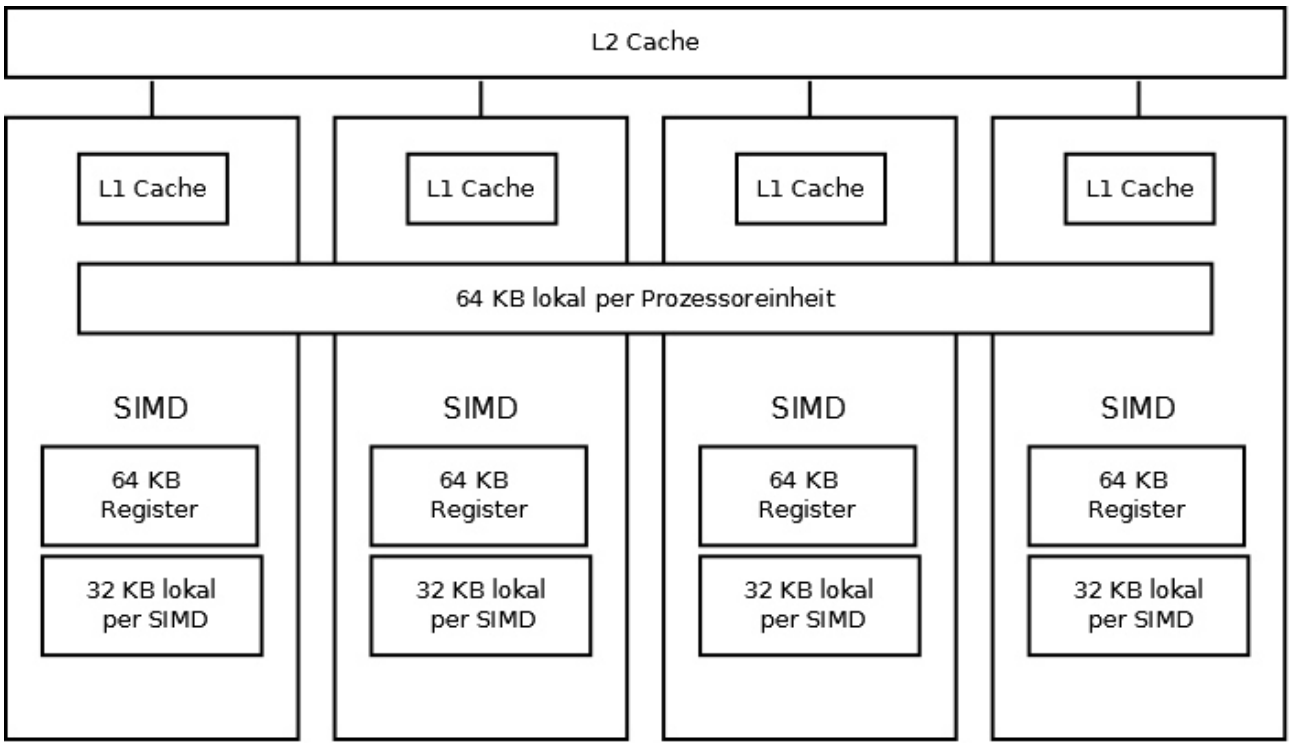


**Abbildung 7** *Prozessoreinheit der R7 260X*

**Host Prozessor:**

Der OpenCL Host-Code wird auf dem Intel Core i7-870 mit 4 Prozessorkernen, einem Cache von 8 MB und einer Taktfrequenz von 3-3.6 GHz ausgeführt. Der Prozessor ist an 12 GB DDR3 Arbeitsspeicher mit einer maximalen Speicherbandbreite von 21 GB/s angebunden. Die Kommunikation mit der Grafikkarte findet über den PCI Express 2.0 Bus mit maximal 8 GB/s statt.

## 3.8 Erwarteter Aufwand

Das erforderliche periodische Kopieren von Positionsdaten aus dem Arbeitsspeicher der Grafikkarte in den Hauptspeicher der Host-CPU ist zur Speicherung der Trajektorien der Fußgänger auf der Festplatte unentbehrlich. Die aus dem Grafikspeicher kopierten Positionsdaten können durch die CPU, unabhängig von den am Grafikprozessor stattfindenden weiteren Berechnungen, ausgewertet werden. Bei genügend feiner Zeitauflösung der Simulation ist dies jedoch nur bei einem Vielfachen eines einzelnen Zeitschritts nötig und kann konkurrent durchgeführt werden. Sämtliche Schritte des Algorithmus sollen auf der Grafikkarte stattfinden da die Anzahl von Recheneinheiten bei einer einzelnen GPU in der Größenordnung der maximalen Anzahl von simulierten Fußgängern (~1000) liegt. Bei größeren Menschenmengen wird der Vorteil der zusätzlichen Recheneinheiten mehrerer

Grafikkarten durch die Notwendigkeit der Synchronisation der Daten zwischen den Karten beeinflusst. Ein Vorteil gegenüber der Dauer $T$ eines Simulationsschritts auf einer einzelnen GPU würde theoretisch erst entstehen falls die Anzahl der simulierten Fußgänger $N_{ges}$ die Anzahl $N_{CU}$ der Recheneinheiten pro Grafikprozessoren derart übersteigt, dass mit Synchronisationszeit $T_{sync}$ der benötigten Körperpositionen zwischen den Karten pro Simulationsschritt folgende Beziehung gilt:

$$\frac{T}{N_{ges}/N_{CU}} + T_{sync} < T$$

Da die Synchronisationsdauer eines Simulationszeitschritts $T_{sync}$ unter Umständen länger benötigen würde als die tatsächliche Berechnung eines Zeitschritts, wäre dies ein den Simulationsbedarf dieser Arbeit übersteigender Aufwand.

# 4 Simulationsalgorithmus

## 4.1 Parallelen der Fußgängersimulation zur Molekulardynamik

Die Computersimulation von Molekülen ist seit der Entwicklung von modernen Computerprozessoren aktives Forschungsgebiet in der Physik und hat unzählige Methoden und Algorithmen hervorgebracht. Abhängig von den jeweiligen Potential- und Bewegungsgleichungen, der Reichweite der auftretenden Kräfte und der Teilchenanzahl und Zeitauflösung sind verschiedenste Simulationsalgorithmen auf geeigneter Hardware anwendbar. Es stellt sich die Frage, wie die einzelnen auftretenden Berechnungen auf die verfügbaren Recheneinheiten (in Bezug auf Speicherbedarf und Speicherlokalität) optimal aufgeteilt werden können

## 4.2 Reichweite des Interaktionspotential

Ob eine Reaktion und Pfadänderung zweier sich aufeinander zubewegender Fußgänger stattfindet hängt von der voraussichtlichen Zeit bis zur Kollision (bei Beibehaltung der momentanen Relativgeschwindigkeit) ab. Nach den Ergebnissen von Karamouzas, Skinner und Guy **[2]** gibt es eine gewisse Maximalzeit ab der keine Beschleunigung der Körper auftritt. Die Berechnung der wechselseitigen Interaktion zweier Passanten ist also, abhängig von deren Geschwindigkeit, erst ab einem gewissen Relativabstand durchzuführen. Die maximale Reaktionszeit von 3 s, bei einer maximalen Gehgeschwindigkeit von 2 m/s, entspräche beispielsweise einem Relativabstand von 6 m zum Kollisionspunkt. Große Simulationsgebiete übersteigen den Wirkungsbereich der Kraft also um ein Vielfaches, es handelt sich um verhältnismäßig kurzreichweitige Kräfte die bei der Simulation des Interaktionspotentials auftreten. Dadurch reduziert sich auch die Anzahl der paarweisen Interaktionen **[5],** die für einen einzelnen Körper pro Zeitschritt auszuwerten sind. Die derzeit schnellste und einfachste Methode für möglichst exakte Berechnung langreichweitiger, mit dem Vergleich aller Körper untereinander ist somit keine attraktive Möglichkeit zur Berechnung der Interaktionen, da sämtliche Teilchenpositionen für optimalen Speicherzugriffen nacheinander in Blöcken **[6]** geladen und unnötig verglichen werden. Auch Baumstruktur Algorithmen wie der Algorithmus von Barnes-Hut **[7]**, die gute Näherungen für die Wechselwirkung mit weit entfernte Körpergruppen liefern, bringen keinen Vorteil.

## 4.3 Vergleich gängiger Methoden

Für zum Simulationsgebiet verhältnismäßig kurzreichweitiger Kräfte, wie sie nach im verwendeten Model auftreten, sind folgende Strategien geläufig:

**Aufteilung nach Simulationskörpern:**
Jede Recheneinheit erhält eine Gruppe von Körpern, die ihr über den Zeitraum der Simulation fest zugeordnet sind. Da während des Simulationsvorgangs jeder Fußgänger mit jedem anderen interagieren könnte, muss eine Liste **[8],** in welcher die paarweisen Interaktionen der entsprechenden Teilchen pro Zeitschritt gespeichert sind, angelegt werden.
Die Paarliste muss permanent durch Vergleich und Berechnung der aktuellen Teilchenabstände und voraussichtlichen Kollisionszeiten aktuell gehalten werden. Weiters müssen die pro Recheneinheit nötigen Eingangsdaten der mit der Gruppe potentiell wechselwirkenden Körper periodisch lokal geladen werden. Bei auf die Recheneinheiten verteiltem privaten Speicher, wie es bei den schnellen lokalen Speicherbereichen der einzelnen Einheiten der Grafikkarte der Fall ist, bedeutet dies einen hohen Synchronisationsaufwand.

Eine zentrale Verwaltung aller Positions- und Geschwindigkeitsdaten im Hauptspeicher der Grafikkarte, mit ständigem direktem Zugriff aller Prozessoreinheiten, bedeutet verhältnismäßig hohe Latenzen zum Laden und Speichern der jeweils benötigten Körperdaten pro Zeitschritt. Das größte Problem liegt in der ungleichen Auslastung der Recheneinheiten, welche bei massiv parallelen Prozessoren zu starkem Performanceverlusten führen kann. Während es Gruppen von stark interagierenden Körpern gibt kann gleichzeitig für weiter entfernte Körper keine Interaktion zu berechnen sein. Die jeweils verantwortlichen Schaltungen wären unter Umständen beliebig stark unterschiedlich ausgelastet. Da im Vorhinein nicht ohne weiteres vorhersagbar ist, wie sich diese Ungleichheit entwickelt, entfällt im schlimmsten Fall der Performancevorteil gegenüber einer seriellen Implementierung.

**Aufteilung nach Kraftberechnungen:**
Die in einem Zeitschritt auftretenden einzelnen Kraftberechnungen werden gleichmäßig über die Recheneinheiten verteilt. Dies führt zur optimalen Auslastung der ALUs. Da jedoch unterschiedlich viele Summationen von Kräften für unterschiedliche Körper auftreten, die beliebig über die Recheneinheiten verteilt sein können, muss jede einzelne Kraft gespeichert und schließlich in jedem Zeitschritt von unterschiedlichen Speicherstelle und Recheneinheit aufsummiert werden. Dieser Vorgang erhöht die benötigte Speichermenge und deren Verwaltungsaufwand und führt zwangsläufig

zu willkürlich verteilten Speicherzugriffen. Dies ist ein Performanceproblem **[9]** da Zugriffe auf benachbarte Speicherstellen von kollektiven Ladeoperationen und dem vorhandenen Cache des der Grafikprozessoren profitieren.

**Räumliche Aufteilung der Berechnungen:**
Bei diesem Ansatz wird das Simulationsgebiet in Bereiche/Sektoren unterteilt, wobei jede Prozessoreinheit für die Berechnung der Wechselwirkungen in diesem Bereich verantwortlich ist. Anhand der Position der Fußgänger werden sie periodisch dem entsprechenden Sektor und der entsprechenden Prozessoreinheit zugeordnet. Da Körper auch über Bereichsgrenzen wechselwirken, müssen die Positionsdaten benachbarter Bereiche in maximal berücksichtigter Kraftreichweite regelmäßig abgeglichen werden. Um eine gleichmäßige Auslastung des Prozessors zu gewährleisten sollten sich in jedem Sektor gleichviele Körper befinden. Dies ist nur dann möglich, wenn die Dimensionen jedes Bereichs dynamisch während der Laufzeit an die jeweilige Teilchendichte angepasst werden, z.B durch dynamisches räumliches Bipartitionieren**.** Die Synchronisation über Bereichsgrenzen sowie das Verfahren zu dynamische Berechnung der Raumteilung sind für den Overhead dieser Methode maßgeblich.

Jede Teilchenposition und Geschwindigkeit muss, zumindest zur regelmäßigen Kontrolle der Daten, durch die Host-CPU im Arbeitsspeicher abgelegt werden. Ist jede Teilchenposition an einer über den Simulationszeitraum festgelegten Adresse gespeichert, ergibt sich folgendes Problem: Obwohl die Daten zweier Körper im Speicher an benachbarten Adressen liegen, befinden sich die Körper räumlich gesehen eventuell an komplett unterschiedlichen Stellen des Simulationsgebiet und wechselwirken nicht miteinander. Dadurch sind die Daten für die Berechnung der Kraft möglicherweise an komplett unterschiedlichen Speicherstellen. Ein derartig zufälliger Speicherzugriff ist für die breiten Ladeoperationen von Grafikprozessoren ungünstig.

Um räumlich benachbarte Körper auch benachbart im Speicher abzulegen, wäre eine periodische Umordnung des Speichers nach Position notwendig. Diese Umordnung jeden Zeitschritt zu vollziehen scheint jedoch wenig attraktiv da sie genauso viele Speicheroperationen die uncoalesced geschehen beinhaltet. Findet sie nur alle paar Zeitschritte statt, kann sie je nach Simulationscharakter einen Geschwindigkeitsvorteil bringen. Das im folgenden vorgestellte Verfahren, angelehnt an den Algorithmus für kurzreichweitige Kräfte von Wang **[11],** versucht alle verwendeten Datenstrukturen möglichst coalesced zu speichern und zu laden.

## 4.4 Der bin-/pairlist hybrid Algorithmus für kurzreichweitiger Kräfte

Zur Bestimmung der in einem Zeitschritt auftretenden Interaktionen wird eine Paarliste verwendet, in welcher für jeden Körper eine Liste von Partnern gespeichert wird von denen eine beeinflussende Kraft ausgeht. Dazu werden einfach jene Körper in die Liste Paarliste eingetragen, deren räumlicher Abstand zum Zielkörper unter der maximalen gesetzten Interaktionsreichweite liegt. Um diese Paarliste zu erzeugen, wäre ohne weitere Beschränkung der paarweise Positionsvergleich aller Fußgänger nötig. Durch die Symmetrie der Abstandsberechnung entfällt zwar theoretisch die Hälfte der Vergleiche, es bleibt aber eine Vergleichsanzahl $O({N_{ges}}^2)$ **[12]**. Um die Erzeugung der Paarliste zu beschleunigen, wird das Simulationsgebiet zuerst in ein gleichmäßiges Gitter unterteilt. Die resultierenden rechteckigen Zellen sind derart dimensioniert, dass die maximale kreisförmige Interaktionsumgebung eines Fußgängers innerhalb der 8 umliegenden Zellen liegt **[13]**.

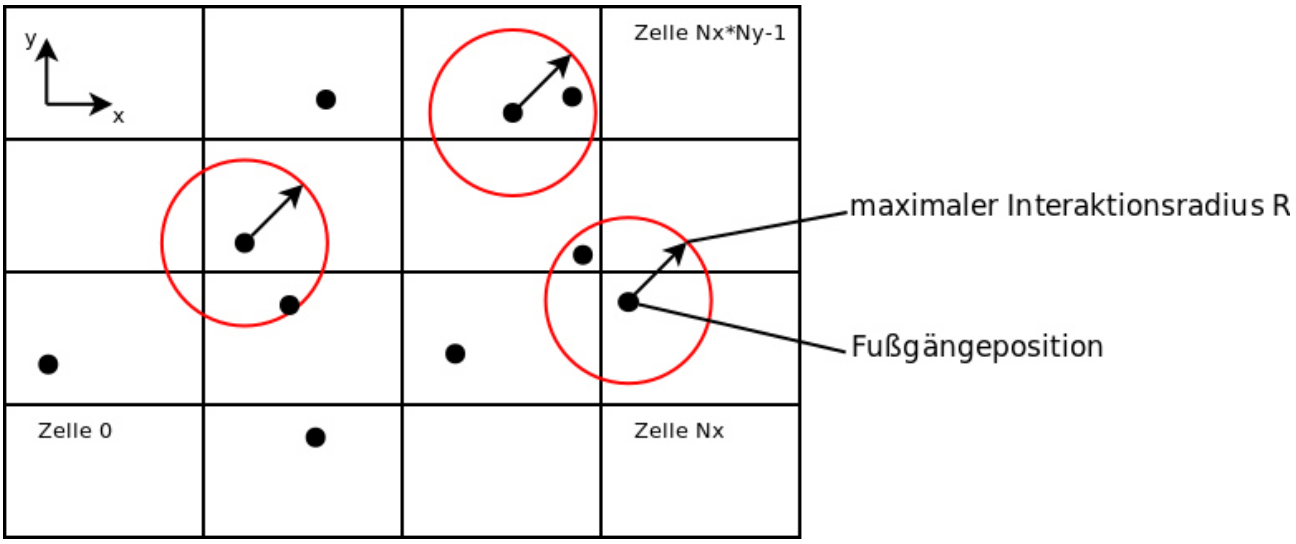


**Abbildung 8** *Zellen und Interaktionsradius*

Für jeden Fußgänger wird aus der derzeitigen Position die zugehörige Zelle. Zur Berechnung der Paarliste des Fußgängers müssen, wie in Abb. 8 ersichtlich, nun nur noch die umliegenden 8 Zellen untersucht werden. Jeder Körper kann nun mit seiner Paarliste an eine Recheneinheit übergeben werden, um die Kräfte und Positionsveränderungen zu berechnen. Dazu müssen die Positionen und Geschwindigkeiten der Körper in der Paarliste aus dem Hauptspeicher lokal geladen werden. Eine gleichmäßige Aufteilung der einzelnen Kraftberechnungen (unabhängig vom zugehörigen Körper) auf die Recheneinheiten erzielen theoretisch eine bessere Auslastung des Prozessors, die dadurch nötigen Speicheroperationen relativieren diesen Vorteil jedoch.

Für die Durchführung eines Simulationszeitschritts müssen 5 Phasen, dargestellt in Abb. 9, durchlaufen werden, die durch entsprechende OpenCL Kernel implementiert sind. Sollte in einem

Zeitschritt tatsächlich keine Kraftberechnung für einen Fußgänger notwendig sein, bleibt die zugeordnete Recheneinheit während dieser Phase unausgelastet.


**Abbildung 9** *Simulationszeitschritt mit einzelnen Phasen*

### 4.4.1 Binning

Jeder Körper wird einer Recheneinheit zugeordnet. Das Simulationsgebiet wird durch ein Gitter gleichmäßig in N durchnummerierte Zellen eingeteilt. Zu beachten ist, dass die Seitenlängen der Zellen den maximal berücksichtigten Interaktionsradius eines Körpers übersteigen. Nach Laden der entsprechenden Körperposition wird für jeden Fußgänger die Identifikationsnummer der zugehörigen Zelle berechnet.

### 4.4.2 Sortierung

Die Identifikationsnummern (IDs) der Körper werden mit den Identifikationsnummern der jeweils zugehörigen Zellen als Schlüsselwerten (in aufsteigender Reihenfolge) sortiert, sodass die IDs der Körper derselben und angrenzender Zellen auch benachbart im Speicher abgelegt sind.

Die Sortierung selbst findet ebenfalls parallel auf der Grafikkarte statt und verwendet eine optimierte Implementierung des parallelen RadixSort Algorithmus der freien OpenCL Bibliothek libcl **[14]**. Gleichzeitig werden auch die IDs der zugehörigen Zellen sortiert um in der nächsten Phase eine Indizierung der Start- sowie Endadresse der Körper jeder Zelle zu erzeugen.

### 4.4.3 Indizierung

Jede Speicherposition im Array der aufsteigend sortierten Körper-IDs wird einer Recheneinheit zugeordnet. Danach wird von derselben Recheneinheit die ID der Zelle, welche sich an derselben relativen Position des in vorherigen Phase mitsortierten Array der Zellen-IDs befindet, geladen. Da die Zellen-IDs auf die gleiche Weise als Schlüsselwerte mitsortiert wurden, entspricht die geladene ID der Zelle in welcher sich der entsprechende Fußgänger der Recheneinheit räumlich befindet. Nun wird die geladene Zellen-ID mit der Zellen-ID der vorherigen Speicherstelle verglichen. Sind die zwei Werte unterschiedlich, handelt es sich um eine Zellengrenze und die Speicherposition der Recheneinheit wird in einem dritten Index-Speicherarray abgelegt. Dieses Speicherarray enthält pro Zelle zwei Einträge für Start und Endposition im Speicherarray der Körper. Die Endposition der vorherigen Zelle wird ebenfalls von der Recheneinheit in das Speicherarray der vorhergehenden Zelle eingetragen. Somit ist aus dem neuen Index-Speicherarray durch Laden der Start und Endposition einer Zelle die schnelle gemeinsame Adressierung der zugehörigen Körper aus dem Körper-ID Speicherarray möglich. Weiters können die Körper-IDs einer bestimmten Zelle in einer Ladeoperation für die entsprechende Prozessoreinheit coalesced geladen werden.

### 4.4.4 Berechnung der Paarliste

Für jeden Fußgänger müssen alle ihn umgebenden Körper (in maximal gesetzter Interaktionsreichweite) in die ihm zugeordnete Paarliste eingetragen werden. Dazu ist die Untersuchung der eigenen Zelle aber auch der umliegenden acht Zellen nötig Zellen in größerem Abstand befinden sich nach Voraussetzung außerhalb der maximalen Interaktionsreichweite und werden nicht berücksichtigt. Für jede einzelne Zelle werden die

entsprechenden Körper-IDs, durch Referenzierung aus der in der Indizierungsphase erstellten Index-ID, in den lokalen Speicher einer Prozessoreinheit geladen. Für jeden dieser Körper ist nun eine Recheneinheit zuständig. Die Recheneinheiten laden nun einmalig die Positionen ihres jeweiligen Fußgängers und behalten diese während der gesamten Phase gespeichert. Nach dem synchronisierten Abschluss des Ladens der Positionen vergleicht jede Recheneinheit paarweise seine zugeordnete Körperposition mit den Positionen der anderen Recheneinheiten und entscheidet ob der entsprechende Körper sich innerhalb des Interaktionsradius befindet. Ist dies der Fall, wird die der Position entsprechende KörperID zur Paarliste hinzugefügt wird. Nach Abschluss dieses Vorgangs werden die Körper-IDs der angrenzenden Zellen nacheinander auf gleiche Weise geladen. Jede Recheneinheit vergleicht nun seine Körperposition aus der eigenen Zelle mit den Positionen aus der neu geladenen Zelle. Nach eventuellem Hinzufügen der entsprechenden Körper-IDs zur eigenen Paarliste werden diese Positionen wieder verworfen und der Vorgang wird analog für die restlichen 7 angrenzenden Zellen wiederholt. Nach Vergleich mit der letzten angrenzenden Zelle ist die Paarliste fertig erstellt und die nötigen Interaktionsberechnungen für die nächste Phase dadurch bestimmt.

### 4.4.5 Kraftberechnung

Wie bereits in den Phasen davor wird jeder Körper einer Recheneinheit zugeordnet. Diese hat nun die Aufgabe für jeden Eintrag der Paarliste dieses Körpers nacheinander die entsprechenden Positions- und Geschwindigkeitsdaten zu laden und die voraussichtliche Kollisionszeit der beiden Körper zu berechnen **[10]**. Findet vorrausichtlich eine Kollision statt, muss anschließend die entsprechende Kraft berechnet werden. Da auf den betrachteten Fußgänger sowie auf seinen beeinflussenden Partner betragsmäßig dieselbe Kraft wirkt, welche sich nur um ein Vorzeichen unterscheidet wird diese Kraft von derjenigen Recheneinheit, der dieser Interaktionspartner zugeordnet ist, ebenfalls berechnet. Der Synchronisierung- und Speicherverwaltungsaufwand der nötig wäre um diese Berechnung simulationsweit nur einmal durchzuführen und zu speichern ist jedoch zu hoch, um sich ohne doppelte Berechnung der Kraft Laufzeit zu sparen. Nach Abarbeitung aller Interaktionspartner, also allen Einträgen der Paarliste eines Körpers, werden auf analoge Weise die abstoßenden Kräfte die von statischen Objekten ausgehen berechnet.

Die treibende Kraft die den Fußgänger auf sein Ziel zusteuern lässt, wird schließlich zur Gesamtkraft addiert. Zur Berechnung dieser "antreibenden Kraft" ist die gewünschten Zielposition des Körpers, für welchen die Recheneinheit verantwortlich ist, erforderlich. Hat der Fußgänger sein Ziel bereits erreicht, wird natürlich keine Kraftberechnung durchgeführt.

### 4.4.6 Berechnung der Positionen und Geschwindigkeiten

Geschwindigkeit, Position und Gesamtkraft für jeden Passanten sind nun bekannt und das Positionsupdate kann tatsächlich durchgeführt werden. Dazu werden die Berechnungen der numerischen Integration für jeden Körper parallel von der zugeordneten Recheneinheit durchgeführt und die sich ergebenden neuen Positionen und Geschwindigkeiten in den Arbeitsspeicher der Grafikkarte abgelegt.

Die Reihenfolge und Verwaltung der Speicherpositionen der Fußgängerdaten, das Zugriffschema und die Berechnung der Zellpositionen sowie die Indizierung und Kraftberechnung wurden in dieser Arbeit eigens entwickelt. Der hier vorgestellte Algorithmus ist stark an den von Wang entwickelten Algorithmus **[11]** zur Berechnung von Molekulardynamik in CUDA angelehnt.

Ein Durchlauf dieser Phasen mit abschließender numerischen Integration der Bewegungsgleichungen ergibt den Eingangszustand der Körper im Phasenraum für den nächsten Zeitschritt **[15]**. Der vorgestellte Algorithmus wird für jeden simulierten Zeitschritt angewendet, um durch die gesamte Simulation zu iterieren. Hat jeder Fußgänger seine Endposition erreicht ist die Simulation abgeschlossen. Eine detaillierte Beschreibung des Verfahrens folgt in der Beschreibung der einzelnen OpenCL Kernel.

# 5 Das Simulationsprogramm

## 5.1 Konstanten und Parameter

Folgende globale Simulationsparameter sind für den Programmablauf festgelegt und bestimmen, zusammen mit Anfangsbedingungen für die Positionen der einzelnen Fußgänger, den Ablauf des simulierten Szenarios. Sämtliche Parameter sind in Host- und GPU-Code verfügbar:

- **dt** - der bei der numerischen Lösung der Bewegungsgleichungen für jeden Fußgänger auftretende Zeitschritt. Der Durchlauf aller 5 Simulationsphasen ergibt die Positions- und Geschwindigkeitsänderung während dieser Zeit.
- **timeSteps** - Anzahl der maximal durch die Simulation durchgeführten Zeitschritte, sollten nicht alle Fußgänger ihr angesteuertes Ziel erreicht haben. Der maximal simulierte Zeitraum ergibt sich als dt*timeSteps.
- **Cutoff** - maximal berücksichtigter Interaktionsradius. Kraftberechnungen finden nur in diesem Umkreis statt, alle Körper die sich außerhalb dieses Radius um einen Fußgänger befinden werden nicht berücksichtigt
- **objectN** - Anzahl der Objekte im Simulationsgebiet
- **bodyN** - Anzahl der simulierten Fußgänger
- **goalRadius** - Dieser Wert bestimmt den Mindestabstand eines Fußgängers zu seinem Ziel, sodass er dieses auch tatsächlich erreicht hat.
- **radius** - Bei der Idealisierung der Körpergeometrie des Fußgängers als Kreis entspricht dieser Wert dem Radius eines Körpers. Befinden sich zwei Fußgänger in einem Abstand, der dem doppelten Radius entspricht, kollidieren sie.
- **maxAccel** - Die maximale Beschleunigung zu der ein Fußgänger im Zuge einer Interaktion in der Lage ist. Wird eine höhere Beschleunigung berechnet, so wird der Wert auf maxAccel gesetzt.

## 5.2 Datenstrukturen und Verwendung

Im Arbeitsspeicher der Host-CPU werden verschiedene Speicherbereiche reserviert, die für Simulationsdaten und Steuerung der einzelnen Phasen des Programms benötigt werden. Alle Daten werden im Arbeitsspeicher als eindimensionale Arrays von Ganz- oder Gleitkommazahlen in

einfacher Präzision mit 32 Bit gespeichert, um Kopiervorgänge coalesced durchzuführen. Die im Verlauf des Programms an den Grafikspeicher gesendeten Werte werden dort, von dem am Grafikprozessor ausgeführten Code, mit neuem Datentyp (z.B.: Vektoren) interpretiert. Die eindimensionalen Speicherarrays werden bei Programmbeginn dynamisch alloziert und werden nach entsprechender Initialisierung in den Grafikkartenspeicher gelesen. Während der gesamten Simulation werden lediglich die Positions- und Geschwindigkeitsdaten jedes Fußgängers periodisch in den Arbeitsspeicher der Host-CPU zurückgelesen, alle anderen Daten verbleiben im GPU-Speicher. Sämtliche Phasen der Simulation eines jeden Fußgängers laufen am Grafikprozessor ab und ohne weitere Auswertung und Visualisierung wäre, bei reiner Berechnung der Trajektorien, theoretisch kein regelmäßiger Datenaustausch zwischen Host-CPU und Grafikprozessor nötig.

Folgende Speicherarrays werden zu Beginn der Simulation alloziert:

**bodyID:**

Zu Beginn der Simulation werden alle $N$ simulierten Fußgänger in aufsteigender Reihenfolge durchnummeriert. Das Array bodyID enthält vor dem ersten Zeitschritt $N$ natürliche Zahlen, wobei jeder Wert der Position im Array entspricht. Die Identifikationsnummer $k$ befindet sich also vor Beginn der Simulation an Positon $k$. BodyID benötigt somit $N$*4 Byte (int) an Speicherplatz. Während der Sortierungsphase der Simulation wird dieses Array periodisch auf der Grafikkarte nach räumlicher Position und entsprechender Zelle des Simulationsgebiet sortiert.

**cellID:**

Jeder Speicherstelle in bodyID entspricht eine Speicherstelle in cellID an derselben Position im Array. An dieser Position ist die Zelle des Simulationsgebiet gespeichert, in welcher sich der Fußgänger mit entsprechender Identifikationsnummer nach bodyID befindet. CellID benötigt genau wie bodyID $N$*4 Byte (int) Speicherplatz. Nach diesen Schlüsselwerten wird das bodyID Array in der Sortierphase des Algorithmus angeordnet und ist anschließend nach aufsteigenden Zellennummern geordnet.

**CellIndex:**

Die Anzahl der Speicherstellen des Arrays entspricht der Anzahl der Zellen, in die das Simulationsgebiet in der Binning Phase unterteilt wird. Jede Zelle erhält zwei Speicherstellen an der Position im Array die ihrer Zellennummer entspricht An der ersten Stelle wird in der Phase der Indizierung die Anfangsposition des bodyID Arrays gespeichert, an welcher die IDs der Körper abgelegt wurden, die sich in der Zelle befinden. Die zweite Stelle enthält die inkrementierte

Endposition des letzten in die Zelle sortierten Körpers. Enthält eine Zelle in einem Simulationszeitschritt keinen einzigen Fußgänger, sind beide Speicherwerte der entsprechenden Zelle nach der Indizierungsphase in cellIndex auf 0 gesetzt. Die Anzahl der Körper in einer Zelle ergibt sich somit als Differenz zwischen den beiden Einträgen. Mit Zellenanzahl $cellN$ wird insgesamt $2*cellN*4$ Byte (int) Speicher benötigt.

**bodyPos:**

Dieses Array bestehend aus 32 Bit Gleitkommazahlen und enthält die Körperpositionen im Simulationsgebiet in 2D. Jede Körperposition ist durch dessen Koordinaten im euklidischen Koordinatensystem, das über das Simulationsgebiet gedacht wird, bestimmt. Zwei aufeinanderfolgende Einträge im h_bodyPos Array entsprechen x und y Komponenten der Position. Das Array enthält also insgesamt $N*2*4$ Byte (float). Während der gesamten Simulation bleiben die neu berechneten räumlichen Positionen der Fußgänger an der gleichen Position im Array gespeichert. Die Position des Fußgängers mit Identifikationsnummer $k$ ist also an der Position $k*2$ abgelegt.

**bodyV:**

Analog zu bodyPos werden in diesem Array die Geschwindigkeitsvektoren jedes einzelnen Fußgängers gespeichert. Wieder sind beide Komponenten des Geschwindigkeitsvektors jedes Körpers an der seiner Identifikationsnummer entsprechenden Stelle (multipliziert mit 2) gespeichert. Das Array verbraucht ebenfalls $N*2*4$ Byte (float) an Speicher.

**bodyPrefV:**

Jeder Fußgänger hat das Bestreben sich mit seiner individuellen bevorzugten Geschwindigkeit auf sein derzeitiges Ziel zuzubewegen. Der Betrag dieser bevorzugten Geschwindigkeit wird über den Simulationszeitraum als konstant angenommen und an der Position im bodyV Array gespeichert die der ID des entsprechenden Fußgängers entspricht. Es ändert sich also lediglich die Richtung, welche in der direkten Verbindungslinie zwischen derzeitiger Position und Endziel der Trajektorie liegt. In jedem Zeitschritt wird dieser normierte Richtungsvektor erneut berechnet und mit dem für den jeweiligen Fußgänger im bodyV Array gespeicherten Geschwindigkeitsbetrag multipliziert. Dies wäre die bevorzugte Geschwindigkeit, die ein Fußgänger ohne jegliche Einflüsse annehmen würde. Die Differenz zwischen der bevorzugten und durch die wirkenden Abstoßungskräfte berechnete Geschwindigkeit bestimmt die treibende Kraft, die den Fußgänger aus eigenem Antrieb auf sein Ziel zusteuern lässt. Pro Fußgänger ist ein Geschwindigkeitsbetrag in bodyV zu speichern, das Array besteht also aus $N*4$ Byte (float).

**bodyGoal:**

Das Array enthält konstante 2*4 Byte (float) 2D Positionen, welche den Endzielen jedes Fußgängers im Simulationsgebiet entsprechen. Die Zielposition des Fußgängers mit ID $k$ ist an Position $k$ des bodyGoal Array gespeichert.

**Pairlist:**

In jedem Zeitschritt wird die Paarliste für die Bestimmung der benötigten Interaktionsberechnungen neu erzeugt. Nach Vergleich aller in Frage kommenden Körper werden die entsprechenden Körper-IDs im Pairlist Array abgespeichert. Die räumlich maximal mögliche Anzahl von Interaktionspartnern $N_{max}$ übersteigt keinesfalls das Verhältnis der Fläche der Interaktionsumgebung zur Fläche eines Körpers. Für jeden Fußgänger wird daher einmalig zu Simulationsbeginn ein Speicherbereich von $N_{max} * N * 4$ Byte (int) reserviert. Da es höchst unwahrscheinlich ist, all diese Speicherstellen wirklich in einem Zeitschritt zu benötigen, bleibt dieser Speicher während der Simulation nicht voll besetzt. Bei dynamisch angepasster Allozierung des Speichers in jedem Zeitschritt würde jedoch Verlustzeit für die Reservierung, als auch für den möglicherweise Ladevorgang der nicht zwangsläufig zusammenhängenden Speicherbereiche auftreten, welcher uncoalesced erfolgen würde. Für die simulierten Fußgänger wird der möglicherweise nur dünn befüllter Speicher statisch reserviert. In Anbetracht der Größe des Grafikkartenspeichers von mehreren GB ist dies verkraftbar und führt auch zu keiner Performanceminderung.

**pairsNo:**

Das pairsNo ID enthält einen Eintrag für jeden Körper, in dem die Anzahl der zu berücksichtigten Interaktionspartner in diesem Zeitschritt gespeichert ist. Er bestimmt also, wie viele Einträge aus der Paarliste jedes Fußgängers geladen werden müssen, um alle nötigen Interaktionen eines Zeitschritts zu berechnen. Die Anzahl der Interaktionen des Körpers, dessen ID an Stelle $i$ des bodyID Arrays gespeichert ist, wird während der Erstellung der Paarliste an Position $i$ des pairsNo Arrays abgelegt. Der erforderliche Speicherplatz beträgt also $N$*4 Byte (int).

Die beschriebenen linearen Speicherarrays werden bei Programmstart dynamisch erzeugt und initialisiert. Die dafür benötigten Speicherbereiche sind im Arbeitsspeicher der Host-CPU reserviert. Da die einzelnen Simulationsschritte jedoch auf dem Grafikprozessor ausgeführt werden, müssen ebenfalls entsprechende Speicherbereiche im Hauptspeicher der Grafikkarte erzeugt werden:

```
//create device buffers
d_bodyPos=clCreateBuffer(context,CL_MEM_READ_WRITE,2*CONST_bodyN*sizeof(float),
```

```
NULL,&err);
d_bodyV=clCreateBuffer(context, CL_MEM_READ_WRITE,2*CONST_bodyN*sizeof(float),
NULL, &err);
d_bodyGoal=clCreateBuffer(context,CL_MEM_READ_WRITE,2*CONST_bodyN*sizeof(float),
NULL,&err);
d_bodyPrefV=clCreateBuffer(context,CL_MEM_READ_ONLY,CONST_bodyN*sizeof(float),
NULL, &err);
d_bodyID=clCreateBuffer(context,CL_MEM_READ_WRITE,CONST_bodyN*sizeof(int),NULL,
&err);
d_cellID=clCreateBuffer(context,CL_MEM_READ_WRITE,CONST_bodyN*sizeof(int),NULL,
&err);
d_cellIndex=clCreateBuffer(context,CL_MEM_READ_WRITE,2*(grid->cellN)*sizeof(int)
,NULL, &err);
d_pairlist=clCreateBuffer(context,CL_MEM_READ_WRITE,maxPack*CONST_bodyN*sizeof(i
nt),NULL,&err);
d_pairsNo=clCreateBuffer(context,CL_MEM_READ_WRITE,CONST_bodyN*sizeof(int),NULL,
&err);
d_cellAdd=clCreateBuffer(context,CL_MEM_READ_WRITE,8*sizeof(int),NULL,&err);
d_objects=clCreateBuffer(context,CL_MEM_READ_WRITE,CONST_objectN*3*sizeof(float)
,NULL,&err);
check_ocl_error(err);
```

Nach Reservierung der entsprechenden Bereiche, welche über den gesamten Simulationszeitraum benötigt werden, ist es möglich die Daten aus den gleichstrukturierten Arrays im Arbeitsspeicher der Host-CPU zu übertragen. Klarerweise findet keine implizite Synchronisation der Daten zwischen CPU- und GPU-Speicher statt. Somit werden die Arrays auf dem Host zu einem beliebigen Zeitpunkt veraltete Simulationsdaten enthalten. Die Abfrage der aktuellen Daten findet nach jeweils 100 Zeitschritten der Simulation statt, um die Laufzeit des Programms zu minimieren. Durch die feine Zeitauflösung des numerischen Verfahrens bedingt dies aber keine Einschränkung für die Qualität einer Visualisierung der Trajektorien.

## 5.3 Struktur des Programms

Nach Hinzufügen der Kernel zur OpenCL Warteschlange für den verwendeten Grafikprozessor werden diese in derselben Reihenfolge ausgeführt in der sie eingereiht wurden. Nach jeder Kernelausführung müssen die Ausgangsdaten für die nächste Phase vorliegen. Die dazu nötige Synchronisationsbarriere wird durch die synchrone OpenCL Warteschlange garantiert, die mit der Ausführung des nächst gereihten Kernels erst nach Abschluss aller Operationen des vorherigen Kernels beginnt. Im Gegensatz dazu muss für Synchronisationsbarrieren innerhalb der Kernel deren

Gültigkeitsbereich beachtet werden. Synchronisation zwischen einzelnen Threads die als OpenCL Work-Items auf den Shadern ausgeführt werden kann nur innerhalb derselben Work-Group stattfinden. Erfolgt eine Speicheroperation auf ein und dieselbe Adresse aus zwei verschiedenen Work-Groups gibt es keine Möglichkeit vorherzusagen, in welcher Reihenfolge diese Operationen ausgeführt werden. Wird ein Wert in den lokalen Speicher einer Prozessoreinheit geladen, auf den jedoch im weiteren Verlauf Work-Items derselben Work-Group zugreifen, muss nach der Ladeoperation eine Synchronisationsbarriere stattfinden. Dies muss sicherheitshalber auch dann berücksichtigt werden, wenn sich diese Werte in einem zusammenhängenden Speicherbereich des Arbeitsspeichers befinden und an eine Speicherliniengrenze (aligned access) angepasst sind **[16].**

Jede der vorgestellten Phasen eines Simulationszeitschritts wird durch entsprechende OpenCL Kernel implementiert. Die Kernel werden nacheinander in die Warteschleife der GPU eingereiht und ermöglichen die Berechnung der neuen Positionsdaten. Abb. 10 zeigt diesen Vorgang. Für jeden OpenCL Kernel der Simulation ist eine eigene C++ Klasse definiert die alle erforderlichen Funktionen zur Erstellung und Ausführung eines Kernels enthält:

**BinPositions** - Binning Phase
**oclRadixSort** - Sortierphase
**IndexCells** - Indizierungsphase
**PairlistUpdate** - Phase für Erstellung der Paarliste
**IntegrateForce** - Phase der Kraftberechnung und Positionsupdate

Die Kernel-Programme werden zur Laufzeit von den entsprechenden Objekten per Konstruktor folgendermaßen produziert:

- Setzen der Anzahl von vorgesehenen OpenCL Work-Items und Work-Groups
- Laden des OpenCL Kernel Codes von der Festplatte
- Erstellung eines OpenCL Programmes mit dem entsprechenden Quellcode (clCreateProgramWithSource())
- Kompilieren des geladenen Quellcode zum ausführbaren Code (clBuildProgram())
- Erstellung des Kernel aus dem Programmcode (clCreateKernel())

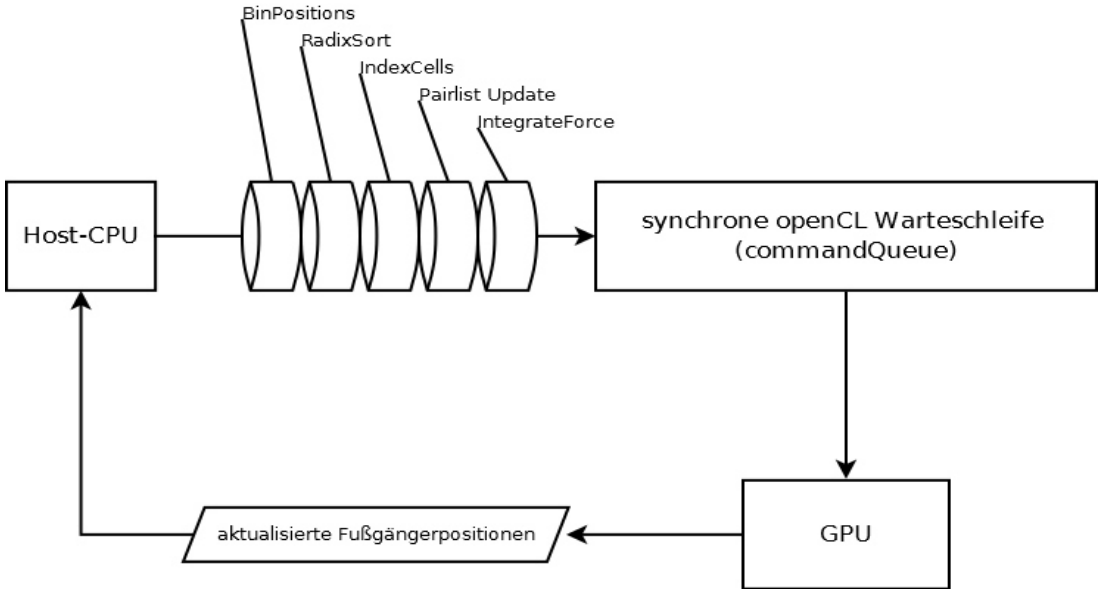


**Abbildung 10** *Einreihung der Kernel in die Warteschleife*

## 5.4 OpenCL Kernel der Simulation

Vor Simulationsbeginn werden die benötigten Kernel, welche die einzelnen Phasen eines Simulationsschritts implementieren, zu Laufzeit im OpenCL Kontext kompiliert, um in eine Warteschlange eingereiht werden zu können. Abb. 11 zeigt diesen Zusammenhang.

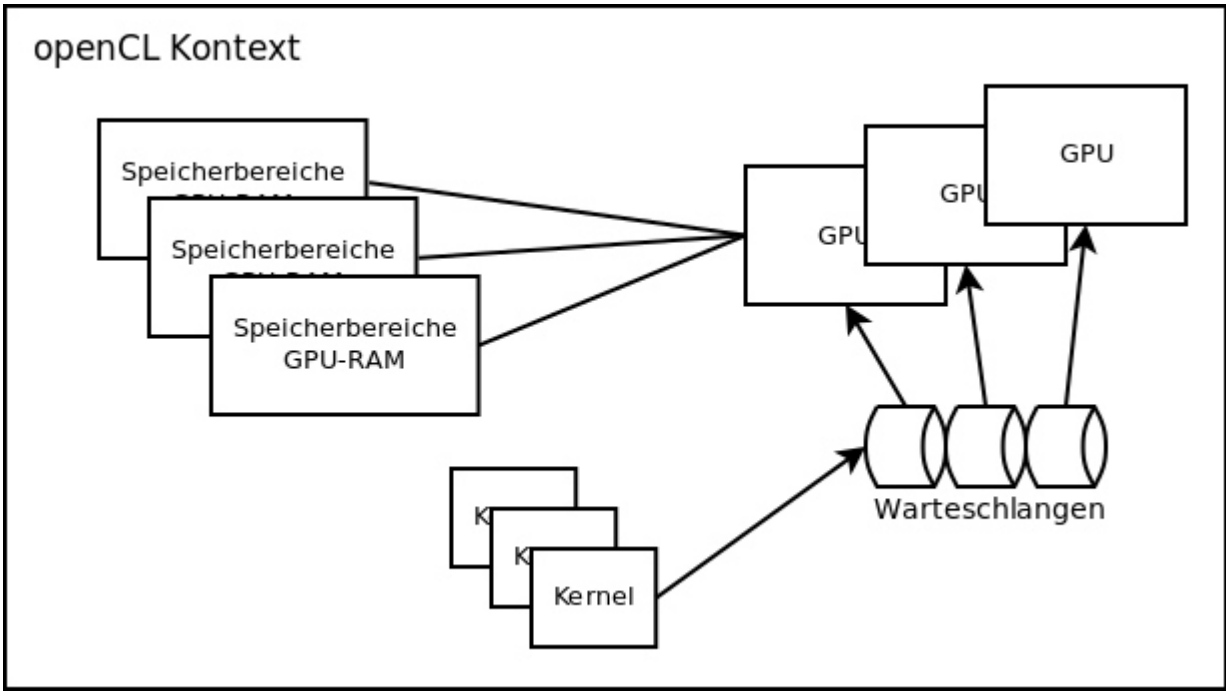


**Abbildung 11** *OpenCL Kontext*

Dazu wird der Kernel bei Aufruf des Konstruktors des zugeordneten Objekts dem übergebenen OpenCL Kontext und entsprechendem OpenCL Programm hinzugefügt und kompiliert.
Der Konstruktor jedes Kernelobjekts liest den Kernel Programm Code ein, erstellt ein entsprechendes OpenCL Programm (clCreateProgramWithSource) daraus, kompiliert den gewünschten Kernel (clBuildProgram) und erstellt (clCreateKernel) das zur Ausführung bereite Kernel-Objekt:

```
program = clCreateProgramWithSource(context,1,(const char **)&kernelString,
(const size_t *)&size ,&err);

check_ocl_error(err);

err =clGetProgramInfo(program, CL_PROGRAM_SOURCE, 0, NULL, &size);
char *buildLog=(char *)malloc(size);
err|=clGetProgramInfo(program, CL_PROGRAM_SOURCE, size, (void*)buildLog, NULL);
check_ocl_error(err);
printf("\n\nBuildlog:   %s\n\n",buildLog);
free(buildLog);

err = clBuildProgram(program, 0, NULL ,"-I ../inc" ,NULL, NULL);

if (err != CL_SUCCESS){
      printf("\n\nINTEGRATE BUILD FAILURE\n\n");
      //check_ocl_error(err);
}

cl_device_id deviceChoice;

clGetCommandQueueInfo(queue,CL_QUEUE_DEVICE,sizeof(cl_device_id),
(void*)&deviceChoice, NULL);

err =clGetProgramBuildInfo(program, deviceChoice, CL_PROGRAM_BUILD_LOG, 0, NULL,
&size);
char *buildLog=(char *)malloc(size);
err|=clGetProgramBuildInfo(program, deviceChoice, CL_PROGRAM_BUILD_LOG, size,
(void*)buildLog, NULL);
check_ocl_error(err);
printf("\n\nIntegrateForce buildlog: %s\n\n", buildLog);
free(buildLog);

integrateKernel = clCreateKernel(program, "integrateForce", &err);
check_ocl_error(err);
```

Nach Erstellung des Kernels und Bekanntgabe der zugeordneten Warteschleife (queue) der verwendeten GPU kann es jederzeit zur synchronen Ausführung gereiht werden. Jeder Kernel besitzt, wie eine gewöhnlich C/C++ Methode, eine Anzahl von Argumenten, die vor der ersten Einreihung zur Ausführung zumindest einmalig gesetzt werden müssen. Die Argumente sind Konstanten oder Zeiger auf bereits reservierte und möglicherweise initialisierte globale oder lokale Speicherbereiche auf der GPU. Durch Aufruf der set() Methode des entsprechenden Objekts werden die benötigten Argumente des Kernels gesetzt. Der Großteil der Argumente der Kernel, genauer die Zeiger auf den entsprechenden Speicherbereich bleibt über den Simulationszeitraum konstant. Lediglich die Speicherpositionen der Fußgängerdaten innerhalb der Speicherbereiche ändern sich. Daher reicht es den Großteil der Kernelargumente einmalig vor Simulationsbeginn festzulegen und die Kernel periodisch in die Warteschlange einzureihen, um unnötige Prozessorzeit zu sparen. Die privaten Variablen eines Kernels werden vorzugsweise innerhalb des Funktionskörper der Kernelfunktion deklariert, können aber auch als privat deklariertes Argument übergeben werden. Private Variablen haben keine Gültigkeit außerhalb eines OpenCL Work-Items. Nach Erstellung der Objekte für die Kernel müssen anschließend die Kernel Argumente gesetzt werden:

```
//initialize kernel objects
BinPositions *binPositions = new BinPositions(context, queue, localSize);
oclRadixSort *radixSort = new oclRadixSort(CONST_bodyN, context, queue);
IndexCells *indexCells = new IndexCells(context, queue, localSize);
PairlistUpdate *pairlistUpdate = new PairlistUpdate(grid->cellN, context, queue,
localSize, cellMaxPack, maxPack);
IntegrateForce *integrateForce = new IntegrateForce(context, queue, localSize);

//set kernel arguments once
binPositions->set(d_bodyID, d_bodyPos, d_cellID, grid);
radixSort->set(d_cellID, d_bodyID);
indexCells->set(d_cellID, d_cellIndex);
pairlistUpdate->set(grid->cellN, d_pairlist, d_pairsNo, d_bodyPos, d_bodyID,
d_cellIndex, d_cellAdd);
integrateForce->set(d_bodyID, d_pairlist, d_pairsNo, d_bodyPrefV, d_bodyPos,
d_bodyV, d_bodyGoal, d_objects);
```

Nach Erzeugung und Setzen der Kernelargumente durch das entsprechende Objekt werden die Kernel eines Simulationsschritts in die Warteschleife eingereiht, wie Abb. 10 veranschaulicht. Durch die synchrone OpenCL Warteschleife liegen die parallel abgearbeiteten Ausgangsdaten jedes Kernels für den nächsten Kernel konsistent bereit. Die Simulationsschleife wird eine gewisse Anzahl von

Zeitschritten durchgeführt, bzw. endet falls alle Fußgänger ihr Endziel erreichen. Durch periodische Einreihung einer Leseanforderung in die OpenCL Warteschleife werden die Trajektorien der Fußgänger periodisch aus dem Hauptspeicher der GPU gelesen:

```
//simulation
for(int i=0;i<CONST_timeSteps;i++)
{
      binPositions->bin();
      radixSort->sort();
      indexCells->index();
      pairlistUpdate->update();
      integrateForce->integrate();

//get buffers
      if(i%P==0){
            err    |=    clEnqueueReadBuffer(queue,    d_bodyID,    CL_TRUE,    0,
CONST_bodyN*sizeof(int), h_bodyID, 0, NULL, NULL);
            err    |=    clEnqueueReadBuffer(queue,    d_bodyPos,    CL_TRUE,    0,
2*CONST_bodyN*sizeof(float), h_bodyPos, 0, NULL, NULL);
                  check_ocl_error(err);
      }
}
```

### 5.4.1 Binning:

Das rechteckige Simulationsgebiet ist durch ein gleichmäßiges Gitter in quadratische Zellen unterteilt. Diese gleichmäßige Einteilung und das Binning dienen jedoch nur der Bestimmung der berücksichtigten benachbarten Fußgänger für die spätere Berechnung der Kräfte in Wirkungsreichweite. Die gleichmäßige Aufteilung verursachen keine load imbalance der GPU, da jeder Rechenkern die Gitterposition genau eines Teilchens berechnet. Beginnend bei der untersten linken Zelle sind diese Zellen mit 0 beginnend durchnummeriert, wie in Abb. 12 dargestellt. Jede Zellenzeile enthält $Nx$- und jede Spalte $Ny$ Zellen. Die Nummerierung erfolgt zeilenweise mit aufsteigender Reihenfolge, wie Abb. 12 zeigt.

| | | | | |
|---|---|---|---|---|
| (Ny-1)*Nx | (Ny-1)*Nx+1 | (Ny-1)*Nx+2 | ... | Ny*Nx-1 |
| ... | ... | ... | ... | ... |
| 2*Nx | 2*Nx+1 | 2*Nx+2 | ... | 3*Nx-1 |
| Nx | Nx+1 | Nx+2 | ... | 2*Nx-1 |
| 0 | 1 | 2 | ... | Nx-1 |

**Abbildung 12** *Nummerierung der Zellen*

Der Binning-Kernel hat die Aufgabe die ID der Zelle mit Seitenlänge $l$ zu bestimmen, in welcher sich ein Fußgänger laut seiner derzeitigen Position befindet. Der Wert der x-Koordinate bestimmt dabei die relative Position in der Zeile des Gitters und die y-Koordinate bestimmt die relative Position bezüglich der Spalte. Abb. 13 zeigt die Berechnung der Zelle. Somit ergibt sich die der Position zugehörige Zellen-ID als:

$$ID = floor\left(\frac{x}{l}\right) + Nx * floor(\frac{y}{l})$$

Die $floor()$ Funktion bestimmt hierbei den nächstkleineren ganzzahligen Wert.

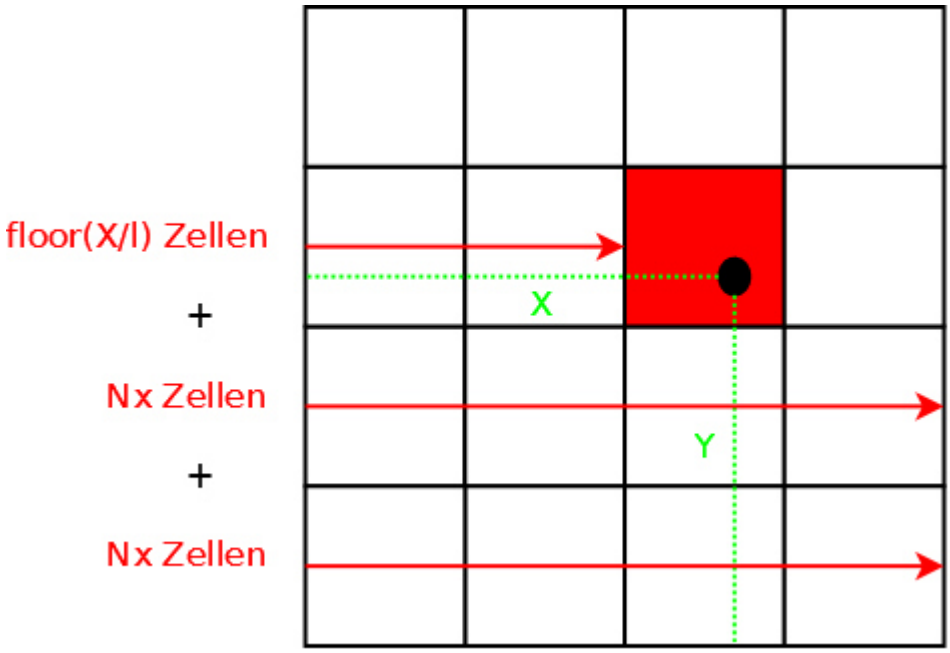


**Abbildung 13** *Berechnung der Zellen-ID aus Körperposition*

Der Kernel benötigt die Anzahl der Spalten sowie die Seitenlänge der quadratischen Zelle als konstante Eingangsparameter. Das Array der Positionen wird dem Binning-Kernel als globaler Speicherbereich als OpenCL Vektor Datentyp in 2 Komponenten übergeben. Die im bodyID Array gespeicherten Körper-IDs sowie das zu initialisierende cellID Array der zugehörigen Zellen-IDs werden als globale Speicherbereiche an den Kernel übergeben. Anschließend lädt jedes Work-Item die Körper-ID, die an der Speicherstelle seiner globalen Work-Item Nummer im Array der Körper-IDs enthalten ist. Somit ist im Vorhinein nicht klar, welcher Fußgänger von welchem Work-Item auf seine Zelle überprüft wird. Lediglich die Position im Array ist durch die globale Reihenfolge der Work-Items vorgegeben. An dieser Speicherposition befindet sich eine beliebige Körper-ID, abhängig der Sortierphase des vorherigen Simulationsschritts. Dies spielt jedoch keine Rolle, da jedes Work-Item eine unterschiedliche Körper-ID und dessen Zellenberechnung übernimmt und jeder Wert im Körper-ID Array, wie in Abb. 14 dargestellt, abgearbeitet wird.

Um die Zellen aller Körper zu bestimmen, sind also insgesamt gleichviele Work-Items wie simulierte Körper nötig. Mit der pro Work-Item geladenen Körper-ID kann die zugehörige Position des entsprechenden Fußgängers aus dem Positions Array privat geladen werden.

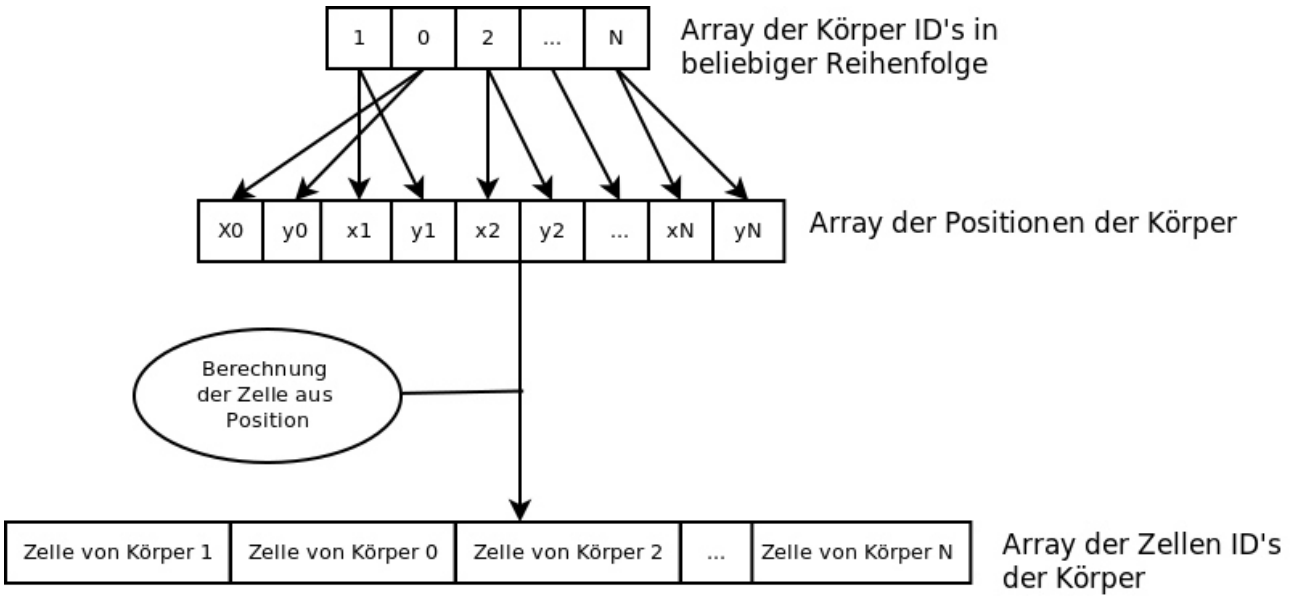


**Abbildung 14** *Speicherbereiche der Zellenberechnung*

Nach Bestimmung der Zellen-ID wird diese an die Position im globalen Zellen-ID Array geschrieben, die der aus dem Körper-ID Array geladenen Position gleichkommt. Ist also die Körper-ID $i$ an Position $p$ im Körper-ID Array gespeichert, wird auch die ID der Zelle, in der sich der Fußgänger mit Körper-ID $i$ räumlich befindet, an Position $p$ im Zellen-ID Array abgelegt. Diese Speicherordnung macht die gemeinsame Sortierung nach Zellen der nächsten Phase möglich und erfordert dazu keinen zusätzlichen Verwaltungsaufwand einer Referenzierung nach den Identifikationsnummern der Fußgänger. Die Bestimmung der Zellen-ID erfolgt durch folgenden Kernel:

```
__kernel void binPositions(const   int cellNx,
                                   const float cellLength,
                                   __global float2 *bodyPos,
                                   __global int *bodyID,
                                   __global int *cellID){
      int id = get_global_id(0);
      int body;
      float2 pos;
      if(id<CONST_bodyN){
            body=bodyID[id];
            pos=bodyPos[body];
            cellID[id]=floor(pos.x/cellLength)+cellNx*floor(pos.y/cellLength);
      }
}
```

### 5.4.2 Sortierung

Zur Sortierung des Körper-ID Arrays mit den Einträgen des Zellen-ID Arrays als Schlüsselwerten wird die High-Performance OpenCL Bibliothek libcl verwendet **[11].** Die Sortierung erfolgt mittels der RadixSort Methode, die eine Sortierung nach dem Radix Sortieralgorithmus mit 4 Bit Schrittweite implementiert. Wie Abb. 15 zeigt, sind die Körper-IDs nach dem Sortiervorgang nach Zellen geordnet, im Zellen-ID Array befinden sich die zugehörigen Zellen-IDs nach aufsteigender Reihenfolge.

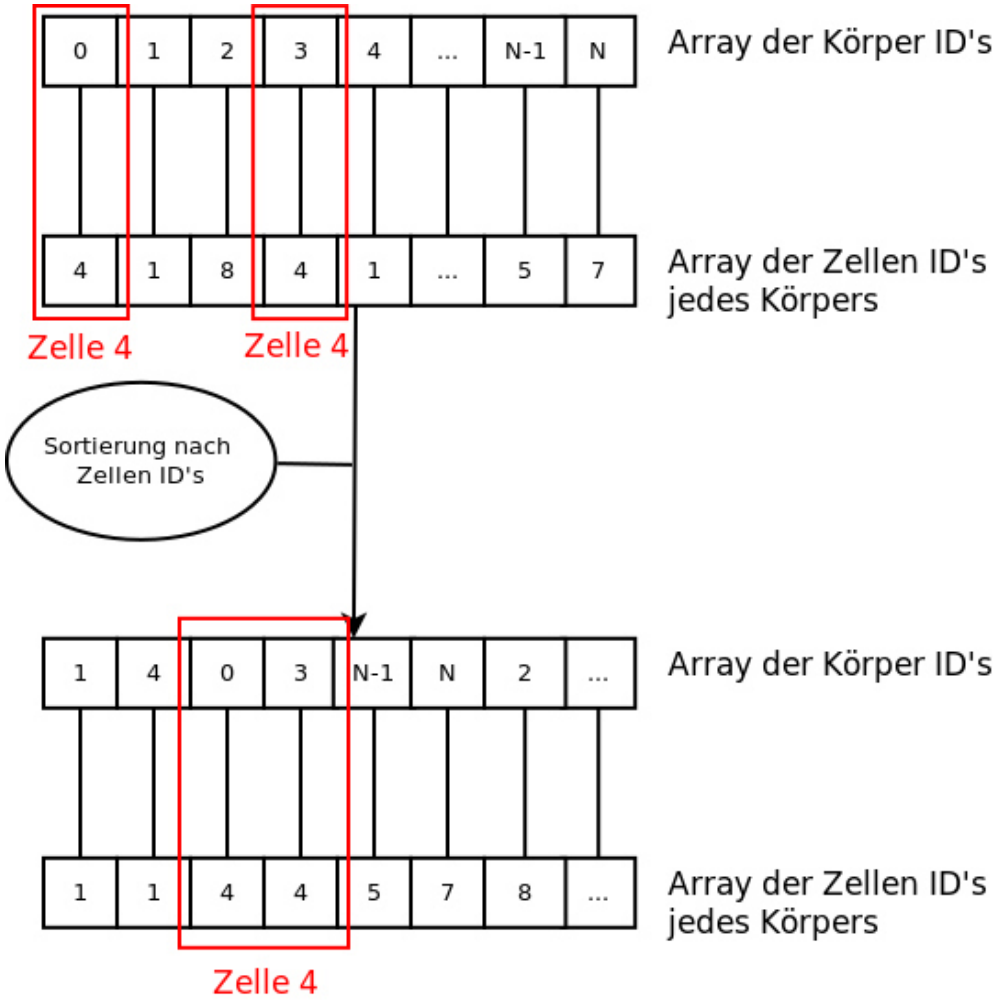


**Abbildung 15** *Sortiervorgang nach Zellen*

### 5.4.3 Indizierung

Nachdem die Körper-IDs nach aufsteigender Zellen-ID sortiert wurden, bestimmt der Indizierungs-Kernel die Speicherpositionen im Körper-ID Array, die Fußgänger einer bestimmten Zelle enthalten. Das dabei zu bestimmende Index-Array hält für jede Zelle zwei Einträge vor. Diese Einträge markieren die Start- und Endposition des Speicherbereichs im Körper-ID Array, in welchem die Körper-IDs der Fußgänger gespeichert sind, die sich momentan räumlich in der Zelle befinden. Die

Zelle mit ID *N* hat also zwei Einträge im Index-Array an den Positionen 2**N* und 2**N*+1 über die Körper-IDs der Zelle referenziert werden. Befinden sich die Fußgänger zweier angrenzende Einträge im Körper-ID Array in verschiedenen Zellen, so müssen auch nach der Sortierphase an denselben zwei Positionen im Zellen-ID Array unterschiedliche Zellennummern gespeichert sein. Somit ist bei Auffindung einer solchen Zellengrenze im Zellen-ID Array die Anfangsspeicherposition der Zelle sowie die Endposition der Körper-IDs der nächstbesetzten Zelle mit kleinerer Identifikationsnummer gegeben. Diese Zellengrenzen werden vom Indizierungskernel gefunden, indem jeder Speicherposition im Zellen-ID Array ein Work-Item zugeordnet wird. Jede Recheneinheit lädt nun den an dieser Position sowie den an der vorherigen Position gespeicherten Wert.

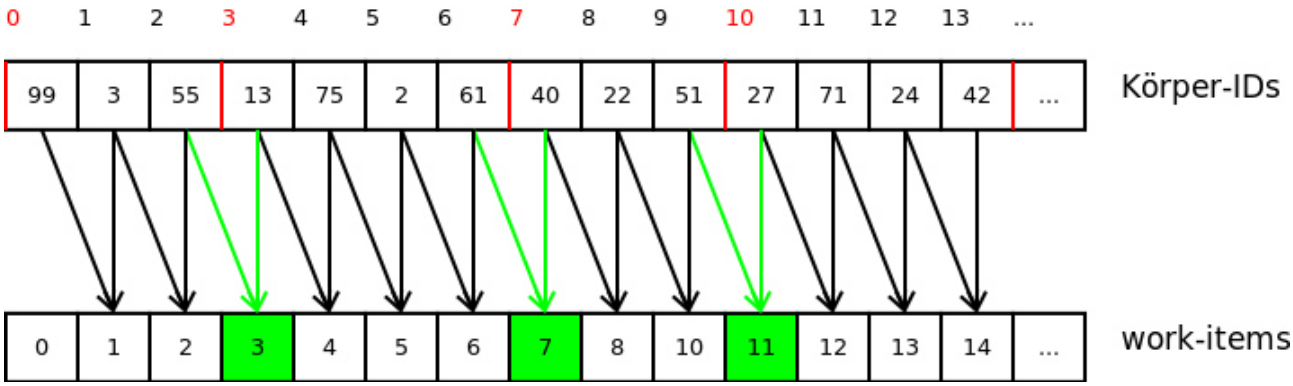


**Abbildung 16** *Auffindung von Zellgrenzen*

Sind die beiden geladenen Zellen-IDs unterschiedlich, handelt es sich offensichtlich um eine Zellengrenze und die Position, welche der globalen Work-Item Nummer entspricht, ist im Index-Array abzuspeichern. Die Position wird im Index Array als Anfangsposition der ersten geladenen Zellen-ID und als Endposition der geladenen Zellen-ID der nächstbesetzten kleineren Zelle gespeichert. Abb. 16 zeigt diesen Vorgang.

Falls beide von einem Work-Item geladenen Zellen-IDs übereinstimmen, sind an denselben Positionen IDs von Fußgängern der gleichen Zelle gespeichert. Da es sich also weder um Anfang- noch Endposition einer Zelle im Array handelt, werden diese von einem anderen Work-Item ins Index Array geschrieben. Ist eine Zelle komplett leer, so ist die ID dieser Zelle zu Beginn der Indizierung an keiner Position des Zellen-ID Array gespeichert und die zugehörigen Anfangs- und Endpositionen werden nicht beschrieben.

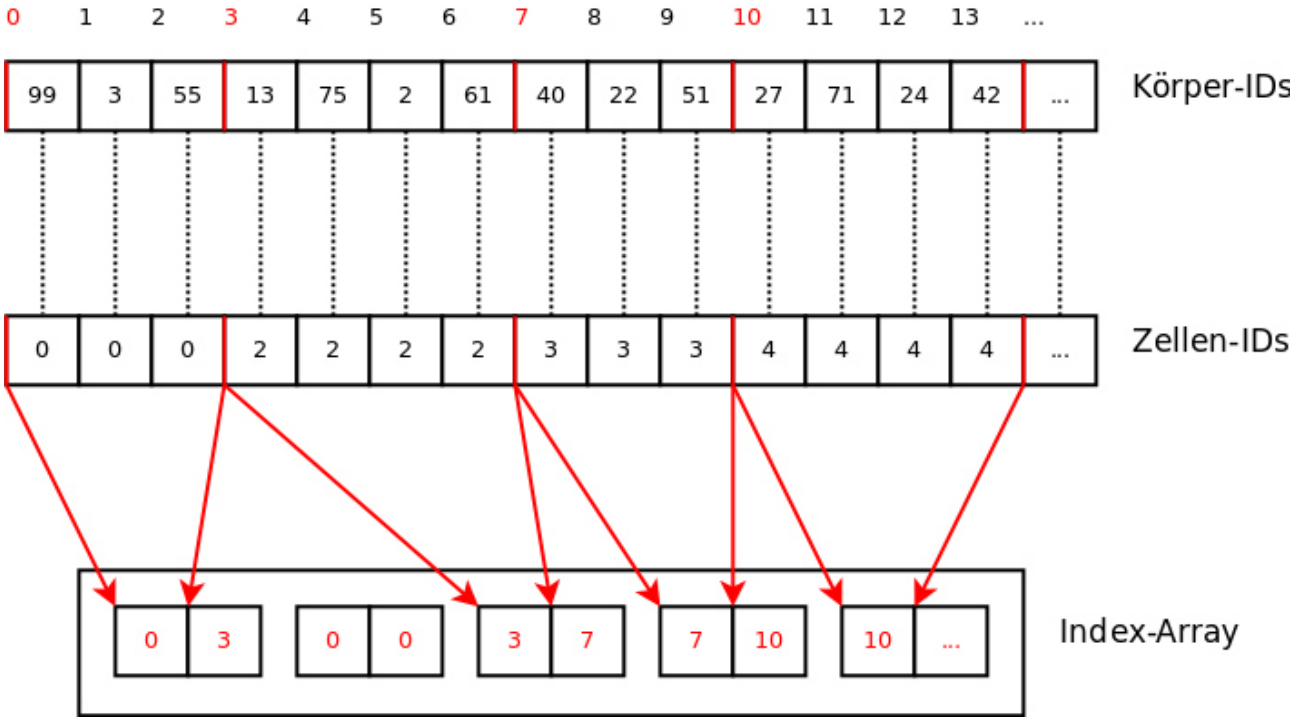


**Abbildung 17** *Speicherbereiche der Indizierung*

Sind alle Einträge des Index-Arrays, wie in Abb. 17 dargestellt, bestimmt, ergibt sich die Anzahl der in einer Zelle enthaltenen Fußgänger durch Differenzbildung der Endspeicherposition und Anfangsspeicherposition. Ist in einer Zelle kein Fußgänger zugegen, sind sowohl Anfangs- als auch Endspeicherposition 0. Nach jedem Simulationsschritt muss das gesamte Index-Array wieder auf 0 initialisiert werden, um die richtige Berechnung der in den Zellen enthaltenen Fußgänger zu gewährleisten, da vom Indizierungskernel bei vorhandenen Bereichsgrenzen nur auf den Speicherpositionen der beteiligten Zellen geschrieben wird. Der Wert der Anfangsposition im Index Array der Zelle 0 bleibt für die gesamte Simulation 0, während die Endposition (falls nötig) von einem anderen Work-Item gesetzt wird. Für den letzten Eintrag im Zellen-ID Array, also die letzte Zelle nach aufsteigender Nummerierung, die Fußgänger enthält wird die Endposition im Index Array auf $N$+1 gesetzt, wobei $N$ der Anzahl von Speicherstellen im Zellen-ID Array entspricht. Der entsprechende Kernel ergibt sich zu:

```
__kernel void indexCells(__global   int* ID, __global int* index){
      int cell, prevCell;
      int globalID = get_global_id(0);

      if(globalID<CONST_bodyN){
            cell = ID[globalID];
```

```
            if(globalID!=0 && cell!=(prevCell = ID[globalID-1])){
                  index[2*cell] = globalID;
                  index[2*prevCell+1]=globalID;
            }
      }
      if(globalID==CONST_bodyN-1)
                  index[2*cell+1]=CONST_bodyN;
}
```

### 5.4.4 Aktualisierung der Paarliste

Zur Erzeugung der Paarliste werden dem pairlistUpdate-Kernel die konstante Nummer der Zellen des Simulationsgebiets, die Zeiger auf die globalen Speicherbereiche für Paarliste (pairlist), Anzahl der Interaktionspartner jedes Fußgängers (pairsNo), die Positionen (Pos), Körper-Ids (bodyID), das Indizierungs-Array (cellIndex) sowie auf weitere lokale Speicherbereiche übergeben (die Notwendigkeit der lokalen Speicherbereiche des Kernels wird später genauer erläutert):

```
__kernel void pairlistUpdate( const int cellN,
                              __global int *pairlist,
                              __global int *pairsNo,
                              __global float2 *Pos,
                              __global int *bodyID,
                              __global int2 *cellIndex,
                              __global int *g_cellAdd,
                              __local int *localMem,
                              __local float2 *localMem2,
                              __local int *cellAdd)
```

Zur Erzeugung der Paarliste reicht es, die potentiellen Interaktionspartner der eigenen und der umliegenden 8 Zellen auf ihre Entfernung zu untersuchen. Jede OpenCL Work-Group hat die Aufgabe die Paarliste aller Fußgänger einer Zelle zu erzeugen. Dazu wird jedem Fußgänger einer Zelle ein Work-Item der entsprechenden Work-Group zugeordnet. Damit diese Zuordnung in jedem Zeitschritt funktioniert, muss die Anzahl der Work-Items in jeder Work-Group zumindest so groß wie die maximal mögliche Fußgängeranzahl, die räumlich in eine Zelle passen, sein. Jeder Work-Group wird diejenige Zellen-ID zugeordnet, die auch ihre eigene globale Work-Group Nummer entspricht. Dadurch ist jede Work-Group für einen zusammenhängenden Speicherbereich (Speicheroperationen können coalesced erfolgen) des Körper-ID Arrays zuständig, der alle Körper-IDs der in der

zugeordneten Zelle befindlichen Fußgänger enthält und der Speicherzugriff findet coalesced statt. Die Work-Group lädt diesen, in Speichergröße je nach enthaltenen Fußgängern variablen Bereich, durch seine Work-Items in den lokalen Speicher der verwendeten Recheneinheiten, wie in Abb. 18 veranschaulicht. Dazu lädt zuerst jedes Work-Item die Anfangs- und Endposition der Zelle aus dem Index-Array (cellIndex), welches durch den Indizierungs-Kernel geschrieben wurde. Durch Differenzbildung ergibt sich die Anzahl der in der Zelle enthaltenen Körper. Die Anzahl der in der Work-Group tatsächlich beschäftigten Work-Items entspricht dem errechneten Wert. Jedes Work-Item lädt nun die Körper-ID einer Speicherstelle aus dem seiner Work-Group zugeteilten Bereich der Zelle im Körper-ID Array. Die relative Position der Speicherstelle in diesem Bereich ist dabei die lokale Nummer des Work-Items innerhalb seiner Work-Group.

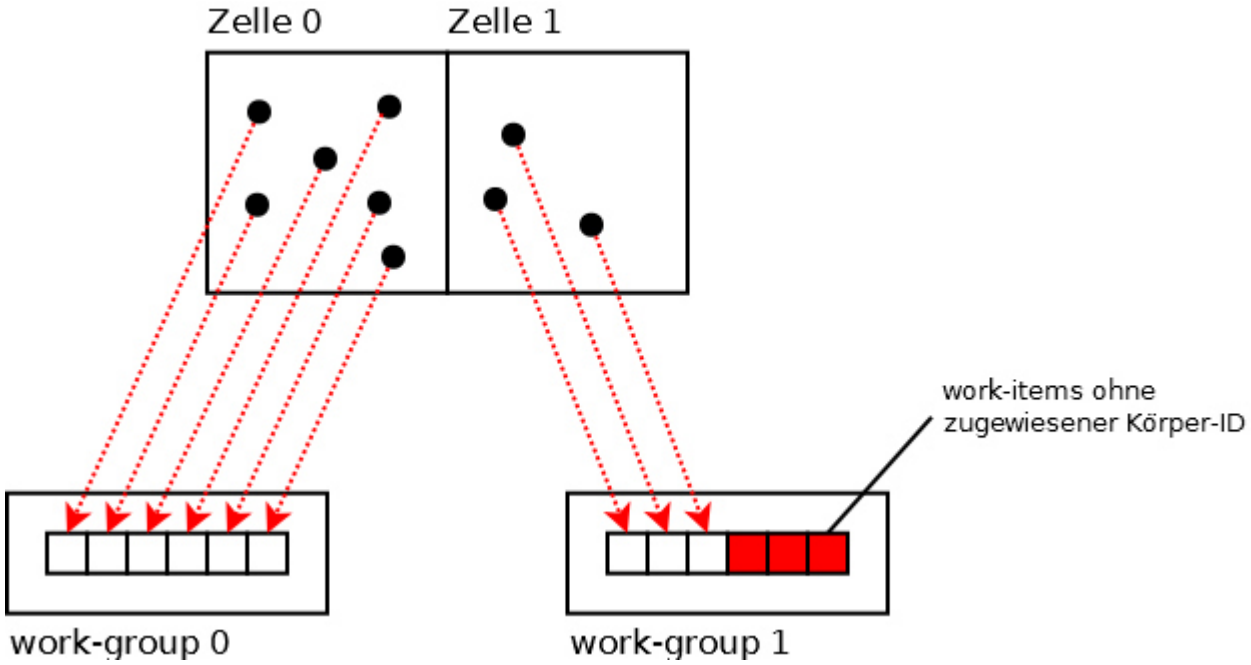


***Abbildung 18*** *Zelleinteilung in work-groups*

Nach Laden der Körper-ID lädt nun jedes Work-Item die zu dieser ID gehörende räumliche Position (Pos) des Fußgängers aus dem globalen Speicher. Nachdem alle Work-Items der Work-Group die Körper-IDs und Positionen in der Zelle in den lokalen Speicher der Prozessoreinheit geladen haben, muss eine Work-Group eine interne Synchronisationsbarriere erzwingen, um diese übergreifend zwischen den parallelen Work-Items zu nutzen. Da nach der Synchronisationsbarriere alle Positionsdaten und Körper-IDs in der Zelle der Work-Group vorhanden sind, können die Work-Items mit der paarweisen Abstandsberechnung fortfahren.

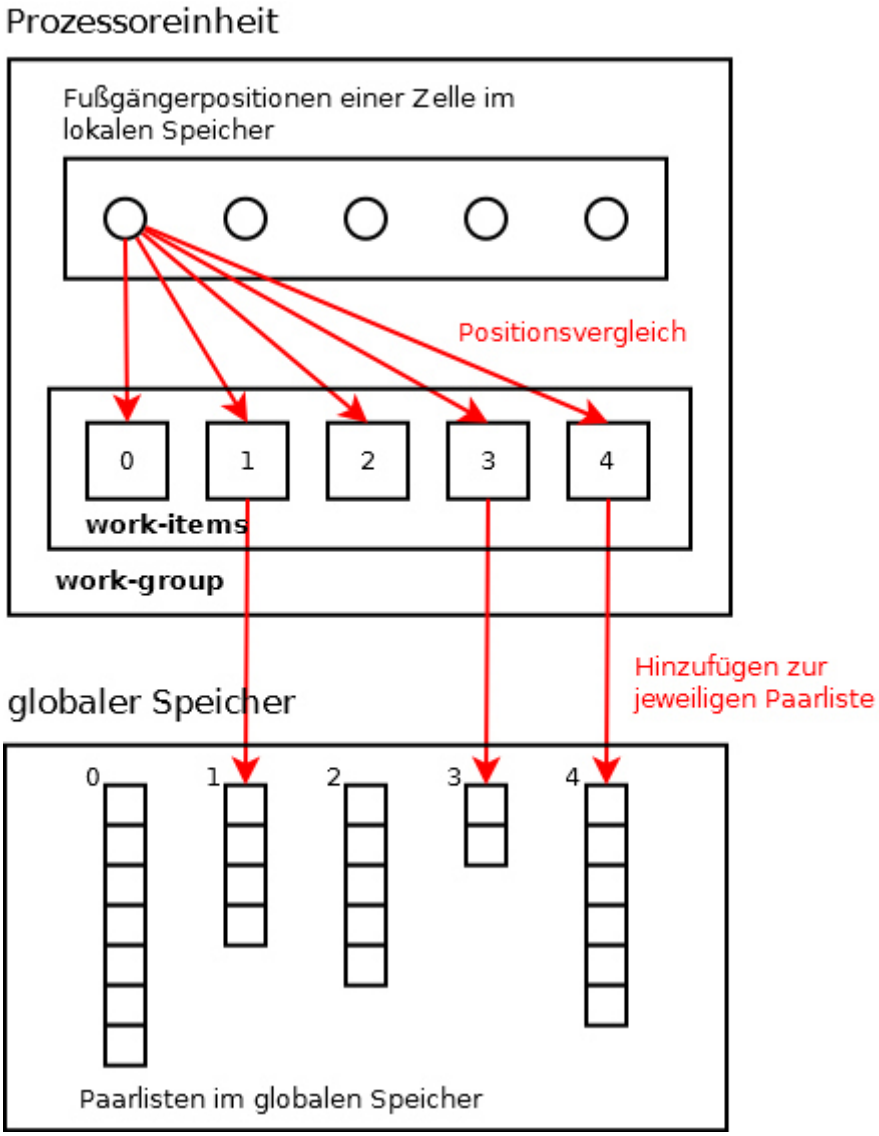


**Abbildung 19** *Erzeugung der Paarliste*

Dazu iteriert jedes Work-Item über alle geladenen lokalen Speicherstellen der Work-Group für Körper-ID und Position und berechnet in jedem Iterationsschritt die euklidische Distanz zu der Position des eigenen Fußgängers. Liegt diese Distanz innerhalb des Interaktionsradius, wird die zugehörige Körper-ID des Interaktionspartners in die globale Paarliste gespeichert, wie in Abb. 19 dargestellt und die Partneranzahl inkrementiert.

Die Position im Speicherbereich für die Paarliste eines Körpers entspricht der absoluten Position der Körper-ID im Körper-ID Array. Das Speicherschema der Paarliste ist in Abb. 20 ersichtlich.

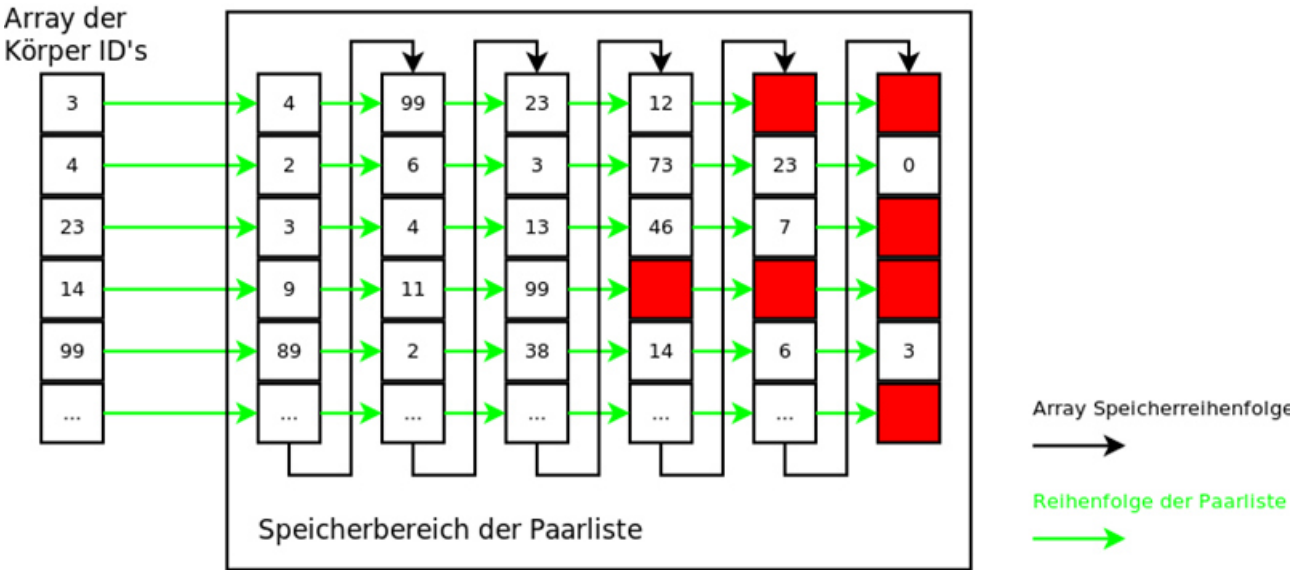


**Abbildung 20** *Speicherschema der Paarliste*

Der OpenCL-Code zur Berechnung der Paarliste ergibt sich als:

```
indexVec = cellIndex[groupID];
count = indexVec.y - indexVec.x;

if(localID<count){
      index = indexVec.x+localID;
      localMem[localID]=bodyID[index];

      position          = Pos[localMem[localID]];
      localMem2[localID] = position;
}

barrier(CLK_LOCAL_MEM_FENCE);

if(localID<count){
      for(i=1;i<count;i++){
            partner = (localID+i)%count;
            if(norm(localMem2[partner]-position)<=CONST_cutoff+4*FLT_EPSILON){
                  pairlist[CONST_bodyN*j+index]=localMem[partner];
                  j++;
            }
      }
}
```

Sind sämtliche paarweisen Vergleiche innerhalb der eigenen Zelle abgeschlossen, werden nach ähnlichem Schema die Körper-IDs und Positionen der Fußgänger in den 8 umliegenden Zellen geladen. Dazu wird die Nummer einer Nachbarzelle aus der Nummer der eigenen Zelle und Work-Group sowie den cellAdd Array bestimmt, welches die Rechenoperation zur Berechnung der Nachbarzellen enthält. Für jede der acht Nachbarzellen erhält man die entsprechende Zellen-ID durch addieren des korrespondierenden Wert im cellAdd Array. Die entsprechenden Werte der Nachbarzellen sind in Abb. 21 dargestellt.

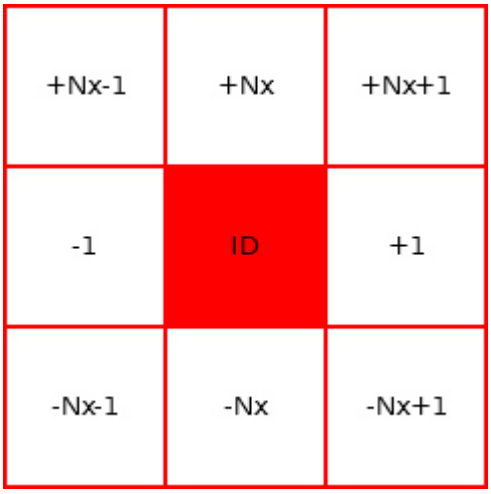


***Abbildung 21*** *Berechnung der Nachbarzellen*

Ist die Nachbarzelle bestimmt lädt wieder jedes Work-Item für seine Recheneinheit die Körper-ID eines Fußgängers aus dem Speicherbereich der Nachbarzelle im Körper-ID Array. Das Work-Item mit lokaler Nummer *N* innerhalb der Work-Group lädt dabei auch den Speicherinhalt der relativen Position *N* innerhalb des Speicherbreichs der Nachbarzelle. Anschließend wird die zur geladenen Körper-ID zugehörige Position in den lokalen Speicher geladen. Sind alle Daten geladen, garantiert durch eine weitere Synchronisationsbarriere, vergleicht jedes Work-Item wieder die private Position seines bei Kernelbeginn geladenen Fußgängers mit den Positionen der aus der Nachbarzelle in den lokalen Speicher geladenen Körper-IDs. Wieder entscheidet der Abstand über das Hinzufügen zur Paarliste. Dieser Vorgang wird in einer Schleife für alle verbleibenden acht Nachbarzellen ausgeführt:

```
for(k=0;k<8;k++){
      if(count!=0){
      indexVec = cellIndex[groupID+cellAdd[k]];
      activeCount = indexVec.y - indexVec.x;
```

```
            if(localID<activeCount){
                  localMem[localID] = bodyID[indexVec.x+localID];

                  localMem2[localID] = Pos[localMem[localID]];
      }
}

      barrier(CLK_LOCAL_MEM_FENCE);

      if(localID<count){
            for(i=0;i<activeCount;i++){
                  if(norm(localMem2[i]-position)<=CONST_cutoff+4*FLT_EPSILON){
                        pairlist[CONST_bodyN*j+index]=localMem[i];
                        j++;
                        }
            }
      }
}
```

### 5.4.5 Kraftberechnung und Positionsupdate

Der Kernel zur Erzeugung der Paarliste legt alle zu berechnenden Interaktionen fest. Der integrateForce Kernel hat nun die Aufgabe alle wirkenden Kräfte auf einen Körper zu bestimmen, diese aufzusummieren und die daraus resultierende Positions- und Geschwindigkeitsänderung in diesem Zeitschritt anzuwenden. Dazu werden die Kräfte, ausgehend von allen Interaktionspartnern, bestimmt und anschließend die von Objekten ausgehenden Abstoßungskräfte aufsummiert.

Die Kernelargumente sind die Zeiger auf die benötigten globalen Speicherbereiche der Körper-IDs (bodyID), der Paarliste (pairlist) und der Anzahl der zu berücksichtigenden Interaktionspartner (pairsNo), die in der vorigen Phase bestimmt wurden. Weiters werden aktuelle Position (Pos), aktuelle Geschwindigkeit (v), Ziel (goal) und die Liste der Objekte als Zeiger auf globale Arrays übergeben. Die Ziele der einzelnen Fußgänger sind nötig, um vor dem Positionsupdate die treibende Kraft, welche den Fußgänger wieder auf sein Ziel zusteuern lässt, zu bestimmen. Die Arrays der Positionen, Geschwindigkeiten, Ziele und Objekte werden als OpenCL 2D Vektor Datentyp übergeben:

```
__kernel void integrateForce(   __global int *bodyID,
                                __global int *pairlist,
```

```
                                __global int *pairsNo,
                                __global float *velPref,
                                __global float2 *Pos,
                                __global float2 *V,
                                __global float2 *goal,
                                 __global float2 *objects)
```

Jedes Work-Item lädt eine Körper-ID aus dem bodyID Array. Alle Berechnungen für den Fußgänger mit der geladenen ID werden von diesem Work-Item übernommen. Nach Laden der Anzahl der zugehörigen Interaktionspartner aus dem pairsNo Array wird in jeder Iteration der Schleife die resultierende paarweise Kraft zur Gesamtkraft summiert. Dazu muss die ID des derzeitigen Interaktionspartners aus der Paarliste geladen werden, um anschließend die zugehörigen Positions- und Geschwindigkeitsvektoren zur Berechnung der Relativposition bzw. Relativgeschwindigkeit zu referenzieren. Nach diesen Ladeoperationen sind alle nötigen Daten des Interaktionspartners bekannt. Nun werden alle benötigten Skalarprodukte aus Positions- und Geschwindigkeitsvektoren ausgewertet, um die voraussichtliche Kollisionszeit mit dem Fußgänger des derzeitigen Work-Items zu bestimmen. Nur wenn die resultierend Kollisionszeit positiv ist, wird die resultierende Kraft zur Gesamtkraft addiert. Insgesamt ergibt sich die Interaktionskraft $\vec{F}_i$ . Bei negativen Werten entfernen sich die beiden Fußgänger voneinander und es findet keine Interaktion und Geschwindigkeitsanpassung statt. Der entsprechende OpenCL-Code zur Berechnung der Kraft:

```
v*=velPref[body]/sqrt(a);
F=(v-vel)/CONST_ksi;
count = pairsNo[globalID];

for(i=0; i<count; i++){

      partner = pairlist[CONST_bodyN*i+globalID];
      w=Pos[partner]-pos;
      a=dotProduct(w, w);
      b=CONST_radius+CONST_radius;
      d=b*b;

      if(a<d){
            b=CONST_radius+CONST_radius-sqrt(a);
            d=b*b;
}
```

```
      v = vel - V[partner];
      d = a-d;
      a = dotProduct(v,v);
      b = dotProduct(w,v);
      d = b*b - a*(d);

      if(d > .0f &&){
            d = sqrt(d);
            t = (b-d)/a;
            if(t>0)
                  F+=-CONST_k*exp(-t/CONST_t0)*(v-(b*v-
a*w)/d)/(a*powr(t,(float)CONST_m))*(CONST_m/t + 1/CONST_t0);
      }
}
```

Nach analoger Abarbeitung aller anderen Interaktionspartner werden die von statischen Objekten ausgehenden Kräfte berechnet, diese ergeben insgesamt die von Objekten ausgehende Kraft $\vec{F}_o$. Zuerst wird geprüft ob eine Kollision stattfindet, also ob sich der Fußgänger auch tatsächlich auf das Objekt zubewegt. Ist dies der Fall wird die voraussichtliche Kollisionszeit bestimmt. Befindet sich der Fußgänger zu dieser Zeit vorrausichtlich an der Objektgrenze wird, die resultierende Kraft zur Gesamtkraft addiert. Ist das begrenzte Objekt jedoch zum potentiellen Kollisionszeitpunkt nicht innerhalb des Ausdehnungsradius des Körpers, wird der Fußgänger nicht beeinflusst:

```
a=sqrt(dotProduct(e,e));
e=e/a;
a=dotProduct(e,(pos-v));

//calculate nearest point on obstacle
w=v+e*a-pos;
b=sqrt(dotProduct(w,w));
w=w/b;
d=dotProduct(vel,w);

if(d > .0001f){
            t = (b-CONST_radius)/d;
            b=dotProduct(vel,e)*t-a;
            if(b<CONST_radius&&b>sqrt(dotProduct(e,e))+CONST_radius)
                        F+=CONST_k*exp(-
t/CONST_t0)/(powr(t,(float)CONST_m)*d)*(CONST_m/t + 1/CONST_t0)*w;
```

Nachdem alle abstoßenden Kräfte $\overrightarrow{F_i} + \overrightarrow{F_o}$ berechnet sind, bleibt die treibende Kraft $\overrightarrow{F_t}$, die vom Ziel des Simulationspfades ausgeht zu bestimmen. Nach Laden der bevorzugten Geschwindigkeit und resultierender treibender Kraft ergibt sich die Gesamtkraft $\vec{F}$ auf den dem Work-Item zugeordneten Fußgänger als:

$$\vec{F} = \overrightarrow{F_t} + \overrightarrow{F_o} + \overrightarrow{F_i}$$

Da jeder Fußgänger nur eine gewisse maximale Beschleunigung umsetzen kann, muss die Gesamtkraft anschließend angepasst werden und der Kraftüberschuss subtrahiert werden. Dazu wird der Betrag des Kraftvektors gegebenfalls normiert und anschließend mit dem Betrag der maximal möglichen Beschleunigung skaliert. Zur Berechnung der Positionsänderung wird das vorgestellte diskrete Verfahren auf die Kraft angewendet, wobei die "Masse" $m$ jedes Fußgängers durch das Kraftgesetz auf 1 skaliert ist. Die Positionsänderung wird durch das semi-implizite Euler-Verfahren berechnet:

```
a = sqrt(dotProduct(F,F));


if(a>CONST_maxAccel)
      F*=(CONST_maxAccel/a);

      vel = vel + F*CONST_dt;
      V[body] = vel;


      pos += vel*CONST_dt;
      Pos[body] = pos;
```

## 5.5 Serielle Implementierung

Zum Vergleich der Laufzeit des parallelen Programms auf der Grafikkarte wird die Simulation ebenfalls in serieller Form auf der CPU ausgeführt. Die paarweisen Kräfte werden auf gleiche Weise berechnet, die vorgestellten Phasen für Binning, Sortierung und Sortierung werden nicht durchgeführt, da weder auf die Speicherhierarchie, noch den vielen Recheneinheiten des Grafikprozessors Rücksicht genommen werden muss. In den Grenzen der Genauigkeit der jeweils implementierten mathematischen Funktionen stimmen die Simulationsdaten also überein.

# 6 Optimierung

## 6.1 Speicherzugriffsmuster coalesced/uncoalesced

Das Zugriffsmuster auf Daten aus dem globalen Speicher hat entscheidenden Einfluss auf die Performance der Kernel. Findet der Zugriff nach einem zufälligem Muster statt müssen die eventuell breit verteilten Zugriffe, wie in Abb. 22, unter Umständen einzeln und nacheinander erfolgen.

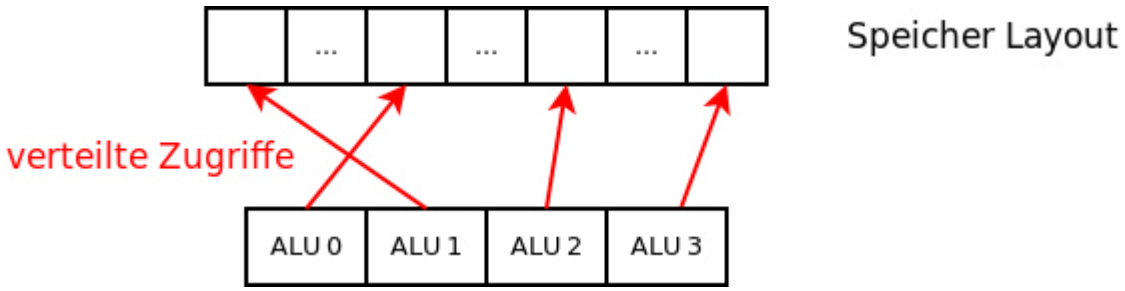


**Abbildung 22** *Ladeoperationen uncoalesced*

Finden Speicherzugriffe jedoch coalesced statt, also benötigen benachbarte Work-Items auch Daten aus benachbarten Speicherstellen, kann dies mit Lade- bzw. Schreiboperationen, welche coalesced möglich sind, bewerkstelligt werden. Diese Lade/Speicheroperationen sind natürlich um ein vielfaches effizienter, als mehrere Speicheroperationen auf komplett unterschiedliche Stellen, da dort jeweils eine komplette Speicherlinie geladen werden muss. Speicherlinie bezeichnet hier die kleinste, durch eine Lade/Speicheroperation ansprechbare Einheit des Arbeitsspeichers.

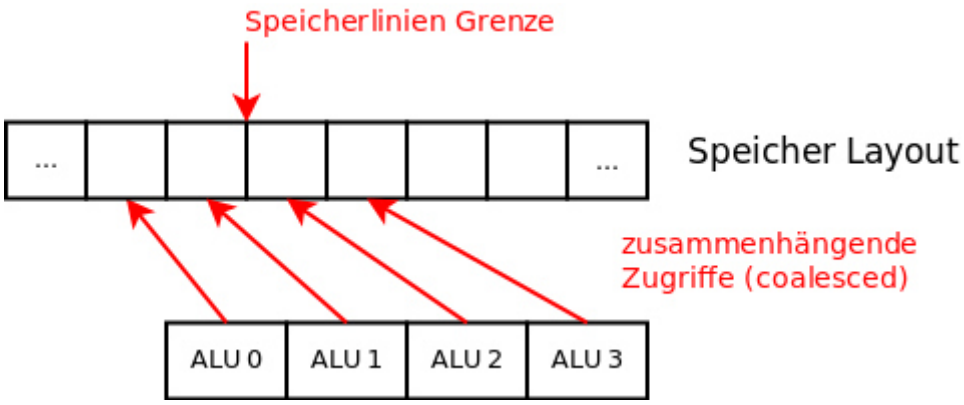


**Abbildung 23** *Ladeoperationen coalesced*

Sind Daten über eine Speicherliniengrenze wie in Abb. 23 hinweg gespeichert, also unaligned, sind mehrere Ladeoperationen erforderlich, da eine Speicheroperation nur an bestimmten Speicherliniengrenzen coalesced sein kann. Durch den in modernen Grafikprozessoren verfügbaren Cache werden die Daten der vorherigen Ladeoperation jedoch wiederverwendet und es tritt kein

signifikanter Zeitverlust auf, deshalb wurde im entwickelten Programm meist kein explizites Alignment berücksichtigt. Sind die Daten wie in Abb. 24 an die Speicherliniengrenzen angepasst abgespeichert, erfolgt der Zugriff auf diese auch aligned.

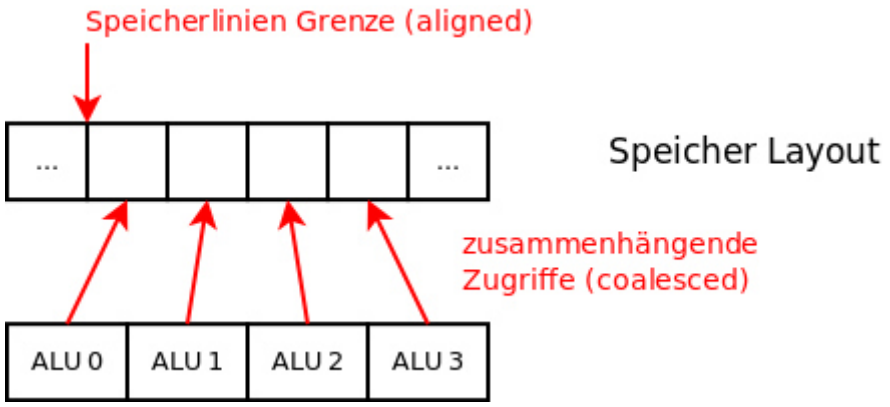


**Abbildung 24** *Ladeoperation coalesced und aligned*

Liegen die benötigten Daten nicht explizit im lokalen Speicher für die mehrmalige Verwendung mit geringeren Latenzen, müssen sie immer aus dem um Größenordnungen langsameren globalen Speicher geladen werden, wenn sie durch vorherige Operationen nicht im Cache zwischengespeichert sind, illustriert in Abb. 25. Die hier angestellten Überlegungen werden in jedem Kernel ausgenutzt, um möglichst alle Schreib und Ladezugriffe coalesced durchzuführen. Ist dies nicht möglich, wird es explizit bei der Beschreibung der Kernel und der jeweiligen Speicheroperation in den entsprechenden Kapiteln angeführt

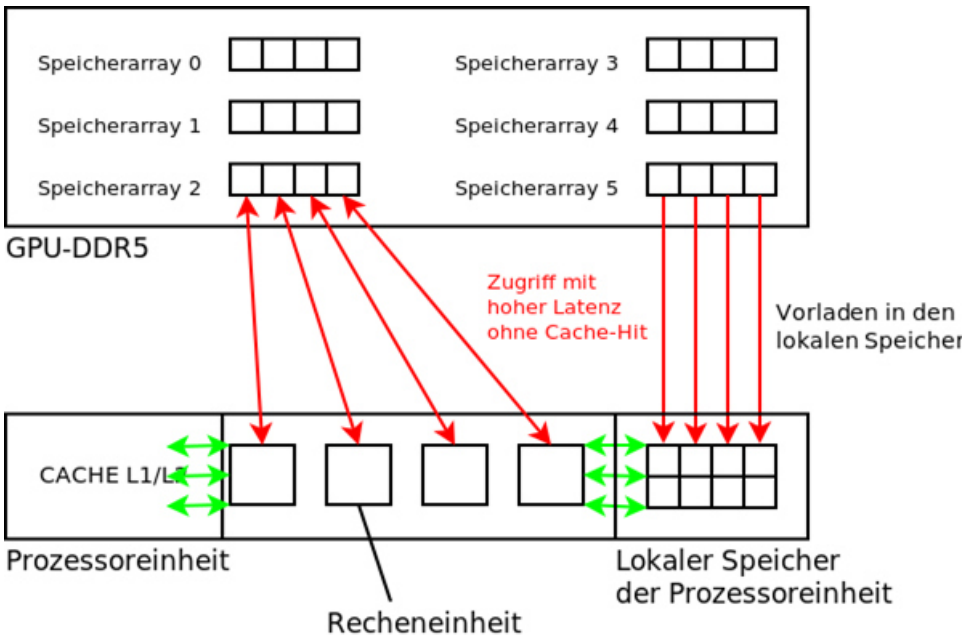


**Abbildung 25** *globale Speicherzugriffe*

## 6.2 Divergenter Kontrollfluß

Falls mehrere Work-Items einer Wavefront unterschiedlichen Ausführungszweige nach einer Fallunterscheidung (z.B. durch ein if/else Konstrukt) folgen, kann dies nicht parallel nach dem SIMD Prinzip funktionieren. Es werden daher beide Ausführungszweige schlimmstenfalls, wie in Abb. 26 dargestellt, hintereinander ausgeführt und die Anweisungen der beiden Zweige in zwei verschiedene Wavefronts eingeteilt. Ist einer der Ausführungszweige nach einer Konditionsauswertung jedoch leer, gibt es in der Regel keine zusätzliche Rechenzeit zu verbuchen. Der Code des Programms ist absichtlich derart gestaltet, dass sämtliche vorkommende Fallunterscheidungen keinen divergenten Kontrollfluss verursachen. Dies bedeutet, dass sämtliche unausweichliche if-Kontrollstrukturen nur auf bestimmte Threads zutreffen und für sämtliche verbleibende Threads derselben Wavefront keine Anweisungen, abhängig von diesen Bedingungen, auszuführen sind. Es kann damit kein divergenter Kontrollfluß, der in erhöhter serieller Abarbeitung der Threads einer Wavefront resultiert, stattfinden.

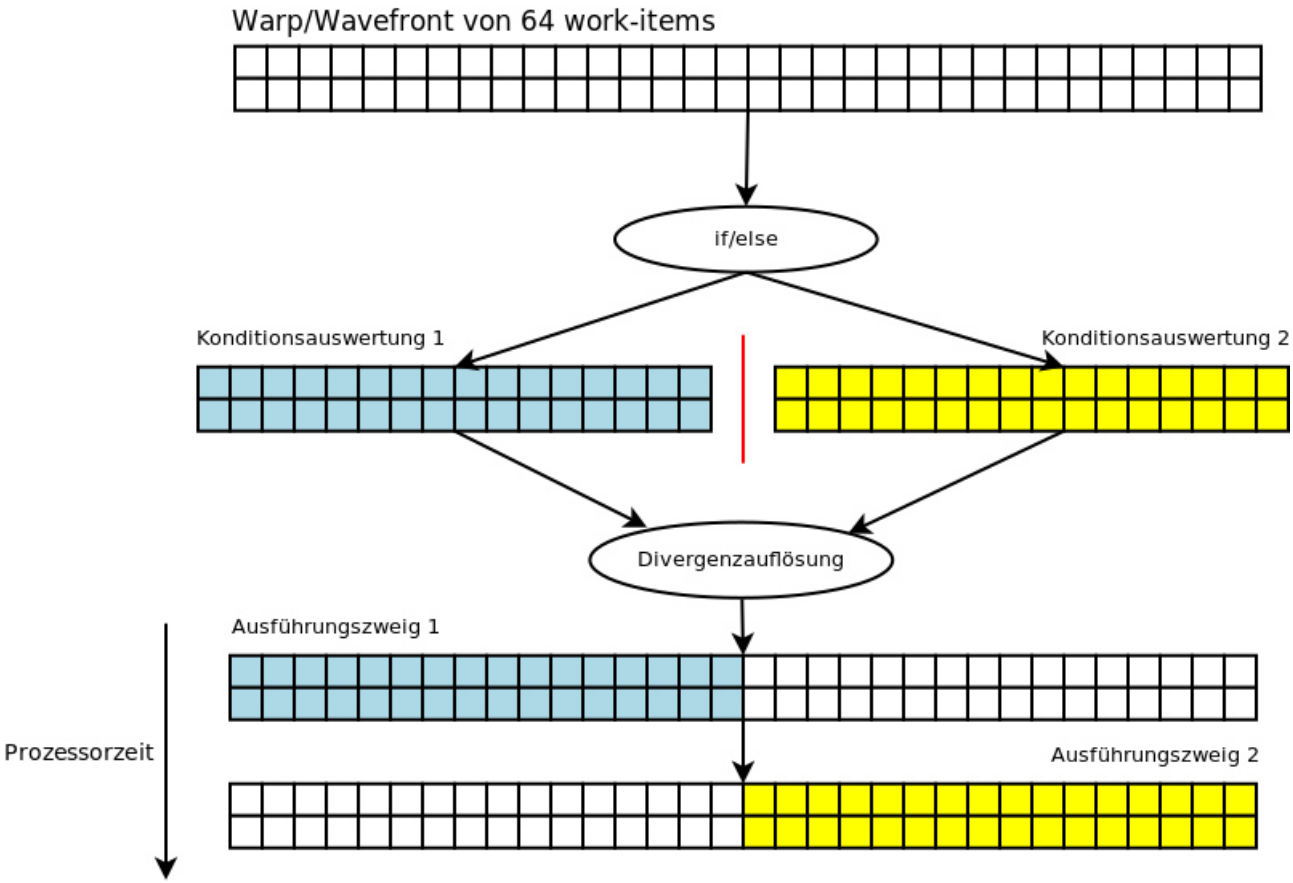


**Abbildung 26** *Ausführung bei divergentem Kontrollfluß*

### 6.3.1 Binning Performance:

Zur Speicherung der globalen ID des Work-Items und der Körper-ID des zugeordneten Fußgängers sind zwei 32 Bit Ganzahlvariablen im privaten Speicher des Kernels reserviert. Das Laden der zugehörigen Position erfordert einen weiteren privaten 64 Bit 2D Vektor. Für die Zellkonstanten werden zwei Ganzzahlargumente an die Kernelfunktion mit insgesamt 64 Bit übergeben. Insgesamt sind also 192 Bit privater Speicherplatz pro Work-Item in Verwendung. Selbst bei einer maximalen für den Bonair Grafikchip zulässigen Anzahl von 256 Work-Items pro Work-Group ergibt sich insgesamt ein privater Speicher von nur 6 kB. Bei 64 kB verfügbarem lokalem und privaten Speicher pro Vektoreinheit ist damit genug Platz zur Vorhaltung der Daten der Wavefronts, um schnellen Wechsel des Ausführungskontextes zu garantieren. Den 64 kB entsprechen dem Speicherbedarf von ungefähr 10 Work-Groups. Position, Zellen-ID und Körper-ID werden aus dem globalen Speicher geladen. Es sind keine lokalen Speicherbereiche zur Work-Item übergreifenden Verwendung innerhalb einer Work-Group nötig. Der Zugriff auf die Werte des Körper-ID Arrays erfolgt coalesced, da benachbarte Work-Items auf benachbarte Speicherpositionen zugreifen. Durch diese Körper-IDs referenzierten Positionsvektoren werden allerdings nicht coalesced geladen. Diese Ladeoperation erfolgt uncoalesced und stellt den größten Ladezeitaufwand dar. Schreibzugriffe auf die Positionen des Zellen-IDs erfolgen coalesced, benachbarte Work-Items schreiben auch benachbarte Zellen-IDs. Da die zusammenhängenden Lade- und Schreiboperationen elementweise über das gesamte Array und alle verfügbaren Work-Items ohne Lücken ausgeführt werden, ist das Speicherlayout hier optimal ausgenutzt.

Um die Performance des Kernels weiter zu erhöhen wäre ein Ladevorgang der Körperpositionen wünschenswert, welcher coalesced ausgeführt wird. Da die Sortierung des Körper-ID/Zellen-ID Arrays jedoch nicht direkt von den Körper-IDs abhängt, würden nach aufsteigenden Körper-IDs geladene Positionen im Gegenzug Schreibvorgänge, welche uncoalesced geschehen, zur Folge haben und den scheinbaren Vorteil damit aufheben.

### 6.3.2 Indizierung Performance

Jedes Work-Item speichert zwei Zellen-IDs in zwei privaten 32 Bit Ganzzahlvariablen, die seiner eigenen und der vorigen Position im Zellen-ID Array entsprechen. Die eigene globale Work-Item Nummer erfordert eine weitere private 32 Bit Variable. Insgesamt ergeben sich 96 Bit privater Speicherplatz pro Work-Item. Bei 64 kB privatem Speicher pro Vektorprozessor also 4*64 kB pro Prozessoreinheit der GCN Architektur sind selbst bei maximaler Work-Group Größe von 256 Work-

Items 256*96 Bit = 3 kB per Work-Group nötig. Somit entsprächen 64 kB dem gleichzeitigen privaten Speicherbedarf von 21 Work-Groups. Bei einer Wavefront Größe von 64 also 4 Wavefronts pro Work-Group ist theoretisch ein Wechsel des Ausführungskontexts zwischen 84 Wavefronts ohne nennenswerten Zeitverlust möglich. Um die nötigen Werte zu überprüfen, werden die beiden benötigten Arrays für Zellen-IDs und Indizierung als Zeiger auf den globalen Speicherbereich an den Kernel übergeben. Das Laden der beiden Zellen-IDs erfolgt nacheinander, wobei paralleles Laden der ersten Zellen-ID an der eigenen Position jedes Work-Items coalesced erfolgt. Benachbarte Work-Items laden auch benachbarte Positionen des Zellen-ID Arrays. Nach dieser Operation wird auch die vorherige Speicherposition von jedem Work-Item benötigt. Diese Position wird vorrausichtlich schneller geladen, falls sie durch ein benachbartes Work-Item bereits privat gespeichert und somit im Cache der entsprechenden Prozessoreinheit vorhanden ist. Das erste Element einer Work-Group hat keine Möglichkeit, gemeinsam mit einem Work-Item der benachbarten Work-Group Daten innerhalb der Prozessoreinheit lokal zu teilen. Einmaliges lokales Laden jeder Zellen-ID ist daher zur Bestimmung der Zellgrenzen nicht ausreichend.

Eine zusätzliche Fallunterscheidung für die letzte Position der Zellen-ID ist nötig, da kein anderes Work-Item mit höherer globaler Nummer existiert, das die Endposition für diese Zelle bestimmt. Diese zusätzliche Operation zum Schreiben der letzten Zellengrenze wird ausschließlich vom letzten zugewiesenen Work-Item durchgeführt, es entsteht also keine echte Thread-Divergenz. Es muss lediglich die zusätzliche Prozessorzeit dieser Operation veranschlagt werden. Die übrigen Work-Items führen in dieser Zeit (keine den Programmverlauf beeinflussende) Operation aus. Auch für den Fall, dass die beiden verglichenen Zellen-IDs eines Work-Items übereinstimmen, bleibt das entsprechende Work-Item unbeschäftigt, während andere Work-Items des selben Wavefront die eventuell nötigen Operationen ausführen. Bei Zuordnung jeder Position des Zellen-ID Arrays an ein Work-Item ist dies unumgänglich. Feiner ist das Problem der Auffindung der Zellgrenzen mit dieser Datenstruktur nicht zu parallelisieren.

### 6.3.3 Paarliste Performance

Der private Speicherbereich des Kernels zur Erzeugung der Paarliste besteht aus:

2 32 Bit Ganzzahlvariablen als Schleifenzähler und Zähler der Interaktionspartner
64 Bit Gleitkommavektor für die Position des Fußgängers
32 Bit Ganzzahl der Position im Körper-ID Array
2 32 Bit Ganzzahlen für Work-Group Nummer und lokale Work-Item Nummer

2 32 Bit Ganzzahlen für potentielle Interaktionspartner der eigenen Zelle, anderer Zellen
64 Bit Ganzzahlvektor für Start und Endposition der Zelle im Körper-ID Array

Insgesamt werden mindestens 352 Bit privater Speicher für jedes Work-Item benötigt. Zur korrekten Erzeugung der Paarliste muss die Anzahl an Work-Items pro Work-Group die Anzahl der maximal in eine Zelle passenden Fußgänger wiedergeben bzw. übersteigen. Diese Zahl soll jedoch möglichst klein bemessen sein, da zusätzliche Work-Items keine zugewiesenen Fußgänger haben bzw. vergleichen und sich somit keine Laufzeitverringerung ergeben kann. Bei einer in der Simulation verwendeten Zellbreite von bis zu maximal 10 m wird eine Work-Group Größe von 128 benötigt. Daher werden 128*352 Bit also ungefähr 6 kB pro Work-Group, wie bereits bei Berechnung des Speicherbedarfs des Binning-Kernels, veranschlagt. Die Anzahl der insgesamt vorhandenen Work-Groups entspricht der Anzahl der Zellen. Dünn besetzte Zellen in welchen sich lediglich ein paar Fußgänger aufhalten verschwenden daher die Kapazität der meisten Work-Items, die für diese Zelle zuständig sind. Da jedoch pro vorhandenem Körper nur eine Abstandsberechnung nötig ist, verhält sich die Auslastung der Recheneinheiten direkt proportional zum Rechenaufwand der Abstandberechnungen der Zelle.

Der Vorteil der zellweisen Aufteilung der Work-Group ist die Möglichkeit der Nutzung des gemeinsamen lokalen Speichers der Prozessoreinheit. Das Laden der Körper-IDs und zugehörigen Fußgängerpositionen sowie der zur Zellnachbarbestimmung nötigen Rechenoperationen kann synchron von allen Work-Items einer Zelle übernommen werden.

Befinden sich die Daten im lokalen Speicher, können alle Abstandberechnungen dieser Zelle ohne Zugriff auf den Hauptspeicher der GPU durchgeführt werden. Lediglich der Eintrag einer Körper-ID in die entsprechende Paarliste fordert einen Schreibzugriff auf den globalen Speicher. Um diese unzusammenhängenden Schreibzugriffe zu vermeiden, wäre eine lokale Kopie der Paarliste jedes Fußgängers eines Work-Items auf der Prozessoreinheit nötig, um die gesamte Paarliste am Ende der Kernelfunktion coalesced zurückzuschreiben. Die Größe der Paarliste eines Fußgängers würde bei einem maximalen Interaktionsradius von 10 m, im Fall der dichtesten Flächenpackung, fast 200 Interaktionspartner ergeben. Jedes Work-Item würde somit einen lokalen Speicherplatz von 200*32 Bit = 0,8 kB beanspruchen. Bei voll besetzten Zellen muss dieser Speicherbedarf für jedes Work-Item gedeckt werden und der lokale Speicher der Recheneinheit wäre mit nur einer Work-Group überfüllt. Der tatsächlich benötigte lokale Speicher für alle Körper-IDs und Positionen einer Zelle, also eine 32 Bit Ganzzahl und ein 64 Bit Gleitkommavektor für jeden potentiellen Fußgänger in der Zelle, beträgt 128*(64+32) Bit = 1,5 kB. Dies stellt in Anbetracht von benötigten 6 kB privatem Speicher pro Work-

Group keine Limitation für die Prozessoreinheit dar welche über mehr als ¼ des privaten Speichers an lokalem Speicher besitzt.

### 6.3.4 Kraftberechnung Performance

Folgende private Variablen des Kernels zur Kraftberechnung werden benötigt:

4 32 Bit Ganzzahlen für Schleifenzähler, Anzahl der Interaktionspartner und geladener Körper-IDs
5 32 Bit Gleitkommazahlen für voraussichtliche Kollisionszeit und Zwischenwerte
6 64 Bit Gleitkommavektoren für geladene Position, Geschwindigkeiten, deren Relativwerte sowie Kraftvektoren
32 Bit Wert der globalen Work-Item Nummerierung

Es ergeben sich somit mindestens 704 Bit für den privaten Speicherbedarf eines Work-Items. Für den Kernel sind keine lokalen Speicherbereiche der Prozessoreinheit nötig, die übergreifend zwischen Work-Items genutzt werden. Jeder Fußgänger besitzt seine eigene Paarliste und greift auf die darin gespeicherten Werte zu. Allerdings ist es sehr wahrscheinlich das benachbarte Work-Items auf ähnliche Positionsdaten zugreifen, da im Körper-ID Array räumlich nahe Fußgänger durch den Sortiervorgang in nahen Zellen gespeichert werden und benachbarte Work-Items auf benachbarte Positionen des Körper-ID Arrays zugreifen. Die für die Interaktionsauswertung benötigten Positionen sind daher mit höherer Wahrscheinlichkeit im Cache zu finden, als dies zwischen weit entfernten Fußgängern der Fall wäre. Für die genaue Abfolge der Ladeoperationen und Ähnlichkeit der Paarlisten von Fußgängern benachbarter Work-Items besteht jedoch keine Garantie. Unterscheidet sich beispielsweise die Geschwindigkeitsrichtung zweier Fußgänger die sich räumlich direkt nebeneinander befinden, kann die Durchschnittsmenge der Interaktionspartner durchaus leer sein. Bewegen sie sich hingegen auf parallelen Pfaden so stimmen die in den beiden Paarlisten abgelegten Werten eher überein.

Alle benötigten Werte werden in den privaten Speicher der Work-Items geladen, wobei die nach räumlichen Zellen sortierte Anordnung der Körper-IDs im globalen Körper-ID Array nach der Sortierungsphase Cache-hits beim Laden der Positionen der Interaktionspartner wahrscheinlicher macht. Das Laden der Positionen und Geschwindigkeiten sämtlicher Interaktionspartner ist dennoch uncoalesced und die potentiell langsamste Ladeoperation aus dem globalen Speicher. Die große Anzahl an arithmetischen Operationen, die zur Berechnung der von einem Interaktionspartner ausgehenden Kraft nötig sind, ergeben jedoch ein gutes Verhältnis von Ladezeit zu, von der

Ladeoperation abhängiger, aufgewendeter Prozessorzeit.

Folgende arithmetische Operationen werden zur Gesamtkraftberechnung benötigt: Für die Berechnung einer paarweisen Kraft zwischen zwei Fußgängern sind mit Auswertung aller auftretenden Relativgrößen, Skalarprodukte und Bestimmung der voraussichtlichen Kollisionszeit 17 Additionen/Subtraktionen und 20 Multiplikationen/Divisionen nötig. Weiters muss die OpenCL Wurzelfunktion sqrt() zweimal, die Potenzfunktion powr() und Exponentialfunktion exp() jeweils einmal aufgerufen werden. Die Anzahl der Einträge der Paarliste entspricht der Anzahl der durchgeführten Schleifendurchläufe. Nach Laden des Paarlisteneintrags, der Position und Geschwindigkeit ergibt sich ein Verhältnis von über 40 arithmetischen Operationen zu 3 Ladeoperationen pro Interaktionsberechnung.

Die Berechnung der jeweils von statischen Objekten ausgehende Kraft erfordert pro Objekt 35 Additionen/Subtraktionen sowie 36 Multiplikationen/Divisionen, zusätzlich muss die OpenCL Wurzelfunktion sqrt() zweimal und die Potenzfunktion powr() und Exponentialfunktion exp() wieder jeweils einmal aufgerufen werden. Nach dem Ladevorgang des jeweiligen Objekteintrags ergibt sich ein Verhältnis von über 71 arithmetischen Operationen zu 2 Ladeoperationen pro Interaktionsberechnung mit einem Objekt. Zur Auswertung der treibenden Kraft und Limitation der Gesamtkraft benötigt man 4 Additionen/Subtraktionen, 6 Multiplikationen/Divisionen sowie eine Wurzelauswertung sqrt(). Zur Aktualisierung der Position und Geschwindigkeit bleiben 2 Additionen/Subtraktionen und 2 Multiplikationen/Divisionen.

## 6.4 Laufzeitvergleich

Zum Vergleich der Beschleunigung des vorgestellten OpenCL Programms gegenüber der seriellen Ausführung auf der Host-CPU wird das Kreisszenario gewählt. Die Anzahl der Kraftberechnungen für dieses Szenario ist eher ungünstig für die Parallelisierung, da zwar im Mittelpunkt des Kreises viele paarweise Kraftberechnungen auftreten, je weiter man sich jedoch vom Kreiszentrum entfernt, umso dünner besetzt sind die Paarlisten der einzelnen Körper und der zusätzliche Aufwand der Simulationsphasen eines Schrittes wird unnötig betrieben. Gerade deshalb wird dieses Szenario zum Laufzeitvergleich herangezogen, um zu prüfen ob das Programm in der Lage eine gute Beschleunigung gegenüber dem seriellen Code auch bei ungünstigen Bedingungen zu erzielen.

Im Kreisszenario stehen sämtliche simulierte Fußgänger in gleichmäßigen Abständen am Umkreis eines Kreises mit 200 m Durchmesser und bewegen sich durch den Mittelpunkt des Kreises auf den gegenüberliegenden Umkreispunkt zu.
Dieses Szenario ist wohl in der Wirklichkeit kaum zu beobachten, stellt aber für den Vergleich der Laufzeit, aufgrund der kurzen Begegnungen der Fußgänger um den Mittelpunkt des Kreises, ein wertvolles Szenario dar. Abb. 27 zeigt den Laufzeitvergleich des Szenarios der seriellen Version, ausgeführt durch den Intel i7-870 Prozessor, verglichen mit dem entwickelten OpenCL Programm, ausgeführt durch die AMD R7 260X GPU.

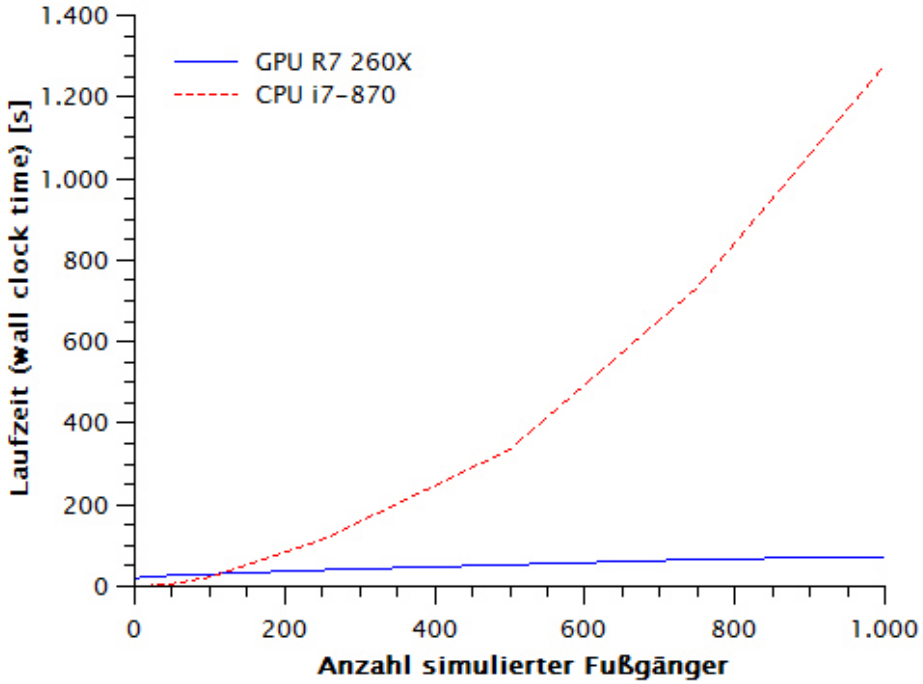


***Abbildung 27*** *Laufzeitvergleich GPU-CPU*

Das serielle Programm wird nur auf einem Prozessorkern ausgeführt und terminiert bis zu einer Fußgängeranzahl von 112 Fußgängern schneller als die OpenCL Version. Für Simulationen mit mehr als 112 Personen übertrifft das vorgestellte OpenCL Programm die serielle Version wie erwartet.

Die Zeitmessung für sämtliche hier angeführte Laufzeiten beinhaltet alle Programmschritte, inklusive Programmende und Abschluss sämtlicher Simulationsergebnisse, sowie deren Speicherung auf der Festplatte. Die gemessene Laufzeit beinhaltet jedoch nicht die Zeit, die zum Kompilieren der OpenCL Kernel vor Programmstart benötigt wird. Das nicht mitberücksichtigte Kompilieren der verwendeten Kernel vor Start des OpenCL Programms benötigt jedoch jedenfalls weniger als eine Sekunde.

Gemessen wird die tatsächlich verstrichene Echtzeit (wall clock time) von Programmstart bis Programmende. Es handelt sich also nicht lediglich um die reine Rechenzeit der GPU, sondern inkludiert auch die für alle Kopiervorgänge der Daten (aus dem Speicher der GPU in den Arbeitsspeicher des Rechners) benötigte Zeit. Somit ist sichergestellt, dass die hier angeführten Werte nicht für tatsächliche Anwendungen irrelevante Benchmarks einzelner paralleler Kernel auf der GPU, sondern die tatsächlichen Laufzeiten des gesamten Programms darstellen.

Bei 100 simulierten Körpern benötigt die GPU 29,9 s während die CPU nach 22,345 s das Simulationsende erreicht. Durch die flache lineare Abhängigkeit der Laufzeit von der Fußgängeranzahl auf der GPU übernimmt diese jedoch bei größeren Körperzahlen die Oberhand. Bei 250 simulierten Körpern benötigt die GPU lediglich 40 s, während die Ausführung der seriellen Version 114 s benötigt. Dies entspricht also einer Beschleunigung um den Faktor 2,9. Ab 1000 Fußgängern ergibt sich sogar eine um mehr als den Faktor 10 verringerte Laufzeit der OpenCL Version. Der serielle Code benötigt bei 30.000 Simulationsschritten mehr als 21 Minuten während der GPU Code die Berechnung des Szenarios nach 73 s abschließt. Es wird also ein Speedup um den Faktor 17,3 erzielt. Bei Szenarien mit großen Menschenansammlungen ist also je nach Interaktionsdichte eine noch höherer Speedup durch das OpenCL Programm zu erwarten. Die Beschleunigungen des OpenCL Programms gegenüber dem seriellen Programm übertrifft diesen Speedup in den im folgenden vorgestellten Beispielszenarien um ein Vielfaches.

Ein Vergleich der Laufzeit des OpenCL Programm mit demselben Programm ohne die in den vorigen Kapiteln vorgestellten Optimierungen wäre wünschenswert. Genau diese Optimierungen bestimmen jedoch den verwendeten Algorithmus und die Umsortierung der Werte sowie das Speicherschema der Positionsdaten und der entsprechenden Datenstrukturen. Dadurch wird das verwendete Berechnungsschema erst möglich. Somit wäre ein fundamental anderer Algorithmus mit ineffizienter Auslastung der Rechenkerne der GPU und ein grundverschiedenes OpenCL Programm zum Vergleich nötig. Die Sinnhaftigkeit des Erstellens und des Vergleichs eines solchen Programms mit der optimierten Version erscheint fragwürdig und wird hier nicht durchgeführt.

# 7 Diskussion der berechneten Beispielszenarien und Ergebnisse

Die berechneten Trajektorien jedes Teilchens werden auf der Festplatte als .xyz-Positionsfile zu den diskreten Zeitschritten gespeichert und werden mit dem Visualisierungsprogramm VMD ausgelesen und visualisiert **[17]**. Abb 28 zeigt eine Momentaufnahme der Visualisierung des Fluchtszenarios.

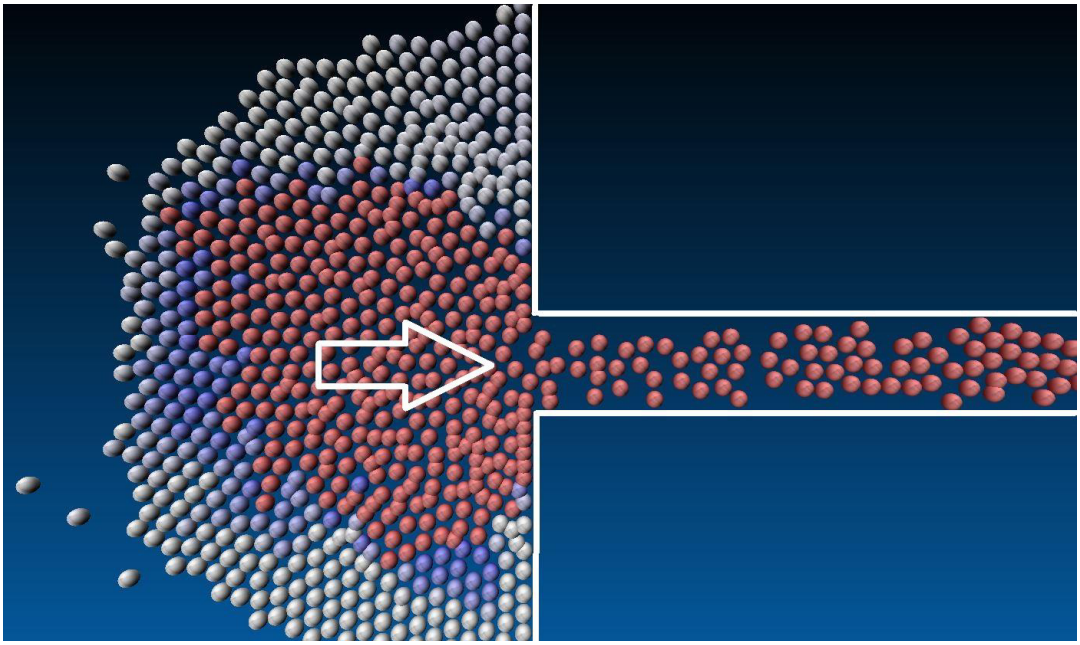

**Abbildung 28** *VMD Visualisierung der Trajektorien des Fluchtszenarios*

## 7.1 Fluchtszenario

Bei Simulation des Fluchtszenarios werden Personen annähernd gleichverteilt paarweisem Abstand in einem quadratischen Raum mit 64 m Seitenlange angeordnet und versuchen dann gleichzeitig durch einen schmalen Fluchtweg mit 2 m Breite den Raum zu verlassen.

Selbst bei 1000 simulierten Fußgängern ergibt sich eine Laufzeit des Programms von unter 40 Sekunden. Die serielle Version benötigt aufgrund der hohen Anzahl an Interaktionen über 30 min zur Berechnung aller Fußgängerpfade. Dies entspricht einem beeindruckenden Speedup von 45.

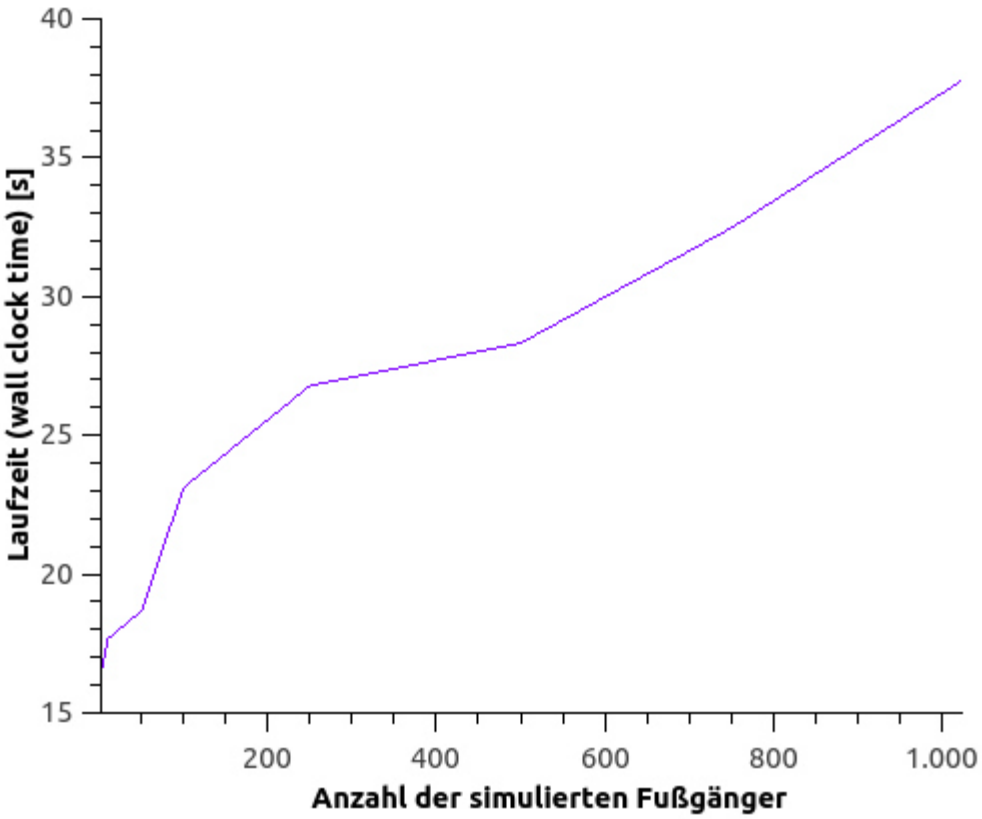


**Abbildung 29** *Laufzeit des Fluchtszenarios*

Die Gesamtanzahl der Simulationsschritte die nötig sind, um das gesamte Szenario durchzuspielen und den Raum zu evakuieren hängt von der Fußgängeranzahl und initialen Verteilung der Personen in dem Raum ab. Bei gleichmäßiger Personendichte in dem Raum ergibt sich die in Abbildung 29 gezeigte Beziehung für die Laufzeit. Hierbei entspricht ein Zeitschritt einer Dauer von 0,006 s. Die Anzahl der Simulationszeitschritte in Abb. 30 multipliziert mit der Zeitschrittweite ergibt die Zeit die alle Personen brauchen, um den Raum zu verlassen. 10.000 simulierte Zeitschritte bei einer verwendeten Zeitschrittweite von 0,006 s entsprechen also einer Evakuierungszeit des 64 m breiten quadratischen Raumes von 1 Minute.

Kommt es jedoch aufgrund der hohen Personendichte am Ausgang des Raumes zu Massenpanik, gilt das benutzte Kraftgesetz nicht mehr und es kann zu weit größeren Kräften als in der Simulation berücksichtigt kommen. Die Simulation kann durch Analyse der Trajektorien das Entstehen von Bereichen hoher Dichten vorhersagen, das tatsächliche resultierende Verhalten in diesen Zonen extremer Personendichte wird vom Interaktionsgesetz jedoch nicht zuverlässig vorausgesagt **[2]**.

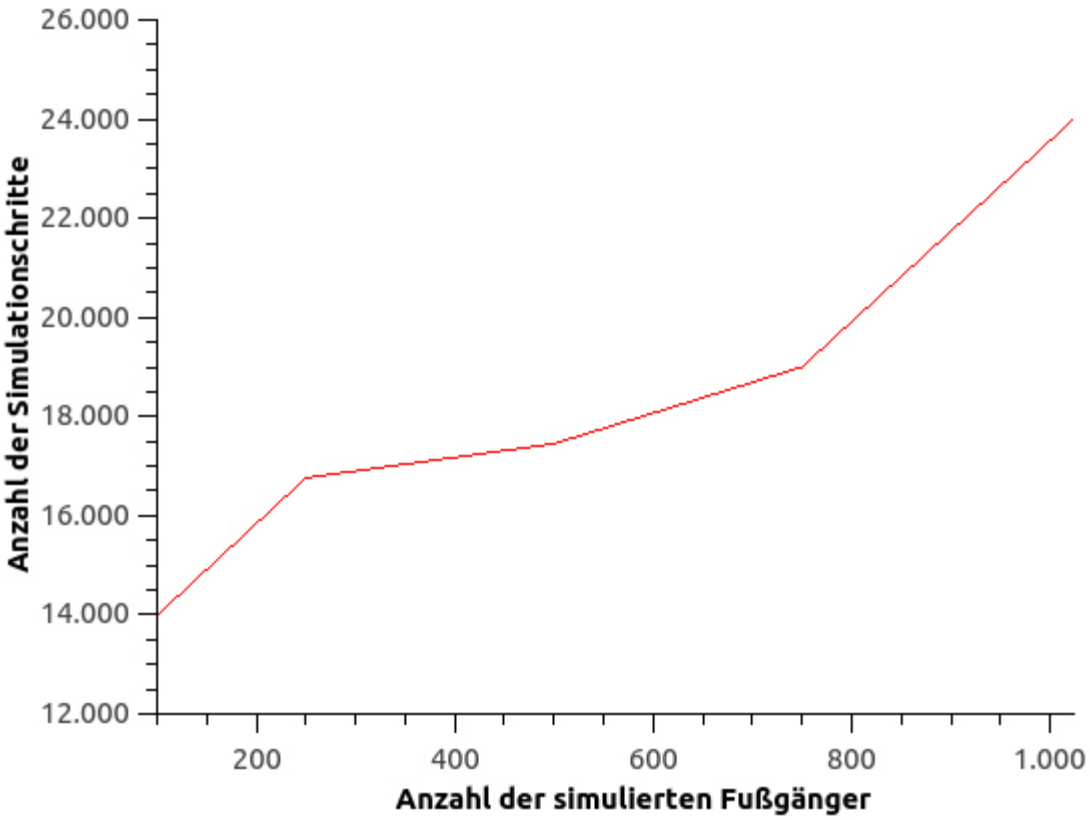


**Abbildung 30** *Simulationsschritte in Abhängigkeit der Fußgängeranzahl des Fluchtszenarios*

## 7.2 Bühnenszenario

Ähnlich dem Fluchtszenario drängt beim Bühnenszenario eine große Anzahl von Menschen an eine Bühne. Natürlich sind die Personen bestrebt möglichst nahe zum Mittelpunkt der Bühne zu gelangen. Bei tatsächlicher Kollision mit einer anderen Person wird jedoch keine weitere Kraft in Kollisionsrichtung berücksichtigt da davon ausgegangen wird das die Menschen an der Stelle der Kollision verharren. Erst, wenn sich ein neuer Platz näher an der Bühne ergibt findet eine Bewegung in diese Richtung statt. Die Berechnung des Szenarios endet, wenn sich ein dem relativen Abstand der Bühnenbesucher nach mittlerer konstanter Wert einstellt, also dem Zustand der während der Bühnendarstellung vorliegt. Dies bedeutet kleinere Bewegungen in der Menschenmenge vor der Bühne, aber keinen großen Menschenstrom zur Bühne bzw. davon weg. Der tatsächliche Interaktionsradius also die Berechnung der voraussichtlichen Kollisionszeit ist dabei auf die nähere Umgebung mit einem Radius von 0,2 m beschränkt. Es sind aufgrund der Kreisgeometrie der Modelpersonen maximal 6 kollidierende Personen um einen Fußgänger möglich.

Wie in Abb. 31 dargestellt ergibt sich bei einer Anzahl von 500 Personen eine mittlere Laufzeit von unter 13 Sekunden. Bei 1000 simulierten Personen beträgt die Laufzeit unter 16 Sekunden (bis sich ein Gleichgewicht der Personenverteilung einstellt). Die benötigten Simulationszeiten liegen weit unter der tatsächlichen Zeit die die Menschenmenge braucht um sich vor der Bühne zu versammeln, wird also dem Anspruch einer Berechnung in Echtzeit gerecht.

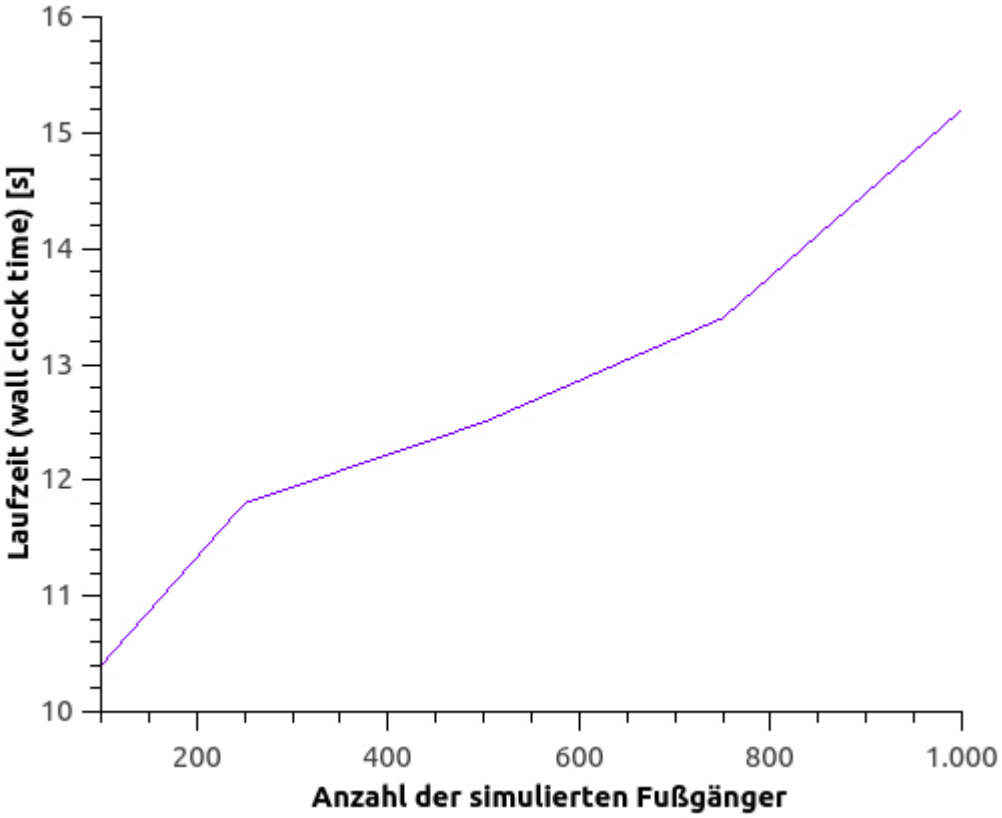


**Abbildung 31** *Laufzeit des Bühnenszenarios*

## 7.3 Korridorszenario

Beim Korridorszenario bewegen sich hunderte Personen auf einem Gehsteig oder in einem Korridor in beide mögliche Richtungen, ihre Wege kreuzen sich dabei permanent. Zur Simulation starten zwei Menschengruppen in entgegengesetzten Richtungen und gehen im 10 m breiten Korridor aneinander vorbei. Haben alle Personen das entgegengesetzte Ende des Korridors erreicht endet die Simulation. Die Länge des simulierten Korridors beträgt 120 m. Wieder beträgt die zur Simulation benutzte Zeitschrittweite 0.01 s. Auf einem echten, hochfrequentierten Gehweg würden aus beiden Richtungen permanent Menschen nachströmen. Dies kann durch periodische Randbedingungen des Korridors

erreicht werden, sobald also eine Person das Simulationsgebiet (den Korridor) verlässt, erscheint sie an derselben relativen Position am anderen Ende des Korridors um erneut an der Simulation teilzunehmen. So ließe sich beispielsweise eine hochfrequentierte Einkaufsstraße simulieren.

Abb. 32 zeigt die Laufzeit des Programms in Abhängigkeit der Anzahl der simulierten Personen. Die Simulation terminiert bei einer Fußgängerzahl von 500 nach unter 34 Sekunden. Bei 1000 simulierten Personen beträgt die mittlere Laufzeit nur beachtliche 44 Sekunden. Die Interpretation der auftretende maximale Personendichte zu jedem Zeitpunkt und die benötigte Zeit jeder Person zur Erreichung ihres individuellen Zieles durch den Korridor ist nicht Gegenstand dieser Arbeit. Alle dafür nötigen Positionsdaten und Trajektorien werden jedoch vom Simulationsprogramm berechnet und gespeichert.

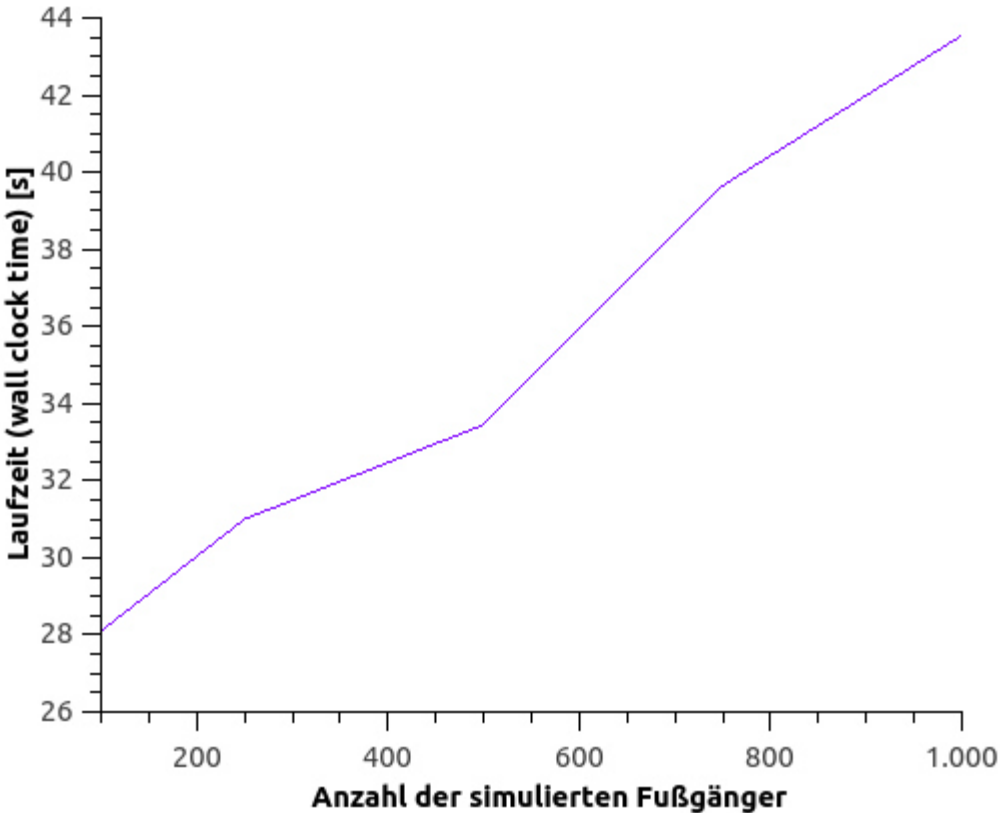


**Abbildung 32** *Laufzeit des Korridorszenarios*

## 7.5 Güte der Simulationsdaten

Als Maß für die Güte der von der Simulation produzierten Positionsdaten wird der Betrag der Abweichung jedes Fußgängers vom erwarteten Pfad für Korridor-, Bühnen- und Fluchtszenario über den Simulationszeitraum gemittelt. Der erwartete Pfad entspricht dabei den von Karamouzas, Skinner

und Guy **[2]** angegebenen Positionswerten. Die Abweichung steigt zwar durch numerische Fehler des Integrationsverfahrens und Genauigkeitsgrenzen der Berechnungen in single precision (32 Bit) im Laufe der Simulation an, übersteigt jedoch nie 0,5 m. Abb 33. zeigt den Verlauf der Abweichung während der Simulation.

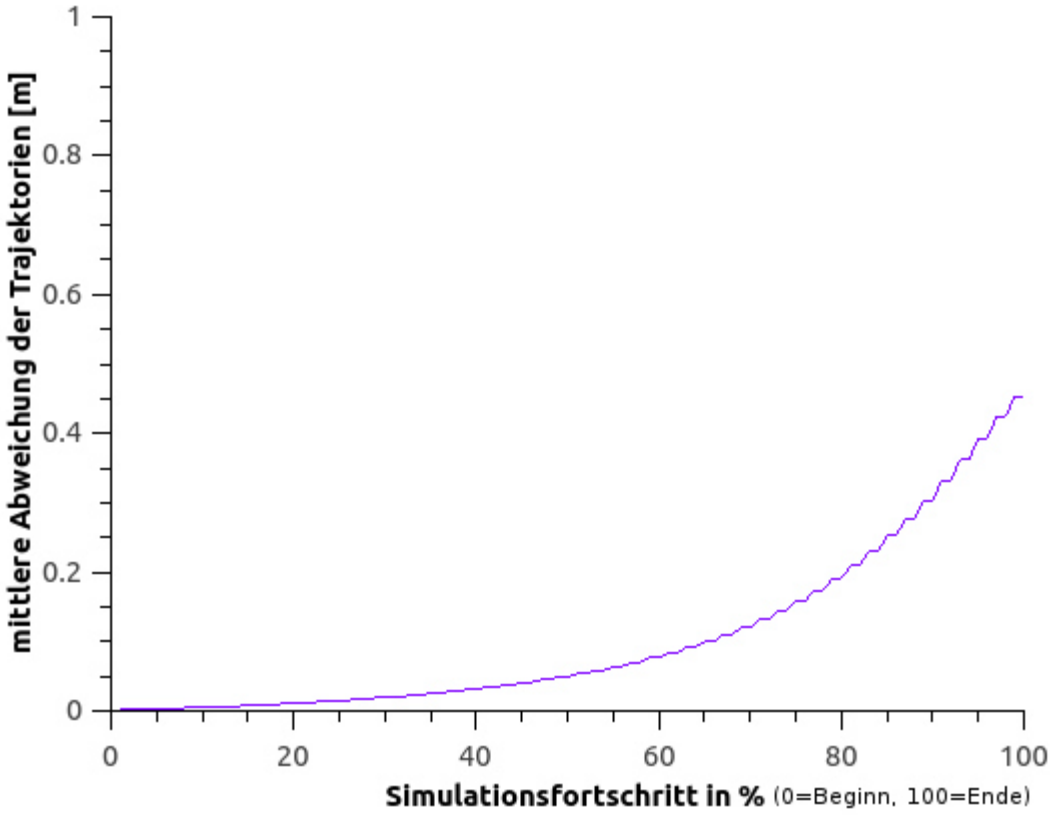


**Abbildung 33** *mittlere Pfadabweichung der Fußgänger von den erwarteten Simulationsdaten*

Dem Anspruch einer Bestimmung der Pfade der Körperschwerpunkte aller Fußgänger mit Abweichungen von wenigen Zentimetern werden Simulationen über längere Zeiträume nicht gerecht. Dies spielt jedoch aufgrund des zu Beginn der Arbeit beschrieben statistischen Charakters sämtlicher Fußgängersimulationen keine Rolle, solange die Positionsdaten nicht um ein Vielfaches der Körperdimensionen einer Person abweichen.

# 8 Zusammenfassung

Das erreichte Ziel dieser Arbeit war die Entwicklung eines geeigneten massiv parallelen Programms, welches die kurzreichweitige Wechselwirkung von Fußgängern nach einem vorgegebenen paarweisen Kraftgesetz simuliert und die sich ergebenden Bewegungspfade möglichst in Echtzeit vorausberechnet. Die Implementierung des vorgestellten Algorithmus und Ausführung auf der GPU (AMD R7 260X) resultiert, verglichen mit der seriellen Programmversion auf der verwendeten CPU (Intel i7-870), bei 1000 simulierten Fußgängern selbst bei ungünstigsten Bedingungen in einem beeindruckendem Speedup um den Faktor 17. Weit höhere Beschleunigungen treten bei größeren und dichteren Menschenmassen auf. Sämtliche simulierten Beispielszenarios sind in einem Bruchteil der tatsächlichen Ablaufzeit des Szenarios berechnet, eine Simulation in Echtzeit ist somit jedenfalls gewährleistet. Der Speedup der parallelen GPU Version sowie die erreichten Laufzeiten übertreffen die Erwartungen.

Die Güte der simulierten Daten, also die Übereinstimmung mit empirischem Datenmaterial, ist abhängig vom verwendeten Kraftgesetz und sorgfältig gewählten Parametern für das entsprechende Szenario. Das in dieser Arbeit verwendete Interaktionspotential und dessen Gültigkeit bzw. Korrelation mit dem bekannten empirischen Datenmaterial der verschiedenen repräsentativen Szenarien, wie der Flucht in einen engen Gang oder der Massendynamik auf freien Flächen ist bereits nachgewiesen **[2]**. Die Simulationsdaten können die im empirischen Datenmaterial zu beobachtenden Korrelationen und Verhaltensmuster der dynamischen Entwicklung der Fußgänger richtig wiedergeben. Evakuierungszeiten, gefährliche Fußgängerdichten und Engpässe, sowie die Geschwindigkeiten der Fußgänger und das Erreichen derer Ziele sind attraktive Ergebnisse der Simulation des Interaktionspotential.

Vergleich der Ergebnisse der Simulation mit den von Karamouzas, Skinner und Guy **[2]** angegebenen Positionswerten ergibt eine mehr als zufriedenstellende Übereinstimmung. Die simulierten Trajektorien stimmen nahezu überein. Die Abweichung der Körper von den Simulationspfaden steigen zwar im Laufe der Simulation an, beträgt aber im Durchschnitt nie mehr als 0,5 m, wie in Abb. 33 dargestellt.

Es existiert bisher keine bekannte Implementierung einer Fußgängersimulation auf Grafik-prozessoren in OpenCL. Der vorgestellte Algorithmus ist eine stark modifizierte Version der nach Wang **[11]** beschriebenen Methode zur Simulation der Molekulardynamik mit kurzreichweitigen Kräfte. Die durchgeführten Fußgängersimulationen liefern mehr als

zufriedenstellende Ergebnisse, dass entwickelte Programm ist somit eine attraktive Möglichkeit für zukünftige Simulationen mit ähnlichen Anforderungen.

Die in dieser Arbeit verwendete Hardware entspricht leistungstechnisch nicht dem derzeit verfügbaren Stand, aktuelle Grafikprozessoren wie die GTX 1070 von NVIDIA erzielen bei derartigen Simulationen aufgrund der hohen Geschwindigkeit und Anzahl der verfügbaren Rechenkerne sowie höherer Speicherbandbreiten ein Vielfaches an Datendurchsatz. Simulationen von mehreren 100.000 Fußgängern sind in Echtzeit möglich, der Bedarf der Simulation von solch großen Menschenmengen ist jedoch fragwürdig. Die in dieser Arbeit durchgeführten Simulationen sind selbst mit heutigen Heimcomputern um wenige hundert Euro in Echtzeit durchführbar. Der tatsächliche Aufwand für die Durchführung eines bestimmten Szenarios ist gering, lediglich die Simulationsumgebung und die Startpositionen und Ziele der einzelnen Personen sind anzugeben.

Auch wenn die Simulation in dieser Arbeit exklusiv für ein einziges Interaktionspotential und Kraftgesetz durchgeführt ist, soll die entwickelte Simulationssoftware nicht auf dieses eine Kraftgesetz beschränkt betrachtet werden. Jede paarweise arithmetisch berechenbare Kraft kann für die Durchführung der Simulation mit dem OpenCL Programm verwendet werden und wird, je nach Anzahl der benötigten Rechenoperationen des Kraftgesetzes, verringerte Laufzeit gegenüber einer seriellen Implementierung auf derzeit verfügbaren CPUs ermöglichen.

Die Simulation der Bewegung großer Menschenmassen in verhältnismäßig kurzer Zeit ermöglicht die Erhöhung von Verkehrssicherheit durch gewissenhafte Planung der Verkehrswege, die Voraussage kritischer Gebiete bei großen Veranstaltung, die Planung von sicheren Räumlichkeiten und Fluchtwegen in Gebäuden und nicht zuletzt die realistische Animation von Menschenmengen in Echtzeit. Das in dieser Arbeit entwickelte OpenCL Programm ist somit für sämtliche diese Aufgaben betreffenden Anwendungen von Wert.

# Literaturverzeichnis

**[1]** L. F. Henderson - The Statistics of Crowd Fluids, Nature 229, 381 - 383 (1971)

**[2]** I. Karamouzas, B. Skinner, S. J. Guy - A universal power law governing pedestrian interactions, Phys. Rev. Lett. 113, 238701 (2014)

**[3]** D. Helbing, I. Farkas, and T. Vicsek - Simulating dynamical features of escape panic, Nature 407, 487-490 (2000)

**[4]** AMD Reference Guide, Sea Islands Series Instruction Set Architecture, AMD (2013)

**[5]** R. Meyer - Efficient Parallelization of Short-Range Molecular Dynamics Simulations on Many-Core Systems, Phys. Rev. E 88, 053309 (2013)

**[6]** R. Yokota, L. A. Barba - Fast N-body Simulations on GPUs, CoRR, abs/1108.5815 (2011)

**[7]** J. Dubinski - A Parallel Tree Code, Elsevier New Astronomy Volume 1, Issue 2, 133-147 (1996)

**[8]** L. Verlet - Computer 'experiments' on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules, Phys. Rev. 159, 98-103 (1967)

**[9]** S. Plimpton - Fast Parallel Algorithms for Short–Range Molecular Dynamics, J. Comput. Phys. 117, 1-19 (1995)

**[10]** C. Ericson - Real-Time Collision Detection, The Morgan Kaufmann Series in Interactive 3-D Technology, Elsevier (2004)

**[11]** W. Brown, P. Wang, S. J. Plimpton, A. N. Tharrington – Implementing Molecular Dynamics on Hybrid High Performance Computers – Short Range Forces, Elsevier Computer Physics Communications 183, 449-459 (2012)

**[12]** J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, S. C. Glotzer - Strong scaling of general-purpose molecular dynamics simulations on GPUs, Elsevier Computer Physics Communications 192, 97-107 (2015)

**[13]** C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, W. W. Hwu - GPU Acceleration of Cutoff Pair Potentials for Molecular bin sizes, Conference Paper, Proceedings of the 5th Conference on Computing Frontiers, 273-282 (2008)

**[14]** libcl-Documentation, http://www.libcl.org/docu.html (2016)

**[15]** D. P. Playne, K. A. Hawick - Classical Mechanical Hard-Core Particles Simulated in a Rigid Enclosure using Multi-GPU Systems, CSTN Computational Science Technical Note Series, Technical Report CSTN-139 (2012)

**[16]** L. Nyland, M. Harris, J. Prins - Fast N-Body Simulation with CUDA, GPU Gems 3, NVIDIA (2007)

**[17]** VMD-Documentation, http://www.ks.uiuc.edu/Research/vmd/current/docs.html (2016)