technische universität
dortmund

Diplomarbeit

Entwurf einer Softwarebibliothek zur Entwicklung portabler, hardwareorientierter HPC Anwendungen am Beispiel von Strömungssimulationen mit der Lattice Boltzmann Methode

Dirk Ribbrock

4. August 2009

Gutachter:
Prof. Dr. Heinrich Müller
Prof. Dr. Stefan Turek

Fakultät für Informatik
Graphische Systeme (LS VII)
Technische Universität Dortmund
http://ls7-www.cs.tu-dortmund.de

Fakultät für Mathematik
Angewandte Mathematik und Numerik (LS III)
Technische Universität Dortmund
http://www.mathematik.tu-dortmund.de/lsiii/

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In den letzten Jahren zeichnet sich ab, dass die Rechenleistung der einzelnen klassischen CPU nicht mehr stark ansteigt, da sie sehr stark durch Wärmeverlustleistung und andere physikalische und elektrotechnische Grenzen eingeschränkt wird. Gleichzeitig hinkt die Geschwindigkeit des Arbeitsspeichers und seiner Anbindung an die CPU dem Bedarf wissenschaftlicher Rechnens weit hinterher. Ein typisches System, dass eine Fließkommaleistung in einfacher Genauigkeit von 96 GFlop/s [1], aber nur eine Speichertransferrate von 12 GByte/s (das entpricht $3 \cdot 10^9$ Fließkommazahlen in einfacher Genauigkeit pro Sekunden) besitzt, müsste also mit jeder transferierten Fließkommazahl mindestens 32 Berechnungen durchführen, um die theoretische Maximalleistung (*peak performance*) zu erreichen. Dies lässt noch außer Acht, dass das Ergebnis der meisten Berechnungen auch noch in den Arbeitsspeicher zurückgeschrieben werden muß. Um für eine Operation einschätzen zu können, wie stark sie durch dieses Missverhältnis zwischen Rechen- und Speichergeschwindigkeit beeinflusst wird, eignet sich die arithmetische Intensität, die das Verhältnis von durchgeführten Fließkommaoperationen zu notwendigen Speichertransfers angibt. Im obigen Beispiel entspricht die arithmetische Intensität $\frac{N_{\text{flop}}}{N_{\text{transfer}}} = \frac{96}{3} = 32$. In der Praxis sind allerdings arithmetische Intensitäten von 1 oder weniger sehr verbreitet. So hat etwa die komponentenweise Summe von zwei Vektoren eine arithmetische Intensität von $\frac{N_{\text{flop}}}{N_{\text{transfer}}} = \frac{1}{3} = 0,33$. Hier ist es nämlich notwendig jeweils beide Operanden zu laden, die Addition durchzuführen und danach das Ergebnis wieder zu speichern. Ihre Ausführungsgeschwindigkeit ist also sehr stark durch die Speichertransferrate und weniger durch die eigentliche Geschwindigkeit der CPU beschränkt. Dieses Problem wird auch als *memory wall* [56] bezeichnet.

Inzwischen gibt es verschiedene Entwicklungen um diesem Problem zu begegnen. So wird bei Multicore CPUs nicht nur die Anzahl der Rechenkerne sondern auch die Anzahl der Speichercontroller erhöht. Gleichzeitig gibt es komplette Neuentwicklungen wie die Cell BE, bei der schon während des Entwurfs auf eine gute Speicheranbindung geachtet wurde. Außerdem sind Grafikkarten, die ursprünglich nur zur Beschleunigung und Berechnung von Computergrafik entworfen wurden, inzwischen in der Lage, auch Berechnungen auszuführen, die früher der CPU vorbehalten waren. Zuletzt verspricht das Vernetzen von

[1] 1 GFlop/s = 10^9 Flop/s (*Floating point operations per second*)

sehr vielen herkömmlichen Rechnern zu einem Cluster eine deutliche Erhöhung der Ausführungsgeschwindigkeit bei sehr großen Problemstellungen, da die theoretische Rechenkapazität deutlich erhöht wird.

Um diese heutige Hardware optimal auszunutzen, muss man bei der Entwicklung von Applikationen auf die speziellen Eigenschaften der Hardware eingehen. Moderne Architekturen wie Cell BE, GPU oder ganze Computercluster haben auf den ersten Blick nicht viele Gemeinsamkeiten. Da man eine Applikation aber nicht für jede Hardware komplett neu schreiben will, wird versucht, durch Basisbibliotheken von der Hardware zu abstrahieren. Allerdings kann üblicherweise die Rechenkapazität jeder Hardware nicht voll ausgenutzt werden, indem man eine neue Applikation auf einer Reihe von gegebenen hardwareoptimierten Basisoperationen aufsetzt (z.B. BLAS [19]). Dieser zunächst naheliegende Ansatz erlaubt es oftmals nicht, die Möglichkeiten der jeweiligen Hardware in Verbindung mit den Eigenarten der speziellen Anwendung voll auszunutzen.

1.2 HONEI LBM

Die in dieser Arbeit vorgestellte Softwarebibliothek namens HONEI [11] versucht eine Lösung für diese Problemstellung zu finden, indem ein Kompromiss zwischen Portabilität und Hardwarenähe erreicht wird.

Die Tauglichkeit von HONEI soll am Beispiel von Strömungssimulationen (*Computational Fluid Dynamics, CFD*) mit der Lattice Boltzmann Methode (LBM) [60] evaluiert werden. Dabei wird die ein vereinfachtes Stömungsmodell, die sogenannten Flachwassergleichungen, verwendet. Hierbei werden nur solche Strömungen betrachtet, bei denen vertikale Effekte bei der Berechnung der Fluidbewegung vernachlässigt werden können, also die Wellenlänge wesentlich größer als die Wellenhöhe ist. Diese Methode erscheint insbesondere deshalb geeignet, da sie sehr lokal auf ihren Daten arbeitet, ein hohes Parallelisierungspotential besitzt und als rechenintensiv gilt, das heißt eine hohe arithmetische Intensität besitzt. Demgegenüber steht der relativ hohe Speicherverbrauch der Lattice Boltzmann Methode, da zur Berechnung eines jeden Elements der Wasseroberfläche sehr viele Geschwindigkeits- und Verteilungsfunktionen gespeichert werden müssen.

Obwohl die verwendeten Flachwassergleichungen durch die genannten Anforderungen scheinbar nur auf wenige Szenarios beschränkt sind, besitzen sie in der Praxis eine Vielzahl von verschiedenen Anwendungen. Insbesondere lassen sich damit nicht nur Wasserströmungen berechnen, sondern alle Phänomene, die ein inkompressibles Fluid beschreiben. Mögliche Anwendungen sind die Vorhersage von Tsunamis oder Sturmfluten und die Simulation von atmosphärischen Strömungen nicht nur auf der Erde sondern auch auf anderen Himmelskörpern. Daneben gibt es viele weitere Anwendungen im Bereich der CFD, wie das Um- bzw. Durchströmen von Fahrzeugen und Geräteteilen bis hin zu Anwendungen in der Unterhaltungsindustrie und sogar der Simulation des Blutflusses etwa durch eine

künstliche Herzklappe. Ausserhalb der CFD kann die LBM zum Beispiel auch zu Simulationen im Bereich der Quantenmechanik [46] verwandt werden. Dazu muss natürlich die in dieser Arbeit verwendete Flachwassergleichung durch eine andere ersetzt werden. Die wesentlichen hier erarbeiteten Eigenschaften und Konzepte behalten aber ihre Gültigkeit.

Schon 1950 berechneten John von Neumann et al. [8] auf dem ENIAC, einem der ersten damaligen Großrechner, eine Wettervorhersage für 24 Stunden. Dazu wurde ein 15×18 Gitter mit einer quadratischen Zellengröße von 736 km und Zeitschritte im Bereich von ein bis drei Stunden verwendet. Die eigentliche Berechnung dauerte mehrere Stunden und konnte im Anschluß mit der tatsächlichen Wetterentwicklung verglichen werden. Die Ergebnisse wichen zwar noch weit von einander ab, zeichneten aber die Richtung der rechnergestützten Wettervorhersage vor. Damals wie heute beschränkt die verfügbare Rechenleistung die erzielbare Genauigkeit der Ergebnisse im wissenschaftlichen Hochleistungsrechnen (*high performance computing, HPC*).

Im Rahmen dieser Arbeit wurde aufbauend auf den Ergebnissen der Projektgruppe SmartCell [42] an der TU Dortmund die Softwarebibliothek HONEI stark erweitert. Insbesondere die Komponenten `Util` und `LA`, sowie die hardwarenahen Bibliotheken für SSE und Multicore wurden wesentlich überarbeitet, erweitert oder komplett neugeschrieben. Dies wurde notwendig um softwaretechnische Einschränkungen, die sich während der ursprünglichen Entwicklung gezeigt hatten, zu beheben. Gleichzeitig war es nur durch wesentliche Veränderungen möglich, insbesondere die neu entwickelte Speicherverwaltung mit den restlichen Komponenten zu verbinden, da diese im ursprünglichen Entwurf der Projektgruppe nicht vorgesehen war. Die gesamte Multicore Unterstützung hatte sich bereits im Laufe der Projektgruppe als nicht performant erwiesen und musste deshalb neu konzipiert werden.

Zusätzlich wurden HONEI folgende neue Komponenten hinzugefügt:

- eine Speicherverwaltung, insbesondere für Systeme mit getrennten Speichern, wie es etwa zwischen Hauptspeicher und Grafikkartenspeicher der Fall ist
- hardwarenahe Bibliotheken für Cluster und Grafikkarten
- zwei eigenständige Lattice Boltzmann Löser
- eine Applikation zur Visualisierung der durch die Lattice Boltzmann Löser berechneten Strömungssimulationen

1.3 Überblick

Kapitel 1 enthält diese Einleitung und eine Zusammenfassung ähnlicher Veröffentlichungen. Kapitel 2 gibt einen Überblick über die in dieser Arbeit betrachteten Hardwarearchitekturen. Aufbauend auf der gewöhnlichen CPU werden SSE, Multicore und Cluster vorgestellt. Außerdem werden Architekturen wie Grafikkarten und die Cell BE erläutert, die zunächst

garnicht für diese Art von Berechnungen entwickelt wurden. Kapitel 3 stellt die Lattice Boltzmann Methode vor, die die Grundlage der in dieser Arbeit entwickelten Löser darstellt. Kapitel 4 beschreibt die HONEI Bibliothek und geht im zweiten Teil besonders auf die in dieser Arbeit neu entworfenen Komponenten ein. Kapitel 5 schließlich stellt den eigentlichen optimierten Lattice Boltzmann Löser vor, der darauf in Kapitel 6 ausführlich auf vielen Architekturen untersucht wird. Kapitel 7 fasst zuletzt die erarbeiteten Ergebnisse und gewonnenen Erkenntnisse zusammen und erörtert offene Fragestellungen.

1.4 Verwandte Arbeiten

Es existieren viele Arbeiten zur Beschleunigung der LBM auf verschiedenen Architekturen.

Die Leistung aktueller Prozessoren von Intel, AMD und IBM mit Blick auf die LBM untersuchen Wellein et al. [54]. Sie vergleichen diese mit alten und neuen dedizierten HPC-Architekturen.

Einen Schritt weiter gehen Zeiser et al. [57] indem sie einen LBM Löser als Benchmarksystem für eben diese Prozessoren einsetzen und über die Leistung des LBM Lösers auf die generelle Leistungsfähigkeit für HPC Anwendungen schließen.

Die Optimierung eines LBM Lösers auf einen x86-64 Prozessor beschreibt im Detail Hausmann [23]. Er stellt insbesondere die besondere Bedeutung von vektorisierten Operationen auch für gewöhnliche x86 Prozessoren heraus, um hohe Leistungen zu erreichen.

Wilke et al. [55] behandeln dem gegenüber insbesondere die Optimierung eines LBM Lösers auf die Speicherhierachie aktueller Prozessorsysteme.

Speziell Multicore Architekturen werden durch Donath [10] untersucht. Hierbei kommt zur Optimierung der Kommunikation ein raumteilender Baum zum Einsatz, wie er normalerweise in der Computergrafik zum schnellen Ausschluss nicht zu rendernder Objekte genutzt wird.

Die Eignung der LBM für stark parallele Großrechner untersuchen Pohl et al. [44] und Wellein et al. [53]. Erstere untersuchen neben der Skalierbarkeit auch die wirtschaftlichen und umwelttechnischen Kosten der einzelnen Architekturen in Hinblick auf die Leistungsfähigkeit der verwendeten LBM Löser. Letztere versuchen einen möglichst kompletten Überblick über alle zur Zeit aktuellen Großrechner zu geben und deren jeweilige Eignung für die LBM einzuschätzen.

Die direkte Berechnung der LBM durch Grafikkarten mittels der eigentlich zur Grafikberechnung vorgesehenen Programmierschnittstellen beschreiben Li et al. [34]. In erster Linie wird hier die generelle Vorgehensweise beschrieben, um Berechnungen, die nicht der Grafikdarstellung dienen, auf einer Grafikkarte durchzuführen. Es werden eine Reihe von Techniken vorgestellt, um auf Basis von Pixel- und Texturoperationen Berechnungen durchzuführen, die zur Lösung der LBM notwendig sind und die notwendigen Datenstrukturen als Texturen zu repräsentieren.

Speziell die Implementierung eines CUDA LBM Lösers behandeln Tölke [52] und Zhao [59]. Hierbei simmuliert Tölke den Fluss durch ein poröses Medium, während Zhao hauptsächlich die LBM zur Bildbearbeitung nutzt. Beide Autoren stellen das CUDA Programmiermodell zur Programmierung der Grafikkarte mit einer sehr großen Anzahl von Threads vor.

Stürmer et al. [50] untersuchen die Simulation von Blutfluss auf der Cell BE um es Medizinern zu ermögliche, die Bildung von Anheurismen besser zu verstehen.

Die Parallelisierung der LBM zur Ausführung auf einem Cluster beschreiben Körner et al. [33]. Hierbei wird insbesondere die lokalität in Zeit und Raum der LBM herausgearbeitet und eine ausführliche Anleitung zur Parallelisierung derselben gegeben.

Speziell die Parallelisierung mit Hilfe von MPI erläutern Thürey et al. [51]. Sie gehen dabei besonders auf die adaptive Zerlegung des Rechengebiets ein um die Rechenlast optimal auf verschiedene Rechner im Cluster verteilen zu können.

Als erste haben Fan et al. [12] die Eignung eins GPU Clusters für wissenschaftliche Berechnungen untersucht und zur Evaluierung einen LBM Löser herangezogen.

Allen vorherigen Arbeiten gemein ist, dass sie ihre Untersuchungen auf eine spezifischen Zielarchitektur ausgerichtet haben. Ein LBM Löser, der ähnlich dem in dieser Arbeit vorgestellten, ganz verschiedene Hardwarearchitekturen unterstützt, wird von Peng et al. [40] vorgestellt und analysiert. Um einen hohen Grad an Parallelität zu erreichen, werden hier jeweils zwei Versionen der eigentlichen Datenstrukturen verwendet und damit der Speicherverbrauch verdoppelt. In die eine Version werden jeweils die aktuellen Ergebnisse geschrieben, während diese aus der anderen, die die Ergebnisse des vorherigen Zeitschritts bereitstellt, gelesen werden. Nach jedem Zeitschritt werden die Rollen vertauscht.

Die Erkenntnisse aus der Arbeit von Mallach [36] bezüglich des Entwurfs eines Multicore Backends, das auch bei geringen Problemgrößen die Berechnung nicht durch einen hohen Verwaltungsaufwand ausbremst, flossen in die Entwicklung des hier gezeigten Multicore Backends ein.

Parallel zu dieser Arbeit erweitert Geveler [16] den in Kapitel 5 vorgestellten LBM Löser um die Möglichkeit mit Bodenprofilen und Festkörpern zu interagieren.

2 HPC Hardware

Im Folgenden werden die unterschiedlichen Hardwarearchitekturen, die im Rahmen dieser Arbeit betrachtet wurden. Im Rahmen des wissenschalftlichen Rechnens geht es dabei insbesondere um ihre Eignung für Fließkommaoperationen.

2.1 CPU

Der althergebrachte Weg, um die Rechenleistung einer CPU zu beschleunigen war die Erhöhung ihrer Taktfrequenz. Mit Taktfrequenzen um vier GHz stießen die Hersteller aber zunehmend an elektrotechnische und physikalische Grenzen, insbesondere weil sich die Wärmeverlustleistung und damit auch die Energieeffizienz in nicht mehr vertretbare Dimensionen entwickelte [43] und die Signallaufzeit an sich nicht mehr erhöht werden konnte. Analog zu der schon angesprochenen memory wall, wird dieses Phänomen als *power wall* bezeichnet. Gleichzeitig erscheint es nicht mehr weiter möglich, durch gleichzeitiges Abarbeiten mehrerer im Programmquelltext aufeinander folgender Befehle einen weiteren Geschwindigkeitsgewinn zu erreichen. Viele Befehle beeinflussen den nachfolgenden Programmfluss und dadurch sind der Parallelität auf Instruktionsebene (*instruction-level parallelism*) sehr häufig Grenzen gesetzt. Diese Grenze wird als *ILP wall* bezeichnet.

Im Folgenden sollen verschiedene Techniken vorgestellt werden, wie trotz der genannten Grenzen die Rechenleistung einer CPU weiter erhöht werden kann.

FPU Sehr alte x86 Prozessoren (bis einschließlich Intel 80486) mussten Fließkommaberechnungen in Software auf den Fixpunktrecheneinheiten simulieren, da sie keine Recheneinheit für Fließkommazahlen, die sogenannte *Floating Point Unit* (*FPU*), besaßen. Da dies sehr langsam war, gab es die Möglichkeit eine FPU als externen Koprozessor nachzurüsten, um die CPU zu entlasten. Die FPU arbeitet auch heute noch mit einem Stackspeicher und kann deshalb nur eine Fließkommaoperation gleichzeitig ausführen. Ab dem Intel Pentium Prozessor ist diese FPU auf Grund sinkender Kosten in der Produktion von Prozessoren fest in die CPU integriert.

MMX Um die Anzahl der möglichen Rechenoperationen pro Sekunde neben den normalen Fortschritten durch einen beschleunigten Prozessortakt weiter zu erhöhen, wurde zunächst von Intel die MMX-Technologie entwickelt und erstmals im Intel Pentium Prozessor ver-

baut. Hierbei werden die schon vorhandenen 80 Bit Register des Stackspeichers der FPU auf acht 64 Bit Integerregister abgebildet, auf die wahlfrei zugegriffen werden kann. Diese Vektorregister können einen 64 Bit Integer oder zwei 32 Bit Integer oder vier 16 Bit Integer speichern. Damit wird es möglich, auf allen in einem Register gespeicherten Daten ein und dieselbe Rechenoperation durchzuführen. Durch Anwendung dieses sogenannten *Single Instruction Multiple Data* (*SIMD*) Prinzips [14] vervierfacht sich in diesem Fall die Anzahl der pro Sekunde ausgeführten Rechenoperationen. Zum Beispiel können die in den Registern A und B gespeicherten Zahlen $A = (a_3, a_2, a_1, a_0)$ und $B = (b_3, b_2, b_1, b_0)$ durch die SIMD Operation $C = A + B$ gleichzeitig elementweise addiert und in dem Register $C = (c_3, c_2, c_1, c_0)$ gespeichert werden:

$$
\begin{aligned}
c_0 &= a_0 + b_0 \\
c_1 &= a_1 + b_1 \\
c_2 &= a_2 + b_2 \\
c_3 &= a_3 + b_3
\end{aligned}
$$

Diese Technik wird schon seit langem in Großrechnern, genauer Vektorrechnern [27] verwendet. Da die MMX und FPU Register physikalisch identisch sind, ergeben sich allerdings Probleme, da nur entweder MMX oder FPU genutzt werden können. Gleichzeitig können mit MMX keine Fließkommaoperationen sondern nur Integeroperationen ausgeführt werden.

SSE Diese Einschränkungen führten zu der Entwicklung von SSE (*Streaming SIMD Extensions*) [28], die erstmals im Intel Pentium II verwendet wurden und acht neue physikalische 128 Bit Register in der CPU hinzufügten. Prozessoren, die die x86-64 Erweiterung (AMD64 ab Athlon 64 / Intel64 ab Pentium 4) beinhalten, besitzen insgesamt sechzehn 128 Bit SSE Register. Diese SSE Register können jeweils vier 32 Bit Fließkommazahlen in einfacher Genauigkeit oder zwei 64 Bit Fließkommazahlen in doppelter Genauigkeit speichern. Die nachfolgenden Revisionen SSE2, SSE3 und SSE4, sowie ähnliche Initiativen von AMD (3DNow!, SSE4a) haben sowohl eine Unterstützung von SSE für Fixkommaoperationen eingeführt, als auch den Befehlsumfang über arithmetische Operationen auf Anwendungen der graphischen Datenverarbeitung und Multimedia erweitert. Außerdem werden nach und nach anfängliche Restriktionen, wie die Form der Speicheradressen, von denen Daten zwischen Hauptspeicher und SSE Register transferiert werden können, reduziert.

2.1.1 Multicore

Ein anderer Ansatz die Anzahl der gleichzeitigen Berechnungen zu steigern, ist die Anzahl der gleichzeitig ausgeführten Operationen auf unterschiedlichen Daten zu erhöhen. Dieses Prinzip wird *Multiple Instruction, Multiple Data* (*MIMD*) [14] genannt. Dies bedeutet auf

Seite der Software üblicherweise, dass das Programm in mehrere parallelen Handlungsstränge, sogenannte *Threads*, aufgeteilt wird. Diese Threads sollen nun von der CPU möglichst gleichzeitig ausgeführt werden.

Hyper-Threading Um den Aufwand an zusätzlichen Prozessorkomponenten möglichst gering zu halten, kann man sich zunächst darauf beschränken, nur die Registersätze zu duplizieren. Somit kann eine physikalische CPU als zwei logische CPUs vom Betriebssystem benutzt werden, da der Prozessor die Ausführung zwischen zwei Threads hin- und herschalten kann. Der Status des Threads bleibt währenddessen in seinen eigenen Registern gespeichert. Dies lohnt sich, da ein Thread sehr häufig auf andere Hardwarekomponenten warten muss, wie etwa den Arbeitsspeicher oder die Festplatte, und zu dieser Zeit selbst gar nicht die Recheneinheiten der CPU in Anspruch nimmt.

Multicore Echte Mehrkernprozessoren besitzen nun mehrere vollständig unabhängige Recheneinheiten (*Multicore*), also Kerne (*Cores*) mit eigenen Rechenwerken, Caches und teilweise auch Speichercontrollern. Theoretisch besitzt ein Zweikernprozessor die doppelte Rechenleistung seines einkernigen Gegenstücks. Einen Schritt weiter gehen Multiprozessorsysteme, die mehrere Mehrkernprozessoren auf einer Hauptplatine vereinen.

Betrachtet man die Mehrkernprozessoren der Hersteller Intel und AMD, so lassen sich insbesondere bei der Speicherhierachie wesentliche Unterschiede erkennen.

Bei den Prozessoren der Pentium D und Intel Core2 Familie besitzt jeder Kern einen eigenen L1-Cache aber alle Kerne teilen sich einen gemeinsamen L2-Cache. Der Zugriff auf den Hauptspeicher geschieht uniform über einen gemeinsamen Frontsidebus, der die Verbindung zum Speichercontroller auf der Hauptplatine herstellt. Im Gegensatz dazu besitzen Prozessoren der Intel Core i7 Familie und Prozessoren der AMD K9 Familie eigene L1- und L2-Caches pro Kern. Alle Kerne teilen sich lediglich einen L3-Cache. Gleichzeitig geschieht der Zugriff auf den Hauptspeicher nicht uniform über jeweils eigene Speichercontroller, die direkt im Chip integriert sind. Diese Art der Speicherarchitektur nennt man *Non Uniform Memory Architecture* (NUMA), da hier jeder Kern seinen eigenen Speicherbereich besitzt. Versucht ein anderer Kern auf diesen Bereich zuzugreifen, so geschieht dies über den erstgenannten Kern. Das bedeutet in der Regel, dass der Zugriff auf den eigenen Speicherbereich für einen Kern deutlich schneller ist, als auf die Speicherbereiche der anderen Kerne. Damit Daten, die ein Kern aus dem Speicherbereich eines anderen Kerns erhalten hat, nicht durch Schreib- oder Lesezugriffe anderer Kerne invalidiert werden, sorgt die CPU für sogenannte Cache-Kohärenz. Das bedeutet, dass sobald ein Kern ein Datum verändert, dies auch in den Caches aller anderen Kerne, die dasselbe Datum in ihrem eigenen Cache vorliegen haben, passiert. Dieses Verhalten, das auch als *cache-coherent NUMA* (ccNUMA) bezeichnet wird, vereinfacht die Programmierung dieser Systeme erheblich, da man immer davon aus-

gehen kann, dass der Zugriff auf eine Speicherstelle ein valides Ergebnis liefert. Die Core i7 und K9 Prozessoren gehören zu diesen ccNUMA Architekturen.

2.1.2 Cluster

Im Gegensatz zu den oben genannten NUMA Architekturen, bei denen die Synchronisation und Kommunikation auf einem Chip und über den L3-Cache realisiert ist, stellen die einzelnen Rechner (Knoten) eines Clusters [1] komplett eigenständige Computersysteme dar. Die Kommunikation zwischen den Knoten geschieht hier über die Netzwerkverbindung, durch die die einzelnen Knoten verbunden sind und ist damit deutlich langsamer als die Kommunikation innerhalb eines Multicorechips. So besitzt zum Beispiel der LiDO-Cluster [29] an der Universität Dortmund 224 einzelne Rechner (Knoten), die miteinander vernetzt sind. Zusammen besitzen die Knoten 1,216 TB Arbeitsspeicher. Dies ermöglicht das Bearbeiten von deutlich größeren Problemen, als es etwa mit einem einzelnen Computer möglich wäre, der heute üblicherweise maximal 8 bis 16 GByte Arbeitsspeicher besitzt. Gleichzeitig können bei derart großen Problemen tatsächlich alle eingesetzten Knoten die Gesamtrechenzeit verkürzen, ohne dass ein zusätzlich hinzugefügter Knoten mehr Verwaltungsaufwand erzeugt, als er die Rechenzeit verkürzt. Dieser Verwaltungsaufwand ist häufig notwendig, um die einzelnen rechnenden Knoten zu synchronisieren und zwischen ihnen die Rechenergebnisse der anderen Knoten zu kommunizieren.

Da in einem Cluster ein Knoten nicht direkt, wie es bei den Kernen einer Multicore-CPU in den bisher vorgestellen NUMA Architekturen der Fall war, auf Speicherbereiche eines anderen Knoten zugreifen kann, spricht man hier auch von einer *Distributed Memory* Architektur. Hier ist es notwendig, dass die Programme, die auf den Knoten ausgeführt werden, einen notwendigen Transfer von Daten aus dem Arbeitsspeicher eines Knoten in den Arbeitsspeicher eines anderen Knoten explizit anstoßen. Um dies zu vereinfachen, gibt es Kommunikationsstandards, wie das *Message Passing Interface* (MPI) [49] [21], das den Nachrichtenausstausch und damit die Kommunikation zwischen den Knoten eines Clusters beschreibt. Bei MPI geschieht die Kommunikation durch Nachrichten, die zwischen zwei oder mehr Knoten verschickt werden. Diese Nachricht enthalten dann zum Beispiel den Inhalt eines Speicherbereichs, den ein Knoten einem anderen zur Verfügung stellt, damit dieser die Daten bearbeitet und eventuell später wieder zurückschickt. Durch die Möglichkeit, den Programmfluß von Sender und Empfänger zu blockieren, können die einzelnen Knoten ihre Programmausführung synchronisieren, da alle beteiligten Knoten darauf warten, dass die Kommunikation abgeschlossen wird.

2.2 GPU

Überblick Grafikkarten [13] müssen in der Lage sein, eine sehr große Menge an Daten sehr schnell zu verarbeiten, um Grafiken in Echtzeit darstellen zu können. Die Art und

Weise wie diese Berechnungen durchgeführt werden, hat zu einer sehr fein parallelisierten Hardwarearchitektur geführt, bei der das Hauptaugenmerk auf einem maximalen Durchsatz und weniger auf der schnellen Berechnung eines einzelnen Pixelwertes liegt. Die hierdurch erreichte theoretische Fließkommmaleistung liegt bei aktuellen Grafikkarten, wie der NVIDIA GeForce GTX 285, bei etwa 1000 GFlop/s in einfacher Genauigkeit. Der Zugriff auf den Speicher der Grafikkarte erfolgt mit einer Transferrate von etwa 160 GByte/s.

Diese eigentlich aus dem Bedarf der Grafikverarbeitung entstandene Rechenleistung möchte man nun auch für andere Berechnungen nutzbar machen. Die Verwendung einer Grafikkarte um allgemeine Berechnungen durchzuführen, die gewöhnlich nur auf der CPU ausgeführt werden, wird als *general-purpose computing on graphics processing units* (GPGPU) [39] bezeichnet.

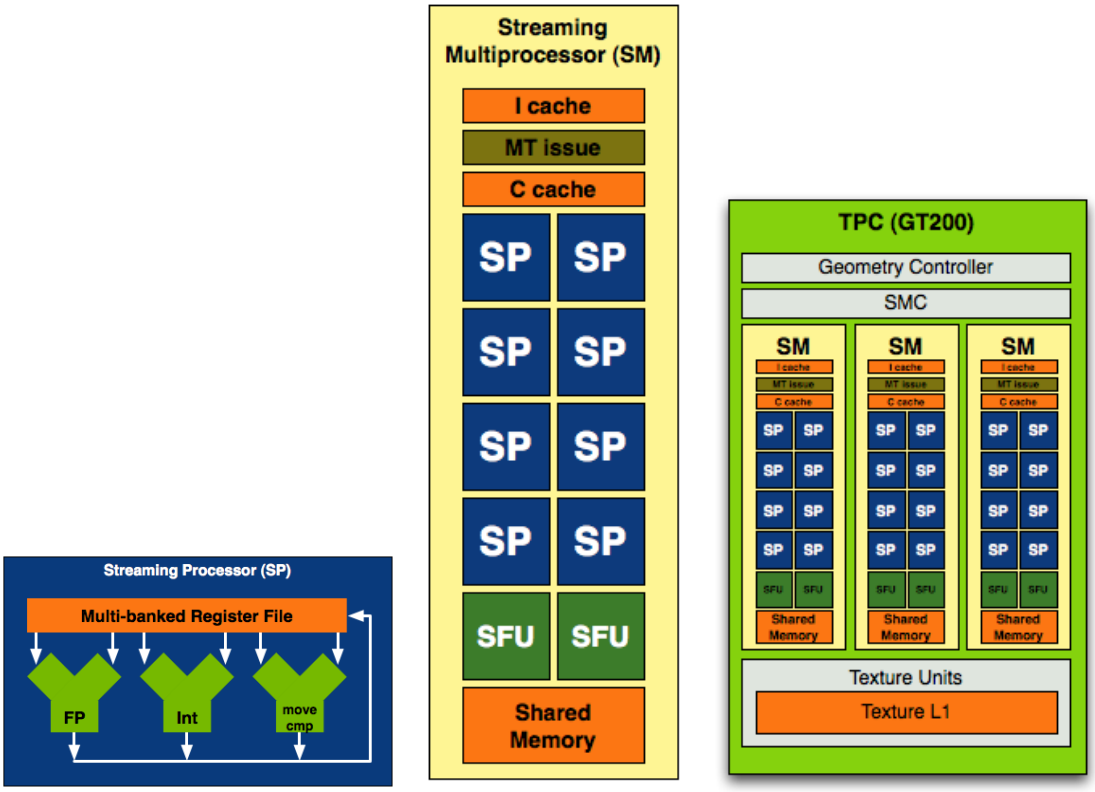
Da die Untersuchungen in dieser Arbeit auf aktuellen Grafikkarten von NVIDIA [35] durchgeführt wurden, soll im Folgenden die Architektur dieser Hardware genauer beschrieben werden.

Aufbau Wie weiter oben angeführt, besitzt eine Grafikkarte eine sehr stark parallelisierte Architektur. Um dies zu erreichn, folgt ihr Aufbau dem Schema, eine große Menge sehr kleiner Recheneinheiten zu immmer größeren Strukturen zusammenzufassen.

Das atomare Bauteil der GT 200 Serie, zu der auch die oben erwähnte GeForce GTX 285 gehört, ist der sogenannte *Streaming Processor* (*SP*) (Abbildung 2.1a), ein Mikroprozessor mit einer Einheit für Vergleiche und zur Berechnung von Sprungadressen, einer Integer- und einer Fließkommarecheneinheit. Der SP besitzt keinen eigenen Cache und vermag nicht viel mehr als mathematische Operationen durchzuführen. Damit könnte man ihn als stark vereinfachte Version einer SPE, wie sie in Abschnitt 2.3 beschrieben wird, auffassen.

Fasst man nun mehrere dieser Streaming Prozessoren zusammen, so erhält man einen *Streaming Multiprocessor* (*SM*) (Abbildung 2.1b). Ein SM besteht aus acht SPs und zwei weiteren sogenannten *Special Function Units* (*SFU*), die für trigonometrische Funktionen und Interpolationen zuständig sind. Alle diese Recheneinheiten erhalten ihre Aufgaben von einer übergeordneten Einheit mit Namen *Multithreaded instruction fetch and issue unit* (*MT Issue*), die die Operationen an die Prozessoren verteilt. Zu diesem Zwecke stehen ein kleiner Befehlscache (*I cache*), ein kleiner Cache für konstante Daten (*C Cache*) und ein zwischen allen Recheneinheiten gemeinsam geteilter 16 kByte großer Speicher (*Shared Memory*) zur Verfügung. Der SM ist in der Lage 1024 Threads parallel auszuführen. Gleichzeitig geschieht das Umschalten zwischen Threads komplett durch die Hardware, weshalb dies ohne Verzögerung möglich ist.

Die SMs werden jeweils zu dritt zu einem *Texture Processor Cluster* (*TPC*) (Abbildung 2.1c) zusammengefasst. Zusätzlich enthält ein TPC wiederum eine Steuereinheit (*Streaming Multiprocessor Controller* (*SMC*)) und einen gemeinsamen Texturspeicher (*Texture L1*).



(a) Streaming Processor (b) Streaming Multiprocessor (c) Texture Processor Cluster

Abbildung 2.1: Schemata der einzelne Grafikkartenkomponenten (Quelle: NVIDIA)

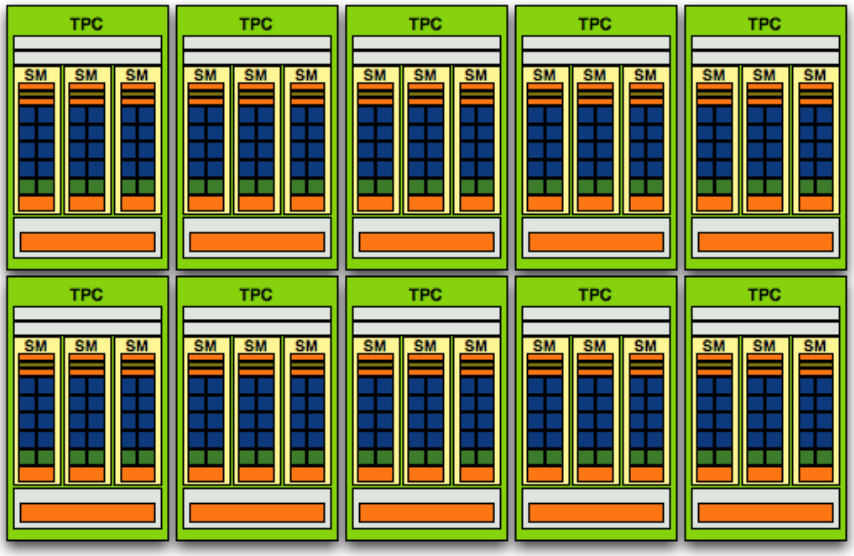


Abbildung 2.2: Schema eines Streaming Processor Array (Quelle: NVIDIA)

Schließlich sind zehn dieser TPC zu einem *Streaming Processor Array* (*SPA*) (Abbildung 2.2) kombiniert. Somit befinden sich auf der Grafikkarte insgesamt 30 MPs bzw. 240 SPs.

Demgegenüber steht eine Speicherhierachie, die auf oberster Ebene ein bis zwei Gbyte globalen Grafikkartenspeicher zur Verfügung stellt. Dieser kann entweder als normaler Arbeitsspeicher aufgefasst oder alternativ auch als Texturspeicher angesprochen werden. Im zweiten Fall kann nur lesend darauf zugegriffen werden. Diese Lesezugriffe werden aber in zwei Dimensionen gepuffert und unterstützen damit optimal die Arbeit mit Texturen und anderen in zwei Dimensionen lokalen Daten. Jeder Chip enthält zusätzlich 64 kByte sehr schnellen sogenannten "konstanten Speicher", auf den alle Threads zugreifen können. Auf die Threads eines SM beschränkt ist der Zugriff auf den schon angesprochenen 16 kByte großen Speicher, den jeder SM besitzt. Schließlich besitzt jeder SM 16384 Hardwareregister, in denen die Daten aller von ihr ausgeführten Threads gespeichert werden. Erscheint diese Zahl zunächst recht hoch, so ergibt sich bei einer vollen Auslastung eines SM mit 1024 Threads eine maximale Registeranzahl von 16 Registern pro Thread.

Ein auf der Grafikkarte ausgeführtes Programm besteht nun aus sehr vielen Threads, die alle dieselbe Folge von Operationen auf unterschiedlichen Daten ausführen. Bezugnehmend auf das oben angeführte SIMD Prinzip, nennt NVIDIA dies *single instruction multiple thread (SIMT)*. Die einzelnen Threads werden in Blöcken organisiert. Viele dieser Blöcke bilden wiederum ein sogenanntes Gitter (*Grid*). Hierbei werden alle Threads eines Blocks auf einem einzelnen SM ausgeführt, können sich also über den geteilten Speicher der SM synchronisieren. Diese Threads sind in sogenannten *Warps* [1] zu je 32 Threads gegliedert. Alle Threads eines Warps laufen gleichzeitig auf ihrem SM und führen jeweils denselben Befehl aus. Es können sich allerdings mehrere Warps von Threads in einem Block bei der Ausführung auf einem SM abwechseln. Die ersten 16 und die letzten 16 Threads eines Warps können gemeinsam auf den Grafikkartenspeicher zugreifen und somit ihre Transfers zu einem einzigen gemeinsamen Speichertransfer (*Coalescing*) zusammenfassen. Dies kann die Speichertransferrate deutlich erhöhen, ist allerdings mit Einschränkungen verbunden. Damit alle 16 Transfers tatsächlich zu einem einzelnen Transfer verbunden werden können, müssen alle 16 in einem Warp benachbarten Threads auf 16 benachbarte Speicherstellen zugreifen. Dies bedeutet, dass wenn ein Thread i auf eine Speicherstelle j zugreift, der Thread $i + 1$ nur auf die Speicherstelle $j + 1$ zugreifen kann. Im Gegensatz zu Grafikkarten der GeForce 8 Serie, bei denen eine kleine Verletzung dieser Regel durch einen der 16 Threads bereits eine sehr starke Aufteilung der Speichertransfers bedeutete, kann der Speichercontroller der GT 200 Serie auch solche Speicherzugriffe zu möglichst wenigen Transfers zusammenfassen.

Jedes Mal wenn ein SM den nächsten Befehl ausführen kann, wird durch die MT Issue Einheit ein Warp von Threads ausgewählt, der gerade bereit ist und nicht auf ausstehende

[1] (Eine Webkette (engl.: Warp) entspricht auf einem echten Webstuhl einer vertikalen Gruppe paralleler Fäden (engl.: Thread)

Datentransfers wartet, da die Latenz beim Zugriff auf den Grafikkartenspeicher durchaus mehr als 1000 Takte betragen kann. An dieser Stelle wird wieder deutlich, dass das grundlegende Design der Grafikkarte auf reinen Durchsatz ausgelegt ist, da es nahezu keine Cache-Hierachie gibt, die etwaige Zugriffe auf den Grafikkartenspeicher beschleunigen würde, wie es ähnliche Cache-Hierachien bei der normalen CPU leisten. Im Optimalfall müssen alle Threads eines Warps denselben Befehl ausführen, sodass alle Threads dies gleichzeitig tun können. Weichen die auszuführenden Befehle der einzelnen Threads durch vorherige Verzweigungen im Programmfluss voneinander ab, so berechnet jeder Thread des Warps beide möglichen Programmpfade und verwirft am Ende das nicht gültige Ergebnis, bis alle Threads wieder denselben Zustand erreicht haben.

2.3 Cell BE

Überblick Die *Cell-Broadband Engine* wurde vom STI Design Center, einer Kooperation der Unternehmen Sony, Toshiba und IBM, entwickelt [30]. Sie wird zum Einen in der Spielekonsole Playstation 3 von Sony verbaut, ist zum Anderen aber auch direkt als Bladeserver erhältlich. So sind zum Beispiel im IBM Blade QS22 [32] zwei miteinander verbundene Cell Prozessoren eingebaut.

Im Gegensatz zu herkömmlichen Mehrkern-Prozessoren besteht der Cell-Prozessor nicht aus mehreren gleichartigen Kernen, sondern aus einer 64bit-*PowerPC*-CPU, dem sogenannten *Power Processor Element* (PPE), und bis zu acht sehr leistungsfähigen Vektorprozessoren, die als *Synergistic Processing Elements* (SPE) bezeichnet werden. Letztere sind nicht zu Steuerungsaufgaben, wie z.B. der Ausführung eines Betriebssystems fähig, aber insbesondere für Fließkommaberechnungen hocheffizient. Abbildung 2.3 zeigt den schematischen Aufbau des Cell-Prozessors.

Jede SPE verfügt im Cell-Prozessor nur über sehr wenig lokalen Speicher (genannt *Local Store (LS)*) und kann nicht direkt auf den Hauptspeicher zugreifen. Die Kommunikation zwischen dem Hauptspeicher und den lokalen Speichern der SPEs über einen gemeinsamen Speicherbus muss vom Programmierer explizit vorgegeben werden. Dies bedeutet einerseits einen sehr hohen Freiheitsgrad für den Programmierer, andererseits aber auch starke Einschränkungen auf Grund des verhältnismäßig kleinen direkt adressierbaren Speichers.

Durch die acht SPEs summiert sich die maximale (theoretische) Fließkommaleistung des Cell-Prozessors auf 230 GFlop/s bei einfacher Genauigkeit.

Die folgende tiefergehende Beschreibung der wesentlichen Komponenten des Cell-Prozessors nimmt Bezug auf die Darstellung von Kahle [30], in der auch weitere Details nachzulesen sind.

Das Power Processor Element Das *Power Processor Element* (kurz: PPE) verfügt über einen 32 kByte L1-Instruktions- und Daten-Cache, sowie eine Taktfrequenz von 3,2 GHz.

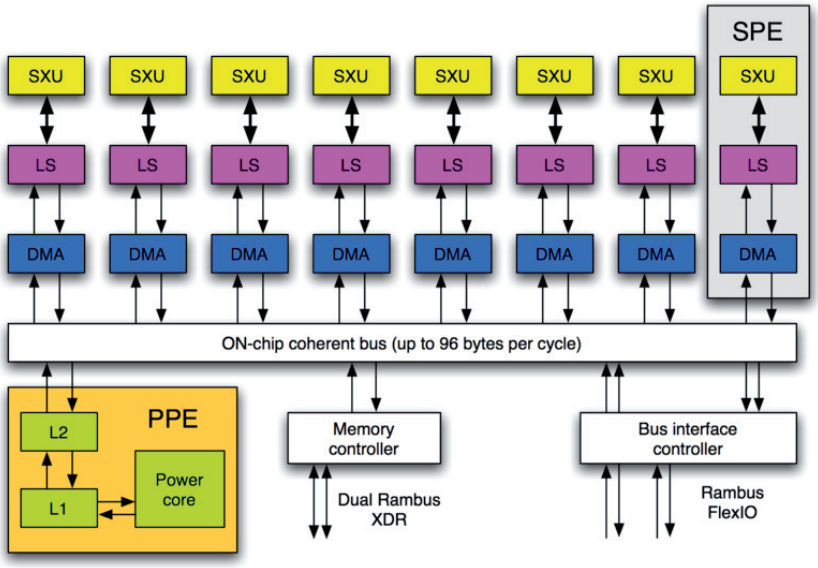


Abbildung 2.3: Schema des Cell-Prozessors (Quelle: Wikipedia)

2 Threads werden nach dem *Round-Robin Prinzip* zur gleichen Zeit verarbeitet (*dual-threaded*) und bis zu 2 Instruktionen können sich gleichzeitig in der Ausführungs-Phase befinden (*dual-issued*), solange sie nicht die gleiche Ausführungseinheit belegen.

Das Hauptmerkmal des Designs ist eine reduzierte Komplexität um eine hohe Taktrate und eine geringe Verlustleistung zu gewährleisten. So wurde zum Beispiel auf *Out-of-Order* Ausführung und eine komplexe Sprung-Vorhersage (*Branch-Prediction*) verzichtet.

Speichertransfers per DMA Wie alle Recheneinheiten des Cell-Prozessors verfügt das PPE über einen Speichercontroller namens *Memory Flow Controller (MFC)*. Über ihn werden alle Transfers per DMA (*Direct Memory Access*) oder die im Verhältnis dazu langsameren MMIO (*Memory-Mapped-I/O*)-Register abgewickelt. Dabei ist es sowohl möglich, dass das PPE Daten zu den SPEs sendet, als auch, dass die SPEs ihrerseits mit ihren *Memory Flow Controllern* Daten vom Hauptspeicher des PPE anfordern. DMA-Speichertransfers sind grundsätzlich an 16-Byte Grenzen ausgerichtet und haben eine maximale Größe von 16 kByte. Sie dienen insbesondere der Übertragung von kontinuierlichen Datenbereichen aus dem Hauptspeicher in die lokalen Speicher der SPEs.

Sollen nicht kontinuierliche Bereiche übertragen werden, so bietet die Cell-Architektur das Konzept der Listentransfers. In diesem Fall werden bis zu 2048 DMA-Transfers mit jeweils 16 kByte Größe über ein vorgegebenes Schema zu einer DMA-Liste zusammengefasst. Alle Adressen einer DMA-Liste müssen dabei in den höherwertigen 32 Bit übereinstimmen. So sind theoretisch mit einer solchen Liste bis zu 32 MByte Übertragung möglich, was der 128-fachen Größe des *Local Stores* entspricht. SPEs können ihrerseits solche DMA-Listen

im *Local Store* anlegen, typischerweise werden die Listen aber vom PPE (auf Grund der Kenntnis von Hauptspeicheradressen) vorbereitet. Anschließend werden sie entweder zur SPE übertragen, oder von der SPE vor dem eigentlichen DMA-Listentransfer per herkömmlichem DMA-Transfer geladen.

Eine andere Möglichkeit zur Kommunikation zwischen dem PPE und den SPEs ist das Mailbox-System, welches ebenfalls über den MFC und die MMIO-Register abgewickelt wird. Jede Recheneinheit besitzt gewissermaßen einen Maileingang und -ausgang. Mit diesem System ist es möglich, 32-bit Nachrichten auszutauschen, die auch unterbrechenden Charakter haben können (*Interrupt-Mails*).

Die Synergistic Processing Elements Die SPEs verfügen als SIMD Recheneinheiten über 128 Vektorregister mit 128 Bit und einen 256 kByte großen *Local Store* anstelle eines Caches. Die Ausführungs- und Zugriffszeiten sind dadurch vollständig vorhersagbar und nicht von Trefferraten im Cache abhängig. Die Anzahl der vorhandenen Register erlaubt eine hohe Ausnutzung der SIMD-Vektoreinheit für große Probleme. Dies ist vor allem für die Anwendungsoptimierung selbst interessant, während bei grundlegenden Operationen wie Additionen oder Multiplikationen eher wenige Register ausgenutzt werden können.

Die SPEs nutzen *dual issue*: Es können Berechnungen und verschiedene Speicheroperationen gleichzeitig durchgeführt werden. Ebenso wurde wie beim PPE auf *In-Order-Execution* gesetzt und Sprünge werden nicht durch die Hardware vorhergesagt. Mit Hilfe von *branch hints* kann der Programmierer oder Compiler hohe Kosten für *branch misses* verhindern. In so einem Fall werden 17 Instruktionen ab der Sprungadresse vorgehalten.

Da die SPEs in der Lage sindi, ein 3-Operanden-*multiply add*, also Multiplikation zweier Operanden inklusive Addition des Ergebnisses auf einen dritten Operanden, in einem Befehl auszuführen, können vektorisiert bis zu vier solche *multiply-add*-Instruktionen pro SPE gleichzeitig ausgeführt werden. Insgesamt handelt es sich dabei also um acht Operationen, wodurch bei 3,2 GHz theoretisch bis zu 25,6 GFlop/s erreicht werden. Aufsummiert für acht SPEs bedeutet dies eine maximale theoretische Leistung von 204,8 GFlop/s. Dieser Wert wurde bei Tests von IBM auch annähernd erreicht.

Der *Local Store* erfüllt ähnlich wie ein Cache die Aufgaben eines sehr schnellen Zwischenspeichers zwischen RAM und Prozessor. Er dient sowohl für den schnellen Zugriff auf per DMA-Transfer geholte Daten, wie auch als Ablageort für Zwischenergebnisse. Zum Ende einer typischen Berechnung werden die Ergebnisse dann erneut über DMA-Transfers in den Hauptspeicher des PPE zurückgeschrieben.

Wie bereits erwähnt besitzt jede SPE dazu einen eigenen Speicher-Controller, der die Zugriffe von außen auf den *Local Store* und umgekehrt steuert, also zum Beispiel Datenanfragen an den Hauptspeicher oder andere *Local Stores*.

Bus und Speicheranbindung des Cell Der sogenannte *Element Interconnect Bus* (kurz: EIB) verbindet die einzelnen Elemente des Cell Prozessors (PPE, 8 SPEs, E/A-Geräte) durch einen Ringbus miteinander. Es handelt sich dabei um 4 unidirektionale 16-Byte-Kanäle, die paarweise in entgegengesetzte Richtungen verlaufen. Die Daten werden pro Bustakt (halber Prozessor-Takt) einen Schritt weitergegeben, wodurch bis zu drei Übertragungen gleichzeitig je Kanal durchgeführt werden können. Somit sind bis zu 12 parallele Übertragungen und sehr hohe Datentransferraten möglich, was erforderlich ist, um die SPEs kontinuierlich mit Daten zu versorgen. Von IBM wurden über 200 GByte/s Datenrate bei 1,6 GHz Bustakt erreicht.

Auch die Anbindung an den Hauptspeicher muss genug Bandbreite bieten, um die neun Prozessoreinheiten mit Daten zu versorgen. Dafür sorgt das *XIO Interface von Rambus*, das im Dual-Channel-Betrieb Datenraten von bis zu 25,6 GByte/s erreicht. Die Kommunikation mit anderen Hardwarekomponenten geschieht über das *FlexIO-Interface* der Firma Rambus. Insgesamt sind bei 3,2 GHz Datenraten von 44,8 GByte/s ausgehend und 32 GByte/s eingehend möglich. Neben Verbindungen zur *North-* und *Southbridge* können so auch zwei Cell-Prozessoren direkt miteinander verbunden werden.

3 Die Lattice Boltzmann Methode für die Flachwassergleichung

Bei der Lattice Boltzmann Methode (LBM) handelt es sich um eine Ende der 1980er Jahre entwickelten Methode zur numerischen Strömungssimulation. Sie verbindet die stark vereinfachte Teilchen-Mikrodynamik der Lattice Gas Automaten mit der Boltzmannschen Transportgleichung. Dieses Kapitel faßt in erster Linie die mathematischen Ergebnisse der Ausführungen von Zhou [60] zusammen, die notwendig sind, um Flachwassersimulationen mit der Lattice Boltzmann Methode durchzuführen. Dort sind auch weitergehende Informationen zu finden. Es wird bewusst auf die mathematischen Hintergründe und Herleitungen verzichtet, da diese Arbeit die Anwendung der LBM mit Hinblick auf hardwareeffiziente Berechnungen behandelt.

Die Flachwassergleichungen können angewendet werden, wenn die horizontalen Ausmaße (Wellenlänge λ) deutlich größer als die vertikalen (Wellenhöhe Δh) sind, also

$$1 \gg \frac{\Delta h}{\lambda} \tag{3.1}$$

gilt. Da die Tiefenwirkung einer Strömung nach obiger Bedingung vernachlässigt werden kann, kann das Modell auf mathematischer Ebene einfach gehalten werden. Die wesentliche Größe des Flachwassermodells ist die Wasserhöhe $h(\mathbf{x}, t)$ eines Fluids zum Zeitpunkt t an der Stelle $\mathbf{x}$, aus der die zu errechnende Fluidoberfläche konstruiert werden kann. Die Höheninformationen einer zweidimensionalen Fluidoberfläche werden auch als Höhenfeld bezeichnet. Der Zustand dieses Fluids wird vollständig durch seine Momentangeschwindigkeiten $u_i(\mathbf{x}, t)$ in kartesischen Richtungen i an jeder Stelle $\mathbf{x}$ zum Zeitpunkt t beschrieben.

Üblicherweise werden die Flachwassergleichungen in der Erhaltungsform

$$\frac{\partial}{\partial t}Q + \frac{\partial}{\partial x}F(Q) + \frac{\partial}{\partial y}G(Q) = 0 \tag{3.2}$$

angegeben. Hierbei entspricht

$$Q = \begin{pmatrix} h \\ hu_x \\ hu_y \end{pmatrix} \tag{3.3}$$

dem Vektor der konservativen Variablen, also der Wasserhöhe und den Geschwindigkeiten. Die Funktionen

$$F = \begin{pmatrix} hu \\ hu_x^2 + \frac{1}{2}gh^2 \\ hu_x u_y \end{pmatrix} \tag{3.4}$$

und

$$G = \begin{pmatrix} hu_x \\ hu_x u_y \\ hu_y^2 + \frac{1}{2}gh^2 \end{pmatrix} \tag{3.5}$$

können als Fluß über ein Kontrollvolumen in die beiden Raumrichtungen interpretiert werden. Der Faktor g entspricht der Gravitationskonstante.

Diese Erhaltungsform beschreibt also die Situation, dass sich die zeitlichen Veränderungen des Fluids in Höhe und Geschwindigkeit in der Summe über alle Orte des Höhenfeldes gegenseitig aufheben. Für eine ausführlichere Darstellung dieser Gleichungen wird etwa auf die Arbeit von Geveler [16] verwiesen.

Dieses Flachwassermodell soll nun mit der Lattice Boltzmann Methode berechnet werden. Die grundlegende Idee der Lattice Boltzmann Methode ist die Simulation der Bewegung von fluiden Partikeln. Hierzu wird über das zu berechnende kontinuierliche Höhenfeld ein rechteckiges Gitter gelegt, um das Rechengebiet zu diskretisieren. Nun werden die auf dem Gitter liegenden Partikel in vorgegebene Richtungen von Gitterpunkt zu Gitterpunkt bewegt. Diese Richtungen werden durch die Struktur der Gitterzelle (*Lattice*) vorgegeben. Im Rahmen dieser Arbeit wird ein zweidimensionales Lattice mit neun vorgegebenen Richtungen $\alpha \in (0, ..., 8)$ verwendet (D2Q9), wie es in Abbildung 3.1 zu sehen ist.

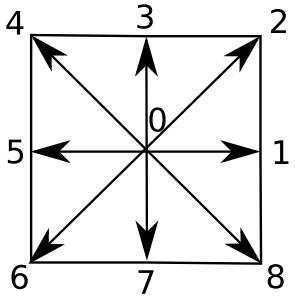


Abbildung 3.1: D2Q9 Lattice

Die Partikel können sich auf diesem Lattice in die Richtungen Eins bis Acht bewegen oder sich nicht bewegen (Richtung Null). Des weiteren gibt es zum Beispiel auch noch Lattices der Form D2Q5, D2Q7 oder im Dreidimensionalen ein Lattice der Form D3Q19.

Die Wechselwirkung der fluiden Partikel wird durch die folgende Lattice Boltzmann Gleichung beschrieben:

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) - f_\alpha(\mathbf{x}, t) = -\frac{1}{\tau}(f_\alpha - f_\alpha^{\mathrm{eq}}) \tag{3.6}$$

Hierbei entspricht f_α der zu errechnenden Verteilungsfunktion (Distributionsfunktion) in den neun Richtungen $\alpha \in (0, ..., 8)$. τ stellt die Relaxationszeit dar, die durch einen Lösungsschritt berechnet wird. Der Geschwindigkeitsvektor $\mathbf{e}_\alpha$ entspricht:

$$\mathbf{e}_\alpha = \begin{cases} (0,0) & \alpha = 0 \\ e[\cos\frac{(\alpha-1)\pi}{4}, \sin\frac{(\alpha-1)\pi}{4}] & \alpha = 1,3,5,7 \\ \sqrt{2}e[\cos\frac{(\alpha-1)\pi}{4}, \sin\frac{(\alpha-1)\pi}{4}] & \alpha = 2,4,6,8 \end{cases} \tag{3.7}$$

Die lokale Equilibriumverteilungs- oder Gleichgewichtsfunktion (*equilibrium distribution*) f_α^{eq} bestimmt, welche Flußgleichung gelöst wird. Hier wird die folgende von Zhou [60] angegebene Funktion verwendet, um mit der LBM die Flachwassergleichung (3.2) zu berechnen. Die Funktion lautet:

$$f_\alpha^{\text{eq}} = \begin{cases} h - \frac{5gh^2}{6e^2} - \frac{2h}{3e^2}u_i u_i & \alpha = 0 \\ \frac{gh^2}{6e^2} + \frac{h}{3e^2}e_{\alpha i}u_i + \frac{h}{2e^4}e_{\alpha j}u_i u_j - \frac{h}{6e^2}u_i u_i & \alpha = 1,3,5,7 \\ \frac{gh^2}{24e^2} + \frac{h}{12e^2}e_{\alpha i}u_i + \frac{h}{8e^4}e_{\alpha j}u_i u_j - \frac{h}{24e^2}u_i u_i & \alpha = 2,4,6,8 \end{cases} \tag{3.8}$$

Hierbei fließen die wesentlichen Größe Wasserhöhe h und Momentangeschwindigkeiten u_i der Flachwassergleichung in die Funktion ein. Um später unnötige Berechnungen zu vermeiden, wird dieser Term vereinfacht, indem h nach Möglichkeit ausgeklammert wird:

$$f_\alpha^{\text{eq}} = \begin{cases} h(1 - \frac{5gh}{6e^2} - \frac{2}{3e^2}u_i u_i) & \alpha = 0 \\ h(\frac{gh}{6e^2} + \frac{e_{\alpha i}u_i}{3e^2} + \frac{e_{\alpha j}u_i u_j}{2e^4} - \frac{u_i u_i}{6e^2}) & \alpha = 1,3,5,7 \\ h(\frac{gh}{24e^2} + \frac{e_{\alpha i}u_i}{12e^2} + \frac{e_{\alpha j}u_i u_j}{8e^4} - \frac{u_i u_i}{24e^2}) & \alpha = 2,4,6,8 \end{cases} \tag{3.9}$$

Der Term $-\frac{1}{\tau}(f_\alpha - f_\alpha^{\text{eq}})$ ist der Kollisionsoperator, der steuert, wie sich die Orientierung der Geschwindigkeiten ändert, wenn sich Partikel gegenseitig beeinflussen. Formt man die Lattice Boltzmann Gleichung (3.6) nach $f_\alpha(\mathbf{x} + \mathbf{e}_\alpha\Delta t, t + \Delta t)$ um, so erhält man den Kollisions- und Strömungsschritt (*collide and stream*):

$$f_\alpha(\mathbf{x} + \mathbf{e}_\alpha\Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{\text{eq}}) \tag{3.10}$$

Wenn Partikel sich über den Rand des Rechengebietes hinausbewegen würden, müssen an diesen Rändern Korrekturen vorgenommen werden (*update velocity directions*). Die naheliegenste von Zhou angegebene Korrektur schlägt vor, Partikel, die einen Rand erreichen, an den gegenüberliegenden Rand zu transferieren und ihre Bewegungsrichtung beizubehalten. Dieses Verfahren nennt sich *periodic boundary correction*.

Ein komplizierteres Korrekturschema (*no-slip boundary correction*) lässt die Partikel, die einen Rand erreichen, von diesem reflektiert werden. Abbildung 3.2 zeigt ein Partikel, dass im Zeitschritt $t = i$ den linken Rand des Rechengebiets erreicht. Um das Partikel zu reflektieren, werden nun seine nach links weisenden Distributionsfunktionen f_6, f_5 und

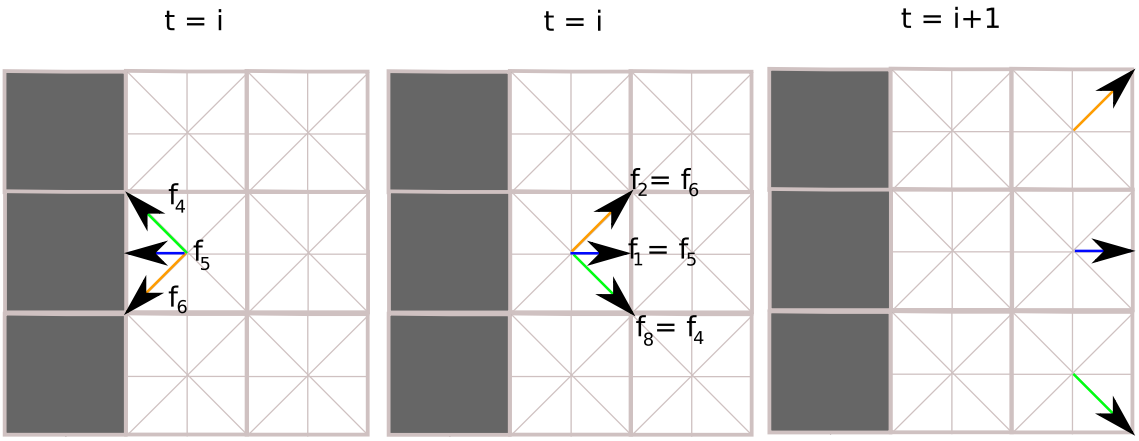


Abbildung 3.2: no-slip boundary correction

f_4 auf die nach rechts weisenden Distributionsfunktionen f_2, f_1, und f_8 kopiert, um die Bewegungsrichtung des Partikel umzukehren.

Allgemein kann diese Regel für Funktionen f in Richtung β, die in Richtung $-\beta$ reflektiert werden, formuliert werden als

$$f_{-\beta} = f_{\beta} \tag{3.11}$$

mit

$$-\beta = \begin{cases} \beta + 4, & 1 \leq \beta \leq 4 \\ \beta - 4, & 5 \leq \beta \leq 8. \end{cases} \tag{3.12}$$

Nachdem die Lattice Boltzmann Gleichung (3.6) gelöst und die Randbedingungen angewendet wurden, können die physikalischen Quantitäten extrahiert werden (*extraction*). Dies sind zum Einen die Wasserhöhe

$$h(\mathbf{x}, t) = \sum_{\alpha} f_{\alpha}(\mathbf{x}, t) \tag{3.13}$$

auf jedem Gitterpunkt und zum Anderen die Geschwindigkeitswerte in kartesischen Richtungen i:

$$u_i(\mathbf{x}, t) = \frac{1}{h(\mathbf{x}, t)} \sum_{\alpha} e_{\alpha i} f_{\alpha} \tag{3.14}$$

Ein kompletter Zeitschritt des Lattice Boltzmann Lösers besteht also aus den vier Teilschritten

- *equilibrium distribution* (Gleichung (3.9))
- *collide and stream* (Gleichung (3.10))
- *update velocity directions* (Randbehandlung)
- *extraction* (Gleichungen (3.13) und (3.14)).

Um korrekte Ergebnisse bei der Berechnung zu erhalten, ist es notwendig gewissen Anfangs- und Stabilitätsbedingungen zu genügen.

Zunächst sollten die physikalischen Quantitäten $h(\mathbf{x}, 0)$ und $u_i(\mathbf{x}, 0)$ initial vorgegeben werden, um mit ihnen initiale Gleichgewichtsfunktionen f_α^{eq} berechnen zu können. Gleichzeitig müssen zu Beginn alle $f_\alpha(\mathbf{x}, 0)$ $\forall \alpha \in [0, 8]$ auf 0 gesetzt werden.

Eine besondere Bedeutung kommt der Wahl korrekter Werte für die Zeitschrittweite Δt und die Relaxationszeit τ zu. Durch die Methode und die verwendeten Flachwassergleichungen ergeben sich nach Zhou [60] hierzu einige Stabilitätsbedingungen. Zunächst entspricht die kinematische Viskosität ν des D2Q9 Modells

$$\nu = \frac{e^2 \Delta t}{6}(2\tau - 1) > 0 \tag{3.15}$$

woraus sich direkt ergibt, dass

$$\tau > \frac{1}{2} \tag{3.16}$$

gelten muss. Zusätzlich beschränkt die Ausdehnung einer Gitterzelle die Geschwindigkeiten der Fluidpartikel durch

$$u_j u_j = e^2 \tag{3.17}$$

und

$$gh = e^2. \tag{3.18}$$

Zuletzt ergibt sich durch Gleichung (3.14), dass

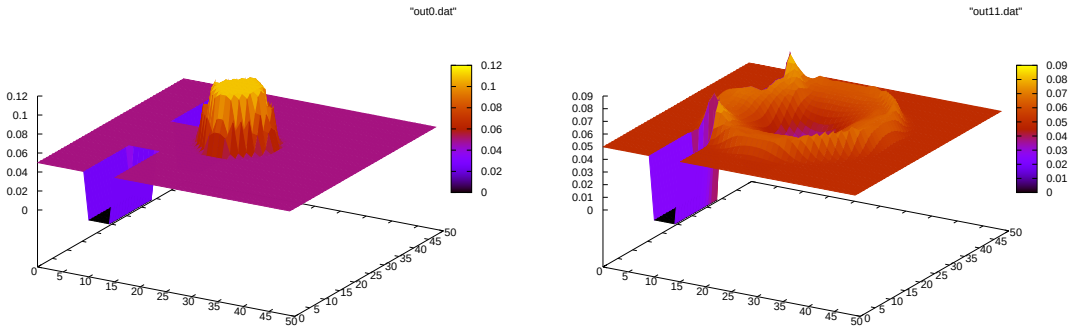
$$h(\mathbf{x}, t) > 0 \tag{3.19}$$

gelten muss.

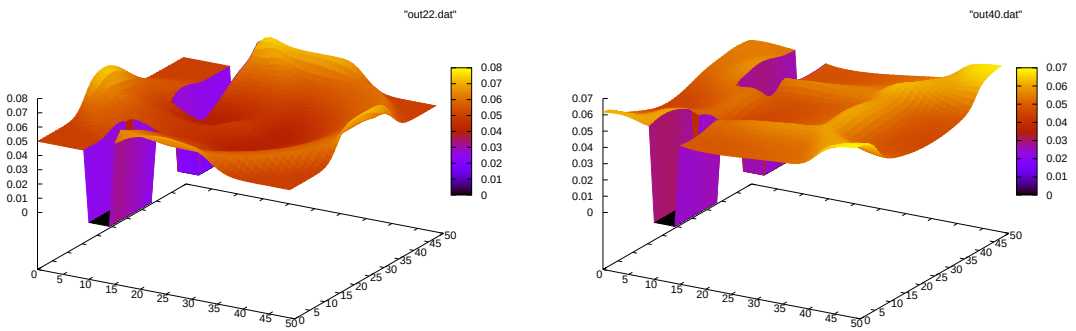
Da die Lattice Boltzmann Methode noch immer theoretisch erforscht wird und einige Stabilitätsbedingungen sich durch die gewählte Gleichgewichtsfunktion unterscheiden, garantiert das Einhalten der oben genannten Bedingungen nicht eine korrekte Berechung; es erleichtert lediglich die manuelle Wahl der korrekten Parameter, die sich von Szenario zu Szenario unterscheiden können.

Abbildung 3.3 zeigt das Ergebnis einer tatsächlichen Berechnung des in Kapitel 5 vorgestellten LBM Lösers `SolverLBMGrid` auf einem quadratischen 100×100 Höhenfeld.

Hierbei ergießt sich eine initiale Wassersäule in ein Becken (zirkulärer Dammbruch), dass auf der linken oberen Seite durch eine Wand mit einem kleinen Durchgang in der Mitte begrenzt ist. Da die Zellen, die die Wände enthalten kein Wasser enthalten, entspricht dort die Wasserhöhe dem Wert Null. Man sieht also das Negativ der eigentlichen Wand. In Zeitschritt 11 ist das Wasser auf die Wand getroffen und fließt gleichzeitig durch den Durchgang in der Wand hindurch. In Zeitschritt 22 trifft das Wasser sowohl hinter dem Durchgang als auch auf allen anderen drei Seiten gegen die äußere Wand. Im letzten hier gezeigten Zeitschritt wurde das Wasser von allen Wänden zuruckgeworfen und beginnt, sich langsam zu beruhigen.



(a) t = 0 (b) t = 11



(c) t = 22 (d) t = 40

Abbildung 3.3: Flachwassersimmulation mit der LBM

4 HONEI

4.1 Übersicht

HONEI (*hardware-oriented numerics efficently implemented*) ist eine Sammlung von Softwarebibliotheken, die einen einheitlichen Zugang für verschiedene Anwendungen aus dem Bereich des wissenschaftlichen Rechnen auf verschiedenen Hardwarearchitekturen ermöglichen. Mit der Entwicklung von HONEI wurde ursprünglich im Rahmen der Projektgruppe 512 „SmartCell: Clevere Algorithmen für den Cell Prozessor“ [42] an den Fakultäten Informatik und Mathematik der TU Dortmund begonnen. Die wichtigsten Grundideen wurden bereits in einem Artikel unter Mitwirken des Autors der vorliegenden Arbeit [11] veröffentlicht.

Im Laufe der Projektgruppe stellte sich heraus, dass der anfängliche Ansatz, eine breite Hardwareunterstützung durch Bereitstellen von stark optimierten Basisoperationen aus der numerischen linearen Algebra, aus denen dann komplexere Anwendungen erstellt werden können, nicht in der Lage war, der darauf basierenden Anwendung einen hinreichend direkten Zugang zur jeweilige Hardware zu ermöglichen. Nur durch Kenntnis der Berechnungen und des Datenflusses über die gesamte Anwendung können insbesondere redundante Speicher- und Rechenoperationen vermieden und Datenzugriffsmuster gewählt werden, die die Speicherhierachie des Systems optimal ausnutzen. So mag eine stark vereinfachte Anwendung um $a = a + b * c$ zu berechnen zunächst ein Produkt $r = b * c$ berechnen, das Ergebnis speichern und in einem zweiten Schritt das Ergebnis wieder laden und die Summe $a = r + c$ berechnen. Das Schreiben des Zwischenergebnisses r in den Speicher durch die erste Operation und das direkt im Anschluss durchgeführte erneute Laden von r durch die zweite Operation können vermieden werden, indem man die beiden Operationen zu einer komplexeren zusammenfasst. Dies ist insbesondere bei Architekturen, die keine schnellen Caches besitzen, sinnvoll, wie etwa der Cell BE oder einer Grafikkarte, aber selbst bei einer normalen CPU bringt der vorhandene Cache keinen Nutzen, wenn man sich die obigen Berechnungen als Vektoroperationen vorstellt und die Vektorgrößen als ein Vielfaches der Cachegröße annimmt.

Um dies zu ermöglichen wurde in HONEI das Konzept des Anwendungskernels eingeführt. Ein solcher Kernel fasst jeweils einen wesentlichen Teil der Berechnungen einer Anwendung zusammen und ermöglicht so die Optimierung weit über Operationsgrenzen hinweg. Gleichzeitig stellt der Kernel den kleinstmöglichen Teil der Anwendung dar, der

für jede Hardware reimplementiert und optimiert werden muss. Das eigentliche Anwendungsgerüst ruft dann diese Kernel auf und muss deshalb nicht für jede Hardware reimplementiert werden. Der Großteil von HONEI ist in C++ implementiert und nutzt sehr stark das in C++ vorhandene *Template*-Konzept und aus der *Standard Template Library* (STL) [48] bekannten Konzepte, wie zum Beispiel Iteratoren. Um die Anpassung an das jeweilige Zielsystem zu vereinfachen und den Erstellungsprozess der Bibliotheken zu automatisieren, kommt das *GNU configure and build system* (Autotools) [7] zum Einsatz. Insbesondere das skriptbasierte Erstellen der SPE Programme des Cell Backends macht hiervon sehr starken Gebrauch.

4.2 HONEI Komponenten

HONEI besteht aus einer Reihe von Bibliotheken, die aufeinander aufbauen und dadurch verschiedene Grade der Hardwareabstraktion erreichen. Abbildung 4.1 zeigt die schematische Struktur von HONEI. Auf unterster Ebene wird die direkte Schnittstelle zur jeweiligen

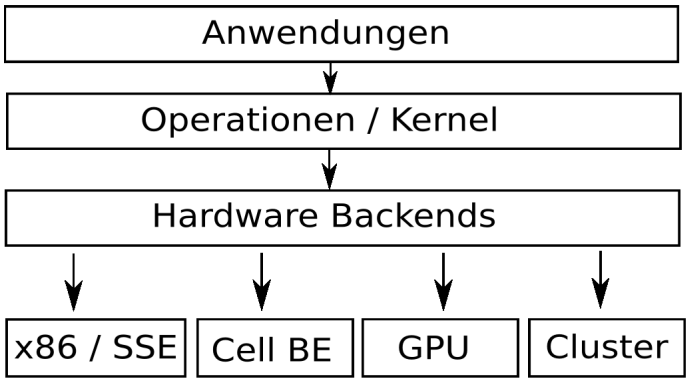


Abbildung 4.1: Struktur der HONEI Bibliotheken mit steigender Hardwarenähe von oben nach unten

Hardware durch die sogenannten *Hardware Backends* hergestellt. Das sind Bibliotheken, die die einzelnen Funktionen der entsprechenden Hardware zur Verfügung stellen, indem sie die vom jeweiligen Hardwarehersteller gelieferten Bibliotheken und hardwarespezifische Spracherweiterungen und Befehle nutzen. Auf diese setzen die Operationen und Kernel der restlichen anwendungsspezifischen Bibliotheken auf. Die Anwendungen wiederum setzen sich aus diesen Operationen und Kerneln zusammen. Durch diesen hierachischen Aufbau kann bei der Anwendungsentwicklung sehr stark von der Hardware abstrahiert entwickelt werden, während über die Operationen bis hin zu den Hardware Backends die Entwicklung immmer hardwarenäher wird.

Die nachfolgenden Abschnitte, deren Namen direkt HONEI entnommen wurden, beschreiben die HONEI Bibliotheken im Detail.

4.2.1 Util

Diese Bibliothek enthält Werkzeuge, die von allen anderen Bibliotheken verwendet werden können.

Partitionierer

Der Partitionierer wird dazu verwendet um beispielsweise einen Vektor in kleinere Teile zu zerlegen. Auf diese Weise können die erzeugten Teilvektoren auf verschiedenen Kernen einer CPU parallel bearbeitet oder über verschiedene Knoten eines Clusters verteilt werden. Um auf die verschiedenen Bedürfnisse der einzelnen Hardwarearchitekturen eingehen zu können, lassen sich eine Vielzahl von Parametern wie das *Alignment* der verschiedenen Partitionen, das heißt die Position der Daten im Speicher, die Anzahl der Partitionen und die minimale und maximale Partitionsgröße einstellen.

Profiler

Der Profiler kann eingesetzt werden, um detailierte Messungen der Ausführungszeit einzelner Programmabschnitte durchzuführen. Hierzu wurde eine einheitliche Schnittstelle entwickelt, die die verschiedenen hardwarespezifischen Techniken und Verfahren kapselt, gleichzeitig aber deren besondere Genauigkeit beibehält. Um Ausführungszeiten auf der CPU zu messen, kommt die Methode `gettimeofday` zum Einsatz, die die aktuelle Systemzeit in Mikrosekunden ausgibt. Allerdings ist die Auflösung nicht festgelegt und hängt von der verwendeten Systemuhr ab. Sie liegt gewöhnlich bei etwa einer Millisekunde und ist damit für detaillierte Messungen sehr ungenau. Auf einer SPE der Cell BE wird die Methode `spu_read_decrementer` genutzt, um die Ausführungszeit von dort ausgeführten Programmteilen zu messen. Diese gibt den Stand eines Zählers aus, der mit jedem Rechentakt um eins erniedrigt wird und ist somit sehr genau. Die Ausführungszeit auf der Grafikkarte wird mit der Methode `cudaEventElapsedTime` als verstrichene Zeit zwischen zwei vorher durch den Profiler erzeugten Ereignissen gemessen und liefert eine Auflösung von zirka $0,5$ μs. So kann zum Beispiel über die normalen Mittel des Anwendungsprogrammierers hinaus genau gemessen werden, wie lange Programmteile auf der Grafikkarte oder einer SPE ausgeführt werden.

Debug

Die Klassen `Log`, `Exception` und `Assertion` ermöglichen eine einheitliche Fehlerverfolgung und Erkennung. Hierbei entsprechen `Exception` (Ausnahme) und `Assertion` (Zusicherung) in weiten Teilen ihren bekannten Pendanten aus C++. Sie wurden allerdings so erweitert, dass sie mit einem sogenannten Kontextstack zusammenarbeiten. Im Quelltext können Sinnabschnitte durch das `CONTEXT` Schlüsselwort markiert werden, die dann während der Pro-

grammausführung erkannt werden. Auf diese Weise kann eine ausgelöste Ausnahme oder eine verletzte Zusicherung direkt anzeigen, auf welchem Wege im Programmfluß (*stacktrace*) sie erreicht wurde. Zum Beispiel werden so zur Laufzeit Fehler wie ungenügende Ressourcen oder invalide Containergrößen erkannt und lokalisiert. Gleichzeitig können zur Fehlersuche detaillierte Informationen zum Beispiel über Datentransfers zwischen SPE und PPE des Cell Prozessors ausgegeben werden. Dazu genügt es, das `LOGMESSAGE` Schlüsselwort zu verwenden, um beliebige zum entsprechenden Zeitpunkt dem Programm zur Verfügung stehende Daten auszugeben. Die Klasse `Log` sorgt nun dafür, dass diese Daten auf dem Bildschirm oder in eine Datei ausgegeben werden. Um die Fehlersuche bei mehreren ausgeführten Threads zu erleichtern, wird diese Ausgabe mit der laufenden Nummer des Threads und dem oben angesprochenen Kontextstack ausgegeben. Im Falle von Systemen mit verteiltem Speicher, sorgt die Klasse `Log` ebenfalls dafür, dass die relevanten Daten automatisch aus dem Speicher der Grafikkarte oder einer SPE mit Hilfe des jeweiligen Hardware Backends in den Hauptspeicher transferiert werden, da sie nur von dort aus ausgegeben werden können.

Thread

Die Klassen `Thread`, `Lock`, `Mutex` und `ConditionVariable` stellen eine einheitliche Schnittstelle da, um nebenläufige Programmstukturen zu erzeugen. Sie basieren in erster Linie auf den sogenannten PThreads [4] und werden in erster Linie vom Multicore Backend aber auch vom Cell Backend eingesetzt. Hierbei stellt die Klasse `Thread` eine einfache Möglichkeit zur Verfügung, eine dem Konstruktor übergebene Methode als eigenständigen Thread auszuführen und wieder zu beenden. Die Klasse `Lock` ermöglicht es mehreren Threads sich zu synchronisieren, indem ein Thread den Zugriff auf eine Ressource für alle anderen Threads sperren kann bis er selbst die Ressource wieder freigibt. Um eine Quelle von Verklemmungen, sogenannten *Deadlocks*, zwischen Threads zu vermeiden, zerstört sich eine solche Sperre nach Beenden der jeweiligen Methode selbst und gibt die Ressource wieder frei; so muss der Programmierer dies nicht manuell zu tun. Die Klasse `ConditionVariable` ermöglicht es schließlich einer Menge von Threads ohne ständiges aktives Abfragen einer Statusinformation auf ein Ereignis zu warten und erst wieder aktiv zu werden, wenn das Ereignis eingetreten ist. Auf diese Weise können beispielsweise SPEs der CELL BE oder einzelne Kerne einer Multicore CPU warten (*idle*), bis ihnen neue Arbeit zugeteilt wird, ohne das System durch aktive Anfragen (*polling, busy waiting*) zu belasten.

TypeTraits

Durch die TypeTraits sind die grundsätzlichen Operationen auf Datentypen implementiert. Dabei handelt es sich in erster Linie um optimierte Allokation, Deallokation, Kopier- und Verschiebeoperationen, die sich sowohl die Eigenarten der jeweiligen Hardware zu Nutze

machen, als auch auf die Eigenarten der jeweiligen Hardware eingehen, also zum Beispiel nur an sinnvollen Addressgrenzen ausgerichteten Speicher allokieren.

Memory Arbiter

Der Memory Arbiter verwaltet Speicherorte und Zugriffe. Er wird als Neuentwicklung im Rahmen dieser Arbeit genauer in Abschnitt 4.3.1 beschrieben.

4.2.2 Hardware Backends

Um die für jeweilige Zielarchitektur effizient und hardwarenah programmmieren zu können, ist es meist unumgänglich, die gewünschten Funktionen in einer hardwarespezifischen Sprache zu formulieren und den jeweiligen Algorithmus an die Hardware anzupassen. Aus diesem Grund sind die optimierten Operationen und Kernel zusammen mit den Steuer- und Kontrollbibliotheken in einzelnen Hardware Backends gekapselt.

SSE

Das SSE Backend nutzt die von Intel zusammen mit den jeweiligen SSE Versionen entwickelten *Intrinsics* [28], um die SSE Register und Recheneinheiten direkt zu programmieren. Durch die verhältnismässig hohe Maschinennähe wird es möglich, Speicherzugriffe zu optimieren und die vorhandenen Register optimal auszunutzen. Da das durch eine Operation zu lösende Problem bekannt ist, können durch manuelle Optimierung bessere Ergebnisse erreicht werden, als dies die automatische Compileroptimierung zu leisten vermag.

Diese Intrinsics sind neue Schlüsselworte, die alle gängigen C++ Compiler unterstützen. Listing 4.1 zeigt die Operation $a = a + b * c$ auf drei Vektoren mit je vier Elementen. Zunächst werden je vier Elemente des Vektors `b` in `m1` und vier Elemente des Vektors `c` in

```
float * a = {0, 1, 2, 3};
float * b = {0.0, 0.1, 0.2, 0.3};
float * c = {1.0, 1.1, 1.2, 1.3};
__m128 m1, m2;

m1 = _mm_load_ps(b);
m2 = _mm_load_ps(c);
m2 = _mm_mul_ps(m1, m2);
m1 = _mm_load_ps(a);
m1 = _mm_add_ps(m1, m2);
_mm_store_ps(a, m1);
```

Listing 4.1: SSE Intrinsics

`m2` geladen. `m1` und `m2` stellen für den Compiler die SSE Register dar und sind 128 Byte groß, können also vier Variablen vom Typ `float` speichern. Dementsprechend steht das Suffix `_ps` das den Intrinsics nachgestellt ist für *packed single*; sie rechnen also auf vier

Variablen mit einfacher Genauigkeit (*single*), die zusammen in einem Register gespeichert sind (*packed*). Das Prefix `_mm_` ist historisch zu erklären und bezieht sich noch auf die erste von Intel entwickelte Vektorisierungstechnik MMX. Anschließend werden die vier Werte in `m1` in einem Rechenschritt mit den vier Werten in `m2` multipliziert und die Ergebnisse wieder in `m2` gespeichert. Danach werden vier Elemente des Vektors `a` in `m1` geladen und mit denen `m2` addiert. Das in `m1` gespeicherte Ergebnis wird schließlich zurück in den Vektor `a` geschrieben.

Da ein üblicher x86 Prozessor mindestens acht SSE Register besitzt, ist es bei größeren Vektoren üblich, soviele Elemente gleichzeitig zu verarbeiten, wie Registerplatz zur Verfügung steht. Im obigen Beispiel könnten die Befehle also vier Mal hintereinander auf verschiedenen Daten und Registern ausgeführt werden, um alle acht SSE Register zu nutzen. Eine Schleife auf diese Art abzurollen, wird *loop unrolling* genannt und kann den Gesamtdurchsatz der Operation stark erhöhen, da größere Datenblöcke gleichzeit transferiert und verarbeitet werden können.

Multicore

Das Multicore Backend baut auf den oben genannten Thread Klassen auf und ermöglicht eine effiziente Ausführung von Threads auf verschiedenen Kernen einer und mehrerer CPUs. Innerhalb dieser Threads können wiederum Operationen der anderen Backends ausgeführt werden. Eine übliche Anwendung ist zum Beispiel, auf mehreren Kernen Operationen des SSE Backends auszuführen. Es ist aber auch denkbar, mit geeigneter Hardware zwei Operationen des CUDA Backends parallel auf zwei Grafikkarten auszuführen. Hierbei kann mit Hilfe sogenannter „Threadaffinität“ im Detail gesteuert werden, wo genau ein spezifischer Thread ausgeführt wird, um etwa Speicherzugriffe auf NUMA Architekturen zu optimieren. Gleichzeitig stellt die Verwendung eines *Threadpools* [36] sicher, dass auch bei kleineren Problemgrößen die Verwaltung der Threads die Ausführungszeit nicht wesentlich beeinflusst. Ein Threadpool verwaltet eine Menge von Threads, die dauerhaft existieren, aber nur aktiv werden, wenn sie das Signal (zum Beispiel durch eine `ConditionVariable`) bekommen, etwas zu berechnen. Ansonsten deaktivieren sie sich und beanspruchen keinerlei Ressourcen. Dadurch entfällt der ansonsten nötige Verwaltungsaufwand zur Erstellung und Zerstörung der einzelnen Threads. Zusätzlich werden Mittel geboten, um nicht blockierend auf die Ausführung der aufgegebenen Arbeit zu warten, sodass die Anwendung mit anderen Berechnungen fortfahren kann. Dies wird dadurch realisiert, dass die Anwendung zu jedem Zeitpunkt den Status der an das Multicore Backend delegierten Aufgaben erfragen kann und abhängig von diesem Status neue Aufgaben in Auftrag geben oder eigene davon unabhängige Berechnungen durchführen kann.

CUDA

Das CUDA Backend bildet die Schnittstelle zwischen HONEI und einer mit CUDA programmierbaren Grafikkarte [35] Es wird genauer in Abschnitt 4.3.2 beschrieben.

MPI

Das MPI Backend übernimmt die Organisation und Steuerung der MPI-gestützten Kommunikation und vereinfacht diese soweit wie möglich. Es wird genauer in Abschnitt 4.3.3 beschrieben.

Cell

Da die grundsätzliche Programmierung der Cell BE deutlich aufwändiger ist, als etwa einen Algoritmus in SSE Intrinsics zu überführen, ist das Cell Backend deutlich umfangreicher als die anderen Hardware Backends. Zunächst muss auf Seite der PPE für jede einzusetzende SPE ein korrespondierender Thread gestartet werden. Diese sind wiederum dafür zuständig, die eigentlichen Programme auf den SPEs mit Hilfe der von IBM zur Verfügung gestellten Bibliothek *libspe2* [26] und des IBM Cell BE SDK [41] zu starten. Um nicht für jede Operation erneut ein Programm auf einer SPE starten zu müssen, laufen dort dauerhaft sogenannte SPE Kernel, die je nach den an sie geschickten Instruktionen unterschiedliche Berechnungen ausführen. Ein Scheduler sorgt dafür, dass die einzelnen Instruktionen an freie SPEs versandt werden und dass dort ein Kernel ausgeführt wird, der diese Instruktion auch umsetzen kann. Den eigentlichen Datentransfer per DMA stößt dann die jeweilige SPE selbstständig an, da dies auf der Architektur für deutlich höhere Transferraten sorgt. Gleichzeitig kann die SPE Datentransfers und Berechnungen überlappen, indem sie durch sogenanntes *double buffering* auf einigen Daten rechnet, während die vorherigen Daten zurücktransferiert und die für die nächste Berechnung benötigten Daten hertransferiert werden. Um auch komplexere Aufgaben zu lösen, besteht für die SPEs die Möglichkeit DMA Listentransfers zu nutzen oder auch untereinander Daten auszutauschen.

Da wie weiter oben erwähnt eine SPE nur sehr wenig lokalen Speicher besitzt, in dem sowohl die Daten als auch die Instruktionen gespeichert werden müssen, enthält ein SPE Kernel immer nur Programmcode von einigen wenigen Operationen. Für verschiedene Anwendungen stehen deshalb verschiedene Kernel zur Verfügung, die exakt die notwendigen Operationen unterstützen, um ein notwendiges Austauschen der SPE Kernel auf den SPEs möglichst zu vermeiden. Um die Erstellung dieser Kernel zu vereinfachen, werden diese automatisch auf Basis einer einzelnen Textdatei erstellt, die die für den Kernel gewünschten Operationsnamen enthält. So steht maximal viel lokaler Speicher für die zu bearbeitenden Daten zur Verfügung.

Die eigentlichen auf der SPE ausgeführten Operationen sind schließlich in Intrinsics, ganz ähnlich denen, die zur SSE Programmierung zum Einsatz kommen, verfasst. Listing 4.2

zeigt den notwendigen SPE Quelltext, um die schon in Listing 4.1 behandelte Operation $a = a + b * c$ auf drei Vektoren mit je vier Elementen zu berechnen. Da durch das Cell

```
// DMA Transfer der Daten vom Hauptspeicher in den LS der SPE.
a = spu_madd(b, c, a);
// DMA Transfer der Daten vom LS der SPE in den Hauptspeicher.
```

Listing 4.2: Cell SPE Intrinsics

Backend vor Aufruf der Operation auf der SPE die notwendigen Daten in den Local Store der SPE und nach Abschluss der Operation wieder zurück in den Hauptspeicher transferiert werden, muss die Operation nur die eigentliche Berechnung durchführen. Da die SPE noch mehr als die SSE Einheit der CPU für die vektorisierte Bearbeitung von Daten optimiert ist, existiert die Operation $a = a+b*c$ direkt als Maschinenbefehl, der in einem Takt ausgeführt werden kann. Es ist also nicht notwendig, Addition und Multiplikation voneinander zu trennen.

4.2.3 Unittest

Die Bibliothek zur Erzeugung von Unittests [22] stellt Werkzeuge zur Verfügung, um alle Container, Operationen, Kernel, Anwendungen und Bibliotheken von HONEI automatisiert auf korrekte Funktionsweise zu überprüfen. Hierdurch wird zum Einen die Entwicklung neuer Programmteile erleichtert, da die korrekte Funktionsweise direkt geprüft werden kann. Zum Anderen werden Softwareregressionen durch umfangreiche Regressionstests [58] erkannt und analysiert.

4.2.4 Benchmark

Das Benchmark Framework ermöglicht eine einheitliche Erfassung und Analyse der Laufzeiten und Leistungsdaten von Operationen, Kerneln und Anwendungen auf allen genutzten Hardwarearchitekturen. Hierzu werden für alle Operationen und Kernel die durch sie ausgeführten Transfers und Fließkommaoperationen erfasst und akkumuliert. Somit entfällt beim Erstellen einer Anwendung eine aufwändige Analyse dieser Kennzahlen auf Anwendungsniveau. Gleichzeitig ermöglicht eine periodische Erfassung der Leistungsdaten aller HONEI Komponenten die direkte Identifikation negative Entwicklungen.

4.2.5 LA

Diese Bibliothek stellt eine Vielzahl von Containern und Operationen aus dem Bereich der linearen Algebra zur Verfügung und bildet damit die Grundlage aller anderen anwendungsorientierten Bibliotheken.

Container

Im Folgenden werden die in HONEI vorhandenen Container kurz aufgezählt. Allen Containern gemein ist, dass sie durch die Verwendung von Templates jeden gewünschten Datentyp speichern können; also zum Beispiel `float`, `double`, `bool` oder `unsigned long`. Zu beachten ist, dass eine echte datenunabhängige Kopie nur explizit über die Methode `copy()` erzeugt werden kann. In jedem anderen Fall werden nur die Containerobjekte kopiert, aber auf eine teure und häufig ungewollte Kopie aller Daten wird verzichtet. Außerdem stellen alle Container die Methoden `lock()` und `unlock()` zur Verfügung, um dem Memory Arbiter einen exklusiven Lese- oder Schreibzugriff zu signalisieren.

Vektoren

- `DenseVector`: Gewöhnlicher Vektor, der als kontinuierliches eindimensionales Array seiner Elemente abgespeichert ist.
- `SparseVector`: Dieser Vektor enthält typischerweise nur wenige Nicht-Null-Elemente, die explizit mit ihren Positionen im Vektor gespeichert werden.

Matrizen

- `DenseMatrix`: Gewöhnliche zweidimensionale Matrix, deren Elemente in einem eindimensionalen Array zeilenweise abgespeichert sind.
- `SparseMatrix`: Diese Matrix enthält typischerweise nur wenige Nicht-Null-Elemente, so dass ihre Zeilen jeweils als `SparseVector` abgespeichert werden können.
- `SparseMatrixELL`: Diese Matrix basiert auf dem ELLPACK Format, wie es in der ITPACK Bibliothek [20] vorgeschlagen wird. Hierbei werden die Nicht-Null-Elemente jeder Zeile zusammen mit ihrer jeweiligen Position in einer `DenseMatrix` gespeichert.
- `BandedMatrix`: Quadratische Bandmatrix. Hierbei werden die Daten in Form von Diagonalbändern als `DenseVector` abgespeichert.
- `BandedMatrixQ1`: Bandmatrix, die exakt neun diagonale Bänder an spezifischen Positionen enthält. Sie findet zum Beispiel Anwendung bei der Berechnung des Poisson Problems [11].

Die im Rahmen dieser Arbeit entwickelten LBM Löser verwenden ausschließlich die Container `DenseVector` und `DenseMatrix`.

Operationen

Die Menge der vorhandenen linearen Algebra Operationen reicht von Reduktionen wie der Norm oder dem Skalarprodukt über elementweise Produkte, Summen und Differenzen bis hin zu Matrix-Vektor und Matrix-Matrix Produkten auf den verschiedenen Containern.

4.2.6 Math

Diese Numerikbibliothek stellt verschiedene iterative Löser für lineare Gleichungssysteme sowie Defekt-, Interpolations- und Approximationsoperationen zur Verfügung. Im einzelnen sind dies ein Löser gemäß dem Verfahren der konjugierten Gradienten (CG) und ein Löser nach dem Jacobi Verfahren. Da diese Löser templatisiert sind, lassen sich darauf aufbauend neue Löser wie ein mit Jacobi vorkonditionierter CG-Löser erstellen. Mit Hilfe dieser Löser wurde ein Mehrgitterlöser entwickelt, der das Mehrgitterverfahren [5] einsetzt, um das Problem von einem feinen Gitter rekursiv auf ein grobes Gitter zu reduzieren, es dort zu lösen und unter Beachtung des berechneten Fehlers wieder auf das feine Gitter zurückzukehren. Da alle Löser ausserdem bezüglich der verwendeten Genauigkeit templatisiert sind, lassen sich sehr einfach neue Löser erstellen, die zum Beispiel gemischt genaue Methoden [18] verwenden.

4.2.7 LBM

Kern dieser Bibliothek sind zwei verschiedene Flachwasserlöser auf Basis der Lattice-Boltzmann Methode, wie sie in Kapitel 3 beschrieben wurde. Der erste Löser `SolverLABSWE` wird in Abschnitt 4.3.4 näher beschrieben Der zweite Löser `SolverLBMGrid` sowie seine Datenstrukturen und Komponenten werden im Detail in Kapitel 5 beschrieben.

4.2.8 SWE

Diese Bibliothek beinhaltet einen voll expliziten 2D Flachwasserlöser. Er benutzt das Relaxationsschema, das von Delis und Katsaounis vorgeschlagen wurde [9], mit einer finiten Differenzendiskretisierung von Genauigkeit zweiter Ordnung im Raum und einem Runge Kutta Zeitschrittmechanismus. Der Löser diente als Beispielapplikation im Rahmen der Projektgruppe [42] und eines bereits veröffentlichten Artikels [11].

4.3 Neuentwicklungen

Im Folgenden sollen einige der oben genannten Komponenten genauer beschrieben werden, die schwerpunktmäßig für diese Arbeit neu entwickelt wurden.

4.3.1 Memory Arbiter

Im Gegensatz zu Systemen mit geteiltem Speicher, wie es bei Ein- oder Mehrkernrechnern der Fall ist, sind die Arbeitsspeicher etwa von Grafikkarte und Hauptprozessor komplett physikalisch voneinander getrennt. In der Regel muss davon ausgegangen werden, dass ein Datentransfer zwischen den getrennten Arbeitsspeichern um ein Vielfaches teurer ist, als etwa der Zugriff eines Prozessorkerns auf Daten im Speicherbereich eines anderen Kerns.

Ohne explizite Speichertransfers kann weder die CPU auf Daten der GPU zugreifen, noch ist der umgekehrte Zugriff möglich. Dieser Sachverhalt trifft zwar auch auf PPE und SPE in der Cell BE zu, dort wird aber auf Grund des sehr kleinen lokalen Speichers einer SPE nach einer Operation der Inhalt des SPE Speichers direkt wieder in den Arbeitsspeicher der PPE zurückgeschrieben. Der lokale Speicher einer Grafikkarte hingegen kann mit einer Größe von ein oder zwei GByte durchaus alle zur Berechnung relevanten Daten über die komplette Anwendungslaufzeit halten.

Um nun die Konsistenz der Daten in allen Speichersystemen zu gewährleisten, gibt es grundsätzlich zwei Herangehensweisen. Naheliegend ist die Strategie zu garantieren, dass ein Datum immer nur in einem Speicher gleichzeitig existiert. Dadurch ist die Konsistenz der Daten trivial gegeben. Allerdings birgt diese Strategie einige Einschränkungen. So kann immer nur die Recheneinheit mit dem Datum arbeiten, in deren Speicher es sich gerade befindet. Will eine andere Recheneinheit das Datum auch nur lesen, so muss in jedem Fall ein Datentransfer zwischen den Speichern stattfinden. Zusätzlich müssen die Daten im Quellspeicher gelöscht werden, was sehr wahrscheinlich zu einem späteren Zeitpunkt einen erneuten Transfer in die Gegenrichtung herbeiführen wird.

Diese Einschränkungen führten zu der für HONEI entwickelten Strategie. Wird es notwendig Daten von einem Speicher in einen anderen zu transferieren, so wird im Zielspeicher eine eigene Kopie angelegt. Die Daten im Quellspeicher bleiben davon zunächst unberührt.

Der Klasse `MemoryArbiter` kommt nun die Aufgabe zu, die nun nicht mehr triviale Konsistenz der Daten zu gewährleisten. Möchte eine Operation auf Daten zugreifen, so muss sie dies der Klasse signalisieren und sowohl den gewünschten Speicher (zum Beispiel Hauptspeicher oder Grafikkartenspeicher) als auch die Art des Zugriffs angeben. Der `MemoryArbiter` verwaltet diese Zugriffe in einer internen Datenbank und sorgt dafür, dass die angeforderten Daten aktuell und im gewünschten Speicher vorliegen. Mögliche Zugriffsarten sind

- `lm_read_only`: nur lesender Zugriff,
- `lm_write_only`: nur schreibender Zugriff und
- `lm_read_and_write`: erst lesend und später auch schreibender Zugriff.

Bei nur lesendem Zugriff ist es nicht notwendig, die dafür in einen bestimmten Speicher transferierten Daten zurück zu transferieren, da sie nicht verändert wurden. Bei nur schreibendem Zugriff ist zunächst gar kein Speichertransfer notwendig, da die Daten im Zielspeicher erst erzeugt werden. Lediglich bei der Zugriffsart `lm_read_and_write` ist es notwendig die Daten erst in den Zielspeicher zu kopieren und später eventuell wieder zurück zu transferieren. Dadurch werden Daten in einem Speicher erst gelöscht, falls sie durch Schreibzugriffe in einem anderen Speicher ungültig werden. Durch diese Unterscheidung der Zugriffsarten können sehr viele Speichertransfers eingespart und nicht notwendige Datentransfers

vermieden werden, die eine Anwendung sehr stark verlangsamen könnten. Allerdings ist es erforderlich, gewisse Einschränkungen bezüglich des gleichzeitigen Zugriffs auf ein und dasselbe Datum einzuhalten. So kann eine Operation nur lesend auf ein Datum zugreifen, wenn kein anderer schreibender Zugriff mehr stattfindet. Außerdem können keine zwei Operationen gleichzeitig schreibend auf ein Datum zugreifen. Lediglich der nur lesende Zugriff ist durch mehrere Operationen gleichzeitig möglich. Da Operationen blockiert werden bis der Datenzugriff freigegeben ist, ist es zwingend notwendig, dem `MemoryArbiter` umgehend das Ende eines Datenzugriffs zu signalisieren. Falls kein Speichertransfer notwendig ist, wird eine Operation durch Verwendung der Klasse `MemoryArbiter` nicht verlangsamt, da im Grunde nur ein Eintrag in der internen Datenbank abgefragt wird. Lediglich die tatsächlich ausgeführten Datentransfers beeinflussen die Laufzeit meßbar.

Damit die Klasse `MemoryArbiter` die notwendigen Allokations- und Transferoperationen zwischen ganz unterschiedlichen Speicherarchitekturen durchführen kann, wurde mit der Klasse `MemoryBackendBase` eine allgemeine Schnittstelle geschaffen, um diese Aufgaben zu vereinheitlichen. Jedes Hardware Backend stellt eine eigene Implementierung dieser Schnittstelle zur Verfügung, um auf seinem eigenen Speicher zu arbeiten, und ermöglicht es zusätzlich, ein Datum mittels einer eindeutigen Identifikationsnummer in einem spezifischen Speicher aufzufinden, falls es dort bereits vorliegt. Zur Laufzeit registrieren sich diese Implementierungen dann beim `MemoryArbiter` und erlauben diesem so flexibel mit jeder möglichen Kombination von Speichersystemen (zum Beispiel ein Arbeitsspeicher und zwei Grafikkartenspeicher von zwei verschiedenen Grafikkarten) zu arbeiten.

4.3.2 CUDA Backend

Die von NVIDIA entwickelte Metaarchitektur CUDA (*compute unified device architecture*) [37] [15] ist eine Technik zur Beschleunigung wissenschaftlicher und technischer Berechnungen. CUDA wurde zusammen mit der GeForce 8 Serie [35] entwickelt und im November 2006 veröffentlicht. Seitdem unterstützt jede neue Grafikkarte von NVIDIA auch CUDA.

Grundlage der CUDA Architektur ist die geräteunabhängige Assemblersprache PTX (*parallel thread execution*). Auf sie können verschiedene Programmiersprachen und -schnittstellen aufsetzen, um CUDA kompatible Grafikkarten zu nutzen.

Um CUDA durch HONEI zu nutzen, wird die Sprache „C for CUDA“ [38] eingesetzt. Dabei handelt es sich um eine Erweiterung der Sprache C, die es ermöglicht, Programme für eine Grafikkarte zu schreiben und zusätzlich auf die Ressourcen der Grafikkarte zuzugreifen. Dadurch stellt sie einen wesentlichen Fortschritt gegenüber früheren GPGPU Ansätzen dar, die mit Hilfe von OpenGL oder Direct3D, also Bibliotheken, die zur reinen Grafikbearbeitung entwickelt wurden, arbeiteten und daher nicht auf alle Funktionen der Grafikkarte direkt zugreifen konnten. Um Programme die in „C for CUDA“ geschrieben wurden zu übersetzen, wird der CUDA C Compiler (`nvcc`) eingesetzt. Dieser erzeugt so-

wohl das auf der Grafikkarte auszuführende Programm als auch die notwendigen Routinen, um dieses Programm auf der Grafikkarte auszuführen.

Da, wie in Abschnitt 2.2 aufgezeigt, alle Threads einer Operation, die auf der Grafikkarte ausgeführt wird, jeweils denselben Programmcode ausführen, beschreibt ein typisches CUDA Programm nur die serielle Arbeit eines solchen Threads. Die eigentliche starke Parallelität wird dann durch die große Anzahl dieser Threads in allen Blöcken im Gitter erzeugt. Um die schon bekannte Rechnung $a = a + b * c$ auf den Vektoren a, b, und c durchzuführen, ist der in Listing 4.3 gezeigte CUDA Kernel notwendig. Hierbei entspricht

```
__global__ void cuda_kernel(float * a, float * b, float * c)
{
    unsigned long idx = (gridDim.x * blockDim.x) +
        (blockDim.x * blockIdx.x) + threadIdx.x;
    a[idx] = a[idx] + b[idx] * c[idx];
}
```

Listing 4.3: Beispiel eines CUDA Kernel

`idx` der laufenden Nummer des ausgeführten Threads, die sich aus der Position innerhalb des Threadblocks und der Position dieses Blocks innerhalb des Gitters ergibt. Um nun diese Berechnung auf allen Elementen der Vektoren durchzuführen, muss zunächst eine Startkonfiguration erstellt werden und dann die entsprechende Anzahl von Threads auf der Grafikkarte gestartet werden. Listing 4.4 zeigt den notwendigen Quelltext um eindimensionale Threadblöcke der Größe `blocksize` zu einem eindimensionalen Gitter der minimalen Größe `size` zusammenzusetzen. Zuletzt werden die CUDA Kernel Threads entsprechend dieser Konfiguration gestartet und ausgeführt.

```
    dim3 grid;
    dim3 block;
    block.x = blocksize;
    grid.x = (unsigned)ceil(sqrt(size/(double)block.x));

    cuda_kernel<<<grid, block>>>(a, b, c);
```

Listing 4.4: CUDA Kernelkonfiguration

Zu beachten ist, dass es sich bei `a`, `b`, und `c` um Zeiger auf Daten im Grafikkartenspeicher handelt. Um diese dorthin und auch wieder zurück in den Hauptspeicher transferieren zu können, bietet das CUDA Backend gemäß der `MemoryBackendBase` Schnittstelle Methoden zur Allokation und zum Transfer von Daten. Außerdem ermöglicht es unter Verwendung der CUDA Methode `cudaGetLastError` Fehler während der Ausführung von Programmen auf der Grafikkarte zu erkennen.

4.3.3 MPI Backend

Das MPI Backend bietet eine templatisierte C++ Schnittstelle für die in C verfassten MPI Funktionen [21].

Ein Programm das auf einem Cluster ausgeführt wird, wird gleichzeitig auf jedem einzelnen Knoten des Clusters ausgeführt. Jede einzelne Programminstanz wird nun durch eine laufende Nummer identifiziert, die es jedem Knoten ermöglicht einem unterschiedlichen Ausführungspfad zu folgen. In dem für diese Arbeit gewählten Kommunikationsmodell gibt es einen sogenannten „Master“, der zentrale Verwaltungsaufgaben übernimmt und die restlichen Knoten, „Slave“ genannt, mit Arbeit versorgt. Zum Programmstart werden also alle Programme bis auf den Master darauf warten, von diesem die notwendigen Daten zu erhalten. Nachdem die so angestoßenen verteilten Berechnungen beendet sind, werden die Teilergebnisse wieder zurück an den Master zur Weiterverarbeitung geschickt.

Grundsätzlich kann zwischen zwei verschiedenen Arten der Übertragung unterschieden werden. Zum Einen existiert die blockierende Kommunikation, das heißt, Sender und Empfänger werden in ihrer Programmausführung blockiert, bis der Transfer abgeschlossen ist. Dies kann nützlich sein, um dafür zu sorgen, dass mehrere Knoten synchronisiert werden. Zum Anderen gibt es die nichtblockierende Kommunikation, bei der Sender und Empfänger nach Anstoßen des Datentransfers in ihrem jeweiligen Programm fortfahren können. Über gesonderte Methoden kann anschließend der Status der Übertragung abgefragt werden. Dies ermöglicht es, Datentransfers und Berechnungen zu überlappen und somit die Latenz, die durch die relativ langsame Netzwerkanbindung an den Rest der Knoten entsteht, zu verstecken. Zusätzlich ist es möglich, zum Beispiel zur Initialisierung der Slave-Knoten, eine Nachricht an eine Menge von Knoten zu verschicken oder von mehreren Knoten eine Nachricht zu empfangen. Dies entspricht den MPI Methoden `MPI_Send` und `MPI_Recv` zur blockierenden Kommunikation, `MPI_Isend` und `MPI_Irecv` zur nichtblockierenden Kommunikation und `MPI_Bcast`, um mit mehreren Knoten zu kommunizieren. Zusätzlich sind für Verwaltungsaufgaben die Methoden `MPI_Init`, `MPI_Finalize`, `MPI_Comm_size` und `MPI_Comm_rank` notwendig.

Auf Grund seiner Aufgaben bezüglich des Datentransfers ist das MPI Backend eng mit der Klasse `MemoryArbiter` verbunden, um die Konsistenz der verschickten Daten zu gewährleisten. So ist es bei geeigneter Hardware etwa möglich, dass nicht die CPUs sondern die Grafikkarten in den einzelnen Knoten zur Berechnung herangezogen werden. Dies führt dann zu einem Transfer von Daten aus dem Grafikkartenspeicher in den Hauptspeicher, durch die Netzwerkschnittstellen in den Hauptspeicher eines anderen Knoten und von da wiederum in den Grafikkartenspeicher des zweiten Knoten.

4.3.4 SolverLABSWE

Der Flachwasserlöser `SolverLABSWE`) orientiert sich an dem in Fortran verfassten LBM Lösers von Zhou [60] und basiert damit auf der in Kapitel 3 vorgestellten Lattice Boltzmann Methode mit den vier Hauptteilen

- *equilibrium distribution*
- *collide and stream*
- *update velocity directions*
- *extraction.*

Ziel bei der Erstellung des Lösers war es, mit seiner Hilfe die Lattice Boltzmann Methode in der Praxis zu verstehen und auf Möglichkeiten von Vektorisierung und Parallelisierung hin zu untersuchen und mit den daraus gewonnenen Erkenntnissen den in Kapitel 5 vorgestellten Löser `SolverLBMGrid` zu entwickeln. Gleichzeitig dient er als Referenzimplementierung hinsichtlich der numerischen Ergebnisse des zweiten Lösers.

Da, wie in Kapitel 5 erläutert, die von diesem Löser verwendete Matrixrepräsentation der Daten zu diesem Zweck nicht geeignet ist, wurde er nur für normale CPUs auf Basis der linearen Algebra Komponenten von HONEI implementiert. So lässt sich die Extraktion der Wasserhöhe

$$h(\mathbf{x}, t) = \sum_{\alpha} f_{\alpha}(\mathbf{x}, t) \tag{4.1}$$

durch eine Folge von Matrixsummen realisieren, die in Listing 4.5 zu sehen ist. Da jede

```
h =  Sum::value(h, f_0);
h =  Sum::value(h, f_1);
h =  Sum::value(h, f_2);
h =  Sum::value(h, f_3);
h =  Sum::value(h, f_4);
h =  Sum::value(h, f_5);
h =  Sum::value(h, f_6);
h =  Sum::value(h, f_7);
h =  Sum::value(h, f_8);
```

Listing 4.5: SolverLABSWE Extraktion der Wasserhöhe

Matrixsumme alle Elemente der Matrix `h` erst lädt und dann wieder speichert bevor die nächste Summeation die Matrix `h` wieder lädt, erzeugt diese Implementierung relativ viele unnötige Transferoperationen.

Listing 4.6 zeigt einen Ausschnitt aus dem *equlibrium distribution* Schritt, der

$$f_0^{\mathrm{eq}} = h - \frac{5gh^2}{6e^2} - \frac{2h}{3e^2} u_i u_i \tag{4.2}$$

berechnet. Die zweidimensionalen Matrizen `f_eq_0`, `h`, `u` und `v` besitzen Methoden um ihre Zeilen- und Spaltenanzahl abzurufen. Außerdem kann mit dem `operator(i, j)` auf das Element in der Zeile `i` und der Spalte `j` zugegriffen werden. Hier wird deutlich wie nah sich diese Implementierung an die Vorgaben der mathematischen Formulierung hält.

```
for(unsigned long i(0); i < h.rows(); ++i)
{
    for(unsigned long j(0); j < h.columns(); ++j)
    {
        f_eq_0(i,j) = h(i,j) -
            ((5. * g * h(i,j) * h(i,j)) / (6. * e * e)) -
            ((2. * h(i,j)) /(3. * e * e) * (u(i,j) * u(i,j) + v(i,j) * v(i,j)));
    }
}
```

Listing 4.6: SolverLABSWE *equilibrium distribution* in Richtung 0

Sehr wohl konnten aber verschiedene Kompositionen von Anwendungskerneln und ihr Einfluss auf die numerische Stabilität des Lösers getestet werden. Dabei stellte sich heraus, dass die oben gegebene Unterteilung der LBM in vier unabhängige Teile nicht mehr sinnvoll weiter zusammengefasst werden kann. Die Teile *equilibrium distribution* und *collide and stream* unterscheiden sich in ihrem Zugriffsmuster sehr stark von einander, da *equilibrium distribution* elementweise auf den Matrizen rechnet wohingegen *collide and stream* alle acht Nachbarelemente für die Berechnung benötigt. Zur Berechnung des *update velocity directions* Schritt muss der *collide and stream* Schritt abgeschlossen sein, da dort zur Reflektion alle Ergebnisse des vorherigen Schrittes vorliegen müssen. Abschließend kann der *extraction* Schritt erst ausgeführt werden, wenn alle notwendigen Reflektionen berechnet wurden, da diese nach keinem festen Muster über die Matrizen iterieren und somit nicht sicher bekannt ist, wann ein Element nicht mehr verändert werden wird.

Um später einen Vergleich mit anderen Lösern zu ermöglichen, die nicht die Flachwassergleichungen sondern die Navier Stokes Gleichungen verwenden, wurde zusäztlich ein zweites *equilibrium distribution* Modul implementiert, das genau die letzgenannten Gleichungen verwendet. Aufgrund der modularen Struktur sowohl von HONEI als auch der LBM muss der Rest des Lösers zu diesem Zweck nicht verändert werden. Wenn nicht explizit erwähnt, ist in dieser Arbeit allerdings immer von den Modulen zur Berechnung der Flachwassergleichung die Rede.

4.3.5 Visualisierung

Um die berechneten Ergebnisse der LBM Löser visuell darstellen zu können, wurde eine grafische Applikation entwickelt, die das simulierte Wasser in Echtzeit anzeigen kann. Hierbei kommt die Bibliothek QT [45] zum Einsatz, die die betriebssystemunabhängige Programmierung grafischer Benutzeroberflächen ermöglicht. Um die eigentliche Grafik in

dem mit QT erzeugten Fenster zu generieren, wird die Grafikbibliothek OpenGL [47] verwendet, deren Ausgabe sich direkt in ein QT Fenster einbetten lässt.

Mit Hilfe dieser Applikation lassen sich verschiedene Szenarios simulieren, die aus einer zentralen Szenariodatenbank ausgeählt werden können. Zusätzlich kann das zu verwendende Hardware Backend, soweit die Hardware es unterstützt, ausgewählt werden. Es ist möglich die Simulation jederzeit zu pausieren oder neu zu starten. Um das Verhalten des Lösers an kritischen Rändern visuell genauer zu untersuchen, kann die Ansicht mit der Maus frei rotiert, bewegt und vergrößert werden, etwa um eine Stelle genauer zu betrachten. Durch eine sehr allgemeine Schnittstelle ist es möglich auch die Berechnungen andere Löser mit dieser Applikation zu visualisieren.

Abbildung 4.2 zeig einen Screenshot der Applikation.

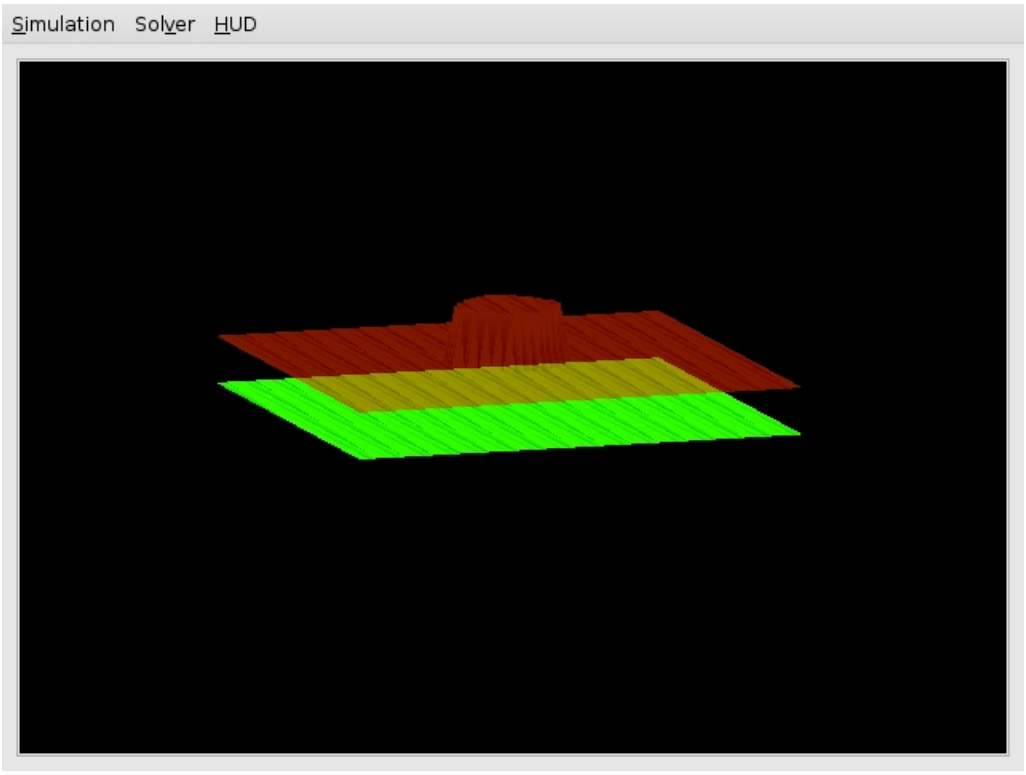


Abbildung 4.2: Screenshot des Visualisierers

Hierbei ist über einem Bodenprofil (grün) ein Fluid (rot) mit einer zentralen Fluidsäule zu sehen. Über die oberen Menüs können Simulationseinstellung, Lösereinstellungen und Anzeigeeinstellungen vorgenommen werden.

5 HONEI LBM Grid

Im Gegensatz zu dem in Abschnitt 4.3.4 vorgestellten Löser `SolverLABSWE`, der sich sehr stark an die vorgegebene Implementierung von Zhou [60] hält und damit keinerlei parallele oder hardwareoptimierte Berechnungen unterstützt, wurde der im Folgenden vorgestellte Flachwasserlöser `SolverLBMGrid` mit genau diesen Zielen entwickelt. Gleichzeitig wurde der Funktionsumfang des Lösers beispielsweise um sogenannte *Obstacles* erweitert, das heißt Zellen im Rechengebiet, die kein Fluid enthalten können.

5.1 Datenstrukturen

Die vier hier vorgestellten Datenstrukturen enthalten alle zur Berechnung notwendigen Informationen und Daten in einer Form, die es erlaubt damit parallel und hardwarebeschleunigt zu arbeiten.

5.1.1 Grid

Die Klasse `Grid` stellt die Ein- und Ausgaben des Lösers dar. Sie enthält in erster Linie das Höhenfeld, das heißt eine Matrix, die die Wasserhöhe der Fluide in jeder Gitterzelle repräsentiert.

Ausserdem kann man durch eine "Obstaclematrix" steuern, in welchen Zellen sich überhaupt Wasser befinden kann. Diese enthält nur die Werte 0 an den Stellen, an denen es fluide Partikel gibt und 1 an den Stellen, die keine fluiden Partikel enthalten können. Um zum Beispiel in der Mitte eines 7×9 Höhenfeldes eine Säule zu modellieren, um die Wasser herumströmen kann, würde man eine Obstaclematrix der Form

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

verwenden. Dies führt zu der in Abbildung 5.1 gezeigten von Wasser umgebenen Säule. Da ein rechteckiges Gitter verwendet wird, kann die Säule auch auf einem sehr feinen Gitter

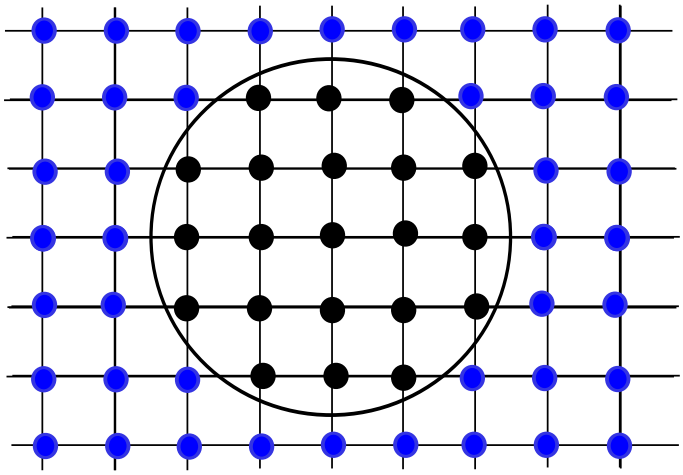


Abbildung 5.1: Säule aus Obstaclezellen umgeben von Fluidzellen

nicht komplett rund sein, sondern entspricht immer nur einer Annäherung. Während der Berechnung können die so markierten Zellen des Höhenfeldes komplett ignoriert werden, da sie niemals Wasser enthalten können.

5.1.2 PackedGridData

Wie in Kapitel 2 ausgeführt, folgen alle betrachteten Hardwarearchitekturen dem SIMD Prinzip. Daher ist es sehr wichtig, dass sämtliche Berechnungen des Lösers vektorisierbar sind. Damit Daten effizient vektorisiert verarbeitet werden können, ist es notwendig, dass sie schon kontinuierlich im Speicher vorliegen. Desweiteren können SIMD Operationen nur angewendet werden, wenn auch alle Elemente des Vektorregisters mit derselben Operation verarbeitet werden sollen. Betrachtet man das in Abbildung 5.1 gezeigte Höhenfeld, so sind die fluiden Zellen rechts und links der Säule durch die Säule voneinander getrennt. Würde man nun die Berechnungen direkt auf dem als Matrix gegebenen Höhenfeld ausführen, müsste man im Bereich der Säule jeweils eine Sonderbehandlung durchführen, da nicht alle fluiden Zellen in der Matrix komplett zusammenhängend vorliegen. Gleichzeitig müssen (zum Beispiel in den Schritten *collide and stream* oder *update velocity directions*) Zellen am Rand der Säule oder am Rand des Höhenfeldes ganz anders als Zellen, die nur fluide Nachbarn besitzen, behandelt werden.

Aus diesem Grund wird das Höhenfeld, bevor der Löser mit der eigentlichen Berechnung beginnt, durch den sogenannten `GridPacker` komprimiert und danach als eindimensionaler Vektor in der Klasse `PackedGridData` gespeichert. Hierzu werden alle Zellen des Höhenfeldes, die gemäß der Obstaclematrix kein Fluid enthalten, gelöscht, da ihre Wasserhöhe implizit bekannt ist und nicht jedesmal neu errechnet werden muß.

Analog zu diesem auf die tatsächlich zu berechnenden Elemente reduzierten Höhenfeldvektor $\mathbf{h}$ werden zusätzlich die beiden Geschwindigkeitswertvektoren $\mathbf{u}$ und $\mathbf{v}$, sowie die

jeweils neun Distributionsfunktionen f_α, f_α^{eq} und f_α^{temp} in dieser Klasse verwaltet. Dies führt zu einem Gesamtspeicherbedarf von etwa $30n$ Fließkommazahlen für ein Höhenfeld mit n fluiden Zellen.

5.1.3 PackedGridInfo

Um nun möglichst große Teilbereiche der Vektoren per SIMD verarbeiten zu können, unterteilt der `GridPacker` diese in logische Abschnitte. Die Grenzen dieser Abschnitte werden als Intervall in einem Vektor `limits` gespeichert. Jeder Abschnitt umfasst Zellen, die ähnliche Nachbarn und Randeigenschaften haben. Das bedeutet, dass innerhalb der Grenzen eines solchen Intervalls alle Zellen gleich behandelt und somit komplett vektorisiert verarbeitet werden können.

Zu jedem Intervall des `limits` Vektors enthält der `types` Vektor eine Bitmaske, die die jeweiligen Randbedingungen kodiert. Dabei bezeichnet ein gesetztes Bit an der Stelle n (von rechts gezählt) einen Rand in der Richtung $n-1$, wie sie das D2Q9 Schema in Abbildung 3.1 definiert. So hat die Bitmaske eines Abschnitts von Zellen, die sich zum Beispiel am oberen Rand des Höhenfeldes befinden und außer am oberen Rand, also in den Richtungen zwei, drei und vier, komplett von fluiden Nachbarn umgeben sind die Form (00001110).

Da die in der ursprünglichen Matrix vorhandenen Nachbarschaftsbeziehungen zwischen den Zellen durch die Reduktion auf einen gepackten eindimensionalen Vektor nicht mehr implizit gegeben sind, enthalten die Vektoren `dir_index_1` bis `dir_index_8` die Abschnittsgrenzen, in denen zusammenliegende Zellen auch zusammenliegende Nachbarn in einer bestimmten Richtung besitzen. Der Index des Nachbarn der ersten Zelle eines Abschnitts ist hierzu in den Vektoren `dir_1` bis `dir_8` explizit gespeichert. Der Index des Nachbarn der nächsten Zelle des Abschnitts entspricht dann dem vorherigen Index plus eins. Hierdurch kann selbst über Nachbarn in diagonalen Richtungen innerhalb der gegebenen Abschnitte vektorisiert iteriert werden.

Abbildung 5.2 zeigt oben ein Höhenfeld, dass einige Obstaclezellen enthält. Der unten durch den `GridPacker` erzeugte Vektor des Höhenfeldes ist um diese Zellen bereinigt. Unterhalb des Vektors sind der Werte- und Bildbereich des Vektors `dir_index_7`, also in Richung nach unten, angezeichnet. Dabei stellt die durchgezogene Linie den Werte- und die gestrichelte Linie den Wertebereich dar. Der hier betrachtete Abschnitt erstreckt sich von Index zwei bis Index neun. Der erste Nachbar des Abschnitts, also der Nachbar der Zelle zwei in Richtung sieben, der im Vektor `dir_7` gespeichert ist, lautet acht. Der Abschnitt endet an der Zelle neun, da die Zelle zehn im ursprünglichen Höhenfeld unter sich eine Obstaclezelle hat und damit die Reihe der nebeneinanderliegenden Nachbarn unterbrochen ist. Die Zelle zehn besitzt innerhalb des Vektors `dir_index_7` keinen Nachbarschaftseintrag in Richtung sieben und wird somit einfach übergangen, was ansonsten

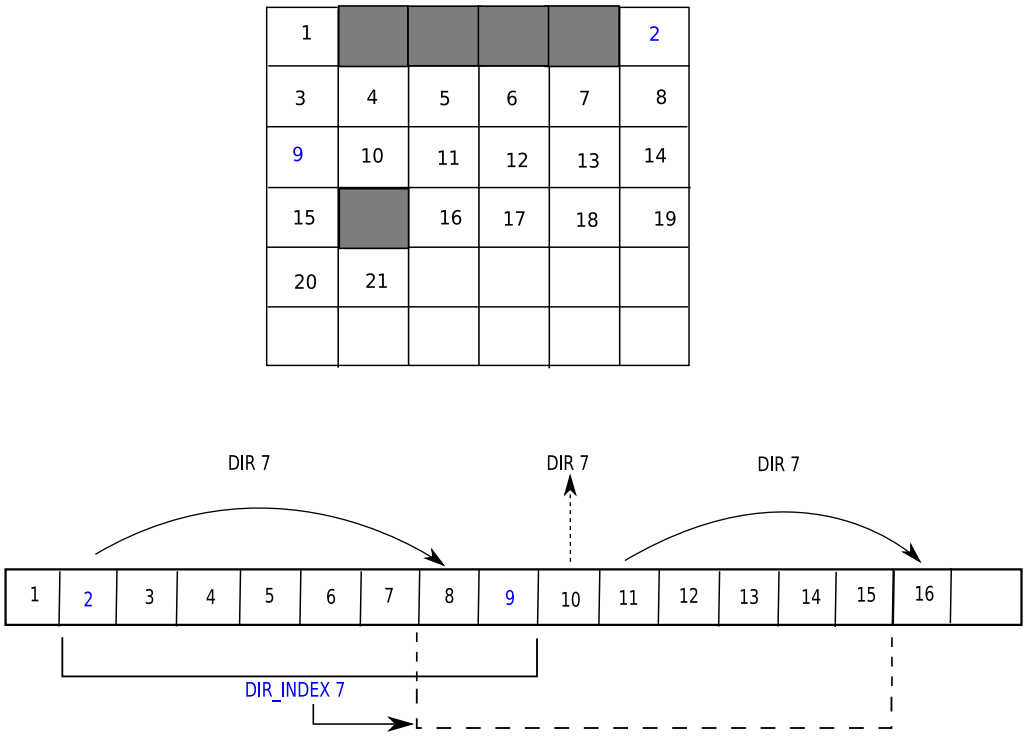


Abbildung 5.2: Aufbau der Grid Datenstruktur

notwendige teure Sonderbehandlungen bei der Berechnung spart. Der nächste zusammenhängende Abschnitt in Richtung sieben beginnt bei Zelle elf mit ihrem unteren Nachbarn sechzehn.

Zusätzlich enthält die Klasse `PackedGridInfo` auch die in Kapitel 3 aufgeführten Größen, wie die Relaxationszeit τ oder den Geschwindigkeitsvektor $\mathbf{e}_\alpha$.

5.1.4 PackedGridFringe

Soll die Arbeit eines Lösers durch mehrere verschiedene Threads oder sogar auf verschiedene Knoten eines Clusters möglichst unabhängig voneinander ausgeführt werden, so muss das Höhenfeld geeignet unterteilt werden. Dadurch wird es möglich, dass mehrere Löser gemeinsam auf einem ursprünglich großen Höhenfeld rechnen. Eine mögliche Unterteilung eines Höhenfeldes in verschiedene Partitionen zeigt Abbildung 5.3. Da zwei Löser, die auf benachbarten Partitionen rechnen, Werte schreiben oder lesen müssen, die ausserhalb ihrer eigenen Partition liegen, werden sogenannte Geisterzellen um den Rand herum eingefügt. Diese Geisterzellen müssen nach jedem Zeitschritt zwischen den Lösern ausgetauscht werden.

Diese Aufgaben übernimmt der `GridPartitioner`. Er legt im Vorfeld für jeden später rechnenden Löser eigene Datenstrukturen der zu bearbeitenden Partition inklusive Geisterzellen an. Dadurch kann auf jedem Knoten beziehungsweise in jedem Thread der

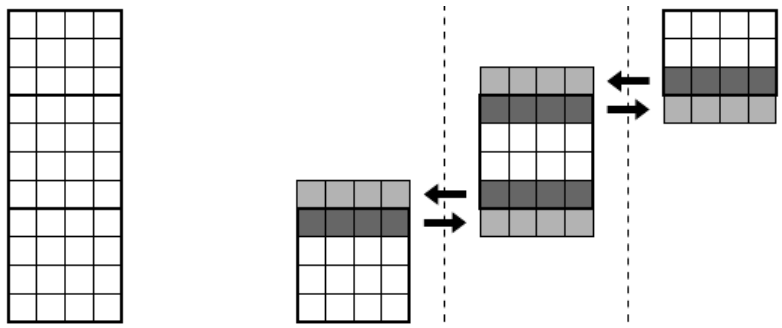


Abbildung 5.3: Partitionierung eines Höhenfeldes (Quelle Körner et al. [33])

normale `SolverLBMGrid` Löser ohne weitere Anpassungen verwendet werden. Nach jedem Zeitschritt muss durch den `GridPartitioner` eine Synchronisation zwischen den einzelnen Lösern durchgeführt werden. Zu diesem Zweck sind in der Klasse `PackedGridFringe` die Elemente gespeichert, die nach dem *extraction* Schritt kommuniziert werden müssen. Zum Einen sind dies die physikalischen Quantitäten **h**, **u** und **v**. Die für einen Löser relevanten Werte, die von anderen Lösern berechnet wurden, sind durch den Vektor `h_index`, der die Indizes der Elemente enthält, und den Vektor `h_targets`, der die laufende Nummer des Quelllösers enthält, gegeben. Zum Anderen benötigen andere Löser für ihre Berechnungen die durch einen Löser erzeugten Distributionsfunktionseinträge. Die genauen Indizes dieser Werte sind in den Vektoren `dir_index_1` bis `dir_index_8` gespeichert. Welchem Löser diese Werte jeweils mitzuteilen sind, ist analog den Vektoren `dir_targets_1` bis `dir_targets_8` zu entnehmen.

Wie in Abbildung 5.3 zu sehen ist, muss jeder Löser bei geeigneter Partitionierung des Höhenfeldes, die durch die Eigenschaften des eindimensionalen Vektors in der Klasse `PackedGridData` automatisch gegeben ist, mit maximal zwei benachbarten Lösern synchronisiert werden. Gleichzeitig handelt es sich immer nur um wenige außenliegende Elemente, die jeweils betroffen sind. In einem Cluster können so alle Kommunikationstransfers zur Synchronisation gleichzeitig angestoßen werden, da sichergestellt ist, dass keine zwei Transfers versuchen, auf dieselbe Speicherstelle zuzugreifen. Da jeder Knoten mehrere Transfers durchführen muss, werden alle Transfers nichtblockierend zur selben Zeit gestartet. Die Synchronisation ist abgeschlossen, wenn alle Transfers beendet sind.

5.2 SolverLBMGrid

Die eigentliche Löserklasse `SolverLBMGrid` besteht aus den drei im Folgenden näher beschriebenen Methoden `do_preprocessing`, `do_postprocessing` und `solve`.

5.2.1 Die do_preprocessing Methode

Hier wird zunächst der Geschwindigkeitsvektor $\mathbf{e}_\alpha$ generiert. Als nächstes wird eine initiale Gleichgewichtsfunktionen f_α^{eq} berechnet und alle $f_\alpha(\mathbf{x}, 0)\ \forall \alpha \in [0, ..., 8]$ werden auf 0 gesetzt. Anschließend werden die restlichen Datenstrukturen durch den `GridPacker` initiiert. Falls der Löser mit den Backends Multicore oder MPI genutzt werden soll, werden an dieser Stelle, wie oben ausgeführt, durch den `GridPartitioner` die einzelnen Partitionen für die Teillöser erstellt. In Falle eines Multicore Lösers werden abschließend mit Hilfe des Hardwarebackends die einzelnen Threads erstellt und im Falle einer NUMA Architektur ihre Daten in den Adressbereich des Threads verschoben. Dies wird dadurch erreicht, dass jeder Löser innerhalb seines eigenen Threads - und damit ausgeführt auf dem ihm zugewiesenen CPU Kern - eine Kopie aller Datenstrukturen anlegt und die Originale löscht. Das Prinzip, dass die Allokation durch den zuerst auf einen Speicherbereich zugreifenden Thread mittels seines eigenen Speichercontrollers geschieht, wird *lazy instantiation* [36] genannt und sorgt automatisch dafür, dass jeder Löser auf seinen Daten in seinem Speicherbereich arbeitet.

5.2.2 Die do_postprocessing Methode

Diese Methode sammelt im Falle eines verteilt rechnenden Lösers die einzelnen Teilergebnisse mit Hilfe des `GridPartitioner` ein und fügt sie zusammen. Danach werden die gepackten Daten durch den `GridPacker` aus der `GridPackedData` Klasse extrahiert und in der `Grid` Klasse gespeichert. Somit liegt das Höhenfeld wieder in der ursprünglichen Matrix vor und kann extern gespeichert oder mit Hilfe der in Abschnitt 4.3.5 vorgestellten Applikation visualisiert werden.

5.2.3 Die solve Methode

Herzstück des Lösers `SolverLBMGrid` ist die Methode `solve`, die jeweils einen einzelnen Zeitschritt berechnet. Innerhalb dieses Zeitschrittes werden die in Kapitel 3 vorgestellten Berechnungen ausgeführt. Die vier dort vorgestellten Teilschritte werden im Löser durch die vier rechts aufgeführten Kernel repräsentiert:

- *equilibrium distribution* — `EquilibriumDistributionGrid`
- *collide and stream* — `CollideStreamGrid`
- *update velocity directions* — `UpdateVelocityDirectionsGrid`
- *extraction* — `ExtractionGrid`

Gemäß dem in Kapitel 4 vorgestellten Konzepts des Anwendungskernels kapseln diese vier Kernel die kompletten durch den Löser durchgeführten Berechnungen und ermögli-

chen somit einen möglichst guten Kompromiss zwischen Generalität der Löserklasse und spezieller Hardwareoptimierung.

Abbildung 5.4 zeigt den Datenfluss zwischen den einzelnen Kerneln innerhalb eines Zeitschritts. Aus den Eingaben `h`, `u` und `v` werden in einem Zeitschritt die Quantitäten des

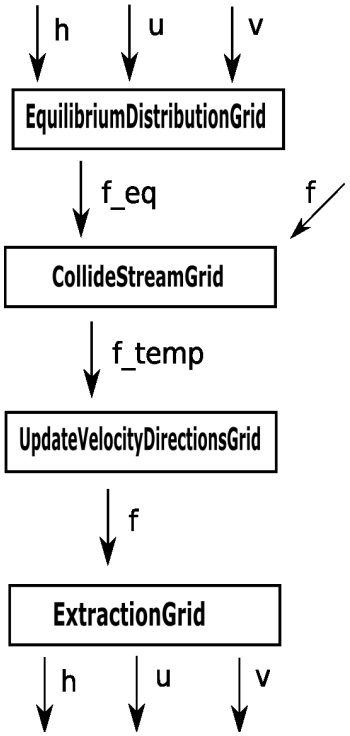


Abbildung 5.4: Datenfluss zwischen den einzelnen Kerneln

nächsten Zeitpunkts berechnet. Es wird deutlich, dass jeder Kernel direkt datenabhängig von seinem Vorgänger ist, da die Eingabe jeweils die Ausgabe des vorherigen Kernels beinhaltet. Gleichzeitig erkennt man, dass die Synchronisation zwischen mehreren Lösern nach dem `ExtractionGrid` Kernel optimal ist, da hier am wenigsten Daten kommuniziert werden müssen (`f`, `f_temp` und `f_eq` stehen hier für die jeweils neun Vektoren f_α, f_α^{temp} und f_α^{eq}).

Summiert man die arithmetischen Intensitäten der im Folgenden vorgestellten Kernel auf, so erhält man eine arithmetische Intensität von $\frac{N_{\text{flop}}}{N_{\text{transfer}}} = \frac{299n}{149n} = 2,0$ für den gesamten Zeitschritt. Dies ist in Hinblick auf die in Kapitel 1 beschriebene Problematik aktueller Hardwarearchitekturen ein vielversprechendes Ergebnis.

5.2.4 EquilibriumDistributionGrid

Dieser Kernel berechnet die lokale Equilibriumverteilungsfunktion

$$f_\alpha^{\text{eq}} = \begin{cases} h(1 - \frac{5gh}{6e^2} - \frac{2}{3e^2}u_i u_i) & \alpha = 0 \\ h(\frac{gh}{6e^2} + \frac{e_{\alpha i}u_i}{3e^2} + \frac{e_{\alpha j}u_i u_j}{2e^4} - \frac{u_i u_i}{6e^2}) & \alpha = 1,3,5,7 \\ h(\frac{gh}{24e^2} + \frac{e_{\alpha i}u_i}{12e^2} + \frac{e_{\alpha j}u_i u_j}{8e^4} - \frac{u_i u_i}{24e^2}) & \alpha = 2,4,6,8. \end{cases} \tag{5.1}$$

Insgesamt sind dazu bei einer Problemgröße von n Fluidelementen eine Anzahl von $45n$ Lade-, $9n$ Speicher- und $228n$ Fließkommarechenoperationen notwendig, wenn man das gegebene mathematische Modell und nicht die jeweils auf die einzelne Hardware optimierte Implementierungen betrachtet. Damit ergibt sich eine arithmetische Intensität von $\frac{N_{\text{flop}}}{N_{\text{transfer}}} = \frac{228n}{54n} = 4,2$. Dies lässt erwarten, dass dieser Kernel weit weniger durch die Speichertransfergeschwindigkeit gebremst wird, als andere Operationen mit einer niedrigeren arithmetischen Intensität. Da für diese Berechnungen nur das jeweils aktuelle Element verwendet und nicht die Werte der Nachbarzellen benötigt werden, lassen sie sich in allen Backends ähnlich implementieren. Um Ausdrücke wie zum Beispiel $6e^2$ oder $\mathbf{e}_{\alpha i}u_i$, die mehrmals in der gesamten Berechnung oder für ein Element benötigt werden, nicht jedesmal neu berechnen zu müssen, werden sie vorberechnet und als Konstanten gespeichert.

Im Folgenden soll die Berechnung eines Elementes von f_0^{eq} genauer betrachtet werden, um auf Gemeinsamkeiten und Unterschiede in den jeweiligen Implementierungen einzugehen.

Listing 5.1 zeigt die Berechnung eines Elements an der Stelle `index` von `f_eq_0` in vereinfachtem C++ Quellcode. Dabei bezeichnet `u` die Geschwindigkeit in x-Richtung (u_x) und `v` die Geschwindigkeit in y-Richung (u_y).

Um diese Operation nun wie in Listing 5.2 in SSE Intrinsics zu überführen, ist es notwendig jeden atomaren Rechenschritt durch Intrinsics zu ersetzen. Gleichzeitig müssen die Daten zunächst in die Vektorregister geladen und zuletzt wieder zurück in den Hauptspeicher transferiert werden. Der Befehl `scal1 = _mm_set1_ps(float(5))` belegt jeden Eintrag der vier Fließkommazahlen langen Variablen `scal1` mit dem Wert `5`. Der Befehl `m2 = _mm_load_ps(h + index)` lädt die nächsten vier Elemente des Vektors `h` ab der Stelle `index` in die Variable `m2`. Der Befehl `_mm_store_ps(f_eq_0 + index, m1)` schließlich speichert das Ergebnis der Berechnungen auf den vier Elementen ab der Stelle `index` im Vektor `f_eq_0`.

Da die SSE Einheit der CPU und die SPE's der Cell BE sich in Funktionsweise und Programmierung gleichen, ist der in Listing 5.3 aufgeführte SPE Code dem in Listing 5.2 gezeigten sehr ähnlich. Lediglich die Division geschieht hier nicht per `_mm_div_ps` sondern durch manuelle Berechung des Kehrwerts des Nenners mit Hilfe von `spu_re` und anschließende Multiplikation mit dem Zähler.

```
// Globale Konstanten
float e2(e * e);
float e23(float(3.) * e2);
float e26(float(6.) * e2);

// Elementweise Konstanten
float u2(u[index] * u[index]);
float v2(v[index] * v[index]);
float gh(g * h[index]);

// Berechnung des Elements
t1 = (float(5.) * gh) / e26;
t2 = float(2.) / e23 * (u2 + v2);
f_eq_0[index] = h[index] * (float(1) - t1 - t2);
```

Listing 5.1: EquilibriumDistribution C++ Code für Richtung 0

```
scal1 = _mm_set1_ps(float(5));
scal2 = _mm_set1_ps(float(6));
m1 = _mm_mul_ps(scal1, gv);
m2 = _mm_load_ps(h + index);
m1 = _mm_mul_ps(m1, m2);
m2 = _mm_mul_ps(e2v, scal2);
t1 = _mm_div_ps(m1, m2);

scal1 = _mm_set1_ps(float(2));
scal2 = _mm_set1_ps(float(3));
m2 = _mm_mul_ps(e2v, scal2);
t2 = _mm_div_ps(scal1, m2);
m1 = _mm_load_ps(u + index);
m1 = _mm_mul_ps(m1, m1);
m2 = _mm_load_ps(v + index);
m2 = _mm_mul_ps(m2, m2);
m1 = _mm_add_ps(m1, m2);
t2 = _mm_mul_ps(t2, m1);

scal1 = _mm_set1_ps(float(1));
m1 = _mm_load_ps(h + index);
m2 = _mm_sub_ps(scal1, t1);
m2 = _mm_sub_ps(m2, t2);
m1 = _mm_mul_ps(m1, m2);
_mm_store_ps(f_eq_0 + index, m1);
```

Listing 5.2: EquilibriumDistribution SSE Code für Richtung 0

```
m1 = spu_mul(five, g_vector);
m1 = spu_mul(m1, h[i]);
m2 = spu_mul(e_vector, six);
vd = m2;
id = spu_re(vd);
td = spu_nmsub(vd, id, one);
vd = spu_madd(td, id, id);
t1 = spu_mul(m1, vd);

m2 = spu_mul(e_vector, three);
vd = m2;
id = spu_re(vd);
td = spu_nmsub(vd, id, one);
vd = spu_madd(td, id, id);
t2 = spu_mul(two, vd);

m1 = spu_mul(u[i], u[i]);
m2 = spu_mul(v[i], v[i]);
m1 = spu_add(m1, m2);
t2 = spu_mul(t2, m1);

m1 = spu_sub(one, t1);
m1 = spu_sub(m1, t2);
f_eq[i] = spu_mul(h[i], m1);
```

Listing 5.3: EquilibriumDistribution SPE Code für Richtung 0

Da der Cell Kernel in erster Linie nicht durch die Speichertransferleistung gebremst wird, sind die einzelnen auf der SPE ausgeführten SPE Programme ungewöhnlich schnell mit ihrer Berechnung fertig. Es stellte sich heraus, dass das Anstoßen der nächsten SPE Berechnung nach Beendigung der vorherigen die Gesamtgeschwindigkeit negativ beeinflusst. Aus diesem Grund werden für diesen Kernel alle nötigen SPE Programmaufrufe vorberechnet und in einer Liste, der sogenannten `SPEInstructionQueue`, direkt zu Anfang an die SPEs übertragen. Dadurch kann eine SPE alle für diesen Schritt notwendigen Berechnungen ausführen ohne auf neue Anweisungen warten zu müssen. Allerdings hat die PPE zur Laufzeit keinen Einfluss mehr auf die SPEs und kann somit die Berechnungen nicht dynamisch verteilen. Auf Grund der Anwendung und verschiedenen Laufzeitmessungen stellte sich eine vorberechnete Partitionierung von einer SPE für jede zu berechnende Richtung als optimal heraus. Dieser Kernel verwendet also immer neun SPEs (beziehungsweise fünf auf der Playstation 3).

Demgegenüber ist der in Listing 5.4 gezeigte CUDA Quellcode wieder deutlich näher am Original orientiert zu sein. Hierbei ist aber zu beachten, dass dieses Programm in sehr vielen Threads auf allen Streamingprozessoren gleichzeitig ausgeführt wird. In diesem Fall entspricht `index` also weniger einer Schleifenvariable als viel mehr der laufenden Nummer des ausgeführten Threads.

```
float u2(u[index] * u[index]);
float v2(v[index] * v[index]);
float gh(g * h[index]);

t1 = (float(5.) * gh) / e26;
t2 = float(2.) / e23 * (u2 + v2);
f_eq_0[index] = h[index] * (float(1) - t1 - t2);
```

Listing 5.4: EquilibriumDistribution CUDA Code für Richtung 0

Da die Grafikkarte keinen schnellen Cache wie eine gewöhnliche CPU besitzt, werden die von allen Threads sehr häufig verwendeten Geschwindigkeitsvektoren $\mathbf{e}_{\alpha i}$ im Shared Memory jedes Multiprozessors gespeichert. Wie in Listing 5.5 zu sehen ist, wird zunächst über das Schlüsselwort `__shared__` ein Bereich des Shared Memory reserviert.

```
extern __shared__ float  distribution_cache[];
float* distribution_x = distribution_cache;
float* distribution_y = distribution_cache + 9;

unsigned long idx =
        (blockDim.y * blockIdx.y * gridDim.x * blockDim.x)
        + (blockDim.x * blockIdx.x) + threadIdx.x;

if (idx < 9)
{
    distribution_x[idx] =
        distribution_x_data[idx % blockDim.x];
    distribution_y[idx] =
        distribution_y_data[idx % blockDim.x];
}
__syncthreads();
```

Listing 5.5: Zwischenspeichern von $\mathbf{e}_{\alpha i}$ in CUDA

Im Anschluß wird die laufende Nummer des aktuellen Threads bestimmt. Die ersten neun Threads eines Blocks, die ja zusammen auf einem Multiprozessor ausgeführt werden, schreiben nun die beiden Geschwindigkeitsvektoren in den Shared Memory und warten danach darauf, dass alle Threads ihre Arbeit ebenfalls beendet haben. Ab hier können nun alle Threads direkt auf diese Daten sehr schnell zugreifen.

5.2.5 CollideStreamGrid

Mit Hilfe dieses Kernels wird der Kollisions- und Strömungsschritt

$$f(\mathbf{x} + \mathbf{e}_\alpha \Delta t, t + \Delta t) = f_\alpha(\mathbf{x}, t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{\mathrm{eq}}) \tag{5.2}$$

berechnet. Bei einer Problemgröße von n sind dazu insgesamt $45n$ Lade-, $9n$ Speicher- und $27n$ Fließkommarechenoperationen notwendig. Dies ergibt eine arithmetische Intensität von

$\frac{N_{\text{flop}}}{N_{\text{transfer}}} = \frac{27n}{45n} = 0,6$. Dieser Kernel wird also sehr wahrscheinlich durch die notwendigen Transfers in seiner Geschwindigkeit begrenzt. Da der Strömungsschritt nicht jedes Element unabhängig voneinander berechnet, sondern die Nachbarn in der jeweiligen Richtung mit einbezieht, werden hierzu die oben erläuterten Vektoren `dir_index_n` und `dir_n` aus der Klasse `GridPackedInfo` verwendet.

Listing 5.6 zeigt die Berechnung von $f_1(\mathbf{x}+\mathbf{e}_1\Delta t, t+\Delta t)$ als C++ Programm. Die äußere

```
for (unsigned long begin(0), half(0) ;
        begin < dir_index_1->size() - 1; begin+=2, ++half)
{
    for (unsigned long i(dir_index_1[begin]), offset(0) ;
        i < dir_index_1[begin + 1] ; ++i, ++offset)
    {
        f_temp_1[dir_1[half] + offset] =
                f_1[i] - (f_1[i] - f_eq_1[i])/tau;
    }
}
```

Listing 5.6: CollideStreamGrid Iteration in Richtung 1

for-Schleife iteriert von Abschnitt zu Abschnitt, in dem jeweils alle Nachbarn in Richtung eins existieren und nebeneinander liegen. Die innere for-Schleife iteriert mit der Variablen `i` durch den jeweiligen Abschnitt. Der Index des Nachbarn in Richtung eins, dessen Wert in `f_temp_1` beschrieben werden soll, ergibt sich durch indirekte Adressierung mit dem Wert, der in `dir_1` gespeichert ist. Die innere Schleife kann nun wieder analog zu dem `EquilibriumDistributionGrid` Kernel für die verschiedenen Backends vektorisiert werden.

5.2.6 UpdateVelocityDirectionsGrid

Dieser Kernel korrigiert die Distributionsfunktionen an den inneren und äußeren Rändern des Rechengebietes. Abhängig von der Bitmaske des Eintrags im `types` Vektor für einen durch den `limits` Vektor gegebenen Abschnitt werden die Einträge in den `f_temp` Vektoren reflektiert. Listing 5.7 zeigt den C++ Quellcodeausschnitt, der die Elemente behandelt, die

```
for (unsigned long index(begin) ; index < end ; ++index)
{
    if((types[index] & 1<<0) == 1<<0)
        for (unsigned long i(limits[index]) ;
                i != limits[index + 1] ; ++i)
        {
            f_temp_5[i] = f_temp_1[i];
        }
}
```

Listing 5.7: Randkorrektur am rechten Rand

in Richtung eins (das niederwertigste Bit im `types` Vektor ist gesetzt) an einen Rand grenzen. Da alle Zellen eines Abschnitts dieselben Randeigenschaften haben, muss die `if`-Anweisung, die den Programmfluss erheblich stören kann, nur einmal pro Abschnitt ausgeführt werden. Somit kann die innere Schleife wieder wie gewohnt vektorisiert werden.

5.2.7 ExtractionGrid

Der letzte Kernel extrahiert die Wasserhöhe

$$h(\mathbf{x}, t) = \sum_{\alpha} f_{\alpha}(\mathbf{x}, t) \tag{5.3}$$

und die Geschwindigkeitswerte

$$u_i(\mathbf{x}, t) = \frac{1}{h(\mathbf{x}, t)} \sum_{\alpha} e_{\alpha i} f_{\alpha}. \tag{5.4}$$

Insgesamt sind bei einer Problemgröße von n dazu $47n$ Ladeo-, $3n$ Speicher- und $44n$ Fließkommarechenoperationen notwendig. Dies ergibt eine arithmetische Intensität von $\frac{N_{\text{flop}}}{N_{\text{transfer}}} = \frac{44n}{50n} = 0,88$. Dieser Kernel wird also diesem Wert zur Folge auch durch die notwendigen Transfers in seiner Geschwindigkeit begrenzt.

Listing 5.8 zeigt den Schleifenkörper des Extraktionskernels. Diese Berechnung benötigt ebenfalls nur das aktuelle Element und ist somit wiederum leicht vektorisierbar.

```
h[index] = f_0[index] + f_1[index] + f_2[index]
        + f_3[index] + f_4[index] + f_5[index]
        + f_6[index + f_7[index] + f_8[index];

u[index] = (distribution_x[0] * f_0[index] +
        distribution_x[1] * f_1[index] +
        distribution_x[2] * f_2[index] +
        distribution_x[3] * f_3[index] +
        distribution_x[4] * f_4[index] +
        distribution_x[5] * f_5[index] +
        distribution_x[6] * f_6[index] +
        distribution_x[7] * f_7[index] +
        distribution_x[8] * f_8[index]) / h[index];

v[index] = (distribution_y[0] * f_0[index] +
        distribution_y[1] * f_1[index] +
        distribution_y[2] * f_2[index] +
        distribution_y[3] * f_3[index] +
        distribution_y[4] * f_4[index] +
        distribution_y[5] * f_5[index] +
        distribution_y[6] * f_6[index] +
        distribution_y[7] * f_7[index] +
        distribution_y[8] * f_8[index]) / h[index];
```

Listing 5.8: Extraktion von h, u und v

6 Ergebnisse

Im Folgenden werden die Ergebnisse der Untersuchungen des in Kapitel 5 vorgestellten Lösers `SolverLBMGrid` vorgestellt. Dazu wird zunächst kurz auf die für diese Tests eingesetzte Hardware eingegangen. Danach werden die numerischen Ergebnisse bezüglich Korrektheit und Stabilität des Lösers vorgestellt. Anschließend werden zunächst die einzelnen Löserkernel im Detail untersucht, um darzustellen, wie die einzelnen Komponenten die Gesamtleistung des Lösers beeinflussen. Darauf aufbauend kann dann die Rechenleistung des gesamten Lösers auf den verschiedenen Architekturen betrachtet werden. Abschließend werden Durchsatz und Skalierbarkeit des Lösers auf Clustern untersucht.

Wenn nicht anders angegeben, wird für alle Messungen das auf die jeweilige Gittergröße skalierte Szenario, das in Kapitel 3 in Abbildung 3.3 zu sehen ist, verwendet. Durch die in diesem Szenario verwendete Wand, auf die der zirkuläre Dammbruch aufläuft, ist eine hinreichende Zahl von nicht fluiden Zellen gegeben, sodass die Fluidzellen viele verschiedene innere Ränder besitzen.

6.1 Verwendete Hardware

Tabelle 6.1 zeigt technische Daten zu der in dieser Arbeit verwendeten x86 Architekturen. Dies sind im Einzelnen ein AMD Opteron 2214 System, ein Intel Core i7 920 System und

	Opteron	Core i7	LiDO
Kerne	2×2	4	2×1
Kerntakt (GHz)	2,20	2,66	2,40
Speicher (GByte)	8	12	8
Speicherbandbreite (GByte/s)	6,4	38	6,4
gcc Version	4.12	4.32	4.21

Tabelle 6.1: Technische Daten der verwendeten x86 Architekturen

als Knoten im LiDO Cluster jeweils ein AMD Opteron DP 250 System. Das LiDO System kommt erst in Abschnitt 6.3.3 zum Einsatz.

Tabelle 6.2 zeigt zusätzlich die technischen Daten der Playstation 3, des QS22 Blade und der beiden NVIDIA Grafikkarten vom Typ GTX 8800 und GTX 285.

	Playstation 3	QS22 Blade	GTX 8800	GTX 285
Kerne	8	2×8	16	30
Kerntakt (GHz)	3,20	3,20	1,3	1,5
Speicher (GByte)	0,256	8	0,768	2
Speicherbandbreite (GByte/s)	25,6	51,2	86	159
gcc Version	4.11	4.12	4.12	4.32

Tabelle 6.2: Technische Daten der sonstigen verwendeten Architekturen

Neben der Anzahl der vorhandenen Rechenkerne und ihrer jeweiligen Geschwindigkeit in GHz zeigen dies Tabellen die Größe des Arbeitsspeichers in GByte und die theoretische Speicherbandbreite in GByte/s zwischen den Rechenkernen und diesem Arbeitsspeicher. Dabei ist zu beachten, dass das Opteron System und das LiDO System aus zwei einzelnen Prozessoren mit zwei beziehungsweise einem Kern bestehen. Ähnlich besteht das QS22 Blade aus zwei miteinander verbundenen Cell Prozessoren, die jeweils 8 SPEs besitzen. Die SPEs beziehungsweise die Multiprozessoren der Grafikkarten zählen in dieser Arbeit jeweils als Rechenkerne des Systems. Die PPE der Cell BE wird dabei ignoriert, da sie nur Koordinierungs- und Synchronisationsaufgaben, aber keine eigentlichen Berechnungen durchführt. Für weitere Details zu den einzelnen Architekturen wird auf Kapitel 2 verwiesen.

Die letzte Zeile zeigt die Version des auf dem jeweiligen System verwendeten `gcc` Compilers. Es hat sich herausgestellt, dass verschiedene Compiler beziehungsweise Compilerversionen keinen wesentlichen Einfluss auf die Geschwindigkeit des erzeugten Programms haben. Dies ist darauf zurückzuführen, dass alle für die eigentlichen Berechnungen wesentlichen Codeteile schon optimiert in Intrinsics vorliegen und damit dem Compiler wenig Spielraum für eigene Optimierungen lassen. Es muss also bei der Verwendung von HONEI keine Rücksicht auf die Nutzung eines spezifischen Compilers genommen werden um wesentlichen Leistungseinbußen zu vermeiden. Gleichzeitig ist es nicht notwendig, dass der verwendete Compiler die exakte verwendete Hardware kennt um eine optimale Leistung zu erhalten. Es ist nämlich häufig so, dass aktuelle Hardware erst Jahre nach ihrem erscheinen von Compilern in vollem Umfang unterstützt und ausgenutzt wird.

Zusätzlich wird zur Programmierung der Grafikkarten GTX 8800 und GTX 285 die CUDA Bibliothek in den Versionen 2.0 beziehungsweise 2.2 eingesetzt. Hierbei ist zu beachten, dass die GTX 8800 Karten in dem Opteron System, die GTX 285 Karten demgegenüber in dem Core i7 System integriert sind.

Die Playstation 3 wird nur für ausgewählte Benchmarks herangezogen, da ihr effektiv nutzbarer Arbeitspeicher von nur 100 bis 150 MByte Benchmarks auf größeren Gittern nicht erlaubt. Dies wird dadurch verstärkt, dass zur Erstellung der eigentlichen Datenstrukturen nochmals deutlich mehr Arbeitsspeicher benötigt wird, als zur späteren Be-

rechnung des Lösers. Da die Erstellung der Datenstrukturen zusäztlich sehr stark linear ist und immer wieder alle bisher berechneten Zwischenergebnisse benötigt, lässt sie sich nur sehr schwer parallelisieren oder in unabhängigen Teilschritten ausführen.

6.2 Korrektheit

Um die numerische Korrektheit des Lösers `SolverLBMGrid` auf allen Architekturen zu garantieren wird eine zweistufige Strategie angewandt. Zunächst wird sichergestellt, dass die Ergebnisse des x86 basierten Lösers `SolverLBMGrid` mit denen des Lösers `SolverLABSWE`, der Referenzimplementierung, übereinstimmen. Darauf aufbauend wird sichergestellt, dass die Ergebnisse der `SolverLBMGrid` Löser auf den restlichen Architekturen mit denen des x86 Lösers übereinstimmen. Dadurch entspricht das Ergebnis des Lösers `SolverLBMGrid` auf jeder Architektur dem des Lösers `SolverLABSWE`.

Es bleibt also zu untersuchen, ob der Löser `SolverLABSWE` in sich korrekte Ergebnisse berechnet. Hierzu werden zwei verschiedene Verfahren angewandt.

Zum Einen kann man ein Szenario derart viele Zeitschritte lang ausführen, bis sich die Wasseroberfläche nicht mehr bewegt (*steady state*). Im Anschluß wird die Volumenveränderung zwischen Start und Ende des Szenarios gemessen. Für die Volumenberechnung kommen Gaussquadraturen und bilineare Interpolationen zum Einsatz, die genau genug Approximieren um Fehler durch die Messung auszuschließen. Das genaue Verfahren wird von Becker [3] ausführlich beschrieben. Falls sie in einem geringen Rahmen bleibt, geht anschaulich „kein Wasser verloren“, wie es insbesondere an den Rändern möglich ist. Tabelle 6.3 zeigt den relativen Volumenfehler $\left|\frac{V_{\text{Start}}-V_{\text{Ende}}}{V_{\text{Start}}}\right|$ für zwei verschiedene Szenarien und verschiedene Gittergrößen nach 500 Zeitschritten, in denen sich die Wasseroberfläche hinreichend beruhigt hat. Dabei handelt es sich jeweils um einen zirkulären Dammbruch,

	relativer Volumenfehler	
Gittergröße	Szenario A	Szenario B
50×50	0,00014	0,0002
500×500	0,000029	0,000043
1000×1000	0,000005	0,000006
2000×2000	0,000004	0,0000046

Tabelle 6.3: Volumenfehler nach 500 Zeitschritten

einmal ohne nicht fluide Zellen (Szenario A) und einmal mit einer an einer Stelle durchlässigen Wand aus nicht fluiden Zellen (Szenario B), wie es in Kapitel 3 in Abbildung 3.3 zu sehen ist. Die Abnahme des relativen Fehlers mit zunehmender Gittergröße ist dadurch zu erklären, dass, wie oben angeführt, besonders Randzellen einen möglichen Volumenfehler erzeugen können. Dies geschieht durch die Korrektur der Geschwindigkeiten an den Rän-

dern. Da der relative Anteil der Randzellen bei kleinen Gittern am größten ist, wird dort der größte Fehler gemessen. Die zusätzliche Wand in Szenario B verstärkt diesen Fehler noch zusätzlich. Umgekehrt wird dieser Fehler auf großen Gittern, die ja in dieser Arbeit betrachtet werden, sehr gering bis nicht meßbar.

Zum Anderen werden die Ergebnisse des Lösers mit denen eines dritten LBM Lösers verglichen. Da keine Referenzdaten eins LBM-SWE Lösers öffentlich zugänglich sind wurden hierzu die Ergebnisse des von Hübner [25] entwickelte LBM Lösers herangezogen. Bei diesem Löser wiederum konnte nachgewiesen werden, dass er bei diesen Tests die Ergebnisse eines Finite Elemente Navier Stokes Lösers aus der FEATFLOW Bibliothek [2] reproduziert.

Da dieser Löser keine Flachwassergleichungen sondern Navier Stokes verwendet, musste hierzu die Berechnung des *equilibrium distribution* Schrittes im `SolverLABSWE` Löser, wie in Abschnitt 4.3.4 erläutert, verändert werden, um auch Navier Stokes zu verwenden. Die restlichen Löserschritte bleiben davon unberührt. Es wird also nicht der eigentliche Flachwasserlöser mit dem von Hübner verglichen, sondern alle Lösermodule bis auf den *equilibrium distribution* Schritt. Es stellt aber eine der wenigen Möglichkeiten dar, überhaupt die implementierten Operationen und Kernel mit anderen Arbeiten zu vergleichen. Auf einen Vergleich mit dem in der in Abschnitt 4.2.8 erwähnten voll explizite 2D-Flachwasserlöser wurde verzichtet, da er zum Einen andere Randbedingungen benutzt und es somit ebenfalls notwendig gewesen wäre, Teile der Löser auszutauschen. Zum Anderen hat der Vergleich mit Lösern einer anderen Bibliothek sicherlich eine höhere Aussagekraft als der Vergleich verschiedener Löser derselben Autoren.

Das verwendete Szenario wird als *lid-driven cavity* bezeichnet [17] und entspricht einem Wasserstrudel, der durch einen der vier Ränder des quadratischen Höhenfeldes erzeugt wird. An diesem Rand wird das Wasser durchgängig beschleunigt, so dass es einen Wirbel formt. Hierzu werden nach jedem Zeitschritt die Geschwindigkeiten an diesem Rand modifiziert um das Wasser zu beschleunigen, bevor der nächste Zeitschritt berechnet wird. Abbildung 6.1 zeigt die Isolinien für die x-Komponente der makroskopischen Geschwindigkeit, die den Wirbel deutlich erkennen lassen. Um dieses Ergebnis mit dem von Hübner zu vergleichen, zeigt Abbildung 6.2 als charakteristische Kurve der Löserausgabe die relative y-Koordinate bezogen auf das gesamte Lattice ($y[lattice]$) angetragen gegen die relative x-Komponente der makroskopischen Geschwindigkeit (u/U). Die Abweichungen zwischen den beiden Kurven sind mit dem bloßen Auge nur schwer zu erkennen. Die euklidische Norm des absoluten Fehlers über alle gemessenen Datenpunkte D beträgt $0,0002$. Der relative Fehler $\left|\frac{D_{\text{Hübner}}-D_{\text{HONEI}}}{D_{\text{Hübner}}}\right|$ beträgt $0,00036$. Die Ergebnisse der Löser stimmen also in hohem Maße überein. Für weitergehende Untersuchungen zur Korrektheit des Lösers `SolverLBMGrid` wird auf die Arbeit von Geveler [16] verwiesen.

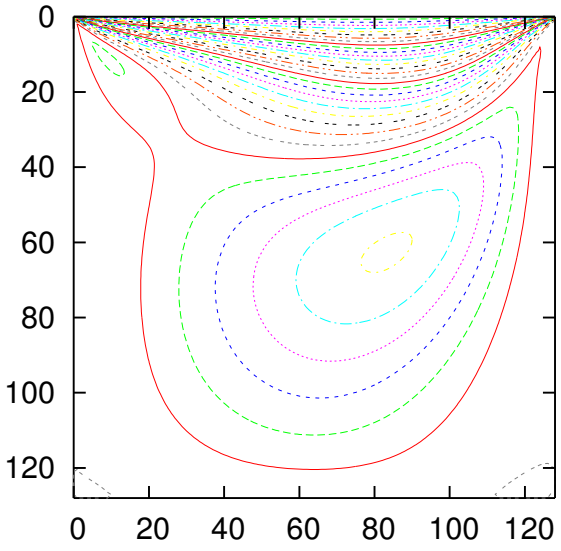


Abbildung 6.1: Driven cavity: Isolinien der Geschwindigkeitswerte

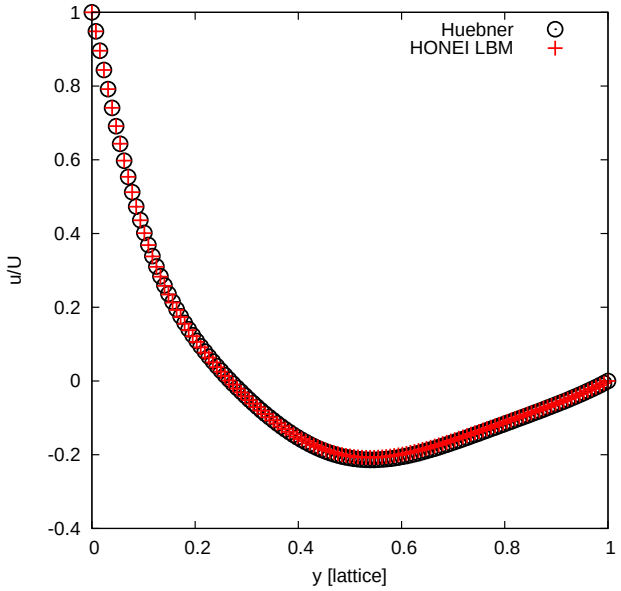


Abbildung 6.2: Driven cavity Ergebnisse

6.3 Benchmarks

6.3.1 Kernelbenchmarks

Tabelle 6.4 zeigt die typische Leistung der einzelnen in Kapitel 5 vorgestellten Löserkernel `EquilibriumDistributionGrid` (EqDist), `CollideStreamGrid` (CollStream) und `ExtractionGrid` (Extraction) auf den verschiedenen Architekturen in Gflop/s für ein Gitter der Größe 2000×2000. Diese Gittergröße hat sich als praktikabel für diese Benchmarks herausgestellt, da die GTX 8800 Grafikkarte mit dem kleinsten Arbeitsspeicher der hier verwendeten Architekturen keine größeren Probleme berechnen kann. Da der Kernel `UpdateVelocityDirectionsGrid` (UpVelDir) keine eigenen Berechnungen im Sinne von Fließkommaoperationen durchführt, sondern nur Daten im Speicher verschiebt, ist er hier nicht aufgeführt. Zu beachten ist, dass die auf den x86 Architekturen ausgeführten Kernel jeweils nur einen Kern nutzen, da der in Kapitel 5 vorgestellte Multithreading Ansatz mit Hilfe des `GridPartitioner` Threads auf Löserebene und nicht auf Kernelebene einsetzt. Zusätzlich sind die aus Kapitel 5 bekannten arithmetischen Intensitäten dieser Kernel aufgeführt.

	EqDist	CollStream	Extraction
arithm. Intens.	4,2	0,6	0,88
Opteron	1,99	0,68	1,24
Core i7	7,19	2,00	4,53
QS22 Blade	11,54	1,27	9,29
GTX 8800	90,65	3,84	19,63
GTX 285	181,54	19,16	46,07

Tabelle 6.4: Typische Rechenleistung der einzelnen Löserkernel in Gflop/s

Tabelle 6.5 zeigt für dieselbe Konfigurationen die Ausführungszeit der einzelnen Löserkernel in Sekunden. Um deutlich ausserhalb der Auflösung der Systemuhren zu liegen, wurden 250 Zeitschritte zusammen gemessen.

Zunächst wird deutlich, dass die arithmetische Intensität in der Tat einen guten Indikator für die mögliche Ausreizung der Rechenkapazitäten darstellt. Die mit Abstand höchste Rechenleistung erreicht auf allen Architekturen der Kernel `EquilibtriumDistributionGrid` (EqDist), der auch die höchste arithmetische Intensität besitzt.

Da der Kernel `CollideStreamGrid` (CollStream) die Nachbarn einer Zelle mit einbezieht, sind hier kompliziertere Speicheroperationen notwendig. Wie in Abschnitt 2.2 erläutert, ist der Speichercontroller der GTX 8800 nicht in der Lage, Speicherzugriffe, die nicht *coalesced* sind, zusammenzufassen. Daher ist die GTX 8800 Grafikkarte bei diesem Kernel um einen Faktor vier langsamer als die GTX 285 Grafikkarte. Damit ist sie hier doppelt so langsam wie bei der Berechnung des `EquilibtriumDistributionGrid` Kernels, bei der zwischen

	EqDist	CollStream	Extraction	UpVelDir
arithm. Intens.	4,2	0,6	0,88	-
Opteron	224,9	39,9	35,6	0,2
Core i7	62,1	13,6	9,8	0,1
QS22 Blade	2,0	21,3	4,86	1,2
GTX 8800	2,3	6,4	2,3	1,3
GTX 285	1,2	1,4	1,0	0,3

Tabelle 6.5: Typische Ausführungszeit der einzelnen Löserkernel in Sekunden für 250 Zeitschritte

beiden Grafikkarten ein Faktor von zwei liegt, der ziemlich genau dem Unterschied zwischen den Speicherbandbreiten der beiden Grafikkarten entspricht. Durch die notwendigen Zugriffe auf die Nachbarzellen ist auch zu erklären, dass das QS22 Blade bei diesem Kernel hinter die Leistung des Core i7 zurückfällt. Der jeder SPE zur Verfügung stehende lokale Speicher ist nämlich deutlich geringer als die Caches des Intel Prozessors, so dass die SPEs häufiger Daten aus dem Hauptspeicher nachtransferieren müssen.

Der Kernel `ExtractionGrid` (Extraction) rechnet hingegen wiederum elementweise. Daher werden hier durch alle Architekturen wieder höhere Leistungsdaten erreicht.

Strukur eines einzelnen Löserzeitschrittes

Tabelle 6.6 zeigt den prozentualen Anteil der einzelnen Kernel an der Gesamtausführungszeit eines Löserzeitschrittes. Bei den x86 Architekturen wurde zusätzlich zu der Version

	EqDist	CollStream	Extraction	UpVelDir
Opteron 1 Thread	75	13	12	0
Opteron 3 Thread	61	24	15	0
Core i7 1 Thread	74	15	11	0
Core i7 4 Thread	48	35	17	0
QS22 Blade	7	72	16	5
GTX 8800	19	52	19	10
GTX 285	32	37	24	7

Tabelle 6.6: Anteil der Löserkernel an der Gesamtrechenzeit in Prozent

mit einem Thread jeweils die in Abschnitt 6.3.2 optimale Threadanzahl für die beiden Rechensysteme hinzugefügt. Da, wie oben angeführt, eigentlich jeder Thread einen kompletten Zeitschritt und nicht jeden Kernel einzeln berechnet, sind die Werte für die Löser mit mehreren Threads nur einem einzelnen Zeitschritt entnommen. Deshalb sind sie nicht über viele Kernelaufrufe gemessen worden und damit potentiell ungenau. Sie dienen hier

nur dem Vergleich mit den restlichen Messungen und sollen gleichzeitig die Rechenzeitverteilung auch in einem Multicorelöser näher beschreiben.

Diese Tabelle bestätigt die oben getroffenen Aussagen zur Leistung der einzelnen Kernel. Die drei unteren besonders rechenstarken Architekturen verbringen im Vergleich eine sehr viel geringere Zeit mit der Berechnung des Kernels `EquilibriumDistributionGrid` (EqDist) als die x86 Architekturen .

Die beiden x86 Löser mit je einem Thread benötigen für jeden Kernel etwa denselben Zeitanteil wie der jeweils andere Löser. Vergleicht man die Löserversionen mit einem Thread und die Varianten mit mehreren Threads so wird deutlich, dass die Erhöhung der potentiellen Rechenleistung durch mehrere Threads besonders gut die Ausführung des EqDist Schrittes beschleunigt.

Insbesondere die Cell BE benötigt auf Grund des kleinen lokalen Speichers der SPEs sehr lange zur Berechnung des Kernels `CollideStreamGrid` (CollStream). Da dieser Kernel sehr viele Speicherzugriffe erfordert, die nicht *coalesced* sind, benötigt auch die ältere GTX 8800 Grafikkarte dafür relativ lange. Die Verteilung der Rechenzeitanteile auf die Rechenschritte EqDist und CollStream ist also zwischen den x86 Architekturen auf der einen Seite und dem QS22 Blade und der GTX 8800 Grafikkarte auf der anderen Seite genau entgegengesetzt. Die x86 Architekturen benötigen in ersterem Rechenschritt sehr lange, während sie letzteren relativ schnell bearbeiten. Die letztegenannten auf hohe Rechenleistung optimierten Architekturen verbringen die meiste Zeit in dem weniger rechenintensiven CollStream Schritt. Da bei der GTX 285 Grafikkarte der Speichercontroller, wie oben angeführt, mit den notwenigen Speicherzugriffen im CollStream Schritt deutlich besser als die GTX 8800 Karte umgehen kann, haben hier beide Kernel denselben Anteil an der Gesamtrechenzeit.

Der Anteil des Kernels `ExtractionGrid` (Extraction) weist auf allen Architekturen keine Besonderheiten auf.

Die im Kernel `UpdateVelocityDirectionsGrid` (UpVelDir) mit sehr vielen Programmverzweigungen verbundene Korrektur der Ränder ist im Gegensatz dazu besonders geeignet für die x86 Architekturen, da diese mit komplexen Sprungvorhersagen und spekulativen Ausführungseinheiten ausgerüstet sind. Demgegenüber müssen in so einem Fall auf den Grafikkarten die ausgeführten Threads alle möglichen Programmpfade berechnen, wie es in Abschnitt 2.2 erläutert wurde. Auf der Cell BE muss jeweils auf die Auswertung der Sprungbedingung gewartet werden, bevor die Berechnung fortgesetzt werden kann, da, wie in Abschnitt 2.3 ausgeführt, die SPEs keine Sprungvorhersage in Hardware besitzen.

Cell Einzelbenchmarks

Da der Kernel `ExtractionGrid` auf der Cell BE für eine variable Anzahl von verwendeten SPEs ausgelegt werden konnte, lässt sich dort gut der Einfluss einer steigenden Anzahl von verwendeten SPEs auf die Rechenleistung des Kernels untersuchen, wie er in Abbildung 6.3

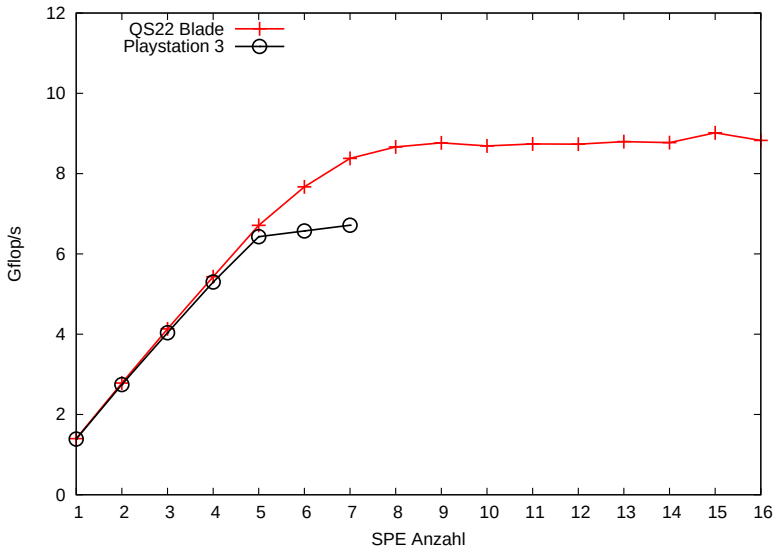


Abbildung 6.3: ExtractionGrid: Skalierung mit SPE Anzahl

zu sehen ist. Hierzu wird ein Höhenfeld von der Größe 1000×1000 verwendet, da größere Höhenfelder mit der Playstation 3 auf Grund des sehr geringen Arbeitsspeichers nicht berechnet werden können. Gleichzeitig ist auf der Playstation die achte SPE von Sony deaktiviert, um den Produktionsprozess zu vereinfachen und kann somit nicht genutzt werden.

Auf dem QS22 Blade, das aus zwei miteinander verbundenen Cell Prozessoren besteht, werden die vorhandenen SPEs immer abwechselnd verwendet. Zum Beispiel berechnen bei vier verwendeten SPEs auf jedem Cell Prozessor jeweils zwei SPEs den ausgeführten Kernel. Bis zu einer Anzahl von sieben SPEs steigt die erreichte Leistung auf dem Blade nahezu linear an. Ab acht SPEs ist durch nachfolgend hinzugefügte SPEs keine weitere wesentliche Leistungssteigerung mehr zu erkennen. An dieser Stelle ist der Speicherbus ausgelastet und alle SPEs warten den Großteil der Zeit darauf, dass sie Daten aus dem Hauptspeicher erhalten. Da der Kernel auf Grund seiner niedrigen arithmetischen Intensität durch Speichertransfers in seiner Leistung begrenzt ist, entspricht dieses Verhalten dem erwarteten Ergebnis. Auf der Playstation 3 steigt die Leistung nur bei bis zu fünf verwerndeten SPEs linear an und flacht dann stark ab. Dies liegt daran, dass die Playstation nur aus einem Cell Prozessor besteht und damit nur die halbe Speichertransferrate des Blade besitzt. Bis zu einer Anzahl von fünf verwendeten SPEs erreichen beide Architekturen dieselbe Leistung, da jede zusätzliche SPE die Gesamtleistung um ihre Rechenleistung erhöht, da sie nicht durch eine zu geringe Speichertransferrate ausgebremst wird. Diese Beobachtung entspricht den Ausführungen von Buttari [6] und Stürmer [50].

6.3.2 Löserbenchmarks

Im Folgenden soll die Rechenleistung des gesamten Lösers `SolverLBMGrid` auf den unterschiedlichen Hardwarearchitekturen untersucht werden. Dazu wird, wie in Arbeiten zu LBM Lösern üblich, die Leistung des Lösers in der Anzahl der in einer Sekunde aktualisierten Gitterzellen (*mega lattice site updates per second, MLUP/s*) gemessen.

Zunächst werden die beiden x86 Architekturen AMD Opteron und Intel Core i7 betrachtet. Wie in Abbildung 6.4 zu sehen ist, verdoppelt sich die Rechenleistung des Lösers auf dem Opteron Prozessor nahezu, wenn anstatt einem Thread zwei verwendet werden. Wird ein zusätzlicher dritter Thread verwendet, so fällt die Leistungssteigerung deutlich

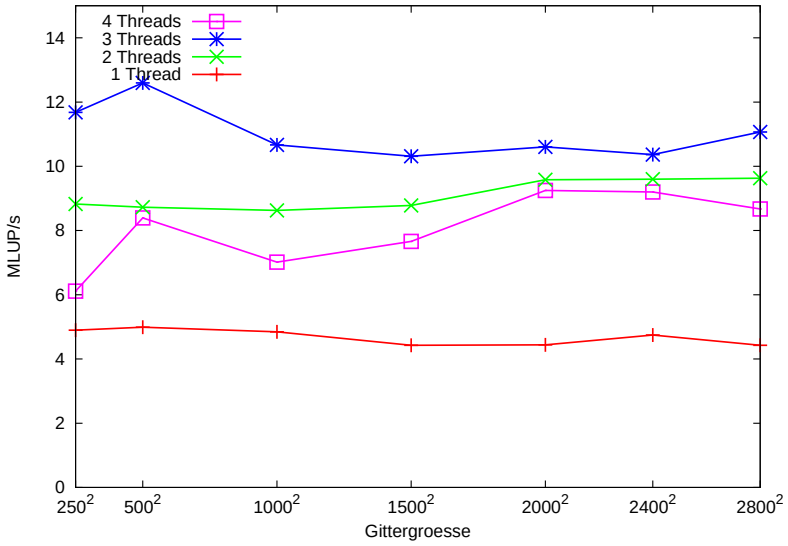


Abbildung 6.4: Rechenleistung des Lösers auf AMD Opteron

geringer aus. Weitere zusätzliche Threads bremsen den Löser sogar wieder auf Grund des höheren Synchronisationsaufwandes aus. Dies ist darauf zurückzuführen, dass dieses System genau zwei unabhängige Speichercontroller besitzt. Da das Multicore Backend die Threads gleichmäßig auf alle verfügbaren CPU Sockets verteilt, können die ersten beiden Threads also jeweils unabhängig voneinander die volle Bandbreite eines eigenen Speichercontrollers nutzen. Kommen weitere Threads hinzu, so müssen sich alle Threads die beiden Speichercontroller teilen.

Abbildung 6.5 zeigt die Rechenleistung des Lösers für verschiedene Threadanzahlen auf dem Core i7 System. Zuallererst fällt die bemerkenswerte Leistung schon eines Threads gegenüber der in Abbildung 6.4 gezeigten Opteron Systems auf. Wie schon bei diesem System, verdoppelt auch hier das Aktivieren eines zweiten Threads nahezu die Rechenleistung. Ein dritter und vierter Thread können zwar die Rechenleistung ebenfalls noch erhöhen, es wird aber auch hier deutlich, dass der Zugriff auf den Speicher die Vielzahl an Threads ausbremst. Da der Core i7 Prozessor die in Kapitel 2 erklärte Hyper-Threading

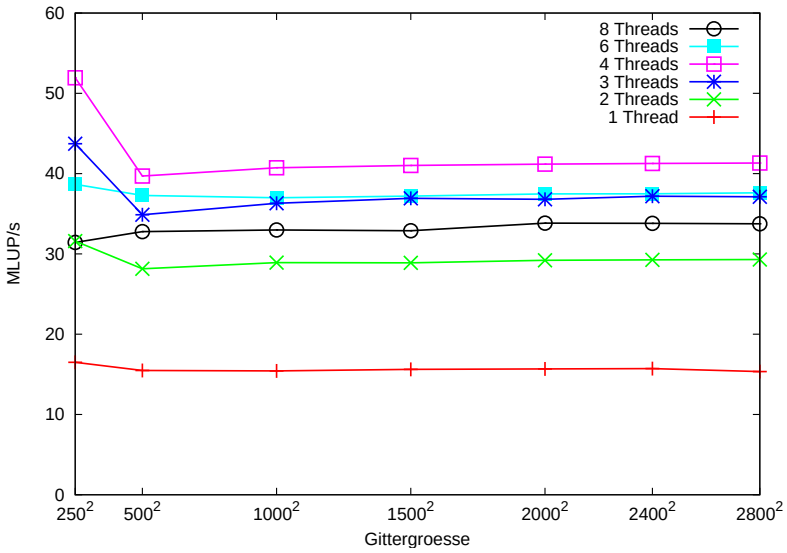


Abbildung 6.5: Rechenleistung des Lösers auf Intel Core i7

Technik unterstützt, können theoretisch bis zu acht Threads auf den vier Kernen parallel ausgeführt werden. Mehr als vier Threads verlangsamen den Löser allerdings wieder auf Grund des hohen Synchronisationsaufwandes, dem kein hinreichender Leistungsgewinn gegenübersteht, da der Speichercontroller die zusäztlichen Threads nicht schnell genug mit Daten versorgen kann.

Abbildung 6.6 zeigt die Rechenleistung des Lösers auf dem QS22 Blade und der Playstation 3 für kleine Gittergrößen unter Nutzung einer optimalen Anzahl von SPEs für jeden Kernel auf der jeweiligen Architektur. Es zeigt sich, dass beide Systeme nahezu dieselbe

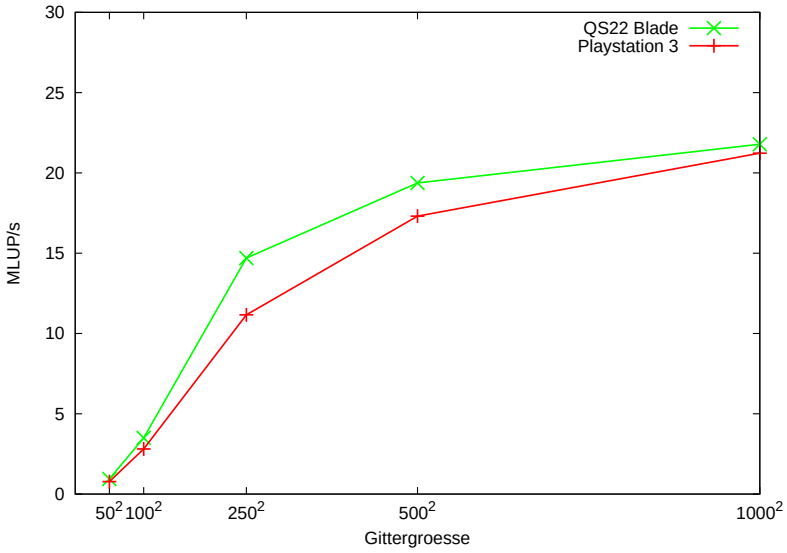


Abbildung 6.6: Rechenleistung des Lösers auf den beiden Cell BE Architekturen

Leistung erbringen, insbesondere für ein Gitter der Größe 1000×1000. Dies ist darauf zurückzuführen, dass wie in Tabelle 6.6 ersichtlich die meiste Rechenzeit zur Berechnung des `CollideStreamGrid` Kernels verwendet wird, und dort die zusäztlichen SPEs des QS22 Blade bei diesem stark datenabhängigen Kernel keinen Vorteil bringen.

Zu beachten ist, dass die Playstation 3 schon ab einer Gittergröße von 500×500 zur Erstellung der Datenstrukturen für die späteren Löserzeitschritte starken Gebrauch von der Auslagerungsdatei machen muss, um den zu geringen Arbeitsspeicher auszugleichen. Bei einer Gittergröße von über 1000×1000 ist der Arbeitsspeicher schließlich für die eigentlichen Rechenschritte zu gering. Auf Grund dieser starken Einschränkung der Playstation 3 durch ihren sehr gering bemessenen Arbeitsspeicher, ist das Cell Blade für gewöhnlich die deutlich bessere Wahl für wissenschaftliche Berechnungen.

Nun sollen alle verschiedenen Architekturen miteinader verglichen werden. Abbildung 6.7 zeigt hierzu die Rechenleistung auf den verschiedenen in dieser Arbeit behandelten Architekturen. Zunächst ist zu beachten, dass die GTX 8800 Grafikkarte auf Grund des

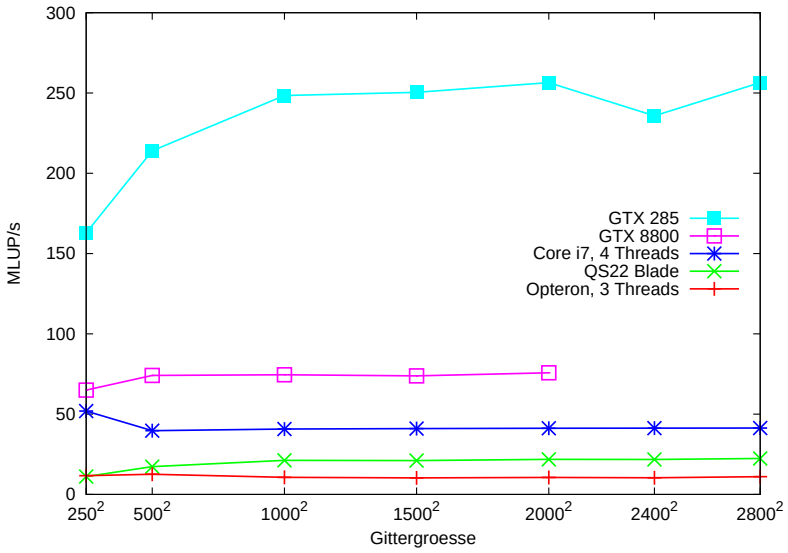


Abbildung 6.7: Vergleich des Lösers auf verschiedenen Architekturen

geringen Grafikkartenspeichers von 768 MByte maximal Berechnungen auf einer Gittergröße von 2000^2 komplett im Grafikkartenspeicher ausführen kann. Grundsätzlich entsprechen die Leistungsunterschiede dem Verhältnis zwischen den Architekturen, das die Tabellen 6.4 und 6.6 in der Summe über alle Löserkernel andeuten. Die auf Kernelebene gemessene Leistung schlägt sich also voll auf die Anwendungsebene durch. Die Leistung der GTX 285 Grafikkarte ist um den Faktor $3,4$ schneller als die der älteren GTX 8800 Grafikkarte. Dies ist neben der erhöhten Rechen- und Speichergeschwindigkeit, wie oben ausgeführt, in erster Linie auf die Verbesserungen des Speichercontrollers im Umgang mit Speicherzugriffe, die nicht *coalesced* sind, zurückzuführen. Auf Grund der in Tabelle 6.6 ersichtlichen hohen

Ausführungszeit des `CollideStreamGrid` Kernels auf dem QS22 Blade, ist die Gesamtleistung des Lösers auf dem Blade nicht sehr viel höher als die des Opteron System. Das Core i7 System ist bereits schneller als das Cell Blade. Beide Architekturen bleiben aber wegen ihrer wesentlich geringeren Speichergeschwindigkeit deutlich hinter der Leistung der Grafikkarten zurück.

Einfluss des GridPacker Algorithmus auf die Löserleistung

Um den Einfluss der Höhenfeldform und des durch den `GridPacker` verwendeten Packalgorithmus zu untersuchen, zeigt Tabelle 6.7 die typische Rechenleistung des Lösers auf dem üblicherweise zu Benchmarks verwendeten quadratischen Höhenfeld ($n \times n$) mit $n = 2000$.

	Leistung			rel. Verlust	
	$1 \times n^2$	$n \times n$	$n^2 \times 1$	$n \times n$	$n^2 \times 1$
Opteron	4,7	4,5	3,4	4	28
Core i7	16	15	10	6	37
QS22 Blade	23	21	4,4	9	81
GTX 8800	83	74	60	11	28
GTX 285	253	248	215	2	15

Tabelle 6.7: Einfluss der Höhenfeldform auf die Rechenleistung in MLUP/s und relativer Verlust zur optimalen Leistung in Prozent

Diese wird verglichen mit der auf einem Höhenfeld, das nur aus einer Zeile besteht ($1 \times n^2$). Bei dieser Höhenfeldform wird der `GridPacker` mit Ausnahme der linken und rechten Zelle alle inneren $n^2 - 2$ Zellen zu einem großen Abschnitt zusammenfassen. Dieser kann dann komplett vektorisiert verarbeitet werden. Hierbei ist eine sehr hohe Leistung des Lösers zu erwarten. Die umgekehrte Situation ergibt ein Höhenfeld, dass nur eine Spalte besitzt ($n^2 \times 1$). Hier wird jede Zelle zu ihrem eigenen Abschnitt zusammengefasst und somit werden viele vektorisierte Rechenschritte erschwert. Zusätzlich ist der relative Leistungsverlust in Prozent angegeben, den ein Löser auf einem nicht optimalen Höhenfeld erleidet.

Betrachtet man das nur schwer vektorisierbare Höhenfeld ($n^2 \times 1$), so zeigt sich ein deutlicher Leistungsverlust im Vergleich zu den anderen Höhenfeldformen. Sehr stark sinkt die Leistung des Core i7 Systems und besonders des QS22 Blade, da die SPEs keinerlei Unterstützung für skalare Operationen bieten und diese durch Vektoroperationen mit nur einem Skalar emulieren müssen.

Es wird aber auf der anderen Seite deutlich, dass ein quadratisches Höhenfeld bereits ein ausreichendes Maß an Vektorisierung ermöglicht, um eine nahezu gleichwertige Leistung im Vergleich zu der als optimal angenommen Leistung auf einem Höhenfeld der Form $1 \times n^2$ zu

erreichen. Gleichzeitig zeigt die Optimierung der Höhenfeldform an den Packalgorithmus, dass umgekehrt ein optimierter Packalgorithmus nochmals zu einer Leistungssteigerung führen kann. Dieser könnte zum Beispiel grundsätzlich alle Zellen desselben Typs zu je einem Abschnitt zusammenfassen und so die maximal vektorisiert berechenbaren Abschnitte maximieren, unabhängig von der ursprünglichen Höhenfeldform. Es ist allerdings zu beachten, dass eine derartige komplette Umstrukturierung des Höhenfeld wiederum eine sehr lineare Operation wäre, die die ohnehin schon kritische Erstellung der Datenstrukturen weiter komplizieren und ihren Speicherverbrauch weiter erhöhen würde.

6.3.3 Cluster und Skalierbarkeit

Um die Leistungsfähigkeit des Lösers `SolverLBMGrid` auf einem MPI-gestützten Cluster zu bewerten, bietet es sich an, die Skalierbarkeit des Lösers auf dem Cluster zu untersuchen. Genauer gibt es zwei verschiedene Definitionen von Skalierbarkeit, die als starke und schwache (parallele) Skalierbarkeit bezeichnet werden. Unter starker Skalierbarkeit versteht man, dass bei einer Verdoppelung der Ressourcen (in diesem Fall die Anzahl der verwendeten Knoten im Cluster) und konstanter Problemgröße sich die Geschwindigkeit verdoppelt und sich somit die Ausführungsgeschwindigkeit halbiert. Demgegenüber bedeutet schwache Skalierbarkeit, dass bei doppelter Ressourcenanzahl und einem Problem doppelter Größe die Laufzeit unverändert bleibt.

Dabei werden hier nur Cluster aus x86 Prozessoren betrachtet und auf diesen alle notwendigen Prozesse durch die MPI Bibliothek gestartet. Damit die Kommunikation zwischen den Prozessen (Jobs) tatsächlich über die Netzwerkverbindung geschieht und damit der Einfluss der notwendigen Kommunikation auf die Löserleistung gemessen werden kann, wird nach Möglichkeit auf jedem Knoten nur ein Job ausgeführt um etwaige Kommunikation über einen gemeinsamen Speicher zu unterbinden. Die Ausführung eines Multicore oder GPU basierten Lösers auf jedem Knoten eines Clusters ist zwar softwaretechnich durch die von HONEI verwendete Templatisierung aller Klassen möglich, ist für diese Anwendung aber nicht ohne weiteres praktikabel:

Im Falle eines Multicore Lösers wird dieser zunächst zwischen seinen Threads eine Synchronisation durchführen, bevor das Gesamthöhenfeld mit den anderen Knoten im Cluster synchronisiert wird. Dadurch wird zum Einen also auf zwei verschiedenen Ebenen eine Synchronisation durchgeführt die zum Anderen den Gesamtsynchronisationsaufwand erhöht, da die nun im Cluster zu synchronisierenden Höhenfelder um die Anzahl der auf einem Knoten verwendeten Threads größer sind. Gleichzeitig steigt der Speicherbedarf unnötig an, da immer auch ein Höhenfeld und damit auch alle korrespondierenden Daten wie Geschwindigkeitsfelder für das Gesamtergebnis eines Knotens verwaltet werden müssen, das die einzelnen Threads des Multicore Lösers berechnen.

Im Falle eines GPU basierten Lösers wird zur Kommunikation über das Netzwerk im Cluster nach jedem Zeitschritt ein Datentransfer von dem Grafikkartenspeicher des einen Knoten in dessen Hauptspeicher und von dort zur Netzwerkkarte erforderlich bevor die Daten in einem zweiten Knoten aus der Netzwerkkarte über den Arbeitsspeicher letztlich in den dortigen Grafikkartenspeicher transferiert werden können. Diese notwendigen Datentransfers können asynchron zu der auf den Grafikkarten ausgeführten Berechnungen durchgeführt werden und müssen somit die Gesamtleistung des Lösers nicht negativ beeinflußen. Da diese vielversprechende Technik zu Begin dieser Arbeit allerdings nicht zur Verfügung stand, konnte sie im Rahmen dieser Arbeit nicht realisiert werden.

Starke Skalierbarkeit

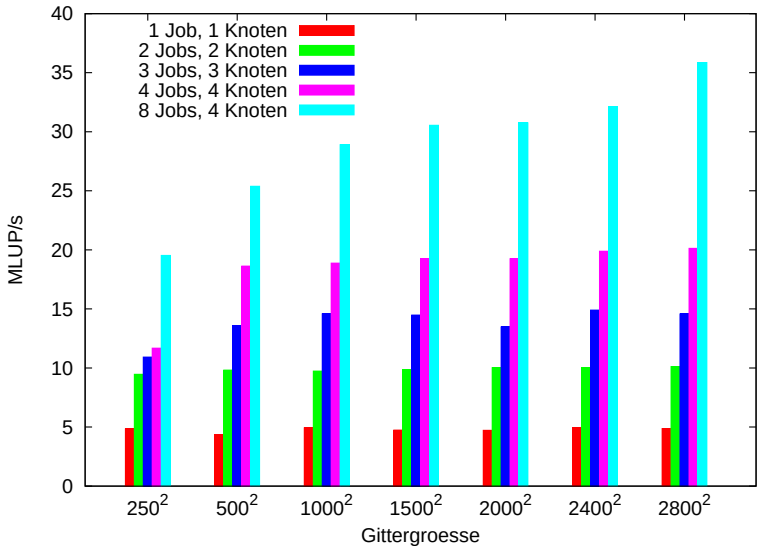


Abbildung 6.8: Starke Skalierbarkeit auf einem Opteron Cluster

Die starke Skalierbarkeit soll anhand eines Clusters aus vier durch ein Inifiniband Netzwerk [24] verbundenen AMD Opteron 2214 Maschinen untersucht werden, deren technische Daten in Tabelle 6.1 aufgeführt sind. Abbildung 6.8 zeigt die Geschwindigkeit verschiedener Clusterkonfigurationen auf verschiedenen Gittergrößen. Hierbei wird eine steigende Anzahl von MPI Jobs, die jeweils einen Thread ausführen auf eine steigende Anzahl von Clusterknoten verteilt. Es zeigt sich deutlich, dass insbesondere das Verdoppeln und Vervierfachen der verwendeten Knoten die Leistung ebenfalls verdoppelt bzw. vervierfacht. Da nur vier Knoten zur Verfügung stehen, wird eine Verachtfachung der Ressourcen durch das Ausführen von zwei MPI Jobs auf jedem Knoten, genauer einem Thread pro Prozessorsocket, realisiert. Mit einer maximalen Geschwindigkeit von 36 MLUP/s wird ein Speedup gegenüber der Geschwindigkeit eines Jobs (4,7 MLUP/s) von 7,6 erreicht. Dieser Abfall ist zum Einen dadurch zu erklären, dass das Gitter für diese größere Anzahl von Jobs noch

zu klein ist (eine gedachte Kurve über die Balken „8 Jobs, 4 Knoten“ in Abbildung 6.8 hat noch nicht ihr Maximum erreicht); der Verwaltungsaufwand zur Synchronisation also im Vergleich zu den berechnenden Teilproblemen einen relevanten Anteil darstellt. Zum Anderen ist aber auch zu beachten, dass in dieser Konfiguration nicht jedem Job ein einzelner Knoten allein zur Verfügung stand.

Eine andere Beobachtung kann bezüglich des Verwaltungsaufwandes zur Synchronisation bei den ersten vier Konfigurationen gemacht werden. Da der Löser perfekt mit den eingesetzten Knoten skaliert, hat die Synchronisation keinerlei negativen Einfluss auf die Geschwindigkeit. Dies ist bemerkenswert, da die Synchronisation hier nicht wie in einem Multicore System über den gemeinsamen Speicher sondern über ein sehr viel langsameres Netzwerk geschieht. Dies bestätigt die in Kapitel 5 getroffene Aussage, dass nur sehr wenige Elemente zwischen den einzelnen Knoten kommuniziert werden müssen. Dies wird leicht dadurch klar, dass stark vereinfacht die oberste und unterste Zeile eines Höhenfeldes synchronisiert werden muss. Da die Anzahl der Zellen auf einem Gitter der Größe $n \times n$ quadratisch mit $\mathcal{O}(n^2)$ ansteigt, die Anzahl der Randzellen demgegenüber aber nur linear mit $\mathcal{O}(n)$ wächst, sinkt der Anteil $\frac{\mathcal{O}(n)}{\mathcal{O}(n^2)}$ der zu synchronisierenden Zellen sogar noch mit steigender Problemgröße.

Gittergröße	Jobanzahl			
	2	3	4	8
250^2	1,9	2,3	2,4	4,0
500^2	2,1	3,1	4,1	5,8
1000^2	2,1	3,0	3,8	5,9
1500^2	2,0	3,0	4,0	6,4
2000^2	2,0	2,9	4,0	6,5
2400^2	2,0	3,0	4,0	6,5
2800^2	2,1	3,0	4,0	7,6

Tabelle 6.8: Gewonnener Speedup zwischen verschiedenen Konfigurationen auf einem Opteron Cluster

Tabelle 6.8 zeigt nun für alle Konfigurationen mit mehreren Jobs den gewonnenen Speedup gegenüber den Löserversionen die nur mit einem Job rechnen. Bis auf den Löser mit acht Jobs erreichen alle Löser insbesondere für große Gitter außergewöhnlich gute Speedup-Faktoren und skalieren damit perfekt. In der letzten Spalte wird deutlich, dass der Speedup-Faktor des Lösers mit 8 Jobs mit wachsender Gittergröße ansteigt und noch nicht sein Maximum erreicht hat. Außerdem ist erkennbar, dass die Löser mit zwei, drei und vier Jobs bei sehr kleinen Gittergrößen atypische Speedup-Faktoren erreichen. Bei einer Gittergröße von 250^2 ist der Sychronisationsaufwand für drei und vier Jobs so groß, dass der Geschwindigkeitsgewinn durch mehrere parallel Rechnende Knoten kaum zum tra-

gen kommt. Demgegenüber rechnen die Löser auf einem Höhenfeld mit einer Gesamtgröße von 500^2 etwas schneller, als naiv betrachtet theoretisch möglich wäre. Das ist dadurch zu erklären, dass Berechnungen auf dem gesamten Höhenfeld durch einen Job den vorhandenen Prozessorcache aufgrund der zu großen Datenmenge nicht so gut nutzen können wie die Berechnungen, die nur einen Teil des Höhenfeldes in jedem der zwei bis vier Jobs verarbeiten müssen. Diese kleineren Daten ermöglichen es die Berechnung in den einzelnen Jobs und damit auch die Gesamtberechnung überdurchschnittlich stark zu beschleunigen. Gleichzeit hat die notwendige Synchronisation zwischen den Knoten bei dieser Gittergröße nun keinen wesentliche Einfluß mehr auf die Gesamtrechenzeit des Lösers mit zwei bis vier Jobs.

Schwache Skalierbarkeit

Die schwache Skalierbarkeit soll mit Hilfe des LiDO Clusters untersucht werden, dessen Knoten aus durch ein Infiniband Netzwerk verbundenen Opteron DP 250 Systemen bestehen, wie sie in Tabelle 6.1 näher beschrieben sind. Hierzu wird zunächst auf einem Knoten ein Problem mit der Gittergröße $n = 442^2$ berechnet und 100 Zeitschritte durchgeführt. Im Anschluß werden Gittergröße und Knotenanzahl mehrfach gleichzeitig verdoppelt. Bei zwei Knoten wird also ein Höhenfeld der Form 625×625 verwendet, da $625^2 = 442^2 \cdot 2$ ist. Diese relative kleine innitiale Größe wurde deshalb gewählt, da es in der aktuellen Implementierung noch notwendig ist, dass ein Knoten zu Anfang alle zu berechnenden Teilhöhenfelder gleichzeitig erstellt. Er muss also die Datenstrukturen des gesamten Höhenfeldes und gleichzeitig die Datenstrukturen aller Teilhöhenfelder im Arbeitsspeicher verwalten. Für 32 Knoten ist damit der maximale Arbeitsspeicher eines Knotens ausgelastet. Abbildung

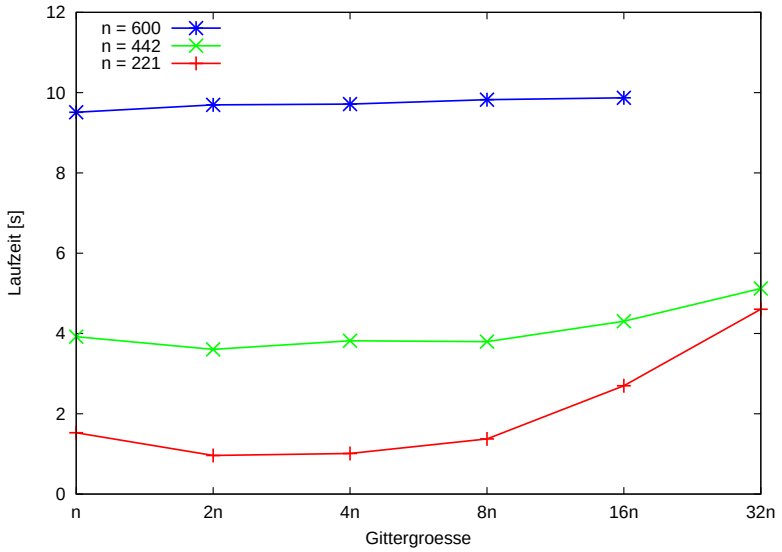


Abbildung 6.9: Schwache Skalierbarkeit auf dem LiDO Cluster

6.9 zeigt die Ausführungszeiten für diese Konfigurationen. Die Ausführungszeit bleibt trotz ansteigender Gittergröße durch den gleichzeitigen Einsatz von zusätzlichen Rechenknoten zunächst nahezu konstant. Jedoch wird bei 32 Rechenknoten deutlich, dass die Laufzeit merklich ansteigt. Dies ist darauf zurückzuführen, dass jeder Knoten sein relativ kleines Teilhöhenfeld sehr schnell berechnet hat, die Synchronisation zwischen der gestiegenen Anzahl an Knoten aber die Gesamtlaufzeit negativ beeinflusst.

Um dies zu verdeutlichen, wurde dieselbe Messung mit einer initialen Gittergröße von $n = 221^2$ erneut durchgeführt. Bei diesem nochmals kleineren Gitter wird der negative Einfluss der Synchronisation auf die Laufzeit bei 16 und 32 verwendeten Knoten noch deutlich stärker sichtbar.

Verzichtet man auf die Nutzung von 32 Knoten, so lässt sich auch ein Gitter mit einer höheren initialen Größe von $n = 600^2$ verwenden. Dabei zeigt sich, dass die Laufzeit nun nahezu konstant bleibt und die Synchronisation den Löser bei dieser Gittergröße also nicht mehr ausbremst.

7 Zusammenfassung

In dieser Arbeit wurden zunächst anhand der in Kapitel 2 erläuterten Hardwarearchitekturen zwei wesentliche Techniken zur Beschleunigung wissenschaftlicher Berechnungen vorgestellt: Die Vektorisierung von gleichen Rechenoperationen auf verschiedenen Daten (SIMD) und die Parallelisierung der eigentlichen Programmausführung, sei es durch Threads in einem Multicoresystem beziehungsweise auf der Grafikkarte oder durch jeweils eigene Programme auf verschiedenen Knoten eines Clusters.

Im Anschluß wurde in Kapitel 4 die HONEI Bibliothek vorgestellt, die die Entwicklung von Anwendungen dieser Art unterstützen und vereinfachen soll, indem häufig benötigte Routinen zur Verfügung gestellt werden. Beispiele hierfür sind die grundsätzliche Verwaltung der Daten in verschiedenen Speicherbereichen oder das Ausführen von Programmen auf einer SPE. Zur Anwendungsentwicklung steht dabei besonders das Konzept des Anwendungskernels im Mittelpunkt, also die Zusammenfassung möglichst großer Teile einer Anwendung, um auf dieser Ebene möglichst stark optimieren zu können.

Bei der Entwicklung einer neuen Anwendung ist es somit möglich sich auf die spezifischen Problemstellungen der Anwendung zu konzentrieren. Gleichzeitig ist es notwendig, die Berechnungen der Anwendung (und ihrer erarbeiteten Kernel) in Hinblick auf die oben genannten Techniken Vektorisierung und Parallelisierung zu formulieren und dafür geeignete Datenstrukturen zu finden.

Dieses Konzept wurde in Kapitel 5 am Beispiel eines Lattice Boltzmann Lösers demonstriert. Dazu wurden vier verschiedene Anwendungskernel identifiziert und Datenstrukturen entwickelt, um es diesen Kerneln zu ermöglichen, die Daten vektorisiert zu verarbeiten. Es wurde deutlich, dass ein sehr hardwarenah implementierter Kernel notwendig ist, um die sehr verschiedenen Eigenschaften und Ressourcen der jeweiligen Hardware voll nutzen zu könnnen. Beispiele sind der geteilte Speicher auf einem Streaming Multiprocessor oder das Ausrollen einer Schleife von SSE Code um genau die Anzahl der verfügbaren Register auszuschöpfen. Damit parallele Berechnungen möglichst selten durch notwendige Synchronisationen unterbrochen werden, wurde ein Parallelisierungskonzept auf Anwendungsebene entwickelt, bei dem viele Löser ihr eigenes Teilproblem lösen und sich erst nach einem kompletten Zeitschritt synchronisieren.

Die Leistung dieses Lösers und der verwendeten Hardware wurde schließlich in Kapitel 6 untersucht. Dabei bestätigte sich die vielversprechende Leistungsfähigkeit von Grafikkarten, die auf Kernelebene bis zu einem gemessenen Faktor 100 schneller rechnen als ein

gewöhnliches x86 System. Auf Applikationsebene beträgt der maximale Speedup immer noch bis zu 60, da sich nicht alle Kernel gleichermaßen beschleunigen lassen. Demgegenüber zeigte der Vergleich zwischen einem Opteron und einem Core i7 System zwar einen deutlichen Fortschritt in der Geschwindigkeit, allerdings bewegen sich die x86 Systeme inzwischen auch mit ihrer immer weiter steigenden Anzahl an Kernen in einer anderen Leistungsklasse als die Grafikkarten. Weiterhin bleibt festzuhalten, dass sowohl der Vergleich Opteron - Core i7 als auch der Vergleich GTX 8800 - GTX 285 zeigt, dass die jeweils nächste Hardwaregeneration die Leistung des Lösers durch neue Techniken deutlich beschleunigt. Hierbei wird neben einer besseren Anbindung an den jeweiligen Arbeitsspeicher besonders aus einer Erhöhung der Recheneinheiten zusätzliche potentielle Leistung gewonnen. Insgesamt hat sich das Konzept von hardwarenahen Anwendungskerneln, die von einer portablen Anwendung aufgerufen werden, bewährt.

Im Rahmen dieser Arbeit haben sich einige zusätzliche Fragestellungen herauskristallisiert, die den Umfang der Arbeit übersteigen, aber in Zukunft von Interesse sein mögen:

Zunächst wird ganz konkret von Geveler [16] der vorgestellte LBM Löser um die Möglichkeit erweitert, um mit einem Bodenprofil und mit festen Objekten innerhalb des Fluids zu interagieren. Dadurch ergibt sich im Anschluss automatisch die Frage, wie weit diese Erweiterungen auf alle vorgestellten Architekturen effizient übertragen werden können.

Die Probleme des LBM Lösers mit einem zu geringen Arbeitsspeicher auf Cell BE und Cluster können durch ein Umformulieren der `GridPacker` Methoden zur Erstellung der Datenstrukturen reduziert werden, so dass der Gesamtspeicherverbrauch nicht wesentlich die Größe der für einen Zeitschritt notwendigen Daten überschreitet. Gleichzeitig vermag ein adaptiv arbeitender `GridPacker` die Zellen des Höhenfeldes komplett neu anzuordnen, um tatsächlich alle vektorisiert zu verarbeitenden Zellen zu identifizieren und zusammenzufassen.

Neueste Entwicklungen in der Technik erscheinen ebenfalls vielversprechend. So gibt es inzwischen Systeme mit mehreren Grafikkarten, die in HONEI als eine Kombination einer Multicore Applikation aufgefasst werden können, deren einzelne Threads keinen x86 sondern einen CUDA Kernel ausführen. Daneben gibt es inzwischen mit OpenCL [31] eine Spezifikation zur einheitlichen Entwicklung von Anwendungen für unter anderem die in dieser Arbeit vorgestellten Hardwarearchitekturen.

Literaturverzeichnis

[1] Bauke, H. und S. Mertens: *Cluster Computing : Praktische Einführung in das Hochleistungsrechnen auf Linux-Clustern.* Springer, September 2005.

[2] Becker, C. und S. Turek: *FEATFLOW – Finite element software for the incompressible Navier-Stokes equations.* User Manual, Universität Dortmund, 1999.

[3] Becker, H.: *GPU-basierte Echzeit-Simulation und Visualisierung von Shallow Water Equations*, 2006. Diplomarbeit. Fakultät Informatik, TU Dortmund.

[4] Bradford, N., D. Buttlar und J. Farrell: *Pthreads programming.* O'Reilly, 1998.

[5] Briggs, W., H. Van Emden und S. McCormick: *A multigrid tutorial (2nd ed.).* SIAM, Philadelphia, PA, USA, 2000.

[6] Buttari, A., P. Luszczek, J. Kurzak, J. Dongarra und G. Bosilca: *SCOP3: A rough guide to scientific computing on the PlayStation 3.* Technischer Bericht, Innovative Computing Laboratory, University of Tennessee Knoxville, 2007. UT-CS-07-595.

[7] Calcote, J.: *Autotools.* No Starch Press, 2009.

[8] Charney, J., R. Fjörtoft und J. von Neumann: *Numerical integration of the barotropic vorticity equation.* Tellus, 2:237–254, 1950.

[9] Delis, A. und T. Katsaounis: *Numerical solution of the two-dimensional shallow water equations by the application of relaxation methods.* Applied Mathematical Modelling, 29(8):754–783, 2005.

[10] Donath, S., K. Iglberger, G. Wellein, T. Zeiser, A. Nitsure und U. Rüde: *Performance comparison of different parallel lattice Boltzmann implementations on multi-core multi-socket systems.* International Journal of Computational Science and Engineering, 4(1):3–11, 2008.

[11] Dyk, D. van, M. Geveler, S. Mallach, D. Ribbrock, D. Göddeke und C. Gutwenger: *HONEI: A collection of libraries for numerical computations targeting multiple processor architectures.* Computer Physics Communications, 40th Anniversary Issue, 2009. (akzeptiert). doi:10.1016/j.cpc.2009.04.018.

[12] Fan, Z., F. Qiu, A. Kaufman und S. Yoakum-Stover: *GPU cluster for high performance computing.* In: *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Seite 47, 2004.

[13] Fatahalian, K. und M. Houston: *A closer look at GPUs.* Communications of the ACM, 51(10):50–57, 2008.

[14] Flynn, M.: *Some computer organizations and their effectiveness.* IEEE Transactions on Computers, C-21:948+, 1972.

[15] Garland, M., S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang und V. Volkov: *Parallel computing experiences with CUDA.* IEEE Micro, 28(4):13–27, 2008.

[16] Geveler, M.: *Echtzeitfähige Interaktion von Festkörpern mit 2D Lattice Boltzmann Flachwasserströmungen in 3D Virtual-Reality Anwendungen*, 2009. Diplomarbeit. Fakultät Informatik, TU Dortmund.

[17] Ghia, U., K. Ghia und C. Shin: *High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method.* Journal of Computational Physics, 48(3):387–411, 1982.

[18] Göddeke, D., R. Strzodka und S. Turek: *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations.* International Journal of Parallel, Emergent and Distributed Systems, 22(4):221–256, 2007.

[19] Goto, K.: *GotoBLAS.* `http://www.tacc.utexas.edu/resources/software/#blas`.

[20] Grimes, R., D. Kincaid und D. Young: *ITPACK 2.0 user guide.* Center for Numerical Analysis, University of Texas, 1979. Technical Report CNA-150.

[21] Gropp, W., S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir und M. Snir: *MPI: The complete reference.* The MIT Press, 2nd Auflage, 1998.

[22] Hamill, P.: *Unit test frameworks.* O'Reilly, 2004.

[23] Hausmann, S.: *Optimization and Performance Analysis of the Lattice Boltzmann Method on x86-64 based Architectures*, 2005. Bachelor's Thesis. Lehrstuhl für Informatik 10 (Systemsimulation), Friedrich-Alexander-Universität Erlangen-Nürnberg.

[24] Haviv, Y.: *Infiniband: past, present and future.* Technischer Bericht, CERN, Geneva, 2004.

[25] Hübner, T. und S. Turek: *Efficient simulation techniques of the Lattice Boltzmann equation on unstructured meshes.* Technischer Bericht, Fakultät für Mathematik, TU

Dortmund, 2007. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 355.

[26] IBM Corporation: *SPE runtime management library.* `http://www-01.ibm.com/chips/techlib/techlib.nsf/pages/main`, 2007.

[27] Inoue, A. und A. Maeda: *The architecture of a multi-vector processor system.* Parallel Computing, 8(1-3):185–193, 1988.

[28] Intel Corporation: *Intel 64 and IA-32 architectures software developer's manual*, 2007.

[29] ITMC Universität Dortmund: *LiDO.* `http://www.itmc.tu-dortmund.de/en/hochleistungsrechnen/lido/index.html`.

[30] Kahle, J., M. Day, H. Hofstee, C. Johns, T. Maeurer und D. Shippy: *Introduction to the Cell multiprocessor.* IBM Journal of Research and Development, 45(4/5):589–604, 2005.

[31] Khronos OpenCL Working Group: *The OpenCL specification, version 1.0.* `http://www.khronos.org/opencl`, Dezember 2008.

[32] Kistler, M., J. Gunnels, D. Brokenshire und B. Benton: *Programming the Linpack benchmark for the IBM PowerXCell 8i processor.* Scientific Programming, 17(1-2):43–57, 2009.

[33] Körner, C., T. Pohl, U. Rüde, N. Thürey und T. Zeiser: *Parallel Lattice Boltzmann Methods for CFD applications.* Numerical Solution of Partial Differential Equations on Parallel Computers, 51:439–465, 2006.

[34] Li, X. und A. Kaufman: *Implementing Lattice Boltzmann computation on graphics hardware.* The Visual Computer, 19:444–456, 2003.

[35] Lindholm, E., J. Nickolls, S. Oberman und J. Montrym: *NVIDIA Tesla: A unified graphics and computing architecture.* IEEE Micro, 28(2):39–55, 2008.

[36] Mallach, S.: *Beschleunigung ausgewählter paralleler Standard Template Library Algorithmen*, 2008. Diplomarbeit. Fakultät Informatik, TU Dortmund.

[37] Nickolls, J., I. Buck, M. Garland und K. Skadron: *Scalable parallel programming with CUDA.* ACM Queue, 6(2):40–53, 2008.

[38] NVIDIA Corporation: *NVIDIA CUDA Compute unified device architecture programming guide (version 2.2).* `http://www.nvidia.com/cuda`, 2009.

[39] Owens, J., D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn und T. Purcell: *A survey of general-purpose computation on graphics Hardware.* Computer Graphics Forum, 26(1):80–113, 2007.

[40] Peng, L., K. Nomura, T. Oyakawa, R. Kalia, A. Nakano und P. Vashishta: *Parallel Lattice Boltzmann Flow Simulation on Emerging Multi-core Platforms.* Springer-Verlag, Berlin, Heidelberg, 2008.

[41] Perrone, M. und T. Sowadagar: *Cell BE software programming and toolkits.* In: *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

[42] PG SmartCell: *Endbericht der Projektgruppe SmartCell.* Technischer Bericht, Fakultät für Informatik, TU Dortmund, 2008.

[43] Pham, D.: *Key considerations given to the design of a next generation multi-core communications platform.* Integrated Circuit Design and Technology, 2008. ICICDT 2008. IEEE International Conference, 2008.

[44] Pohl, T., N. Thürey, F. Deserno, U. Rüde, P. Lammers, G. Wellein und T. Zeiser: *Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures.* 2004. Supercomputing Conference 04.

[45] Qt Software: *QT.* `http://www.qtsoftware.com/`.

[46] Sauro, S.: *Lattice Boltzmann Equation for relativistic quantum mechanics.* Philosophical Transactions: Mathematical, Physical and Engineering Sciences, 360:429–436, 2002.

[47] Shreiner, D.: *OpenGL programming guide.* Addison-Wesley, Upper Saddle River, NJ, 2006.

[48] Sillicon Graphics: *STL Webseite.* `http://www.sgi.com/tech/stl/`.

[49] Squyres, J.: *Definitions and fundamentals – the Message Passing Interface (MPI).* ClusterWorld Magazine, MPI Mechanic Column, 1(1):26–29, 2003.

[50] Stürmer, M., J. Götz, G. Richter und U. Rüde: *Blood flow simulation on the Cell Broadband Engine using the Lattice Boltzmann Method.* Nummer 07-9. 2007.

[51] Thürey, N., T. Pohl und U. Rüde: *Hybrid parallelization techniques for Lattice Boltzmann free surface flows.* Proceedings of Parallel CFD 2007, Seiten 1–8, 2007.

[52] Tölke, J. und M. Krafczyk: *TeraFLOP computing on a desktop PC with GPUs for 3D CFD.* International Journal of Computational Fluid Dynamics, 22(7):443–456, 2008.

[53] Wellein, G., P. Lammers, G. Hager, S. Donath und T. Zeiser: *Towards optimal performance for Lattice Boltzmann applications on terascale computers.* Parallel Computational Fluid Dynamics: Theory and Applications, Seiten 31–40, 2005.

[54] Wellein, G., T. Zeiser, S. Donath und G. Hager: *On the single processor performance of simple Lattice Boltzmann kernels.* Computer & Fluids, 35:910–919, 2006.

[55] Wilke, J., T. Pohl, M. Kowarschik und U. Rüde: *Cache Performance Optimizations for Parallel Lattice Boltzmann Codes in 2D.* Technischer Bericht 03–3, Lehrstuhl für Informatik 10 (Systemsimulation), University of Erlangen-Nuremberg, Germany, 2003.

[56] Wilkes, M.: *The memory gap (keynote).* 2000. `http://www.ece.neu.edu/conf/wall2k/wilkes1.pdf`.

[57] Zeiser, T., G. Wellein, G. Hager, S. Donath, F. Deserno, P. Lammers und M. Wierse: *Optimized Lattice Boltzmann kernels as testbeds for processor performance.* Technischer Bericht, Lehrstuhl für Informatik 10 (Systemsimulation), University of Erlangen-Nuremberg, 2005.

[58] Zhao, J., T. Xie und N. Li: *Towards regression test selection for aspect-oriented programs.* In: *Proc. 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP 2006)*, Seiten 21–26, 2006.

[59] Zhao, Y.: *Lattice Boltzmann based PDE solver on the GPU.* The Visual Computer, 24(5):323–333, 2008.

[60] Zhou, J.: *Lattice Boltzmann Methods for shallow water flows.* Springer, 2004.