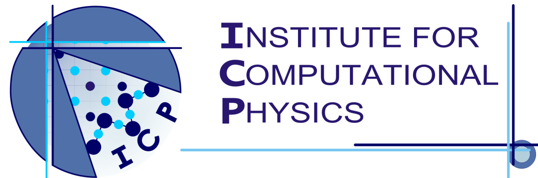
Universität Stuttgart: Fachbereich Physik



Hauptseminar: Moderne Simulationsmethoden in der Physik

Ausarbeitung zum Vortrag

ATOMISTISCHE SIMULATIONEN AUF NEUARTIGEN PROZESSORARCHITEKTUREN - SIMULATIONEN AUF GRAFIKPROZESSOREN

Marco Di Sarno

18. Februar 2010

Inhaltsverzeichnis

1 Einführung **3**
- 1.1 Fortschritt in der Mikroprozessortechnik . . . 3
- 1.2 Special Purpose Computing . . . 3

2 Moderne Prozessorarchitekturen **4**
- 2.1 Field programmable gate arrays . . . 4
- 2.2 Mehrkernprozessoren . . . 5
- 2.3 Grafikprozessoren . . . 6

3 Simulationen auf Grafikprozessoren **7**
- 3.1 Compute Unified Data Architecture . . . 7
  - 3.1.1 Ausführungsmodell . . . 8
  - 3.1.2 Datenflussschema . . . 10
- 3.2 Molekulardynamikpakete . . . 12
  - 3.2.1 GPULib . . . 12
  - 3.2.2 HOOMD-blue . . . 12

4 Anwendungsbeispiel: Lennard-Jones Fluid **13**
- 4.1 Vorbereitung . . . 13
- 4.2 Realisierung auf der GPU . . . 13
- 4.3 Simulation mit HOOMD-blue . . . 15
- 4.4 Fazit . . . 17

5 Abschließende Bemerkungen **17**

Referenzen **R**

1 Einführung

1.1 Fortschritt in der Mikroprozessortechnik

Immer mehr Probleme lassen sich heutzutage mittels Simulation am Computer lösen. Grundlage dafür ist in erster Linie der anhaltende Fortschritt in der Mikroprozessortechnik, der die Entwicklung immer aufwändigerer und damit rechenintensiverer Algorithmen ermöglicht. Hier hat man über die letzten vier Jahrzehnte Leistungssteigerungen vor allem durch eine Erhöhung der Taktfrequenz bei steigender Dichte der integrierten Schaltkreise pro Chip erreicht. Auf diese Art uns Weise konnte der von Moore in den 60er Jahren vorhergesagte (nahezu) exponentielle Leistungszuwachs bis heute aufrecht erhalten werden. Allerdings hat diese Strategie gleich zweierlei Grenzen: Zum einen steigt die Wärmeentwicklung irgendwann über jedes vernünftige Maß, zum anderen lassen sich die Komponenten nicht beliebig klein machen, da sonst Quanteneffekte zu Fehlern führen können[1]. Diese Grenze wird schätzungsweise 20xx erreicht.

Als Reaktion darauf hat man das Augenmerk in den letzten Jahren hauptsächlich auf die Umsetzung neuer Prozessorarchitekturen gelegt. Für den Hausgebrauch bedeutet dies Verwendung von mehreren, parallel laufenden Prozessoren pro Chip. Dabei wird dem Verbraucher suggeriert, er erhalte durch die Verdopplung der Prozessorkerne eine Verdopplung der Rechenleistung, was nur bedingt stimmt; nämlich nur für Programme, in denen nahezu alle Programmteile parallel arbeiten können. Das Amdahlsche Gesetz beschreibt diesen Sachverhalt recht treffend.[2]

1.2 Special Purpose Computing

Eine weitere Möglichkeit, die Leistung zu steigern, ohne sich den oben genannten Problemen stellen zu müssen, besteht darin, problemspezifische Hardware zu entwickeln, beziehungsweise die Recheneinheiten auf die im Problem dominanten Rechenoperationen zu optimieren. Das Arbeitsfeld, das sich mit solchen Architekturen befasst, nennt sich Special Purpose Computing.

Solche Rechner kann man zum Beispiel aus so genannten Anwendungsspezifischen Integrierten Schaltungen (engl. application specific integrated circuit, kurz: ASIC) aufbauen, also Schaltungen die spezielle Aufgaben effektiv bearbeiten können, dafür aber, einmal hergestellt, unveränderlich sind. Einen anderen Ansatz verfolgen die so genannten FPGAs, die weiter unten ausführlich behandelt werden.

Im Folgenden seien zwei Beispiele für Special Purpose Computing genannt:

IBM Deep Blue

Deep Blue war ein von IBM entwickelter Schachcomputer. Er ist der erste Computer, dem es gelang, den Schachgroßmeister Kasparov zu besiegen (1996/97). Diese legendären Partien gingen in die Geschichte des Schachsports ein. Deep Blue war aufgebaut aus 30 IBM/RS6000 Chips[3], die von insgesamt 480 speziellen Schachchips[4] unterstützt wurden. Diese Chips hatten drei Komponenten: Ein Zuggenerator (move generation) spielt in einer zufälligen Abfolge alle möglichen Stellungen durch, eine Evaluierungsfunktion bewertet diese dann und eine Kontrolleinheit (search control) stoppt die Suche, wenn ein hinreichend guter nächster Zug gefunden wurde. (Anmerkung: Deep Blue wurde nach dem Sieg gegen Kasparov demontiert und bleibt somit auf alle Zeit ungeschlagen.)

IBM MD-GRAPE

Der von IBM entwickelte MD-Grape Chip dient zur Berechnung von Molekulardynamiksimulationen. Er ist spezialisiert auf die Berechnung von zwischenmolekularen Kräften in großen Systemen. Die MD-Grape Chips gibt es unter anderem als rechenbeschleunigende PCI-Karten, die auf ein handelsübliches Motherboard passen. So kann der Host (also die CPU) seine volle Rechenkapazität für die Berechnung der Bewegungen der Teilchen während eines Zeitschrittes nutzen, während die MD-Grape-Karte die Gesamtkraft auf jedes einzelne Teilchen liefert. Im Juni 2006 wurde mit 4824 solcher Chips am japanischen Forschungszentrum RIKEN ein Molekulardynamiksupercomputer realisiert, der RIKEN MD-GRAPE-3[5]. Als Host dient hier ein Cluster aus 64 Servern zu insgesamt 256 Intel Dual-Core Intel Xeon Prozessoren. Der MD-GRAPE-3 erreichte in seiner Disziplin bereits die Petaflop-Grenze, während diese erst 2 Jahre später von einem Allzwecksupercomputer, namentlich dem Roadrunner am LANL, erreicht wurde.

2 MODERNE PROZESSORARCHITEKTUREN

In diesem Abschnitt seien drei prinzipiell unterschiedliche Prozessorarchitekturen beschrieben: FPGAs, Mehrkernprozessoren und Grafikprozessoren.

2.1 FIELD PROGRAMMABLE GATE ARRAYS

Aus im Anwendungsfeld programmierbaren Anordnungen von Logikgattern (engl. field programmable gate arrays, kurz: FPGAs) lassen sich nahezu beliebig komplexe logische Schaltungen aufbauen. „Programmierbar“ meint hier nicht die Programmierung der zeitlichen Abfolge von Funktionen im Baustein, sondern vielmehr die Programmierung der zugrunde liegenden Funktionsstruktur. In diesem Sinne spricht man zur begrifflichen Abgrenzung statt dessen oft von der Konfiguration eines FPGAs. FPGAs sind aufgebaut

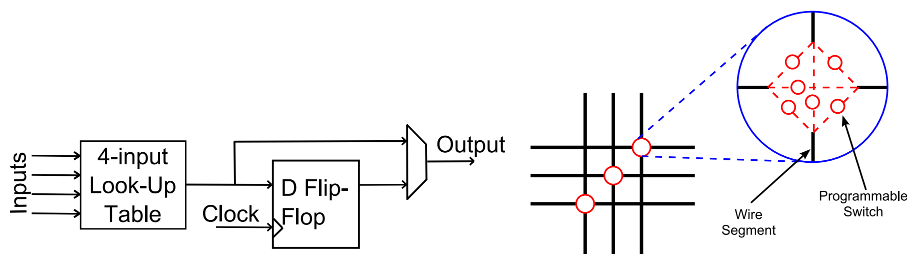


ABB. 2.1: **Links**: Logikblock eines FPGAs, aufgebaut aus einer Lookuptabelle und einem 1-Bit Register. **Rechts**: Schaltmatrix, auf der die Logikblöcke aufgetragen werden. An den Kreuzungspunkten befinden sich programmierbare Schalter.

aus so genannten Logikblöcken , die in ein Netz aus programmierbaren Schaltern (Schaltmatrix) eingebettet sind (Abb. 2.1). Jeder Logikblock verfügt über eine Lookuptabelle und ein 1-Bit Register. Eine Lookuptabelle ist im Prinzip eine als Wahrheitstabelle realisierte logische Funktion, die über einen externen Speicher eingelesen wird. Die Dimension der Tabelle bestimmt sich durch die Anzahl ihrer Eingänge - üblicherweise werden 4- oder 6-Kanallookuptabellen verwendet. Zusätzliche Multiplexerstrukturen eröffnen die Möglichkeit der direkten Kommunikation zwischen Logikbausteinen. Die frei wählbare Verschaltung dieser willkürlich konfigurierter Logikbausteine verleiht solchen Schaltungen ihre

Vielseitigkeit und macht diese damit zum optimalen Baustein für Special Purpose Computer. Auch werden FPGAs als Substrat für komplexere Komponenten, wie zum Beispiel fertige Prozessoren, genutzt, zum Beispiel, um bestimmte Programmkomponenten zu beschleunigen.

Wirtschaftlich gesehen ist der Hauptvorteil von FPGAs die Möglichkeit zur Rekonfigurierung. Während ASICs in aller Regel günstiger in der Herstellung sind, aber unflexibel und damit ungeeignet für Testzwecke sind, werden in frühen Entwicklungs- und Testphasen FPGAs bevorzugt. Entsprechend gern gesehen sind sie daher auch auf dem Feld der Computersimulationen, da sie an nahezu alle Probleme angepasst werden können. Allerdings erfordert die Konfiguration von FPGAs einige Einarbeitung, nicht zuletzt deshalb, weil jeder Hersteller seine eigenen Werkzeuge mitbringt, die sich mitunter stark unterscheiden.

2.2 Mehrkernprozessoren

Als Mehrkernprozessor bezeichnet man einen Chip, auf dem mehrere vollwertige Hauptprozessoren, also Kerne, parallel arbeiten. Dabei sind zumindest die Hauptkomponenten, also eine arithmetisch-logische Einheit (engl. arithmetic logical unit, kurz: ALU) und die Registersätze, für jeden Kern repliziert. Oft gilt dies auch für die Caches. Besteht der Prozessor aus zwei Kernen, spricht man von einem Dual-Core, bei drei Kernen von einem Triple-Core und bei vier Kernen von einem Quad-Core. Darüber hinaus gibt es noch keine gängigen Bezeichnungen, obwohl es längst schon Prozessoren mit acht und mehr Kernen gibt.

Hat ein Mehrkernprozessor n Kerne, so ist theoretisch eine Leistungssteigerung gegenüber nur einem solchem Prozessorkern um das n-fache möglich. Wie bereits erwähnt, hängt dies davon ab, wie gut ein Programm parallelisiert ist - es bleibt allerdings so, dass manche Programmteile sequenziell ablaufen müssen.

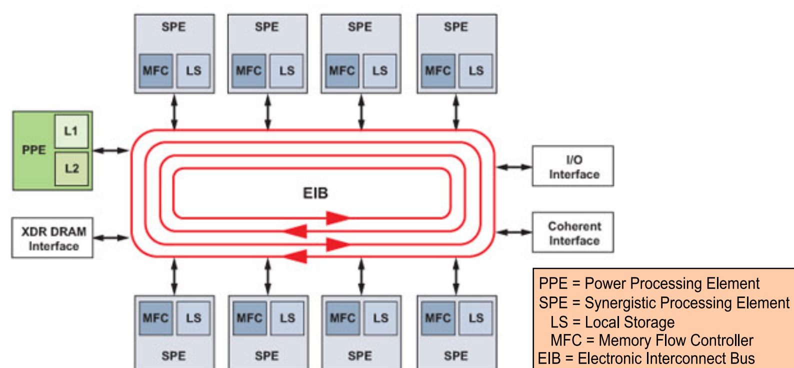


Abb. 2.2: IBM PowerXCell 8i. Quelle: NASA Homepage

Als ein Beispiel für einen aktuellen und sehr leistungsstarken Mehrkernprozessor sei der PowerXCell 8i (Abb. 2.2) von IBM genannt. Er besteht aus einem 64-Bit Prozessor (PowerPC processing element, kurz: PPE) auf Basis der PowerPC-Architektur und insgesamt acht Recheneinheiten, bzw. ALUs, (synergistic processing element, kurz: SPE) mit vierfachem SIMD, sprich jedes SPE kann pro Zyklus dieselbe Operation auf 4 verschiedene Datenströme anwenden. Zusätzlich bietet ein innovativer Bus die Basis für optimalen Datenfluss. Konzipiert wurde der Cell für rechenintensive Anwendungen, wie zum Beispiel Grafikanwendungen, Videodecoder, Kryptographie, etc. Mit diesen Spezifikationen ist der Spagat gelungen: Stark erhöhte Rechenleistung durch acht parallel arbeitende Prozessoren

bei gleichbleibender Kontrolle und Funktionalität.

2.3 Grafikprozessoren

Grafikprozessoren (engl. graphics processing unit, kurz: GPU) zeichnen sich durch eine vergleichweise hohe Zahl an parallel arbeitenden Recheneinheiten (ALUs) aus. In dieser Arbeit wird die GPU exemplarisch anhand des GT200-Prozessors von Nvidia beschrieben (Abb. 2.3). Er enthält 30 so genannte „Streaming Multiprozessoren" zu jeweils acht 32-Bit Recheneinheiten, Skalarprozessoren genannt. Insgesamt gibt es also 240 Kerne auf diesem Prozessor. Hier ist allerdings Vorsicht geboten, will man einen Grafikprozessor mit einem Mehrkernprozessor vergleichen - immerhin sind diese 240 Kerne einfache ALUs, keine vollwertigen Prozessoren mit allen Funktionen und den vergleichsweise großen Caches, die ein solcher mit sich bringt. Außerdem hat jeder dieser Multiprozessoren eine 64-Bit Recheneinheit, sowie zwei Einheiten für spezielle Funktionen (engl. special function unit, kurz: SFU), die spezielle mathematische Operationen wie Sinus, Cosinus oder Invertierung (z.B. durch die Verwendung von Lookuptabellen) deutlich schneller bewältigen können als gewöhnliche ALUs. Jeder Multiprozessor kann bis zu 1024 Datenfäden (Threads) parallel bearbeiten, was insgesamt auf eine Summe von 30720 parallel bearbeitbarer Threads führt.

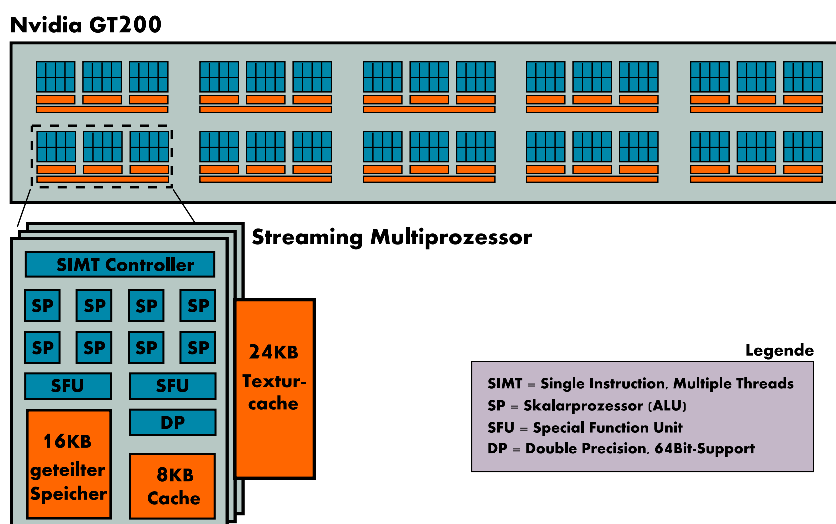


Abb. 2.3: Aufbau eines Grafikprozessors GT200 von Nvidia.

Neben dem globalen Speicher der Grafikkarte, der je nach Hersteller im unteren Gigabytebereich liegt, stehen den Multiprozessoren einige weitere Speicher zu Verfügung:

- ein 8KB constant cache
- ein kleiner geteilter Speicher von 16KB, über den die verschiedenen Threads eines Multiprozessors unter bestimmten Bedingungen kommunizieren können
- ein 24KB großer Texturcache, den sich jeweils 3 Multiprozessoren teilen

Daneben hat jeder Multiprozessor seine eigenen Registersätze. Ein wichtiger Unterschied im Gegensatz zu einer CPU besteht in den Speicherzugriffszeiten. Während diese bei der CPU im Bereich 50 Nanosekunden liegt, liegt sie hier bei 400 Nanosekunden. Programmierer sollten explizit darauf achten, die Caches optimal zu nutzen. Dies ist oft ein Problem, zumal sie im Vergleich zu den Caches einer CPU sehr klein ausfallen.

3 Simulationen auf Grafikprozessoren

Über die letzten Jahre sind Grafikprozessoren, deren eigentliche Aufgabe im Echtzeitrendern von aufwändigen visuellen Effekten in Videospielen besteht, zu ernstzunehmenden Werkzeugen für fließkommaintensive Allzweckberechnungen geworden. Daraus ist ein Arbeitsfeld entstanden, das in der Welt der Wissenschaft unter dem Namen General Purpose computing on Graphics Processing Units' (kurz: GPGPU) einiges an Aufmerksamkeit erhält. Grund dafür ist die immense Rechenleistung dieser Prozessoren (Abb. 3.1). Um sich diese allerdings für solche allgemeinen Berechnungen zu Nutze zu machen ist, musste man sein Problem in ein Problem aus der Bildberechnung verwandeln. Wollte man zum Beispiel eine Molekulardynamiksimulation auf einer GPU durchführen, musste man zunächst ein Objekt finden, in das man Teilchenkoordinaten hineininterpretieren kann. Es boten sich Texturen an: Texturen haben mindestens drei Farb- (R,G,B) und einen Opazitätskanal (alpha). Bei einer Farbtiefe von 32-Bit sind diese Parameter Fließkommazahlen mit einfach Genauigkeit. Verstand man nun den R,G,B-Farbraum als dreidimensionale Box, so war die Verwandlung gelungen: Texturen wurden zu Teilchen, deren R,G,B-Anteile zu den Teilchenkoordinaten. Die nötigen mathematischen Funktionen zur Verrechnung dieser zweckentfremdeten Texturen lieferten dabei die so genannten Fragment Shader. Neben dieser sehr gewöhnungsbedürftigen Art des Programmierens bringen Fragment Shader allerdings weitere Grenzen mit sich, vor allem beim Speichermanagement. Die Shader sind so ausgelegt, dass einzelne Threads ihr Ergebnis nur in jeweils einen Speicherort schreiben können, zudem in einer streng geordneten Art und Weise. Für viele Algorithmen, darunter einige aus dem Bereich der Molekulardynamik, sind allerdings verteilte Schreibzugriffe, teilweise in mehrere Speicherorte gleichzeitig, nötig. Es gab also keine Möglichkeit, die Rechenleistung der GPU direkt für Berechnungen zu nutzen. Dies ändert sich derzeit durch die Entwicklung von low-level-Schnittstellen, die direkten Zugriff auf Hardwarefunktionen haben.

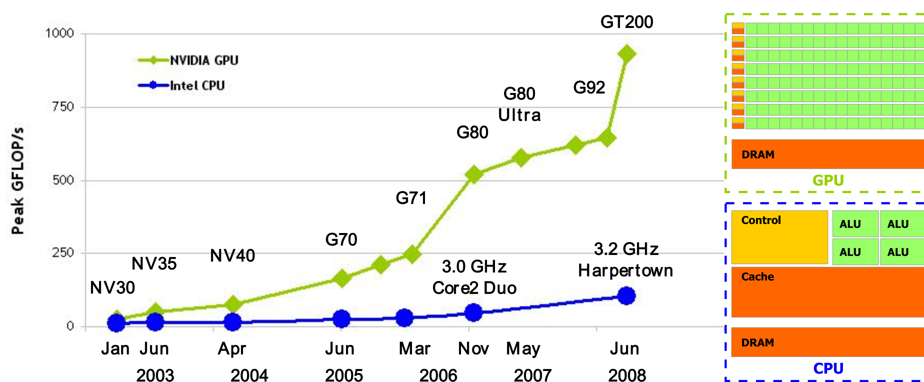


Abb. 3.1: Vergleich der Fließkommaoperationen pro Sekunde zwischen Intel und Nvidia Prozessoren. Daneben: Schematische Darstellung der Unterschiede. Quelle: Nvidia Corp.

3.1 Compute Unified Data Architecture

Eine solche Schnittstelle kommt direkt von Nvidia und nennt sich „Compute Unified Data Architecture“ (kurz: CUDA). Aus Sicht eines Programmierers ist CUDA lediglich eine Code-Erweiterung der Programmiersprache C um einige Nvidia-spezifische Befehle und Konstrukte. Kompiliert wird dieser Code unter Zuhilfenahme eines speziellen Zusatzcompilers von Nvidia (nvcc.exe). Außerdem liefert das CUDA Software Development Kit (CUDA

SDK) verschiedene Bibliotheken für spezielle mathematische Funktionen - eine für Schnelle Fourier Transformation (CUFFT) und eine für Lineare Algebra (CUBLAS) - sodass auch aufwändige Algorithmen effektiv auf der GPU berechnet werden können. Des weiteren gibt es von Drittanbietern Wrapper für so ziemlich jede andere gängige Arbeitssprache (Java, Fortran, Python, .NET, etc.), sogar für High-Level-Sprachen wie IDL und Matlab.

Mit CUDA ist eine Grundlage für das Programmieren auf der GPU gegeben, dennoch kann man hier nicht einfach seine für die CPU geschriebenen Codes verwenden, sondern muss diese an die parallele Natur der GPU anpassen, d.h. den Code so umschreiben, dass die selbe Operation auf möglichst viele voneinander unabhängige Datenstränge ausgeführt wird. Als einfaches Beispiel nehme man die Addition von zwei n-dimensionalen Vektoren: $\vec{a} = \vec{b} + \vec{c}$. Komponentenweise erhält man $a_i \leftarrow b_i + c_i$. In einer gewöhnlichen Implementierung auf der CPU würde man diese Rechnung mit einer Schleife $i = 0 \cdots n$ vornehmen. In CUDA hingegen startet man n verschiedene Threads, die auf nebeneinander liegende Daten in einem float pointer array zugreifen und ihr Ergebnis in ein entsprechendes schreiben. Da diese Berechnungen alle gleichzeitig ausgeführt werden, können die Daten keine Abhängigkeiten untereinander haben, was in der CPU-Implementierung problemlos möglich wäre. Diese Limitierung bereitet bei der Implementierung der meisten Algorithmen Kopfzerbrechen und bedeutet oft den aufwändigsten Teil der Entwicklung.

3.1.1 Ausführungsmodell

Will man effizienten Code für die GPU schreiben, sollte man sich zudem mit dem Ausführungsmodell von CUDA beschäftigen. Threads sind in CUDA zu gleich großen Blöcken zusammengefasst, die je zwischen 32 und 512 Threads enthalten. Die im vorigen Kapitel erwähnte Bedingung für die Interkommunikation von Threads ist die Zugehörigkeit zum selben Block, d.h. nur Threads innerhalb eines Blocks können überhaupt Daten austauschen, Threads verschiedener Blöcke hingegen müssen wirklich völlig unabhängig voneinander sein. Mehrere hundert Blöcke mit insgesamt mehreren tausend Threads sind nötig, um die Leistung der GPU optimal zu nutzen. Jeder Block erhält einen Index (blockID) und jeder der N_B Threads hat einen Index, der seine Position innerhalb des Blocks angibt (threadID). Der Datentyp dieser Indizes nennt sich dim3. Dieser kann je nach Bedarf ein ein-, zwei-, oder dreidimensionales Array sein, was den Vorteil einer intuitiveren Implementierung bietet: Will man zum Beispiel Matrizen multiplizieren, so kann man die Spalten den Index threadIdx.x und den Zeilen den Index threadIdx.y zuordnen. Für die Vektoraddition von oben genügt ein eindimensionaler Index $i \leftarrow blockID \cdot N_B + threadID$. Mit dieser Zuordnung ist sichergestellt, dass jede Komponente der Vektoren von genau einem Thread bearbeitet wird.

Bei der Dimensionierung des Problems muss man noch eine letzte Feinheit des Ausführungsmodells beachten. Intern, d.h. vom Programmierer weder sicht- noch steuerbar, werden von den Multiprozessoren immer genau 32 Threads gleichzeitig aufgerufen. Ein solches Bündel nennt sich Thread Warp. Angenommen, man arbeitet mit 2 Blöcken à 64 Threads (Abb. 3.3), so fasst die GPU diese Threads zu insgesamt 4 Warps zusammen. Von diesen sucht sie sich willkürlich einen heraus, dessen Threads bereits mit Daten bestückt sind und führt ihn aus. Ähnlich wie bei den Zeilen eines Fernsehers wird das Programm also nicht exakt parallel, sondern stückweise ausgeführt und das Ergebnis nach und nach in den Speicher geschrieben. Dieses Modell hat zwei wichtige Konsequenzen.

Zum einen muss man beachten, dass ein Warp die kleinste Einheit im Programm ist, nicht etwa ein Thread. Das bedeutet, dass Blockgrößen von z.B. 5 Threads sehr ungünstig sind, weil diese dann von 27 Threads begleitet werden, die die Operation trotz-

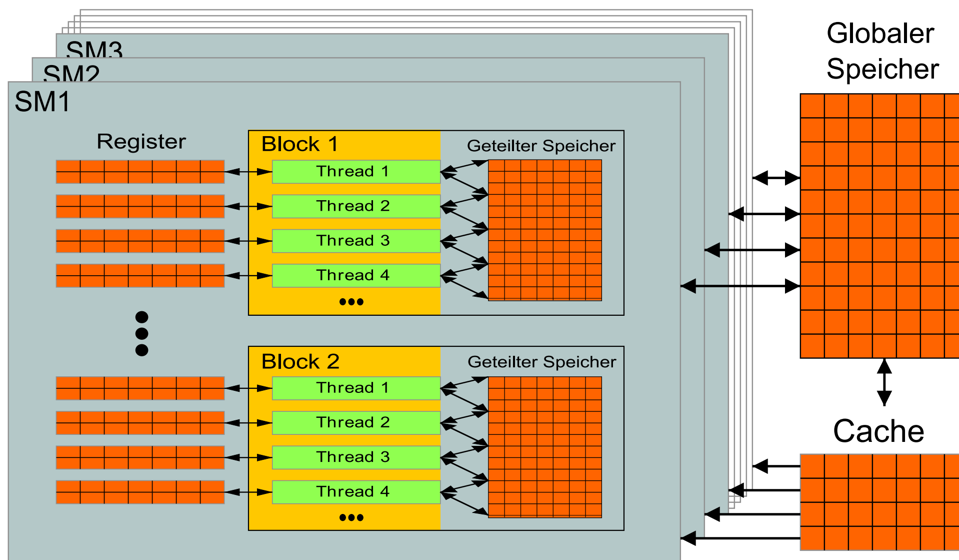


ABB. 3.2: Nvidia Streaming Multiprozessoren mit Blöcken, Threads und Speicher. Quelle: Arnold, ICP, Universität Stuttgart

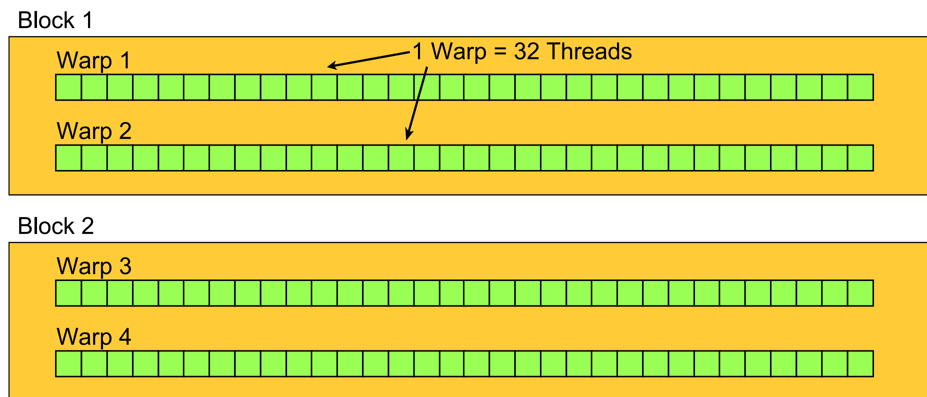


ABB. 3.3: Zusammenspiel von Blöcken und Warps in CUDA.

dem ausführen, ihr Ergebnis aber nirgends niederschreiben, was zu Lasten der Leistung geht. Genauso wichtig ist es, dass alle Threads eines Warps demselben Ausführungspfad folgen. Dies ist zum Beispiel durch Verwendung von if-Ausdrücken oder Schleifen nicht mehr zwingend gegeben und man erhält divergente Ausführungspfade. In diesem Fall findet eine Serialisierung der Ausführung statt, die die Leistung erheblich einbrechen lässt. Hier gibt es allerdings einen im Compiler integrierten Mechanismus, der, sofern die divergenten Pfade nur kurz sind und es nicht zu viele Gabelungen gibt, den gesamten Warp alle Pfade ablaufen lässt, aber durch bestimmte Vorhersagen (engl. predicated instructions) falschen Output verhindert.

Eine zweite wichtige Konsequenz dieses Modells ist, dass es der GPU die Möglichkeit zum Überbrücken der Speicherzugriffslatenz bietet. Der globale Speicher der Grafikkarte hat einerseits eine sehr hohe Bandbreite (> 120GB/s @GTX285), andererseits aber auch eine sehr hohe Latenz (400-600 Taktzyklen[6] = 300-400ns @GTX285, 1.4GHz), also Wartezeit vom Moment der Anforderung eines bestimmten Speicherinhalts bis zu dessen Bereitsstellung. Während dieser Zeit können hunderte von Rechnungen von den Multiprozessoren ausgeführt werden. Der Vorteil des Warp-Modells ist hierbei, dass die einen Warps

bereits ausgeführt werden können, während andere noch darauf warten, dass ihre Daten vom Speicher bereitgestellt werden. So wird erreicht, dass die GPU trotz der Latenzen voll genutzt werden kann.

Bezüglich der Verwendung des globalen Speichers gibt es noch ein weiteres Kriterium, das für die Leistung entscheidend sein kann. Die Speicherzugriffe der Threads auf den globalen Speicher müssen einer bestimmten Ordnung folgen, nämlich müssen benachbarte Threads auf benachbarte Orte im Speicher zugreifen (Abb. 3.4). Dies gilt sowohl für Lese- als auch für Schreibzugriffe. Man spricht dann von der Vereinigung der Threads (engl. coalescing). Nur so entfaltet die Grafikkarte ihre volle Speicherbandbreite - ist dies nicht gegeben, sprich werden Daten in willkürlicher Reihenfolge aus dem Speicher geholt, so bricht die Bandbreite auf etwa ein Zehntel oder ein Zwanzigstel ein. Bei der oben beschriebenen Vektoraddition zum Beispiel erreicht man Vereinigung, wenn gilt: $N_B \mod 32 = 0$, sprich die Blockgröße ein Vielfaches der Warpgröße ist.

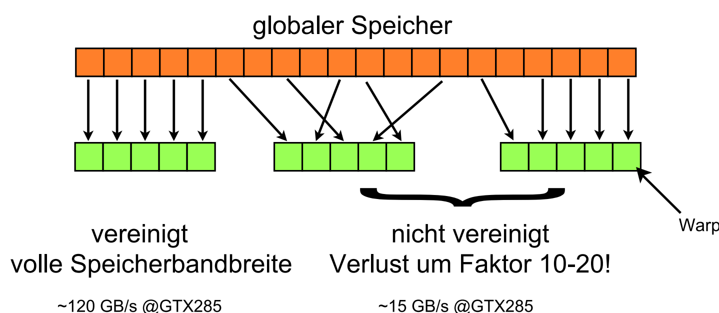


ABB. 3.4: Darstellung zur Verdeutlichung von vereinigten und nicht vereinigten Speicherzugriffen durch Threads.

Zufällige Speicherzugriffe sind für die meisten Algorithmen allerdings nicht völlig vermeidbar. In diesen Fällen kann man auf die verschiedenen Caches zurückgreifen, insbesondere auf den Texturcache. Dieser hat seinen Namen daher, dass er für zweidimensionale Texturen zuständig ist, die auf dreidimensionale Oberflächen gezeichnet werden, wenn die Grafikkarte zur Bildberechnung verwendet wird. Er ist mit 24KB der größte Cache, der den Multiprozessoren zu Verfügung steht und ist als solcher erheblich schneller als der globale Speicher. Dass dies zum Beispiel bei Molekulardynamiksimulationen ein erhebliches Problem darstellt, wird anhand folgender einfachen Rechnung klar: 3 Koordinaten pro Teilchen in 32-Bit Präzision $\hat{=} 3 \cdot 4 = 12$Byte pro Teilchen. Es passt also maximal ein System von etwa 2000 Teilchen in den Texturcache, was für heutige Molekulardynamiksimulationen nur mäßig uninteressant ist. Es ist also auch an dieser Stelle Entwicklungsarbeit nötig, um das Problem diesen Limitierungen anzupassen (siehe Kapitel Anwendungsbeispiel).

3.1.2 DATENFLUSSSCHEMA

Hat man die im vorigen Abschnitt besprochenen Eigenheiten des Ausführungsmodells verinnerlicht, kann man beginnen, seinen Code zu schreiben. Zunächst muss man dazu wissen, wie man überhaupt Daten auf die Grafikkarte bringt, dort berechnet und das Ergebnis ausgibt. CUDA sieht folgendes Datenflussschema vor (Abb. 3.5):

1. Nachdem der nötige Speicherplatz allokiert wurde, können Daten vom Hauptspeicher in den globalen Speicher der Grafikkarte kopiert werden.

2. Kernels werden ausgeführt (Kernels sind Funktionen, die auf der GPU ausgeführt werden)

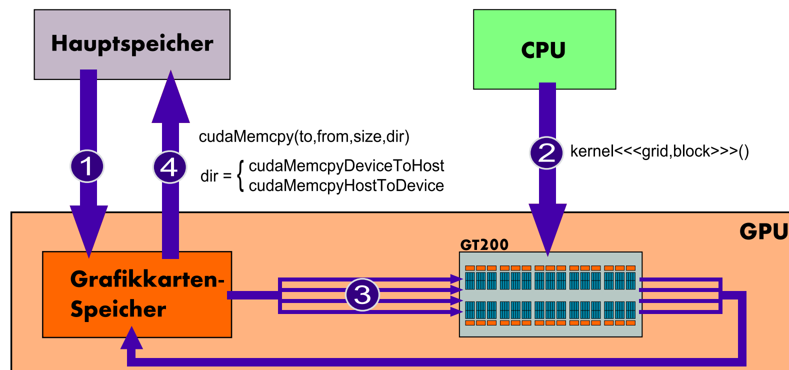


ABB. 3.5: CUDA Datenflussschema

3. Die Ergebnisse werden von den Kernerls in den Grafikkartenspeicher geschrieben.

4. Die Ergebnisse werden in den Hauptspeicher zurück kopiert und können von dort weiterverwendet werden.

```
#include <stdio.h>
#include <cuda.h>

// Kernel, wird auf der Grafikkarte ausgeführt
__global__ void elemente_quadrieren(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}

// Hauptroutine, wird auf der CPU ausgeführt
int main(void)
{
  float* a_cpu;  // Zeiger auf CPU- und GPU-Arrays
  float* a_gpu;
  int N = 10;  // Anzahl der Elemente in einem Array

  a_cpu = (float *)malloc(N*sizeof(float));        // CPU-Array auf Hauptspeicher allokieren
  cudaMalloc((void **) &a_gpu, N*sizeof(float));   // GPU-Array auf Grafikspeicher allokieren

  // CPU-Array initialisieren und auf Grafikkarte kopieren
  for (int i=0; i<N; i++) a_cpu[i] = (float)i;
  cudaMemcpy(a_gpu, a_cpu, N*sizeof(float), cudaMemcpyHostToDevice);
  // Berechnung auf Grafikkarte durchführen (Kernel ausführen)
  elemente_quadrieren<<< n_bloecke, threads_pro_block >>>(a_gpu, N);
  // Ergebnis abfragen und zurückkopieren
  cudaMemcpy(a_cpu, a_gpu, N*sizeof(float), cudaMemcpyDeviceToHost);
  // Ergebnisse ausgeben
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_cpu[i]);
  // Speicher räumen
  free(a_cpu); cudaFree(a_gpu);
}
```

ABB. 3.6: CUDA Codebeispiel: Ein Algorithmus, der alle Elemente eines Arrays quadriert.

An einem kleinen Codebeispiel, das alle Elemente eines Arrays quadriert, soll dies verdeutlicht werden. Siehe dazu Abbildung 3.6. Zunächst werden die nötigen Bibliotheken eingebunden. Für dieses Beispiel benötigt man die Standard input/output Funktionen **#include <stdio.h>** sowie die Standard-CUDA-Erweiterungen **#include <cuda.h>**. An dieser Stelle könnte man bei Bedarf die mathematischen Zusatzbibliotheken CUFFT

und CUBLAS mittels **#include <cufft.h>** und **#include <cublas.h>** einbinden. Als nächstes wird der Kernel geschrieben, der die Berechnung durchführen soll. Der Indikator **__global__** kennzeichnet, dass es sich um einen Kernel handelt. Hier wird zudem die Thread-ID auf die Position im Array gemapped. Es folgt die Hauptroutine, über die zunächst mittels **cudaMalloc()** Speicherplatz auf dem Grafikkartenspeicher allokiert wird, anschließend über **cudaMemcpy(..., cudaMemcpyHostToDevice)** das Startarray auf die Grafikkarte kopiert. Anschließend wird der Kernel ausgeführt. Dazu ruft man wie gewöhnlich eine Funktion auf, allerdings folgt der Zusatz <<< **n_bloecke, threads_pro_block** >>>. Dieser steuert die Anzahl der aufzurufenden Blöcke und die Anzahl der Threads pro Block. Wie im letzten Beispiel beschrieben, ist dieser Punkt kritisch für die Leistung des GPU. Letztendlich wird über **cudaMemcpy(..., cudaMemcpyDeviceToHost)** das Ergebnisarray vom globalen Speicher der Grafikkarte in den Hauptspeicher zurück kopiert. Zum Schluss wird mittels **cudaFree()** der allokierte Speicher wieder befreit.

3.2 Molekulardynamikpakete

Seit der Einführung von CUDA wurde das Programmieren von Allzweckrechnungen auf Grafikkarten deutlich einfacher. Dennoch erfordert die Verwendung von CUDA einiges an Einarbeitungszeit, will man zum Beispiel bereits bestehenden Code auf die Grafikkarte portieren. Vor allem dann, wenn der bestehende Code nicht bereits in C geschrieben war, stößt man schnell an die Grenzen der Motivation. Speziell für Molekulardynamiksimulationen gibt es allerdings bereits einige Pakete, die einem sehr viel Arbeit abnehmen. Zwei grundsätzlich verschiedene solche Pakete sollen im Folgenden etwas näher beschrieben werden.

3.2.1 GPULib

Will man es vermeiden, sich eingehend mit den Paradigmen der parallelen Programmierung zu beschäftigen oder neue Konzepte wie Thread Blöcke, Warps und vereinigten Speicherzugriffen zu erlernen, so bietet GPULib[7] eine Alternative. GPULib setzt CUDA eine abstrakte Arbeitsebene obenauf, die es dem Programmierer ermöglicht, Funktionen in bereits bestehendem IDL oder Matlab Code durch entsprechende Funktionen aus seiner Bibliothek zu ersetzen, um diese auf der GPU ausführen zu lassen. So kann der Programmierer sich voll und ganz auf die Weiterentwicklung seiner Algorithmen kümmern und dennoch den Leistungsbonus moderner Grafikchips nutzen. Außerdem existieren bereits Funktionen für Java und Python, die allerdings vom Support ausgeschlossen sind. Support ist deswegen ein Thema, weil GPULib ein kommerzielles Paket ist, das von Tech-X vertrieben wird. Dies ist für den Programmierer wohl der Hauptnachteil dieses Pakets, da der GPULib Quellcode nicht offen zugänglich ist.

3.2.2 HOOMD-blue

Beschäftigt man sich hauptsächlich mit Molekulardynamiksimulationen, bietet HOOMD-blue einen interessanten Ansatz. Der Name steht für „highly optimized object-oriented many particle dynamics - blue edition“, wobei sich letzteres auf die Farben der University of Michigan bezieht, an dem HOOMD-blue derzeit weiterentwickelt wird. Auch HOOMD-blue baut auf CUDA auf, wobei man sich nirgends mehr mit C-Code auseinandersetzen muss. Denn das Hauptaugenmerk von HOOMD - der Name verrät es bereits - liegt auf dem objektorientierten Ansatz: HOOMD-blue arbeitet mit Pythonskripten, die es dem

Programmierer ermöglichen, mit nur wenigen Zeilen Skriptcode eine vollständig parallelisierte Molekulardynamiksimulation durchzuführen. Neben Integratoren für die verschiedenen Teilchenensembles gibt es eine ganze Reihe von Paarkräften und Paarbindungen für Polymerstrukturen. Ein weiterer großer Vorteil dieses Pakets ist, dass der Quellcode frei verfügbar (open source) ist, um ihn mit eigenen Ideen zu ergänzen oder zu verändern.

4 Anwendungsbeispiel: Lennard-Jones Fluid

Als ein einfacher Anwendungsbeispiel sei hier ein Lennard-Jones Fluid besprochen. Das Hauptaugenmerk soll dabei auf der Implementierung auf der GPU liegen. Es werden jene Konzepte und Lösungen präsentiert, die man auch in HOOMD-blue wiederfindet, um damit anschließend eine Simulation durchführen zu können.

4.1 Vorbereitung

Zugrunde gelegt wird ein NVT-Ensemble von Teilchen einer Sorte, die in einem Lennard-Jones Potential wechselwirken. Dieses Potential ist gewöhnlicherweise ab einem bestimmten Cutoff-Radius r_{cut} abgeschnitten.

$$V(r) = \begin{cases} 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], & r \leq r_{cut} \\ 0, & r > r_{cut} \end{cases}$$

Wie in jeder Molekulardynamiksimulation sind für jedes Teilchen des Systems folgende Schritte durchzuführen:

1. Berechnung einer Liste aller Nachbarn dieses Teilchens
2. Berechnung der Abstände zwischen dem Teilchen und seinen Nachbarn
3. Berechnung der Gesamtkraft auf das Teilchen
4. Vorwärtsintegration des Teilchens in der Zeit

4.2 Realisierung auf der GPU

Zunächst wenden wir uns der Berechnung der Paarkräfte zu. Aufgrund des abgeschnittenen Potentials bietet sich hier eine Nachbarliste an, die für jedes Teilchen mit Index i alle Teilchen im Abstand $r_{max} \geq r_{cut}$ enthält. In einer CPU Implementierung bilden verlinkte Listen das Standarddatenmodell für diese Nachbarliste. Allerdings benötigt eine effiziente Implementierung auf der GPU die Vereinigung von Threads, d.h. die Nachbareinträge müssen vereinigt ausgelesen werden. Deshalb verwendet man hier stattdessen eine Nachbarliste in Matrixstruktur, sprich ein zweidimensionales Array. Gekennzeichnet wird diese Nachbarliste durch NBL_{ji}, wobei j als Index für die Nachbarn des i-ten Teilchens dient. Zudem benötigt man für die Implementierung ein Array NN_i, das die Anzahl der Nachbarn jedes Teilchens enthält.

In der CPU-Implementierung arbeitet man alle Schritte der Simulation nacheinander in einer Schleife über alle Teilchen ab. Die Parallelisierung dieses Algorithmus erfolgt hierbei, indem man die Kräfte auf jedes Teilchen von einem einzelnen Thread berechnen lässt. Zunächst stellt man wie im obigen Codebeispiel den Zusammenhang zwischen dem Teilchenindex i und der Thread- und Block-ID her und initialisiert den Vektor für die Gesamtkraft:

$i \leftarrow blockID \cdot N_B + threadID$
$\vec{F}_{ges} \leftarrow \vec{0}$

Somit repräsentiert jeder Thread i ein Teilchen i. Als nächstes liest man unter Verwendung des Texturcaches die Koordinaten von i, da der Ort dieses Teilchens sicherlich noch von anderen Threads erfragt wird. Die erfolgt auf der GPU mit den Funktionen **tex1Dfetch()**, wobei die 1 kennzeichnet, dass es sich um eine eindimensionales „Textur“, sprich Array, handelt:

$\vec{A} \leftarrow \text{tex1Dfetch}(\vec{R}_i)$

Als nächstes liest jeder Thread die Anzahl der Nachbarn NN aus NN_i. Es folgt eine Schleife, in der nacheinander die Indizes aller Nachbarn von i aus der Nachbarliste gelesen werden:

von $j = 0$ bis $NN - 1$
$k \leftarrow NBL_{ji}$

Dank der Matrixstruktur von NBL_{ji} verläuft dieser Lesezugriff vereinigt. Im Anschluss werden die Koordinaten des jeweiligen Nachbarn gelesen. Hierzu verwendet man wieder den Texturcache, da diese Speicherzugriffe nicht vereinigt stattfinden können:

$\vec{B} \leftarrow \text{tex1Dfetch}(\vec{R}_k)$

Anschließend berechnet man den Abstand der beiden Teilchen und mit einer zuvor definierten Funktion **Kraft**($\vec{r}$) die Kraft $\vec{F}$, die auf i wirkt:

$d\vec{r} \leftarrow \vec{B} - \vec{A}$
$\vec{F} \leftarrow \text{Kraft}(d\vec{r})$

Üblicherweise sind die Nachbarn weiter voneinander entfernt, als der Cutoffradius des Potentials. Dadurch muss die Nachbarliste weniger oft aktualisiert werden, was Rechenzeit spart. Allerdings muss man dann sicherstellen, dass der Cutoffradius nicht überschritten wird:

wenn $|d\vec{r}| > r_{cut}$ **dann** $\vec{F} \leftarrow \vec{0}$

Mit dem Speichern der Kraft à la $\vec{F}_{ges} \leftarrow \vec{F}_{ges} + \vec{F}$ endet der Schleifenkörper und die nächste Iteration der Schleife beginnt. Sind auf diese Weise alle Nachbarn durchlaufen, ist $\vec{F}_{ges}$ die Gesamtkraft auf i und der Thread hat seine Arbeit erledigt.

Für die CPU-Implementierung böte sich hier außerdem eine Optimierung an, indem man sich unter Verwendung von actio-reactio die Hälfte aller Fließkommarechnungen spart. Dies sähe dann in etwa so aus:

$\vec{F} \leftarrow \text{Kraft}(d\vec{r})$
$\vec{F}_i \leftarrow \vec{F}_i + \vec{F}$
$\vec{F}_k \leftarrow \vec{F}_k - \vec{F}$

Allerdings müsste hierzu auf verteilte Speicherorte lesend, bearbeitend und schreibend zugegriffen werden (engl. read-modify-write), was den Geschwindigkeitzuwachs bereits auf der CPU auf etwa 1,5 reduziert. Dies kann man hier allerdings nicht nutzen, da auf der GPU verteilte Speicherzugriffe weitaus teurer sind als auf der CPU und dies die Geschwindigkeit sogar verringern würde.

Bei der Erstellung der Nachbarliste greift man für gewöhnlich auf das so genannte „Binning“ der Teilchen zurück (engl. bin = Kasten). Normalerweise muss jedes Teilchen gegen jedes Teilchen gerechnet werden, was zu einer Laufzeit in $\mathcal{O}(N^2)$ führt. Durch die

Vorsortierung in Kästen entsprechender müssen nur noch die Teilchen benachbarter Größe in die Berechnungen miteinbezogen werden, was die Kosten auf $\mathcal{O}(N)$ reduziert. Da die Zuordnung der Teilchen in die Kästen mit willkürlichen Speicherzugriffen verbunden ist, wird das Binning oft auf die CPU ausgelagert. Es gibt zwar Realisierungen auf der GPU[8], auf diese wird allerdings hier nicht näher eingegangen.

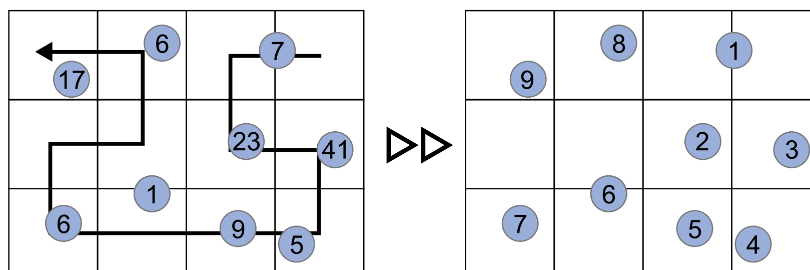


ABB. 4.1: Neusortierung von Teilchenkoordinaten mit einer raumfüllenden Kurve.

Schon bei weniger großen Systemen passen nicht alle Teilchenkoordinaten in den Cache. Nach einiger Zeit laufen die Teilchen allerdings stark auseinander, was dazu führt, dass deren Koordinaten aus dem Cache fliegen. Dies bezahlt man mit Geschwindigkeitseinbußen von bis zu 75%. Dies kann man umgehen, indem man einen Sortieralgorithmus implementiert, der die Teilchenkoordinaten nach einigen Zeitschritten im Speicher neu sortiert. Hier kann man allerdings das Binning ausnutzen, indem man eine raumfüllende Kurve durch alle Kästen zieht (Abb. 4.1) und die Teilchen so sortiert, wie sie die Kurve passieren. Ein rekursiver Algorithmus, der eine solche Kurve implementiert, ist in Referenz [9] gegeben.

Die NVT-Vorwärtsintegration der Teilchen in der Zeit macht auf der GPU keinerlei Probleme, da man die dafür nötigen Speicherzugriffe der Threads leicht vereinigen kann und so das Potential der GPU für diesen Programmteil gut nutzen.

4.3 SIMULATION MIT HOOMD-BLUE

Die im letzten Abschnitt genannten Realisierungen einer Molekulardynamiksimulation sind den Konzepten von HOOMD-blue entnommen. In diesem Abschnitt wird damit die Simulation eines Lennard-Jones-Fluids auf der GPU durchgeführt und die Ergebnisse mit einer Simulation ohne Verwendung der GPU verglichen. Die dabei verwendete Referenzhardware besteht aus einem AMD Athlon X2 Dual Core mit 2,21GHZ und 4GB RAM und einer Sparkle GeForce GTX285 mit 2GB globalem Speicher.

Für die Simulationen werden stets alle Parameter des Potentials auf 1 gesetzt. Es werden Systeme mit einer Größe von 500-100000 Teilchen bei zwei verschiedenen Packungsdichten simuliert. Die Startkonfiguration der Teilchen wurde dabei zufällig gewählt. Das dabei verwendete HOOMD-Skript war:

```
from hoomd_script import *
import math

# Erstelle N Teilchen der Sorte A, mit einer Packungsdichte von phi_p in einer Box
init.create_random(N, phi_p, name='A')

# Wähle ein Lennard-Jones Potential zwischen den Teilchen
lj = pair.lj(r_cut=3.0)
```

```
# Setze alle Konstanten auf 1.0
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=1.0)

# Integriere unter Benutzung eines Nosé-Hoover-Thermostats
integrate.nvt(dt=0.005, T=1.2, tau=0.5)

# Simuliere 10^6 Zeitschritte
run(1000000)
```

Ob auf CPU oder GPU gerechnet wird, entscheidet die Kommandozeilenoption `--mode=[cpu/gpu]`. Für die großen Systeme wurden aus Zeitgründen weniger Zeitschritte simuliert. Verglichen wurde letztendlich die Geschwindikeit der CPU- mit der Geschwindigkeit der GPU-Simulation, gemessen in Zeitschritten pro Sekunde (engl. time steps per second, kurz: TPS). Das Ergebnis dieses Vergleichs ist Abb. 4.2 zu entnehmen. Da durch die Verwendung eines

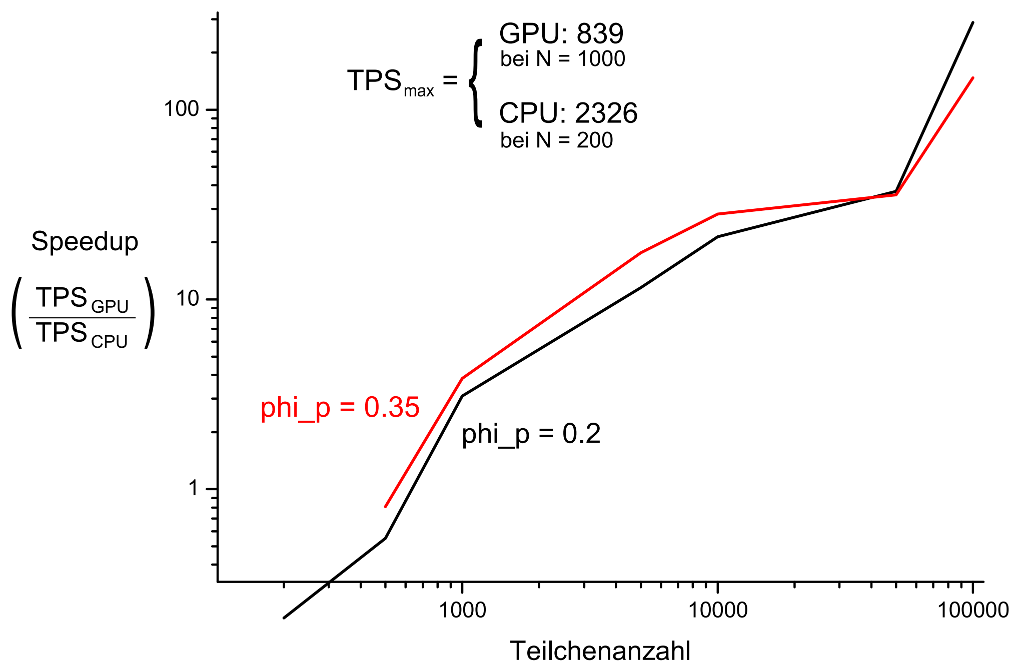


Abb. 4.2: Ergebnis der Simulationsreihe zum Vergleich der Geschwindigkeiten der Referenzhardware.

fertigen Pakets wie HOOMD nicht per se klar ist, ob die beiden Ergebnisse übereinstimmen. Zur Validierung der Ergebnisse wurden die Trajektorien der Teilchen einer CPU- und GPU-Simulation verglichen. Dabei wurde zunächste eine Simulation durchgeführt, um zu einer gemeinsamen Startkonfiguration zu gelangen. Dazu wurde ein Lennard-Jones-Fluid mit 25000 Teilchen 50000 Zeitschritte NVT-integriert. Diese Konfiguration dient als Startkonfiguration für je zwei Simulationen auf der CPU und GPU - jeweils einmal mit einem NVT-Integrator (Nosé-Hoover-Thermostat) und einmal mit einem NVE-Integrator (Velocity-Verlet). Das Abstandsquadrat der Teilchenkoordinaten diente dabei als Maß. In Abb. 4.3 ist das Ergebnis dieser Messung dargestellt. Man kann erkennen, dass die beiden Ergebnisse bereits nach ca. 600 Zeitschritten auseinander laufen. Gründe dafür sind in erster Linie Rechenfehler: Der GPU- und der CPU-Algorithmus addieren die Kräfte auf ein Teilchen in einer anderen Reihenfolge, was durch die Fließkommaarithmetik zu leicht unterschiedlichen Ergebnissen führen kann. Haben sich einmal solche noch so kleinen Abweichungen eingeschlichen, entwickeln sie sich schnell zu sehr großen Fehlern.

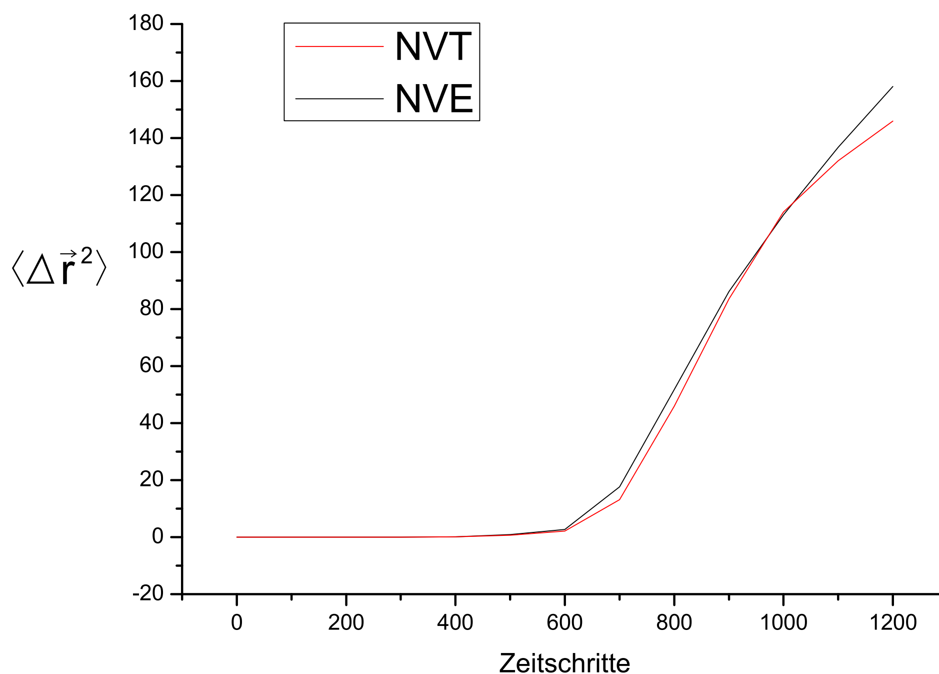


ABB. 4.3: Validierung der Ergebnisse aus GPU- und CPU-Simulationen mittels HOOMD-blue.

4.4 FAZIT

Das Anwendungsbeispiel hat gezeigt, dass Molekulardynamik auf Grafikprozessoren eine echte Alternative darstellt - zumindest, wenn man sich fertiger Pakete bedient, wie dies hier getan wurde. Dass allerdings mehr dahinter steckt, als ein schlichtes Austauschen der Hardware, ist nicht zu übersehen: Der Programmierer sieht sich an vielen Stellen Problemen gegenübergestellt, wo auf CPU-Implementierungen keine waren. Es sei zudem erwähnt, dass die verwendete Referenzhardware in puncto Qualität in keinem Verhältnis zueinandersteht: Während es sich bei der GTX285 Grafikkarte um ein High-End-Produkt handelt, bewegt sich der Athlon X2 allerhöchstens im Mittelfeld.

5 ABSCHLIESSENDE BEMERKUNGEN

Der Ansatz, den Nvidia mit CUDA verfolgt, findet derzeit viel Begeisterung in der Wissenschaft. Dennoch bleibt zu bedenken, ob es ratsam ist, sich von einem einzigen Hersteller abhängig zu machen - in den meisten Fällen ist es das nämlich nicht. ATI verfolgt mit ATI Stream dieselbe Strategie wie Nvidia, befindet sich allerdings derzeit noch in den Kinderschuhen. Ein Ausweg aus dieser Situation bietet die zur Zeit von der Khronos Group entwickelte Open Computing Language (OpenCL), die es sich zum Ziel gemacht hat, Low-Level-Funktionalität für Grafikprozessoren aller Hersteller kostenlos anzubieten. Allerdings ist dies noch ein weiter Weg. Auch Microsoft hat das Potential von Grafikprozessoren erkannt und schlägt mit seinem DirectCompute in die gleiche Kerbe. Allerdings ist auch hier noch viel Entwicklungsarbeit nötig, die allerdings weitreichende Folgen haben könnte: So kursieren beispielsweise (unbestätigte) Gerüchte über die direkte Integration der GPU in zukünftige Betriebsysteme.

REFERENZEN

[1] ZHIRNOV, CAVIN, HUTCHBY, BOURIANOFF: *Limits to binary logic switch scaling - a gedanken model*
Digital Object Identifier: 10.1109/JPROC.2003.818324

[2] AMDAHL: *Validity of the single processor approach to achieving large scale computing capabilities*
http://www-inst.eecs.berkeley.edu/ n252/paper/Amdahl.pdf

[3] MURRAY CAMPBELL ET AL.: *Deep Blue*
http://sjeng.org/ftp/deepblue.pdf

[4] F-H. HSU: *IBM's Deep Blue Chess Grandmaster Chips*
IEEE Micro, p. 70-81, Mar-Apr 1999

[5] RIKEN PRESSEMITTEILUNG: *Completion of a one-petaflops computer system for simulation of molecular dynamics*
http://www.riken.jp/engn/r-world/info/release/press/2006/060619/index.html

[6] NVIDIA CORP.: *CUDA Programming Guide 2.3*
http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[7] TECH-X CORP.: *GPULib: Harnessing the Power of the GPU*
http://www.txcorp.com/pdf/GPULib/documentation/GPULib_UsersGuide.pdf

[8] J.A. VAN MEEL, A. ARNOLD, D. FRENKEL, S.F.P. ZWART, R.G. BELLEMAN: *Harvesting graphics power for MD simulations*
Mol. Sim.

[9] J. WANG, J. SHAN: *Space-filling curve based point clouds index*
Geocomputation 2005