Aufgabe 4: LBM

Martin Aumüller *
Universität zu Köln

9. Juli 2013

1 Themen

In dieser Aufgabe werden folgende Themengebiete bearbeitet:

- GPGPU
- Gitter-Boltzmann-Methode zur Strömungssimulation
- einfache Visualisierungstechniken für Vektorfelder

2 Einführung

In dieser Aufgabe wirst Du sehen, dass Grafikkarten nicht nur zur Grafikausgabe zu gebrauchen sind, sondern dass sie auch allgemeine Berechnungen durchführen können – und das unter geeigneten Umständen sogar sehr schnell.

3 Programmaufbau

Das Programm gliedert sich in folgende Dateien und Klassen:

`main.cpp` Programmstart.

`clock.cpp` In der Klasse *Clock* ist die Abfrage der Echtzeituhr gekapselt.

`applicationwindow.cpp` Der *ApplicationWindow* steuert den Programmablauf und verarbeitet Benutzereingaben.

`qosgviewer.cpp` *QOSGViewer* stellt eine OpenSceneGraph-Szene in einem *QGLWidget* dar.

`overlayviewer.cpp` *OverlayViewer* ist ein angepasster *QOSGViewer*, der Texteinblendungen ermöglicht.

`lbm.cpp` *LBMD3Q19* ist der Kern des Programms mit der Strömungssimulation nach der Gitter-Boltzmann-Methode („Lattice-Boltzmann method“).

`slice.cpp` *Slice* dient der Visualisierung eines Schnitts entlang der Koordinatenachsen des Simulationsvolumens.

`slicevisualisation.cpp` *SliceVisualisation* fasst drei paarweise senkrechte Schnitte durch das Simulationsgebiet zusammen.

`linevisualisation.cpp` *LineVisualisation* stellt die Geschwindigkeit jeder Zelle als Linie dar.

`coordinatebox.cpp` *CoordinateBox* zeigt die Grenzen des Simulationsvolumens.

4 GPGPU und CUDA

4.1 General Purpose Computing on GPUs – GPGPU

Die Programmierbarkeit von Grafikhardware hat im Laufe der Zeit stark zugenommen. Ältere Grafikhardware bestand teilweise noch aus flexibel programmierbaren CPUs, allerdings wurde diese Funktionalität nicht dem Benutzer zugänglich gemacht, so dass später im Zuge weitergehender Miniaturisierung lediglich die Funktionalität der Fixed-Function-Pipeline in Silizium implementiert wurde.

Aber seit dem Aufkommen von Hardware zur 3D-Beschleunigung für Computerspiele auf PCs werden die Grafikchips immer flexibler. Nachdem sie zuerst spezialisierte programmierbare Vertex- und Fragmenteinheiten enthielten, wurden durch vielseitigere Recheneinheiten, die nach Bedarf jede der beiden Funktionen übernehmen konnten, neue programmierbare Stufen zur Erzeugung und feineren Tesselierung von Geometriedaten möglich.

Die Nutzung dieser Recheneinheiten zu allgemeinen Zwecken – daher der Begriff General Purpose Computing on GPUs – war Anfangs nur unter Verrenkungen möglich: die zu verarbeitenden Daten wurden als Texturen zur GPU hochgeladen, dort wurde texturierte Geometrie gerendert, wobei ein Fragment-Programm die gewünschten Berechnungen durchführte, so dass die

*aumueller@uni-koeln.de

Ergebnisse dann aus dem Framebuffer zurückgelesen werden konnten.

Darauf aufbauend gab es bereits Frameworks zur Nutzung von GPUs für allgemeine Berechnungen („GPGPU“), z. B. BrookGPU [1].

4.2 CUDA

Jedoch haben auch Hersteller von Grafikhardware den Markt für GPGPU erkannt. So existieren neben proprietären Frameworks wie CUDA [3] mit OpenCL [2] auch herstellerübergreifende Standards. CUDA ist aber wohl weiter verbreitet, obwohl es nur für Nvidia-Hardware geeignet ist.

C for CUDA, im folgenden nur CUDA genannt, ermöglicht sehr hardwarenahes Programmieren von Nvidia-GPUs.

4.2.1 Streaming Multiprozessor

In den neueren Nvidia-GPUs sind jeweils mind. 8 Recheneinheiten (ALUs) zusammen mit einer „Special Function Unit“ (SFU) für z. B. Inversion, Quadratwurzel und transzendente Funktionen zu einem “Streaming Multiprocessor“ gruppiert. Die ALUs eines Multiprozessors führen in parallel ablaufenden Threads ähnlich einem SIMD-Prozessor (Single Instruction Multiple Data) immer dieselben Instruktionen aus, wobei manche ALUs im Fall von unterschiedlicher Verzweigung auch pausieren können.

Je 32 dieser Threads werden automatisch zu „Warps“ gruppiert, wobei diese immer gemeinsam (oder in unmittelbarer Folge) auf einem Multiprozessor ausgeführt werden.

4.2.2 Speicherhierarchie

Die Speicherhierarchie ist anders als beim Programmieren für CPUs durch die Sprache fast vollständig exponiert.

Register Die Register eines Multiprozessors halten Thread-lokale Variablen und werden aus einem Pool hardwareabhängiger Größe (mind. 8192) einzelnen Threads zugeordnet.

Shared Memory Der „Shared Memory“ eines Multiprozessors steht allen Threads, die auf einem Multiprozessor laufen, zur Verfügung und kann als vom Programmierer gesteuerter Cache verwendet werden.

Konstantenspeicher Im Konstantenspeicher können zur Laufzeit eines Kernels unveränderliche Daten gespeichert werden, der Zugriff darauf wird automatisch gecachet.

Texturspeicher Ähnlich dem Konstantenspeicher dient auch der Texturspeicher dem Ablegen zur Laufzeit des Kernels unveränderlicher Daten mit automatischem Caching. Im Unterschied zum Konstantenspeicher werden Zugriffe mit benachbarten Indizes in *allen* Koordinaten beschleunigt. Außerdem steht zusätzliche Funktionalität beim Laden der Daten aus dem Texturspeicher in die Register wie beispielsweise das Reskalieren von Byte-Daten zu normierten Fließkommawerten zur Verfügung.

lokaler Speicher Reichen die Register eines Multiprozessors nicht für alle Thread-lokalen Variablen aus, so können sie auch im sog. lokalen Speicher abgelegt werden. Das ist – genauso wie der globale Speicher – ein Teil des Framebuffer- und Textur-Speichers der Grafikkarte, der einem Thread zu diesem Zweck zugeordnet ist.

globaler Speicher Auf den globalen Speicher können alle Threads aller Multiprozessoren gleichermaßen zugreifen, wobei Zugriffe auf diesen Speicher um ein Vielfaches langsamer sind als auf Register und Shared Memory.

Durch die hohe Anzahl von Registern innerhalb eines Multiprozessors kann eine große Anzahl von Threads auf einer Funktionseinheit gleichzeitig aktiv sein, d. h. die ihnen zugeordneten Daten verbleiben im Multiprozessor, obwohl evtl. andere Threads ausgeführt werden. Dadurch ist ein sehr schnelles Umschalten zwischen verschiedenen Threads möglich. Das wird von der Hardware z. B. dann, wenn Daten aus dem globalen Speicher angefordert wurden, diese aber noch nicht bereitstehen, automatisch gemacht. Dadurch, dass viele Threads im schnellen Wechsel ausgeführt werden, lassen sich die Wartezeiten beim Zugriff auf den Speicher verstecken, da andere Threads noch sinnvolle Arbeit erledigen können. Hier wird auch deutlich, dass die Hardware einer GPU nicht auf hohe Leistung eines einzelnen Threads sondern auf hohen Gesamtdurchsatz in einer Vielzahl von Threads optimiert ist.

4.2.3 Kernels

Beim Starten eines sog. „Kernels“, d. h. eines GPU-Unterprogramms, gibt man an, wie dieses konfiguriert sein soll: man beschreibt, wie viele Threads gestartet werden sollen und wie diese strukturiert sein sollen. Mehrere Threads können in einem 3-dimensionalen Gitter zu einem sog. Block gruppiert werden, mehrere dieser Blöcke wiederum zu einem 2-dimensionalen „Grid“. Dabei werden die Threads eines Blocks auf einem Multiprozessor ausgeführt. Daher sollte ein Block nicht größer als die maximal zulässige Anzahl von Threads pro Multiprozessor von entweder 768 oder 1024 sein. Da nur

Threads desselben Blocks zu einem Warp gruppiert werden können, ist die Anzahl der Threads in einem Block optimalerweise durch 32 teilbar.

Die Threads eines Blocks können auf denselben „Shared Memory“ zugreifen und eine Barrier-Synchronisation mittels *__syncthreads()* durchführen.

Die Threads verschiedener Blöcke können – sofern ein Multiprozessor über ausreichend Register verfügt – entweder auf demselben Multiprozessor ausgeführt werden oder über verschiedene Multiprozessoren verteilt werden. In keinem der beiden Fälle stehen die oben für Threads innerhalb eines Blocks erwähnten Synchronisationsmechanismen zur Verfügung.

Die Definition eines Kernels muss mit dem Schlüsselwort *__global__* gekennzeichnet werden. Die Größe des Grids und der Blöcke gibt man beim Aufruf in jeweils einer Variablen vom Typ *dim3* an, welcher ein Vektor aus 3 ganzen Zahlen ist. Dazu sind dreifache öffnende und schließende spitze Klammern notwendig:

```
dim3 grid(16, 16);
dim3 block(128); // missing values default to 1
fastKernel<<<grid, block>>>(other, paramters);
```

Der Kernel startet, da aber nicht auf das Ende seiner Ausführung gewartet wird, ist es nicht möglich, einen Rückgabewert oder Fehlerstatus abzufragen.

Je nach Leistungsfähigkeit verfügt eine GPU über einen oder bis zu 16 Multiprozessoren, so dass bis zu 512 Threads tatsächlich gleichzeitig ausgeführt werden können. Um die Leistung der GPU auszunutzen, sollte man daher genügend viele Threads starten, so dass alle Multiprozessoren beschäftigt sind, auch wenn auf ausstehende Speicherzugriffe gewartet werden muss.

Kernels können auch Unterprogramme aufrufen, diese müssen mit *__device__* markiert werden. In den allermeisten Fällen werden diese wie Inline-Funktion direkt in den Kernel eingebunden.

CUDA erlaubt das Mischen von Code für CPU („Host“) und GPU („Device“) in einer `.cu`-Datei. Nicht explizit gekennzeichnete Funktionen werden auf der CPU ausgeführt. Es ist aber auch möglich mit der Sequenz *__host__ __device__* eine Funktion sowohl für CPU als auch GPU zu kompilieren.

4.2.4 Transfer von Daten zwischen CPU und GPU

Für den Transfer von Daten zwischen GPU und CPU steht der Aufruf *cudaMemcpy* zur Verfügung. Das Ziel eines Transfers zur GPU muss vorher mit *cudaMalloc* reserviert und sollte nach Gebrauch wieder mit *cudaFree* freigegeben werden.

Der Aufruf von *cudaMemcpy* bewirkt, dass die CPU darauf wartet, dass alle auf der GPU gestarteten Kernels ihre Ausführung beenden. Das kann man auch mit *cudaThreadSynchronize* erzwingen. Insbesondere beim Debuggen kann das manchmal hilfreich sein, da dieser Aufruf den Code des letzten aufgetretenen Fehlers zurückliefert, der durchaus auch von einem zuvor gestarteten Kernel hervorgerufen werden kann.

4.2.5 Kompilation von `.cu`-Dateien

In der bereitgestellten Datei `cuda.pri` wird `qmake nvcc` als ein weiterer Compiler bekannt gemacht. Er kompiliert die Dateien, welche in der `.pro`-Datei in *CUDA_SOURCES* aufgelistet sind.

Um GPU-Code zu verwenden, müsst Ihr also nur noch eine Datei mit der Endung `.cu` anlegen, dort am besten den Header *<cuda.h>* einbinden – und schon könnt Ihr mit den oben beschriebenen Werkzeugen arbeiten.

Die nötigen weiteren Details zu CUDA findet man neben [3] auch in [4].

5 Strömungssimulation

Bei der Strömungssimulation wird Geschwindigkeit und Druck in einem Fluid, d. h. einer Flüssigkeit oder einem Gas, simuliert.

5.1 Navier-Stokes

Die Navier-Stokeschen Gleichungen beschreiben Strömungsphänomene in einer Vielzahl von Flüssigkeiten und Gasen, wie z. B. Wasser und Luft. Eine analytische Lösung ist i. A. nicht bekannt, deshalb bedient man sich numerischer Methoden zu ihrer Lösung. Dazu ist eine Diskretisierung des Simulationsgebietes, also eine Zerlegung in Zellen erforderlich.

5.2 Gitter-Boltzmann-Methode – LBM

Bei der Gitter-Boltzmann-Methode („Lattice-Boltzmann method“) wird das Simulationsgebiet in gleiche Zellen, z. B. Würfel, zerlegt und zusätzlich die auftretenden Geschwindigkeiten und die Zeitschrittweite so diskretisiert, dass die Partikel des Fluids genau von einer Zelle zur benachbarten „hüpfen“ können.

Das wäre zu ungenau, wenn man nur das Vorhanden- oder Nichtvorhandensein eines Fluidpartikels in einer Zelle betrachten würde. Stattdessen wird in jeder Zelle die Verteilung von Partikeln mit den verschiedenen zulässigen Geschwindigkeiten simuliert.

Die LBM-Simulation lässt sich in zwei unabhängige Schritte zerlegen, die nacheinander in fortwährender Wiederholung durchgeführt werden:

Strömen In diesem Schritt werden die Verteilungen entsprechend ihrer Geschwindigkeit auf die korrespondierenden Nachbarzellen verschoben, ihre Geschwindigkeit bleibt dabei unverändert.

Kollision In diesem Schritt werden die Partikel einer Zelle anderen Geschwindigkeiten zugeordnet, um Ablenkungen durch die ständig auftretenden Partikelkollisionen zu simulieren. Damit Massen- und Impulserhaltung gewährleistet ist, muss ihre Gesamtmenge („Dichte“) und ihr Gesamtimpuls in jeder Zelle unverändert bleiben. Dieser Schritt ist von der Wahl des Relaxationskoeffizienten ω abhängig. Er steuert die Zähigkeit der Flüssigkeit und muss zwischen 0 und 2 liegen.

5.2.1 LBMD3Q19

Die LBM-Simulationen werden entsprechend der Dimension d des Simulationsgebiets und der möglichen Übergänge zu Nachbarzellen q mit LBMdQq bezeichnet. In dieser Aufgabe ist ein LBMD3Q19-Modell implementiert. Das bedeutet, dass das Grundgebiet 3-dimensional ist und dass ein Fluidpartikel beim Strömen zu 19 verschiedenen Nachbarzellen wechseln kann:

- zur Ausgangszelle selbst ($\vec{e}_0$),
- zu einer der 6 entlang der Koordinatenachsen direkt benachbarten Zellen ($\vec{e}_1, \ldots, \vec{e}_6$),
- zu einer der 12 Zellen, die mit einer Kante in Berührung zur Ausgangszelle stehen. ($\vec{e}_7, \ldots, \vec{e}_{18}$).

Ein direktes Wechseln in lediglich entlang der Raumdiagonale benachbarte Zellen ist in diesem Modell nicht möglich.

Dieses Modell mit seinen 19 Nachbarschaftszellen hat sich als ein guter Kompromiss zwischen Berechnungsaufwand (der mit der Zahl der Nachbarschaftszellen steigt) und Zuverlässigkeit erwiesen.

Zu jeder der Zellen des Simulationsgebiets hat man eine Verteilung f_i ($i = 0, \ldots, 18$) mit 19 Werten zu speichern. Mit dieser Verteilungsfunktion lässt sich die Gesamtdichte ρ des Fluids einer Zelle als Summe ihrer Werte bestimmen:

$$\rho = \sum_{i=0}^{18} f_i.$$

Die Gesamtströmungsgeschwindigkeit $\vec{u}$ einer Zelle ergibt sich als Summe der $\vec{e}_i$, wobei als Gewicht jeweils der entsprechende Wert aus der Verteilungsfunktion zu verwenden ist:

$$\vec{u} = \sum_{i=0}^{18} f_i \vec{e}_i.$$

Im Kollisionsschritt muss zunächst die Gleichgewichtsverteilung f_i^{eq} dieser Zelle bestimmt werden:

$$f_i^{\text{eq}} = w_i \left(\rho - \frac{3}{2} \langle \vec{u} | \vec{u} \rangle + 3 \langle \vec{e}_i | \vec{u} \rangle + \frac{9}{2} \langle \vec{e}_i | \vec{u} \rangle^2 \right).$$

Das korrekte Verhalten ist von der sinnvollen Wahl der skalaren Faktoren abhängig, deren Werte wir als gegeben annehmen, die sich aber auch herleiten lassen (siehe z. B. [5]). Außerdem sind die Beiträge der verschiedenen diskreten Geschwindigkeitstypen unterschiedlich zu gewichten:

Elementargeschwindigkeit	Gewicht w_i
$\vec{e}_0$	1/3
$\vec{e}_1, \ldots, \vec{e}_6$	1/18
$\vec{e}_7, \ldots, \vec{e}_{18}$	1/36

Im Kollisionsschritt wird eine neue Verteilung f_i' bestimmt, die näher an der Gleichgewichtsverteilung liegt:

$$f_i' = (1 - \omega) f_i + \omega f_i^{\text{eq}}.$$

Der Relaxationskoeffizient ω steuert, wie schnell diese Gleichgewichtsverteilung erreicht wird.

5.3 Randbedingungen

Zum Modellieren von Randbedingungen lassen sich Zellen unterschiedlicher Typen festlegen, die in der Simulation geeignet zu behandeln sind:

Nasse Zellen Diese enthalten das Fluid und dort läuft die Simulation wie oben beschrieben ab.

No-Slip-Zellen Diese Zellen modellieren Wände, an denen das Fluid haftet, die tangentiale Bewegung der Flüssigkeit wird gestoppt, ihre senkrechte Bewegung reflektiert.

Geschwindigkeitszellen In diesen Zellen wird eine konstante Geschwindigkeit durch äußere Einflüsse fix vorgegeben.

Die Simulation kann mit periodischen Randbedingungen durchgeführt werden. Verzichtet man darauf, kann sie schneller ablaufen – insbesondere auf der GPU, da dann Index-Berechnungen weniger aufwendig sind. Dann ist allerdings sicherzustellen, dass sich am Rand nur No-Slip-Zellen befinden, da andernfalls die Simulation ungültige Ergebnisse liefert.

Ergeben sich negative Dichten, so waren die Randbedingungen ungültig gewählt, so dass sich keine Lösung einstellt. Das geschieht insbesondere dann, wenn eine zu hohe Geschwindigkeit vorgegeben wird (> 0.1) und wenn wegen eines hohen ω (nahe bei 2) zu turbulente Strömungen entstehen.

5.4 Übertragung der Ergebnisse auf die Wirklichkeit

Um das Modell auf ein reales Problem anzuwenden, müssen folgende Größen bekannt sein:

- die dynamische Viskosität η des Fluids in $\frac{\text{Ns}}{\text{m}^2}$ und
- die Dichte ρ des Fluids in $\frac{\text{kg}}{\text{m}^3}$

oder

- seine kinematische Viskosität $\nu = \eta/\rho$ in $\frac{\mathrm{m}^2}{\mathrm{s}}$.

Im LBM-Modell werden nur dimensionslose Größen verwendet, die sich folgendermaßen ergeben:

- die Gitter-Dichte $\rho^* = \rho\frac{\Delta x^3}{\Delta m}$,
- die Gitter-Viskosität $\nu^* = \nu\frac{\Delta t}{\Delta x^2}$.

Die Gitterviskosität ν^* steht mit dem Relaxationskoeffizienten ω in direktem Zusammenhang:

$$\omega = \frac{2}{6\nu^* + 1}.$$

Also können bei der Abbildung auf ein LBM-Modell folgende Größen gewählt werden:

- die Zeitschrittweite Δt,
- die Zellgröße Δx und
- die Einheitsmasse Δm.

Die Einheitsmasse ist in unserem einfachen LBM-Modell, bei dem keine externen Kräfte berücksichtigt werden, ohne Einfluss auf das Modellverhalten, da sie nur eine Skalierung der Dichte bewirkt.

Beispielsweise besitzt Wasser bei einer Temperator von 20° Celsius eine dynamische Viskosität von 10^{-3} $\mathrm{Ns/m^2}$ und somit wegen seiner Dichte von 1000 $\mathrm{kg/m^3}$ eine kinematische Viskosität von 10^{-6} $\mathrm{m^2/s}$. Wählt man als Kantenlänge einer Gitterzelle $0,1$ m, so ergibt sich als Gitter-Dichte $\rho^* = \rho\frac{\Delta x^3}{\Delta m} = 1$. Mit $\omega = 1,95$ ergibt sich $\nu^* = \frac{2/\omega - 1}{6} \simeq 0,0042735$ und somit $\Delta t = (\Delta x)^2\nu^*/\nu \simeq (0,1\ \mathrm{m})^2 \cdot 0,0042735/10^{-6}\ \mathrm{m^2 s^{-1}} = 42,735$ s.

D. h. mit den voreingestellten Werten des Beispielprogramms wird z. B. das Verhalten von Wasser auf einem Gebiet von $3,2 \times 3,3 \times 3,2$ $\mathrm{m^3}$ simuliert, wobei ein Schritt in der Simulation ca. 43 Sekunden in Echtzeit entspricht. Die erzwungene Gitter-Geschwindigkeit von $0,1$ entspricht dabei einer Geschwindigkeit von $0,23$ mm/s.

Genauso hat man aber bei einer Zellgröße von 1 $\mathrm{mm^3}$ und Wahl der Einheitsmasse $\Delta m = 1$ mg das Verhalten von Wasser annähernd in einem Würfel von $3,2$ cm Kantenlänge simuliert. Ein Zeitschritt entspricht dabei aber nur noch ca. $0,043$ s, so dass sich aus der vorgegebenen Gittergeschwindigkeit von $0,1$ dann etwa $2,3$ mm/s ergeben.

5.5 CPU-Implementation

Die CPU-Implementation in der Klasse *LBMD3Q19* ist eine direkte Umsetzung des oben Beschriebenen. Zu jeder von *m_width* × *m_height* × *m_depth* Zellen wird die Verteilungsfunktion gespeichert. Damit dieser Speicher mit *new* allokiert werden kann, sind diese Daten in einem linearen Array abgelegt. Dadurch ergibt sich ein Array, das über 4 Laufvariablen, i für die x-Koordinate, j für die y-Koordinate und k für die z-Koordinate sowie l für die Nummer der Elementargeschwindigkeit, indiziert wird.

Dazu steht bereits eine Methode *index(int i, int j, int k, int l)* zur Verfügung, die den zugehörigen Index im linearen Array bestimmt. Für Werte, die einmal pro Zelle gespeichert werden müssen, gibt es noch eine Methode *index(int i, int j, int k)*, die nur von den drei Koordinatenparametern abhängt.

Zur Speicherung der Verteilungen existieren zwei Arrays, die abwechselnd zum Lesen und Schreiben im Streaming-Schritt dienen: dadurch ist die Reihenfolge, in der das Array beim Strömen durchlaufen wird, unerheblich, da man nicht darauf achten muss, keine Werte, die erst noch in Nachbarzellen übertragen werden müssen, zu überschreiben.

Das Strömen ist in der Methode *streamCpu()*, die Kollision in der Methode *collideCpu()* und die Berechnung von Dichte und Geschwindigkeit in *analyzeCpu()* implementiert.

5.6 GPU-Portierung

Es liegt bereits eine CPU-Implementation der LBM-Simulation vor. Deine Aufgabe ist es, diese vollständig auf der GPU zu implementieren. Du kannst dazu eine Programmiermethode Deiner Wahl verwenden. Beispielsweise könntest Du versuchen, die Simulation in einem Fragment-Programm zu implementieren. Oder aber Du könntest OpenCL verwenden, um die Simulation auf der GPU ablaufen zu lassen. Verfügst Du über eine Grafikkarte von Nvidia, dann könntest Du, genauso wie die Referenzimplementation, auf CUDA zurückgreifen.

Wichtig ist, dass die Simulation vollständig auf der GPU abläuft, dass also für die Methoden *streamCpu()* und *collideCpu()* GPU-Implementationen erstellt werden. Außerdem sollte unnötiger Datentransfer zwischen CPU und GPU vermieden werden und – wie im Referenzprogramm – eine Wechseln zwischen der CPU- und GPU-Implementation möglich sein.

Außerdem solltest Du neben den beiden Simulationsschritten auch noch die Visualisierung der Daten vorbereiten, so dass hierzu nicht übermäßige Kommunikation notwendig ist, also die Methode *analyzeCpu()* portieren.

Dir steht es frei, die gleichen Datenstrukturen wie bei der CPU-Implementation zu verwenden oder auch eine andere Datenorganisation zu wählen. Das kann entscheidend für gute Leistung Deiner GPU-Implementation sein.

Aufgrund der einfachen Struktur des LBM-Algorithmus und der geordneten Speicherzugriffe solltest Du aber ohne die Synchronisationsmechanismen für

Threads innerhalb eines Blocks auskommen.

6 Eigene Erweiterung

Erweitere das Programm in einer sinnvollen Art und Weise. Das könnte z. .B. folgendes sein:

- entkopple Simulation und Benutzerinteraktion, so dass ein Rotieren auch möglich ist, während die Simulation rechnet,
- implementiere weitere Visualisierungsmethoden, wie beispielsweise direktes Volumenrendering, Isoflächen, mit der Maus steuerbare Schnittflächen, Stromlinien oder animierte Partikel, Line Integral Convolution auf 2D-Schnitten, . . . ,
- ermögliche die gleichzeitige Nutzung mehrerer Grafikkarten für beschleunigte Simulationen,
- implementiere ein effizientes Verfahren zur Bestimmung von Maxima und Minima auf der GPU,
- ermögliche eine Interaktion mit der Simulation, so dass beispielsweise interaktiv zur Laufzeit der Simulation Geschwindigkeiten vorgeschrieben werden oder neue Grenzen gesetzt werden können,
- halbiere den Platz, den die Simulation zum Speichern der Geschwindigkeitsverteilungen benötigt,
- ermögliche das Simulieren von Situationen, die größer sind als der auf der GPU zur Verfügung stehende Speicher,
- . . .

7 Zusammenfassung

Mit dem Wissen über Computergrafik, Grafikprogrammierung und Umwidmung von Grafikhardware zu anderen Zwecken bist Du jetzt bestens gerüstet für die letzte Aufgabe!

8 Bewertungsrichtlinien

Wenn Du die folgenden Bedingungen erfüllst, dann kannst Du die Höchstpunktzahl von 20 Punkten erreichen:

1 Punkt Der Quelltext ist, mit Ausnahme der auf der GPU ablaufenden Teile, vollständig in objektorientiertem C++ verfasst. Der Code kompiliert auf dem Referenzsystem mit dem GNU C++-Compiler ohne Fehler und erzeugt mit den Optionen `-Wall -O2` keine unnötigen Warnungen. Auch die anderen Programmierrichtlinien wurden befolgt.

1 Punkte Der gesamte Quelltext ist in englischer Sprache kommentiert und die Kommentare erklären die algorithmische Struktur des C++-Codes. Die Kommentare sind so formatiert, dass sie das Erzeugen einer HTML- und LaTeX-Dokumentation mit `doxygen` erlauben.

2 Punkte Du hast eine zusätzliche interessante Ausgangssituation mit zugehörigen Randbedingungen zur Simulation geschaffen.

2 Punkte Auf der GPU wird Speicher für die Daten angelegt, die Daten werden zur und von der GPU transferiert.

3 Punkte Das Strömen ist auf der GPU korrekt implementiert.

3 Punkte Die Kollision ist auf der GPU korrekt implementiert.

2 Punkte Die GPU-Implementation bewirkt eine Beschleunigung der Simulation (mind. um Faktor 10 gemessen in Mlup/s auf dem Referenzsystem vispw.rrz.uni-koeln.de).

2 Punkte Zur Visualisierung werden auf der GPU Dichte/Druck und Geschwindigkeit der Zellen extrahiert.

4 Punkte Das Programm ist um eine sinnvolle Funktion erweitert und das ist in der Datei `README` dokumentiert.

Literatur

[1] Brookgpu. `http://www.graphics.stanford.edu/projects/brookgpu/`, 2004.

[2] Opencl. `http://www.khronos.org/opencl/`, 2009.

[3] Cuda c programming guide, version 3.2. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`, 2010.

[4] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[5] Niels Thürey. A lattice boltzmann method for single-phase free surface flows in 3d. `http://graphics.ethz.ch/~thuereyn/download/nthuerey_030602_da.pdf`, 2003.