Implementierung und Optimierung von dreidimensionalen Stencil-Skeletten mithilfe paralleler Verarbeitung

Justus Dieckmann
[463347]

Bachelorarbeit

Erstprüfer:	Prof. Dr. Herbert Kuchen
Zweitprüfer:	Prof. Dr. Sergei Gorlatch
Betreuerin:	Nina Herrmann, M.Sc.
Abgabetermin:	14. Februar 2023

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Programmausschnitte

Abkürzungen

CPU Central Processing Unit. 2, 3, 5, 35

GPGPU General Purpose Computation on Graphics Processing Unit. 2

GPU Graphics Processing Unit. 2–6, 11–13, 15, 18, 20, 21, 25–28, 30–32, 35, III

ISL Iterative Stencil Loop. 1, 35

LBM Lattice-Boltzmann-Methode. 9

MIMD Multiple Instruction, Multiple Data. 2, 3

MISD Multiple Instruction, Single Data. 2

MPI Message Passing Interface. 6, 35

Muesli Muenster Skeleton Library. 1, 5, 35

SIMD Single Instruction, Multiple Data. 2, 3

SISD Single Instruction, Single Data. 2

SM Streaming Multiprocessor. 3–5

1. Einführung

Die effiziente Verarbeitung großer Datenmengen wird in den verschiedensten Bereichen, zum Beispiel bei der Bildverarbeitung[1] und Machine Learning[2] immer wichtiger. Eine besondere Bedeutung kommt dabei der parallelen Programmierung zu, mit welcher unter anderem mithilfe von Grafikkarten bei vielen Anwendungsfällen erhebliche Beschleunigungen erreicht werden können. Die Entwicklung paralleler Programme ist allerdings mit vielen zusätzlichen Schwierigkeiten verbunden. Die Muenster Skeleton Library (Muesli) versucht daher eine Schnittstelle mit verschiedenen viel verwendeten Mustern der parallelen Programmierung bereitzustellen, welche dafür sorgen, dass der Entwickler sich nicht mit den vielen technische Details auseinander setzen muss.

Eine wichtige Klasse von Algorithmen innerhalb der parallelen Programmierung stellen sogenannte Iterative Stencil Loops (ISLs) dar, welche zum Beispiel Anwendung bei dem Lösen partieller Differenzialgleichungen und der Simulation verschiedenster Teilcheninteraktionen finden. In diesen Programmen werden die Zellen eines regulären Gitters immer wieder anhand der Werte ihrer Nachbarn aktualisiert. [1]

Diese Arbeit setzt sich zum Ziel, ein effizientes und einfach zu benutzendes dreidimensionales `mapStencil`-Skelett in Muesli zu entwickeln, welches die Implementierung von ISLs vereinfacht. Dazu gibt das zweite Kapitel eine Einführung in die parallele Programmierung mit GPUs und vermittelt alle Grundlagen, die nötig sind, um das Skelett in Muesli entwickeln zu können.

Das dritte Kapitel erläutert zunächst die für diese Arbeit benötigten Teile des bereits existierenden Muesli-Programmcodes. Nach einigen technischen Vorüberlegungen wird das dreidimensionale `mapStencil`-Skelett implementiert und eine Beispielanwendung entwickelt. Außerdem wird die Beispielanwendung ohne die Hilfe von Muesli reimplementiert, um die Leistung und den Entwicklungsaufwand vergleichen zu können. Die Optimierung der beiden Implementierungen schließt die Entwicklung ab.

Im vierten Kapitel wird die Leistung des `mapStencil`-Skeletts für die Berechnung von ISLs mit unterschiedlichen Datengrößen und verschiedenen Anzahlen an benutzten GPUs mit der nativen Referenzimplementierung verglichen und evaluiert.

Abschließend werden die Ergebnisse zusammengefasst und mögliche Ansatzpunkte für die weitere Forschung genannt.

[1] https://opencv.org/about/
[2] https://www.tensorflow.org/

2. Parallele Programmierung auf GPUs

Die Graphics Processing Unit (GPU) wurde ursprünglich entwickelt, um einen Prozessor zu bieten, der deutlich besser als die Central Processing Unit (CPU) dazu geeignet ist, zwei- und dreidimensionale grafische Daten zu generieren und zu verarbeiten. So wurde sie konstruiert, um identische Berechnung für zum Beispiel eine große Anzahl an Pixeln effektiv ausführen zu können. Der Hersteller Nvidia hat 2006 mit der Einführung der Chip-Architektur *Tesla* begonnen, universell nutzbare Rechenkerne (*CUDA-Kerne*) anstelle von spezifischen Recheneinheiten für die verschiedenen Phasen der Grafikverarbeitungspipeline zu verwenden [2]. So wurde es möglich die Leistung der GPU auch für parallele Anwendungen abseits der Grafikverarbeitung zu nutzen. Unter diesem Aspekt hat sich in den letzten Jahren auch der Begriff General Purpose Computation on Graphics Processing Unit (GPGPU) etabliert [3, S. 767]. Im Folgenden wird sich diese Arbeit auf GPUs von Nvidia konzentrieren; die Grafikkarten anderer Hersteller sind aber ähnlich aufgebaut.

2.1. Klassifikation von Rechnerarchitekturen nach Flynn

Zur Einordnung von Rechnerarchitekturen hat Michael J. Flynn eine simple Klassifikation entwickelt, welche Rechner danach kategorisiert, ob sie einen oder mehrere Instruktionsströme ausführen und einen oder mehrere Datenströme gleichzeitig behandeln (siehe Abb. 2.1).

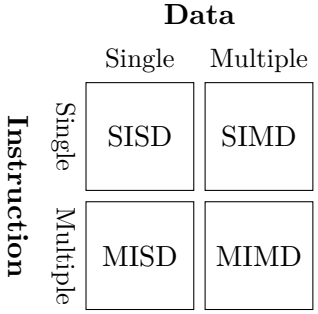


Abbildung 2.1: Klassifikation von Rechnerarchitekturen nach Flynn. Angelehnt an: [4, S. 33].

Die Single Instruction, Single Data (SISD)-Architektur bezeichnet dabei einen Rechner, der mit einem Instruktionsstrom einen Datenstrom bearbeitet, wie es zum Beispiel ein klassischer Einkernprozessor macht. Ein Rechner nach Single Instruction, Multiple Data (SIMD)-Architektur hingegen führt zu jedem Zeitpunkt eine Instruktion auf mehreren Datenströmen aus. In der Fachliteratur ist streitig, ob es für Multiple Instruction, Single Data (MISD)-Architekturen, welche mit mehreren Befehlsströmen einen Datenstrom

behandeln, sinnvolle Anwendungen und Beispiele gibt. Multiple Instruction, Multiple Data (MIMD)-Rechner können gleichzeitig mehrere Befehle auf unterschiedlichen Datenströmen ausführen. Ein Beispiel dafür sind Computer mit Mehrkernprozessoren. [5], [4, S. 32f.]

2.2. Architektur von GPUs

Eine moderne Nvidia-GPU besteht aus mehreren sogenannten Streaming Multiprocessors (SMs), welche wiederum eine Vielzahl an CUDA-Kernen enthalten. Jeder SM ist dabei ein SIMD-Prozessor; es führen also zu jedem Zeitpunkt alle Kerne innerhalb eines SMs die gleiche Instruktion für unterschiedliche Daten aus. Zum einen ermöglicht diese Funktionsweise eine sehr große mögliche Rechenleistung: Im Vergleich zwischen der im Januar 2023 ähnlich teuren GPU Nvidia RTX 3060 und CPU Intel Core i7-12700 kann die GPU rund 12 740 Milliarden Fließkommaoperationen pro Sekunde durchführen [6], während die CPU nur auf circa 403 Milliarden Operationen pro Sekunde kommt [7]. Andererseits zeigt die Funktionsweise auch eine große Limitation der GPU-Programmierung auf: sobald die Berechnungen der CUDA-Kerne innerhalb eines SMs in unterschiedliche Codezweige divergieren, muss der SM diese einzeln nacheinander durchlaufen und deaktiviert dabei alle Kerne, die nicht die derzeitige Instruktion ausführen müssen. [2], [8, Kap. 4]

Um einen Überblick über die Größenverhältnisse einer GPU zu bekommen, lohnt es sich als Beispiel noch einmal die RTX 3060 anzuschauen: Sie beinhaltet 28 SMs mit jeweils 128 CUDA-Kernen, enthält also insgesamt 3584 Kerne [6].

2.3. CUDA als Schnittstelle für GPUs

Um die Rechenleistung einer GPU sinnvoll nutzen und Code auf ihr ausführen zu können, wird eine Schnittstelle benötigt. Für grafische Anwendungen gibt es zum Beispiel schon seit langem das plattformunabhängige Projekt OpenGL[3] und Direct3D[4] für Windows. Um die GPU abseits grafischer Anwedungen benutzen zu können, gibt es OpenCL[5] und CUDA[6]. Letztes wird dieser Arbeit genutzt.

Damit per CUDA die GPU für allgemeine Berechnungen benutzt werden kann, wurde CUDA-C++, eine Erweiterung von C++, entwickelt. Somit kann man Funktionen, welche auf der GPU ausgeführt werden sollen, als C++-Code definieren, und diese mit einem speziellen Syntax aufrufen. Diese Funktionen werden in dem Kontext auch *Kernel* genannt.

In Listing 2.1 wird eine beispielhafte Benutzung des CUDA-Toolkits demonstriert.

[3] https://www.opengl.org/
[4] https://learn.microsoft.com/en-us/windows/win32/direct3d/
[5] https://www.khronos.org/opencl/
[6] https://docs.nvidia.com/cuda/cuda-c-programming-guide/

```
1  const size_t ELEMENTS = 5;
2
3  __global__ void sumInPlace(int *dest, const int *other) {
4      size_t i = blockIdx.x * blockDim.x + threadIdx.x;
5      dest[i] = dest[i] + other[i];
6  }
7
8  int main() {
9      // Setup //
10     int *gpuA, *gpuB;
11     cudaMalloc(&gpuA, ELEMENTS * sizeof(int));
12     cudaMalloc(&gpuB, ELEMENTS * sizeof(int));
13     int a[] = {1, 2, 3, 4, 5};
14     int b[] = {2, 3, 4, 5, 6};
15     cudaMemcpy(gpuA, a, ELEMENTS * sizeof(int), cudaMemcpyDefault);
16     cudaMemcpy(gpuB, b, ELEMENTS * sizeof(int), cudaMemcpyDefault);
17
18     // Kernel call //
19     sumInPlace<<<1, ELEMENTS>>>(gpuA, gpuB);
20
21     // Get results //
22     int result[ELEMENTS];
23     cudaMemcpy(result, gpuA, ELEMENTS * sizeof(int), cudaMemcpyDefault);
24     for (int i : result) {
25         std::cout << i << " "; // Prints 3 5 7 9 11
26     }
27 }
```

Programmausschnitt 2.1: Beispiel für CUDA-Anwendung

Zunächst werden in Zeile 11-12 mit der Funktion `cudaMalloc(void** devPtr, size_t bytes)` zwei Arrays mit jeweils fünf Elementen auf der GPU allokiert. Die Methodensignatur von `cudaMalloc` ist anders, als man es zunächst anhand des Äquivalents `malloc` in C erwarten würde, da fast alle Funktionen der CUDA-API ein Element des Enums `cudaError_t` zurückgeben, über den die sie entweder per `cudaSuccess` Erfolg, oder mit einem anderen Rückgabewert einen Fehler vermelden. Der Rückgabewert sollte normalerweise überprüft werden; dies wird hier aber der Übersichtlichkeit halber weggelassen. Als nächstes werden mithilfe von `cudaMemcpy(void* dst, void* src, size_t bytes, cudaMemcpyKind kind)` Daten aus dem Arbeitsspeicher in die GPU übertragen. [9, Kap. 6.2]

In Zeile 19 findet nun der Kernelaufruf statt. Er besteht aus dem Aufruf einer mit `__global__` qualifizierten Methode mit zusätzlichen in dreifachen Kleiner-/Größerzeichen eingeschlossenen Parametern. Diese geben unter anderem an, wie viele Threads die Kernelfunktion aufrufen. Hierbei wird eine vorher festgelegte Anzahl an Threads in einem Block zusammengefasst. Jeder Funktionsaufruf wird in einem eigenen Thread ausgeführt, wobei immer eine bestimmte Anzahl an Threads in einem Block zusammengefasst werden. Der erste zusätzliche Parameter bestimmt, wie viele Blöcke gestartet werden; der zweite Parameter sagt aus, wie viele Threads jeder Block enthält. Alle Threads innerhalb eines Blocks werden auf dem gleichen SM ausgeführt, sodass die Anzahl der

Threads pro Block dadurch limitiert ist, wie viele Threads ein SM gleichzeitig ausführen kann. Bei modernen GPUs liegt dieses Limit bei 1024. [8, Kap. 2]

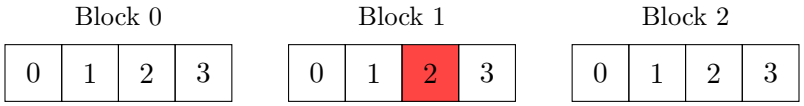


Abbildung 2.2: Threaddiagramm für den Beispielaufruf `kernel<<<3, 4>>>()`

Innerhalb der Kernelfunktion erhält man über vordefinierte Variablen Informationen über den derzeitigen Thread und Block. So ist im markierten Thread in Abb. 2.2 die Anzahl der Blöcke `gridDim.x = 3`, der Blockindex `blockIdx.x = 1`, die Anzahl der Threads pro Block `blockDim.x = 4` und der Threadindex innerhalb des Blockes `threadIdx.x = 2`. [8, Kap. 2]

So wird in dem aufgerufenen Kernel `sumInPlace` in Z. 4 der globale Threadindex berechnet, und in Z. 5 die Zahl an diesem Index im zweiten Array auf die entsprechende Zahl im ersten Array aufaddiert. Zu guter Letzt werden ab Z. 23 die Ergebnisse von der GPU zurück in den Arbeitsspeicher kopiert und ausgegeben.

2.4. Skelett-Programmierung

Die parallele Programmierung ist mit einigen zusätzlichen Schwierigkeiten verbunden. So muss zum Beispiel darauf geachtet werden, dass durch den gleichzeitigen Zugriff auf Daten keine Race-Conditions entstehen und, dass die aktuellen Daten immer an der richtigen Stelle auf den verschiedenen Geräten liegen, auf denen sie gerade gebraucht werden. Zusätzlich ist im Falle eines Fehlers der Fehlerfindungsprozess im Gegensatz zu seriellen, auf der CPU ausgeführten Programmen deutlich erschwert.

Um die Entwicklung paralleler Programme zu vereinfachen, können sogenannte algorithmische Skelette benutzt werden. Diese versuchen technische Details so gut wie möglich zu verstecken, fassen oft verwendete Muster der parallelen Programmierung zusammen und stellen sie als einfach verwendbare Schnittstelle bereit. Durch das häufige Wiederbenutzung der Muster wird Zeit für eine besonders sorgfältige und optimierte Implementierung geschaffen. [10], [11]

In Kapitel 2.5 folgt ein Beispiel einer Skelettbibliothek.

2.5. Muesli

Die Muenster Skeleton Library (Muesli) ist eine in C++ geschriebene, templatebasierte Skelettbibliothek für die Entwicklung paralleler Programme. Die grundlegenden Datenstrukturen in Muesli sind das eindimensionale *DistributedArray* (DA), die zweidimensionale *DistributedMatrix* (DM) und der dreidimensionale *DistributedCube* (DC), welche

mittels Templating beliebige zusammenhängende Daten speichern können. Templating ist ein Konzept in C++, das generische Programmierung erlaubt, ohne aber dabei einen Mehraufwand zur Laufzeit zu erfordern.

Muesli benutzt CUDA, um Berechnungen auf GPUs auszuführen, OpenMP, um die Arbeit auf mehrere Prozessoren aufzuteilen und Message Passing Interface (MPI), damit die Berechnung auf mehrere Computer aufgeteilt werden kann [12, S. 75-77].

Diese Arbeit konzentriert sich auf die Parallelisierung mittels CUDA.

2.5.1. Implementierte Skelette

Muesli implementiert die Skelette `map`, `zip` und `fold` in verschiedenen Varianten:

- Der Ausdruck `Index` im Skelettnamen bedeutet, dass dem Funktor zusätzlich zu den Werten an jeder Stelle auch der Index der Stelle übergeben wird.
- Der Ausdruck `InPlace` im Skelettnamen bedeutet, dass das Ergebnis nicht in eine weitere Datenstruktur, sondern in die Ausgangsdatenstruktur geschrieben wird.
- Die beiden Kennzeichner `Index` und `InPlace` können als `IndexInPlace` kombiniert anzeigen, dass das Skelett dem Funktor sowohl den Index übergibt, als auch das Ergebnis in die Ausgangsdatenstruktur geschrieben wird.
- Eine Skelettvariante ohne `Index` oder `InPlace` im Namen übergibt dem Funktor nicht zusätzlich den Index und schreibt das Ergebnis in eine weitere Datenstruktur.

map

Das Skelett `map` wendet den Funktor auf jedes Element der Datenstruktur an. Muesli implementiert die Varianten `map`, `mapIndex`, `mapInPlace` und `mapIndexInPlace` für alle drei Datenstrukturen. In Abbildung 2.3 wird eine beispielhafte Anwendung skizziert.

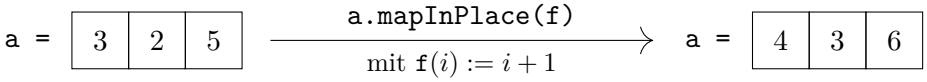


Abbildung 2.3: Beispiel des `MapInPlace`-Skeletts

zip

Das Skelett `zip` ist dafür geeignet, zwei gleichförmige Datenstrukturen zu verarbeiten. Es übergibt für alle Zellen der Datenstrukturen die beiden jeweiligen Elemente der ersten und zweiten Struktur an den Funktor und schreibt das Resultat in die entsprechende Stelle in einer dritten Datenstruktur (oder überschreibt den Wert der ersten Datenstruktur für `InPlace`-Varianten).

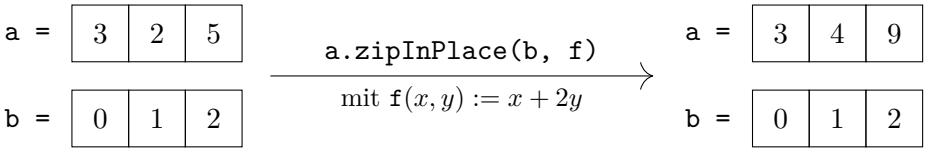


Abbildung 2.4: Beispiel des `ZipInPlace`-Skeletts

fold

Das Skelett `fold` reduziert die Daten einer Datenstruktur zu einem einzelnen Wert, indem immer wieder zwei Werte von dem übergebenen Funktor kombiniert werden, bis nur ein Wert überbleibt.

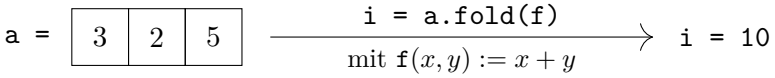


Abbildung 2.5: Beispiel des `Fold`-Skeletts

2.5.2. Beispielanwendung

In der in Programmausschnitt 2.2 gezeigten Beispielanwendung wird in Zeile 14 zunächst die Initialisierungsfunktion von Muesli aufgerufen. Im Anschluss werden Objekte der Funktorklassen `Sum` und `Product` erstellt, welche beide von `Functor2<int, int, int>` erben. Sie nehmen dementsprechend zwei `int`-Argumente entgegen und geben einen Wert vom Typ `int` zurück. In Z. 19 und 20 werden zwei eindimensionale Muesli-Datenstrukturen *DistributedArray (DA)* erstellt, die jeweils vier Elemente vom Typ `int` speichern. Das erste DA wird außerdem mit Dreien gefüllt.

In Z. 21 wird das algorithmische Skelett `mapIndex` benutzt, welches aufgrund der übergebenen Parameter die Zahlen aus `da1`, summiert mit ihren jeweiligen Indices, in `da2` schreibt. Anschließend sorgt der Aufruf des Skeletts `zipInPlace` dafür, dass jede Zahl in `da1` mit ihrem Produkt mit dem entsprechenden Wert in `da2` überschrieben wird.

```
1  class Sum : public Functor2<int, int, int> {
2      public: MSL_USERFUNC int operator() (int x, int y) {
3          return x + y;
4      }
5  };
6
7  class Product : public Functor2<int, int, int> {
8      public: MSL_USERFUNC int operator() (int x, int y) {
9          return x * y;
10     }
11 };
12
```

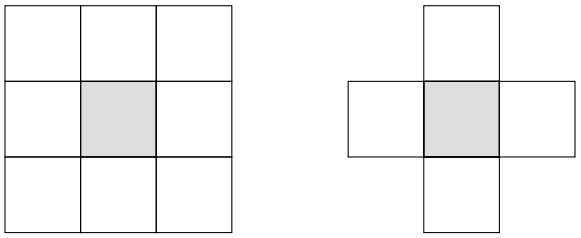
```
13 int main(int argc, char** argv) {
14     msl::initSkeletons(argc, argv);
15
16     Sum sum;
17     Product product;
18
19     DA<int> da1(4, 3);              // da1: [3,  3,  3,  3]
20     DA<int> da2(4);
21     da1.mapIndex(sum, da2);         // da2: [3,  4,  5,  6]
22     da1.zipInPlace(da2, product);   // da1: [9, 12, 15, 18]
23 }
```

Programmausschnitt 2.2: Beispiel einer Muesli-Anwendung

2.6. Stenciloperation

Eine Stenciloperation ist eine Operation, welche für eine Zelle einen neuen Wert auf Grundlage ihrer lokalen Nachbarzellen berechnet.

Welche Nachbarn genau für die Berechnung des neuen Wertes in Betracht gezogen werden, wird durch die Wahl des Stencils bestimmt. Siehe Abb. 2.6 für eine Auswahl von Stencils.



(a) Moore-Nachbarschaft (b) Von-Neumann-Nachbarschaft

Abbildung 2.6: Auswahl von verschiedenen möglichen Stencils

Stenciloperation haben viele verschiedene Anwendungen. Eine Gruppe an einfachen zweidimensionalen Stenciloperationen findet sich in der Bildbearbeitung, in der einige Bildfilter als *Faltung (eng. Convolution)*, eine gewichtete Summe der Pixel in der 3x3 Moore-Nachbarschaft, definiert sind. Mit welchem Faktor ein Nachbarschaftspixel in der Summe multipliziert wird, bestimmt die *Faltungsmatrix*. Beispiele für Faltungen sind der glättende Mittelwertfilter (Abb. 2.7a) oder der Laplacefilter (Abb. 2.7b), welcher der Kantenfindung dient. [13]

Weiterhin werden Stenciloperationen für die Berechnung von zellulären Automaten und für das Lösen partieller differenzieller Gleichungen benutzt.

$$\frac{1}{9} \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

(a) Mittelwertfilter [14] (b) Laplacefilter [15]

Abbildung 2.7: Faltungsmatrizen für Bildfilter

2.7. Beispielanwendung LBM-Simulation

Um die Performance des dreidimensionalen Stencil-Skeletts zu testen, wird als Beispielanwendung in dieser Arbeit eine Implementierung der Lattice-Boltzmann-Methode (LBM), ein Modell zur numerischen Strömungssimulation, benutzt. Die Lattice-Boltzmann-Methode simuliert die Bewegung von Gasen in einem Gitter von Zellen in diskreten Zeitschritten. Es gibt verschiedene Gitter, die mit der LBM benutzt werden können. Diese unterscheiden sich durch ihre Dimensionalität (zwei- oder dreidimensional) und dadurch, welche der Zellen im Umfeld einer Ausgangszelle als Nachbarzellen betrachtet werden, also in welche Zellen Luftteilchen strömen können. Die verschiedenen Gitter werden nach dem Muster DdQq benannt, wobei d die Dimension und q die Anzahl der Nachbarzellen angibt.

Jede Zelle besitzt eine Verteilungsfunktion, welche angibt, was mit den Luftteilchen innerhalb der Zelle im nächsten Zeitschritt geschieht. Die Luftteilchen können in eine der Nachbarzellen übergehen oder in der Zelle bleiben.

Ein Simulationsschritt der LBM besteht aus zwei Phasen: der Kollisionsphase und der Strömungsphase. In der Kollisionsphase interagieren und kollidieren die in der Strömungsphase in jede Zelle eingeströmten Luftströme miteinander, wodurch sich eine neue Verteilungsfunktion ergibt. In der Strömungsphase strömen die Luftteilchen entsprechend der Verteilungsfunktion in ihre Nachbarzellen.

Die Verteilungsfunktion $f_i(x,t)$ gibt für eine Zelle x und einen Zeitpunkt t an, wie viele Luftteilchen sich im nächsten Zeitschritt zum i-ten Nachbarn bewegen werden. Der nullte Nachbar bezeichnet dabei die Zelle selbst.

Es gibt verschiedene mögliche Kollisionsoperatoren, die innerhalb der Lattice-Boltzmann-Methode benutzt werden können. In der folgenden Gleichung wird der Bhatnagar-Gross-Krook (BGK)-Operator benutzt, um den Kollisionsschritt zu definieren:

$$f_i^*(x,t) := f_i(x,t) - \frac{\Delta t}{\tau} \left(f_i(x,t) - f_i^{\mathrm{eq}}(x,t) \right). \tag{2.1}$$

Hierbei ist f_i^* die Verteilungsfunktion nach dem Kollisionsschritt, Δt der Zeitschritt, der in jedem Simulationsschritt simuliert werden soll und τ eine Konstante, die angibt, wie

schnell sich die Simulation einem Gleichgewichtszustand annähert. Damit beeinflusst τ die Viskosität der Gase. Der Gleichgewichtszustand wird berechnet durch

$$f_i^{\mathrm{eq}}(x,t) := w_i \rho \left(1 + \frac{u \cdot c_i}{c_s^2} + \frac{u \cdot c_i}{2c_s^4} + \frac{u \cdot u}{2c_s^2}\right), \tag{2.2}$$

wobei w_i von dem gewählten Gitter abhängende Gewichte und c_i die Positionen der Nachbarzellen relativ zur Ursprungszelle sind. Die Konstante c_s bezeichnet die Schallgeschwindigkeit innerhalb des Modells. Die Massendichte ρ und die Impulsdichte u sind angegeben durch

$$\rho(x,t) = \sum_i f_i(x,t), \qquad \rho u(x,t) := \sum_i c_i f_i(x,t). \tag{2.3}$$

Der Strömungsschritt erzeugt die nächste Verteilungsfunktion f_i und ist durch

$$f_i(x + c_i \Delta t, t + \Delta t) := f_i^*(x,t) \tag{2.4}$$

definiert. [16, S. 61-68]

3. Implementierung des dreidimensionalen Stencil-Skeletts

3.1. Bestehende Implementierung des DistributedCube

Der DistributedCube (DC) stellt in Muesli eine dreidimensionale Würfelförmige Datenstruktur dar, dessen Verarbeitung auf verschiedene Rechner und GPUs verteilt werden kann.

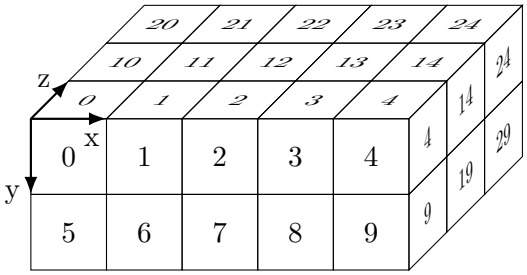


Abbildung 3.1: Indexierung eines DistributedCubes mit Breite 5, Höhe 2 und Tiefe 3

Für die Implementierung des Stencil-Skeletts sind einige Eigenschaften des `DistributedCube` wichtig:

- Die Klasse `DC<T>` des `DistributedCube` ist ein Template, wobei `T` den Typen des in jeder Zelle gespeicherten Elementes bezeichnet.
- In `int ncol, nrow, depth` werden die Abmessung des Würfels in die x-, y-, und z-Dimension gespeichert.
- `T* localPartition` ist ein Pointer zu der Speicherregion, die alle Elemente des Verteilten Würfels auf dem aktuellen Rechner enthält.
  Der Index einer Zelle ist dabei anhand ihrer Koordinaten wie folgt zu berechnen: $i(x, y, z) := z \cdot \texttt{nrow} \cdot \texttt{ncol} + y \cdot \texttt{ncol} + x$.
- `bool cpuMemoryInSync` besagt, ob die Daten in `localPartition` derzeit aktuell sind, oder ob sich neuere Daten auf den GPUs befinden.
- `int ng` spezifiziert die Anzahl der GPUs, auf die die Datenverarbeitung aufgeteilt werden soll.
- `GPUExecutionPlan<T>* plans` ist ein Pointer zu einem Array der Größe `ng`, welches für jede GPU spezifiziert, welche Daten von dieser bearbeitet werden sollen. Dabei enthält jeder `GPUExecutionPlan<T>` unter anderem:
  - `int firstCol, firstRow, firstDepth` bezeichnet die Spalte, Reihe und Tiefe (also x, y, und z-Koordinate) der ersten Zelle, die auf dieser GPU bearbeitet werden soll.

- `int lastCol, lastRow, lastDepth` beschreibt die Koordinaten der letzten Zelle, die auf dieser GPU bearbeitet werden soll.
- `T* d_Data` *(device data)* zeigt auf den Speicherbereich auf der GPU, in dem die Elemente liegen.
- `T* h_Data` *(host data)* zeigt auf den Speicherbereich des Rechners, in dem die entsprechenden Elemente liegen.

3.2. Vorüberlegungen zur Implementierung des MapStencil-Skeletts

3.2.1. MapStencil-Skelett mit verteilten Daten

Bei dem `mapStencil`-Skelett gibt es im Vergleich mit zum Beispiel dem `map`-Skelett zusätzliche Schwierigkeiten bei der parallelen Berechnung mithilfe von GPUs. Zur parallelen Berechnung des `map`-Skeletts reicht es aus, aufzuteilen, welche Zellen auf welcher GPU berechnet werden sollen, die Ausgangsdaten der Zellen auf die jeweiligen GPUs zu kopieren, um dann den Funktor anzuwenden. In Muesli wird ein `DistributedCube` auf GPUs aufgeteilt, indem er entlang der z-Achse in gleich große Stücke bzw. Stücke mit gleicher Anzahl an xy-Ebenen „geschnitten“ wird.

Bei dem `mapStencil`-Skelett muss andererseits bedacht werden, dass für die Berechnung von Zellen, die am Rande des Bereiches einer GPU liegen, auch Nachbarzellen herangezogen werden müssen, welche in dem Bereich einer anderen GPU liegen, wie man in Abb. 3.2 sehen kann. Aus diesem Grund müssen zusätzlich die Nachbarebenen jeder GPU als Puffer mitkopiert werden. Wie viele Ebenen jeweils zusätzlich kopiert werden müssen, definiert der Radius vom Stencil.

3.2.2. GPUs und Funktionen höherer Ordnung

Skelette sind von Natur aus Funktionen höherer Ordnung, also Funktionen, die andere Funktionen als Parameter erwarten [17, Kap. 2.2]. In C++ werden Funktionen als Parameter standardmäßig mit dem Übergeben von Funktionspointern realisiert. Doch in der GPU-Programmierung kommt dort ein Problem auf: Wird ein Funktionspointer erstellt, indem die Funktion auf der CPU referenziert wird, so zeigt der Pointer auf die Stelle im Arbeitsspeicher des Rechners. Wird der Pointer an die GPU übergeben und mit ihm versucht die Funktion aufzurufen, so scheitert dies, da die GPU nicht auf den Arbeitsspeicher des Rechners zugreifen kann und der Pointer auf ungültigen Speicher zeigt.

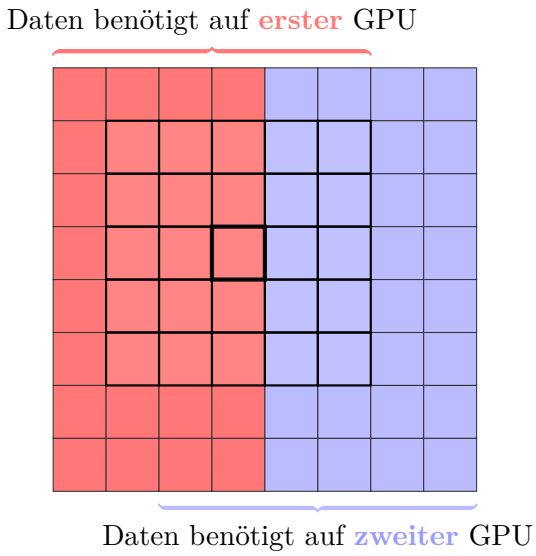


Abbildung 3.2: Verteilte Datenstruktur auf zwei GPUs mit Stencil von Radius 2 um $(4, 4)$.

Einen möglichen Lösungsansatz stellt die Funktion `cudaMemcpyFromSymbol` dar, mit welcher der Inhalt einer `__device__`-qualifizierten Variable von der GPU in den Arbeitsspeicher kopiert werden kann. In Programmausschnitt 3.1 ist ein beispielhaftes Vorgehen zur Übergabe von Funktionspointern zur GPU aufgezeigt.

In Zeile 1 wird der Übersichtlichkeit halber der Typ eines Pointers zu einer Funktion, die keine Parameter erwartet und eine Ganzzahl zurückgibt, als `func_return_int_t` bezeichnet. In Zeile 7 wird eine globale `__device__`-Variable mit einem Pointer auf die `userfunction` in Zeile 3-5 initialisiert. Dadurch, dass die Variable in der GPU residiert, wird der Funktionspointer auch auf die Funktion im Speicher der GPU liegen. Die Adresse der `userfunktion` wird in Zeile 16 zunächst in die lokale Variable `gpu_op` kopiert, um dann in Z. 17 die Kernelfunktion mit einem Thread und `gpu_op` als Parameter zu starten. In der Kernelfunktion selbst ist die Funktion als Parameter aufgelistet, und wird wie eine normale Funktion in Z. 10 aufgerufen.

```
1  using func_return_int_t = int (*)();
2
3  __device__ int userfunction() {
4      return 42;
5  }
6
7  __device__ func_return_int_t pUserfunction = userfunction;
8
9  __global__ void kernel(func_return_int_t func) {
10     int n = func();
11     printf("%i\n", n);
```

```
12 }
13
14 int main() {
15     func_return_int_t gpu_op;
16     cudaMemcpyFromSymbol(&gpu_op, pUserfunction, sizeof(pUserfunction));
17     kernel<<<1, 1>>>(gpu_op);  // Output: 42
18 }
```

Programmausschnitt 3.1: Funktion als Parameter per Funktionspointer

Es fällt auf, dass bei diesem Ansatz nicht nur der Funktor selbst definiert werden muss, sondern zusätzlich eine `__device__`-Variable, welche auf die Funktion zeigt, was eine zusätzliche Unannehmlichkeit für den Endbenutzer bedeutet. Im Programmausschnitt 3.2 folgt eine Möglichkeit, wie Templates ausgenutzt werden können, um eine elegantere Variante zu programmieren. So ist die Kernelfunktion in Z. 8 ein Template, welches eine Funktion erwartet. Dadurch wird `kernel<userfunction>` in Z. 14 beim Kompilieren zu einer Funktion, welche keine Parameter entgegennimmt, und die `userfunction` aufruft.

```
1  using func_return_int_t = int (*)();
2
3  __device__ int userfunction() {
4      return 42;
5  }
6
7  template <func_return_int_t func>
8  __global__ void kernel() {
9      int n = func();
10     printf("%i\n", n);
11 }
12
13 int main() {
14     kernel<userfunction><<<1, 1>>>();  // Output: 42
15 }
```

Programmausschnitt 3.2: Funktion als Parameter per Templateargument

Diese Methode ist für den Endnutzer deutlich einfacher zu handhaben, und wird deshalb in Muesli benutzt.

3.3. Implementierung des MapStencil-Skeletts

Beim Aufruf des `mapStencil`-Skeletts wird zusätzlich zu dem Funktor mit der in Programmausschnitt 3.3 beschriebenen Methodensignatur auch die Größe des Stencils und der neutrale Wert vom Typ `T` übergeben. Dieser bestimmt den Wert der Zellen von Stencils, die außerhalb der Datenstruktur liegen.

Um das Ergebnis für eine Zelle zu berechnen, wird der Funktor mit einer Koordinate und einem Objekt der Klasse `PLCube` *(PaddedLocalCube)* aufgerufen.

```
1     template <typename T>
2     using DCMapStencilFunctor = T(*)(const PLCube<T> &cs,
3                                      int x, int y, int z);
```

Programmausschnitt 3.3: Definition des Typenalias `DCMapStencilFunctor`

3.3.1. Die Klasse PaddedLocalCube

Die Klasse `PLCube` *(PaddedLocalCube)* stellt für den Funktor die Schnittstelle zu der Datenstruktur dar. Mit diesem kann für alle Zellen im angegebenen Stencilradius um die übergebenen Koordinaten der Wert der Datenstruktur abgefragt werden. Ein `PLCube` speichert eine Referenz zum Datenbereich einer GPU, und verwaltet zusätzlich die Puffer mit Daten der beiden anliegenden Datenbereiche. Die Klasse besitzt die folgenden Eigenschaften:

- `int width, height, depth`: die Abmessungen der gesamten Datenstruktur.
- `int stencilSize`: die Stencilgröße, für die der `PLCube` angelegt wurde. Je größer der Stencil, desto größer müssen die Datenpuffer sein.
- `T neutralValue`: der Wert, der für Zellen außerhalb der Datenstruktur zurückgegeben werden soll.
- `int dataStartIndex`: der Index der Datenstruktur, an dem der Datenbereich dieser GPU beginnt.
- `int dataEndIndex`: der Index der Datenstruktur, an dem der Datenbereich dieser GPU endet.
- `int topPaddingStartIndex`: der Index der Datenstruktur, an dem der notwendige nach oben angrenzende Puffer beginnt. Der Puffer endet an `dataStartIndex-1`.
- `int bottomPaddingEndIndex`: der Index der Datenstruktur, an dem der notwendige nach unten angrenzende Puffer endet. Der Puffer beginnt an `dataEndIndex+1`.
- `T* data`: ein Pointer, der auf den Anfang des Datenbereiches zeigt, für den diese GPU verantwortlich ist.
- `T* topPadding, T* bottomPadding`: Pointer, die auf den Anfang der nach oben und unten angrenzenden Puffer zeigen.

```
1 PLCube(int width, int height, int depth, std::array<int, 3> start,
2     std::array<int, 3> end, int device, int stencilSize, T neutralValue,
3     T *data)
4     : width(width), height(height), depth(depth),
5     stencilSize(stencilSize), neutralValue(neutralValue),
6     dataStartIndex(msl::PLCube<T>::coordinateToIndex(start)),
7     dataEndIndex(msl::PLCube<T>::coordinateToIndex(end)),
8     topPaddingStartIndex(msl::PLCube<T>::coordinateToIndex(
```

```
9               std::max(start[0] - stencilSize, 0),
10              std::max(start[1] - stencilSize, 0),
11              std::max(start[2] - stencilSize, 0)
12      )),
13      bottomPaddingEndIndex(msl::PLCube<T>::coordinateToIndex(
14              std::min(end[0] + stencilSize, width - 1),
15              std::min(end[1] + stencilSize, height - 1),
16              std::min(end[2] + stencilSize, depth - 1)
17      )),
18      data(data) {
19  #ifdef __CUDACC__
20      cudaSetDevice(device);
21      cudaMalloc(&topPadding,
22          (dataStartIndex - topPaddingStartIndex) * sizeof(T));
23      cudaMalloc(&bottomPadding,
24          (bottomPaddingEndIndex - dataEndIndex) * sizeof(T));
25  #endif
26  }
```

Programmausschnitt 3.4: Konstruktor der Klasse `PLCube`

```
1  MSL_USERFUNC inline int coordinateToIndex(int x, int y, int z) const {
2      return (z * height + y) * width + x;
3  }
```

Programmausschnitt 3.5: Funktion `coordinateToIndex` der Klasse `PLCube`

Im Konstruktor, abgebildet in Programmausschnitt 3.4, werden die Klassenvariablen `width, height, depth, stencilSize, neutralValue` und `data` auf die jeweils entsprechenden übergebenen Argumente gesetzt. Der Konstruktor erwartet außerdem die Start- und Endkoordinaten als Array von drei Integers. Mithilfe von `coordinateToIndex` (Programmausschnitt 3.5) werden diese in Indices umgewandelt und in `dataStartIndex` und `dataEndIndex` geschrieben. Als nächstes wird der `topPaddingStartIndex` in Z. 8-12 berechnet, indem von allen drei Dimensionen der übergebenen Startkoordinate `stencilSize` abgezogen wird und außerdem sichergestellt wird, dass die resultierenden x-, y- und z-Werte mindestens 0 betragen, und sich somit innerhalb der Datenstruktur befinden. Analog dazu wird der `bottomPaddingEndIndex` in Z. 13-17 berechnet, indem zu allen drei Werten der übergebenen Endkoordinate `stencilSize` hinzuaddiert wird und zusätzlich sichergestellt wird, dass die resultierenden x-, y- und z-Werte maximal `width - 1`, `height - 1` und `depth - 1` betragen. Am Ende wird mittels `cudaMalloc` jeweils ein Speicherabschnitt für den oberen und unteren Puffer allokiert.

```
1  MSL_USERFUNC T operator() (int x, int y, int z) const {
2      if (x < 0 || y < 0 || z < 0 || x >= width || y >= height || z >= depth) {
3          return neutralValue;
4      }
5      int index = coordinateToIndex(x, y, z);
6      if (index >= dataStartIndex) {
7          if (index > dataEndIndex) {
```

```
8               return bottomPadding[index - dataEndIndex - 1];
9           } else {
10              return data[index - dataStartIndex];
11          }
12      } else {
13          return topPadding[index - topPaddingStartIndex];
14      }
15  }
```

Programmausschnitt 3.6: `operator()` der Klasse `PLCube`

Der `operator()`, abgebildet in Programmausschnitt 3.6, wird benutzt, um für eine gegebene Koordinate den entsprechenden Zellenwert des `DistributedCube` abzufragen. Für den Fall, dass die Koordinaten außerhalb der Datenstruktur liegen, wird in Z. 3 die dafür spezifizierte `neutralValue` übergeben. Wenn die Koordinaten innerhalb der Datenstruktur liegen, wird in Z. 5 der Index der geforderten Zelle berechnet, und anschließend überprüft, ob sich die Zelle im primären Datenbereich, oder im oberen oder unteren Datenpuffer befindet. In jedem Fall wird durch Abziehen des Startindexes des jeweiligen Datenbereiches vom Index der Zelle der relative Index innerhalb des Datenbereiches berechnet und so der Wert der gewünschten Zelle zurückgegeben.

```
1  MSL_USERFUNC int3 indexToCoordinate(int i) const {
2      int x = i % width;
3      i /= width;
4      int y = i % height;
5      int z = i / height;
6      return {x, y, z};
7  }
```

Programmausschnitt 3.7: Funktion `indexToCoordinate` der Klasse `PLCube`

Die Funktion `indexToCoordinate` berechnet die 3D-Koordinate für einen gegebenen Index einer Datenstruktur, also die Inverse von `coordinateToIndex`. Da $\mathrm{i}(x, y, z) = (z \cdot \texttt{height} + y) \cdot \texttt{width} + x$, gilt

$$
\begin{aligned}
x(i) &:= i \mod \texttt{width} \\
y(i) &:= \left\lfloor \frac{i}{\texttt{width}} \right\rfloor \mod \texttt{height} \\
z(i) &:= \left\lfloor \frac{i}{\texttt{width} \cdot \texttt{height}} \right\rfloor
\end{aligned}
$$

```
1  [[nodiscard]] inline int getTopPaddingElements() const {
2      return dataStartIndex - topPaddingStartIndex;
3  }
4
5  [[nodiscard]] inline int getBottomPaddingElements() const {
6      return bottomPaddingEndIndex - dataEndIndex;
```

```
}
```

Programmausschnitt 3.8: Funktionen `getTopPaddingElements` und `getBottomPaddingElements` der Klasse `PLCube`

Zusätzlich enthält `PLCube` die Hilfsmethoden `getTopPaddingElements` und `getBottomPaddingElements`, welche die Größe des oberen und unteren Datenpuffers durch Subtraktion der jeweiligen begrenzenden Indizes berechnen.

3.3.2. MapStencil-Implementierung

Für die Umsetzung des `MapStencil`-Skeletts erhält die Klasse `DC` zwei neue Variablen:

- `std::vector<PLCube<T>> plCubes`: ein `PLCube` für jede GPU, der dem Funktor übergeben wird, um dort Zugriff auf die Werte der Datenstruktur zu geben.
- `int supportedStencilSize`: die Stencilgröße, für die die `plCubes` initialisiert wurden. Je größer der Stencil, desto größer müssen die Datenpuffer sein.

Die Funktion `mapStencil` (Programmausschnitt 3.9) ist ein Template, welches zusätzlich zu dem durch die Klasse `DistributedCube` vorgegebenen Templateargument `T` auch das Argument `DCMapStencilFunctor<T> f` erwartet. Dieses ist, wie in Programmausschnitt 3.3 gezeigt, eine Funktion, welche einen `PLCube<T>` und 3D-Koordinaten als Parameter erwartet, und dafür einen Wert vom Typen `T` zurückgibt. Als Funktionsparameter müssen an `mapStencil` die Datenstruktur, in welche das Ergebnis der Operation geschrieben werden soll, die Größe des Stencils und der neutrale Wert vom Typ `T` übergeben werden.

```
template<typename T>
template<msl::DCMapStencilFunctor<T> f>
void msl::DC<T>::mapStencil(
        msl::DC<T> &result, size_t stencilSize, T neutralValue) {
#ifdef __CUDACC__
    this->updateDevice();
    syncPLCubes(stencilSize, neutralValue);
    msl::syncStreams();
    Muesli::start_time = MPI_Wtime();  // For performance testing.
    for (int i = 0; i < this->ng; i++) {
        cudaSetDevice(i);
        dim3 dimBlock(Muesli::threads_per_block);
        dim3 dimGrid((this->plans[i].size + dimBlock.x - 1) / dimBlock.x);
        PLCube<T> cube = this->plCubes[i];
        detail::mapStencilKernelDC<T, f><<<
            dimGrid, dimBlock, 0, Muesli::streams[i]>>>(
            result.plans[i].d_Data, cube, result.plans[i].size);
    }
    msl::syncStreams();
    result.setCpuMemoryInSync(false);
#endif
```

```
22 }
```

Programmausschnitt 3.9: Funktion `mapStencil` der Klasse `DC`

In der Funktion `mapStencil` wird in Z. 6 mithilfe von `updateDevice()` zuerst sichergestellt, dass die Daten auf der GPU aktualisiert werden, falls im Arbeitsspeicher aktuellere Daten vorliegen. Als nächstes wird die Methode `syncPLCubes` aufgerufen, welche gegebenenfalls den Datenpuffer nach oben und unten erstellt und aktualisiert.

```
1  void syncPLCubes(int stencilSize, T neutralValue) {
2      if (stencilSize > supportedStencilSize) {
3          freePLCubes();
4          initPLCubes(stencilSize, neutralValue);
5      }
6      for(int i = 1; i < this->ng; i++) {
7          size_t bottomPaddingSize =
8                  plCubes[i - 1].getBottomPaddingElements() * sizeof(T);
9          gpuErrchk(cudaMemcpyAsync(
10             plCubes[i - 1].bottomPadding,
11             plCubes[i].data,
12             bottomPaddingSize,
13             cudaMemcpyDeviceToDevice,
14             Muesli::streams[i - 1]
15         ));
16
17         size_t topPaddingSize =
18                 plCubes[i].getTopPaddingElements() * sizeof(T);
19         gpuErrchk(cudaMemcpyAsync(
20             plCubes[i].topPadding,
21             plCubes[i - 1].data + (
22                 this->plans[i - 1].size - plCubes[i].getTopPaddingElements()
23             ),
24             topPaddingSize,
25             cudaMemcpyDeviceToDevice,
26             Muesli::streams[i]
27         ));
28     }
29     msl::syncStreams();
30 }
```

Programmausschnitt 3.10: Funktion `syncPLCubes` der Klasse `DC`

In `syncPLCubes` (Programmausschnitt 3.10) wird in Z. 2 durch einen Vergleich des übergebenen Parameters `stencilSize` mit der Klassenvariable `supportedStencilSize` überprüft, ob die `plCubes` mit einer passenden Puffergröße erzeugt wurden. Ist dies nicht der Fall, so werden eventuell bereits bestehende `plCubes` mithilfe von `freePLCubes` gelöscht und anschließend mit `initPLCubes` (Programmausschnitt 3.11) die `plCubes` mit der richtigen Größe erzeugt.

```
1  void initPLCubes(int stencilSize, T neutralValue) {
```

```
2       plCubes = std::vector<PLCube<T>>();
3       plCubes.reserve(this->ng);
4       for (int i = 0; i < this->ng; i++) {
5           plCubes.push_back(PLCube<T>(
6                   this->ncol, this->nrow, this->depth,
7                   {
8                       this->plans[i].firstCol,
9                       this->plans[i].firstRow,
10                      this->plans[i].firstDepth
11                  },
12                  {
13                      this->plans[i].lastCol,
14                      this->plans[i].lastRow,
15                      this->plans[i].lastDepth
16                  },
17                  i,
18                  stencilSize,
19                  neutralValue,
20                  this->plans[i].d_Data
21          ));
22      }
23      supportedStencilSize = stencilSize;
24  }
```

Programmausschnitt 3.11: Funktion `initPLCubes` der Klasse `DC`

Als nächstes werden in der Funktion `syncPLCubes` die Datenpuffer der GPUs aktualisiert. Dazu werden mit der CUDA-Funktion `cudaMemcpyAsync` die ersten `bottomPaddingSize` Bytes des primären Datenbereiches jeder GPU in den unteren Datenpuffer der vorherigen GPU kopiert. Genau so werden die letzten `topPaddingSize` Bytes des primären Datenbereiches jeder GPU in den oberen Datenpuffer der nächsten GPU kopiert. Dies passiert asynchron; es werden zunächst alle Kopieraufträge angelegt, um am Ende mithilfe von `msl::syncStreams()` auf den Abschluss zu warten. Dieses Verfahren erlaubt, dass die GPUs Daten parallel kopieren können.

Der nächste Schritt ist der Kernelaufruf in der Funktion `mapStencil` (Programmausschnitt 3.9). Es müssen `this->plans[i].size` Kernelthreads in einer Blockgröße von `Muesli::threads_per_block` gestartet werden. `Muesli::threads_per_block` beträgt normalerweise 1024. Um die Anzahl der nötigen Blöcke zu ermitteln, muss $\lceil$ `plans[i].size / dimBlock.x` $\rceil$ berechnet werden, was in Z. 13 durch die Ganzzahldivision `(this-> plans[i].size + dimBlock.x - 1) / dimBlock.x` geschieht. In Z. 15 wird die Kernelfunktion `mapStencilKernelDC` aufgerufen und ein Pointer, in den die Ergebnisse geschrieben werden sollen, der `PLCube` und die Anzahl der zu bearbeitenden Elemente übergeben. Da, falls die Blockgröße die Anzahl der benötigten Threads nicht gerade teilt, der letzte Block Threads enthält, deren Index über die zu bearbeitenden Elemente hinausgeht, wird in diesem Fall in der Kernelfunktion `mapStencilKernelDC` (Programmausschnitt 3.12) nach Berechnung des Threadindexes direkt abgebrochen (Z. 4-6). Ansonsten wird die globale Koordinate der zu bearbeitenden Zelle berechnet (Z. 7) und der per Templateargument spezifizierte Funktor mit dem übergebenen `PLCube` und

den errechneten Koordinatenwerten aufgerufen. Das Ergebnis des Funktors wird dann an die korrekte Position des Zielarrays geschrieben.

```
1  template <typename T, msl::DCMapStencilFunctor<T> f>
2  __global__ void mapStencilKernelDC(T *out, const PLCube<T> in, unsigned int
       size) {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      if (i >= size) {
5          return;
6      }
7      int3 coords = in.indexToCoordinate(in.dataStartIndex + i);
8      T v = f(in, coords.x, coords.y, coords.z);
9      out[i] = v;
10 }
```

Programmausschnitt 3.12: Funktion `mapStencilKernelDC`

In der Funktion `mapStencil` (Programmausschnitt 3.9) in Z. 19 wird mithilfe der Methode `msl::syncStreams()` auf die Beendigung der Kernelfunktionen gewartet. Zum Schluss wird in Z. 20 in der Zieldatenstruktur markiert, dass sich die aktuellen Daten auf den GPUs befinden.

3.4. Implementierung der Beispielanwendung LBM-Simulation in Muesli

Als Beispielanwendung wird die Lattice-Boltzmann-Methode mit dem D3Q19-Gitter implementiert. In der Literatur wird bei der LBM-Simulation im Regelfall der Kollisionsschritt vor dem Strömungsschritt aufgelistet [16, S. 61], [18, S. 169]. Da sich die beiden Schritte in einer kontinuierlichen Simulation abwechseln und die Reihenfolge in vielen Anwendungen, wie auch dieser Arbeit, nicht wichtig ist [16, S. 224], wird bei dieser Implementierung die Abfolge der Schritte invertiert, um zunächst den Strömungsschritt und dann den Kollisionsschritt effizient mit einem `mapStencil`-Aufruf ausführen zu können.

In der LBM-Simulation sollen gewisse Zellen besonders behandelt werden, sodass zum Beispiel ausgewählte Zellen als „blockiert“ markiert werden können, um den Luftfluss um statische Gegenstände zu simulieren. Außerdem wird in dieser Simulation die Verteilungsfunktion der Randzellen konstant gehalten, um einen gleichbleibenden Luftstrom zu erzeugen. Um diese Informationen zu speichern, ohne zusätzlichen Speicherplatz für jede Zelle zu benötigen, wird ein Trick angewendet: Der Wert von f_0, also die Menge der Luftteilchen, die in der Zelle bleiben werden, wird für besonders behandelte Zellen nicht benötigt. Dementsprechend werden besondere Zellen mit einem `Not a Number (NaN)` repräsentierenden Gleitkommawert in f_0 markiert. Laut dem Standard IEEE-754 wird jede Gleitkommazahl mit einem maximalen Exponenten und einer von 0 verschiedenen Mantisse als `NaN` gedeutet [19, Kap. 3.4]. So wird für besonders zu behandelnde Zellen das hochwertigste Bit der Mantisse von f_0 gesetzt, damit die Zahl definitiv als `NaN` verstanden wird. Die restlichen Bits der Mantisse können dann frei benutzt werden,

um andere Daten zu speichern. Im Code werden Bitmasken und ein `struct` mit Bitfields definiert, um auf diese Informationen möglichst einfach zugreifen zu können (siehe Programmausschnitt 3.13).

```
const int FLAG_OBSTACLE = 1 << 0;
const int FLAG_KEEP_VELOCITY = 1 << 1;

typedef struct {
    unsigned int mantissa : 23;
    unsigned int exponent : 8;
    unsigned int sign : 1;
} floatparts;
```

Programmausschnitt 3.13: Definitionen, um auf Informationen in der Mantisse von `NaN`-Gleitkommazahlen zuzugreifen

Der Typ der Daten, die für jede Zelle gespeichert werden, ist `array<float, Q>`, wobei `Q` die Konstante 19, die Anzahl der von jeder Zelle erreichbaren anderen Zellen, ist. Für den Typ der Zellendaten wurde ein Alias mit dem Namen `cell_t` angelegt. Da CUDA keine Implementierung von `std::array` mitliefert, wurde eine rudimentäre Version selbst implementiert (siehe Programmausschnitt A.2 im Anhang). Ebenfalls wurde ein dreidimensionaler Vektor `vec3` mit Vektoraddition, skalarer Multiplikation und Skalarprodukt implementiert (siehe Programmausschnitt A.3 im Anhang) und der Typenalias `vec3f` für `vec3<float>` festgelegt.

```
MSL_USERFUNC cell_t update(const PLCube<cell_t> &plCube, int x, int y, int z)
    {
    cell_t cell = plCube(x, y, z);

    auto* parts = (floatparts*) &cell[0];

    if (parts->exponent == 255) {
        if (parts->mantissa & FLAG_KEEP_VELOCITY) {
            return cell;
        }
    }

    // Streaming.
    for (int i = 1; i < Q; i++) {
        int sx = x + (int) offsets[i].x;
        int sy = y + (int) offsets[i].y;
        int sz = z + (int) offsets[i].z;
        cell[i] = plCube(sx, sy, sz)[i];
    }

    // Collision.
    if (parts->exponent == 255) {
        if (parts->mantissa & FLAG_OBSTACLE) {
            cell_t cell2 = cell;
            for (size_t i = 1; i < Q; i++) {
```

```
25                  cell[i] = cell2[opposite[i]];
26              }
27          }
28          return cell;
29      }
30      float p = 0;
31      vec3f vp {0, 0, 0};
32      for (size_t i = 0; i < Q; i++) {
33          p += cell[i];
34          vp += offsets[i] * cellwidth * cell[i];
35      }
36      vec3f v = p == 0 ? vp : vp * (1 / p);
37
38      for (size_t i = 0; i < Q; i++) {
39          cell[i] = cell[i] + deltaT / tau * (feq(i, p, v) - cell[i]);
40      }
41      return cell;
42  }
```

Programmausschnitt 3.14: Funktor des `mapStencil`-Skeletts für die LBM-Simulation.

Der an das `mapStencil`-Skelett übergebene Funktor ist in Programmabschnitt 3.14 abgedruckt. Zunächst wird mit dem übergebenen `plCube` der Inhalt der Zelle an den angegebenen Koordinaten in der Variable `cell` gespeichert und ein `floatparts`-Pointer zum ersten Wert der Zelle angelegt, sodass auf die Bestandteile der Gleitkommazahl zugegriffen werden kann. Dieser Pointer wird benutzt, um in Zeile 6 zu überprüfen, ob die Zahl über einen Exponenten von 255 verfügt, und die sie somit `NaN` ist. Falls dies zutrifft, wird in dem Fall, dass in der Mantisse das von der `FLAG_KEEP_VELOCITY`-Bitmaske selektierte Bit gesetzt ist, der ursprüngliche Zellenwert direkt zurückgegeben, um so die Verteilungsfunktion der Zelle konstant zu halten.

In der in Z. 13 beginnenden `for`-Schleife wird der Strömungsschritt behandelt. Da f_0 den Anteil der Luftteilchen angibt, die in der Zelle verweilen, verändert sich dieser Wert im Strömungsschritt nicht. Somit fängt die Schleife bei `i = 1` an. Die konstante Variable `array<vec3f, Q> offsets` gibt für jeden der 19 Nachbarzellen die Differenz der Koordinaten zur Ursprungszelle an. Mit dieser wird mithilfe des `plCube` der `i`-te Wert der Urspungszelle auf den `i`-ten Wert des `i`-ten Nachbars gesetzt.

Falls f_0 = `NaN` und das Bit gesetzt ist, das von der `FLAG_OBSTACLE`-Bitmaske selektiert wird, so werden alle Luftströme in dieser Zelle reflektiert, um ein Hindernis zu simulieren. Dazu wird das `array<int, Q> opposite` benutzt, welches für jeden Richtungsindex den Index der gegenüberliegenden Richtung angibt, um jeder Komponente der Verteilungsfunktion f den Wert der gegenüberliegenden Komponente zuzuweisen. Für die Hindernis-Zelle wird dann die neu berechnete reflektierte Verteilungsfunktion zurückgegeben.

Ansonsten wird die Kollision der Luftströme mithilfe der diskretisierten Boltzmann-Gleichung, wie in Kapitel 2.7 erläutert, mit der Hilfsfunktion `feq` (Programmausschnitt 3.15) berechnet.

```
MSL_USERFUNC inline float feq(size_t i, float p, const vec3f& v) {
    float wi = wis[i];
    float c = cellwidth;
    float dot = offsets[i] * c * v;
    return wi * p * (1 + (1 / (c * c)) * (3 * dot + (9 / (2 * c * c)) * dot *
     dot - (3.f / 2) * (v * v)));
}
```

Programmausschnitt 3.15: Gleichgewichtsfunktion `feq` für die LBM-Simulation

Die Methode `gassimulation_test` (Programmausschnitt 3.16) misst die Laufzeit der Beispielanwendung. Es werden zunächst zwei `DC`-Datenstrukturen der übergebenen Größe erzeugt, und die erste mit Daten initialisiert[7]. Als nächstes werden Pointer zu den beiden Datenstrukturen erstellt, welche im Folgenden benutzt werden. Dann wird die gewünschte Anzahl an Iterationen ausgeführt: In jeder Iteration wird die `mapStencil`-Methode des `DistributedCube` aufgerufen, auf den der erste Pointer zeigt. Der übergebene Funktor ist die `update`-Funktion und als Zieldatenstruktur wird der Würfel des zweiten Pointers angegeben. Der Stencilradius beträgt eins und der neutrale Wert ist `{}`, also ein mit Nullen gefülltes Array. Es wird die gesamte Laufzeit des Skeletts und, mithilfe der in `mapStencil` gesetzten Variable `Muesli::start_time`, die Laufzeit ohne die anfängliche Speichersynchronisation gemessen und ausgegeben. Da nun der `DistributedCube` des zweiten Pointers die aktuelleren Daten enthält, werden am Ende die beiden Pointer getauscht, sodass der erste Pointer wieder auf die Datenstruktur mit den neuen Daten zeigt und die nächste Iteration starten kann.

```
void gassimulation_test(vec3<int> dimension, int iterations, const std::
    string &importFile, const std::string &exportFile) {
    size = dimension;

    DC<cell_t> dc(size.x, size.y, size.z);

    // Data initialization code replaced; Full code in Appendix.
    fillWithData(dc);

    DC<cell_t> dc2(size.x, size.y, size.z);

    // Pointers for swapping.
    DC<cell_t> *dcp1 = &dc;
    DC<cell_t> *dcp2 = &dc2;

    for (int i = 0; i < iterations; i++) {

        double time = MPI_Wtime();

        dcp1->mapStencil<update>(*dcp2, 1, {});

        double endTime = MPI_Wtime();
        double onlyKernelTime = endTime - Muesli::start_time;
```

[7] siehe Anhang, Programmausschnitt A.4 ab Z. 176.

```
23          double totalTime = endTime - time;
24
25          std::cout << totalTime << " / " << onlyKernelTime << std::endl;
26
27          std::swap(dcp1, dcp2);
28      }
29
30      // Export code removed.
31  }
```

Programmausschnitt 3.16: Funktion mit Aufrufs des `mapStencil`-Skeletts zur LBM-Simulation

3.5. Native Referenzimplementierung

Die Referenzimplementierung muss viele Aspekte der verteilten Programmierung neu entwickeln, die Muesli ohne Mehraufwand für den Benutzer bereitstellt. So berechnet die Funktion `initSimulation` (Programmausschnitt 3.16) zunächst die Verteilung der Daten auf die angegebene Anzahl der GPUs. Zur einfacheren Handhabung werden die Daten nur immer in ganzen x-y-Schichten entlang der z-Achse aufgeteilt. Da die Anzahl der GPUs die z-Dimension, also die Anzahl der Schichten, nicht teilen muss, wird neben der Anzahl der Schichten pro GPU auch die Anzahl der Schichten berechnet, die übrig bleiben. Außerdem werden zwei Arrays für die Datenstruktur im Arbeitsspeicher erstellt. Dadurch, dass die Referenzimplementierung speziell für die `mapStencil`-Anwendung geschrieben wird, kann auf der GPU für den oberen Puffer, den Hauptbereich und den unteren Puffer ein kontinuierlicher Speicherbereich belegt werden. Für jede GPU wird ein `struct` vom Typ `gpu_t` erstellt und mit folgenden Daten initialisiert:

- `int device`: Der Index der GPU.
- `size_t mainGlobalIndex`: Der globale Zellenindex, an dem der Hauptspeicherbereich anfängt.
- `size_t mainLayers`: Die Anzahl der Schichten, die der Hauptspeicherbereich umfasst.
- `size_t mainOffset`: Die Anzahl der Elemente, die der Hauptspeicherbereich vom Anfang des gesamten Speicherbereiches entfernt liegt (also die Anzahl der Elemente im oberen Puffer).
- `size_t bottomPaddingOffset`: Die Anzahl der Elemente, die der untere Puffer vom Anfang des gesamten Speicherbereiches entfernt liegt.
- `cell_t *data1, *data2`: Jeweils zwei komplette auf der GPU liegende Speicherbereiche für oberen und unteren Puffer und Hauptspeicherbereich, sodass sie wie in der Muesli-Anwendung abwechselnd als Quell- und als Zielspeicherbereich genutzt werden können.

Anschließend werden die Speicherbereiche mit Daten gefüllt[8] und mithilfe der Funktion textttsyncStreams auf die Fertigstellung der Speichertransfers zu den GPUs gewartet.

```
void initSimulation(size_t xdim, size_t ydim, size_t zdim, size_t gpus, const
     std::string &importFile) {
    size = {xdim, ydim, zdim};
    cells = xdim * ydim * zdim;

    size_t layersPerGpu = zdim / gpus;
    size_t remainder = zdim - layersPerGpu * gpus;
    elementsPerLayer = xdim * ydim;
    bytesPerLayer = elementsPerLayer * sizeof(cell_t);

    u1 = new cell_t[cells];
    u2 = new cell_t[cells];

    streams = new cudaStream_t[gpus];

    size_t currentLayer = 0;
    gpuStructs = std::vector<gpu_t>();
    gpuStructs.reserve(gpus);

    for (int i = 0; i < gpus; i++) {
        gpuErrchk(cudaSetDevice(i));
        gpuErrchk(cudaStreamCreate(&streams[i]));
        gpu_t gpu{};
        gpu.device = i;
        int toppaddinglayers = i > 0 ? 1 : 0;
        int bottompaddinglayers = i < gpus - 1 ? 1 : 0;
        gpu.mainGlobalIndex = currentLayer * elementsPerLayer;
        gpu.mainLayers = layersPerGpu + (i < remainder ? 1 : 0);
        gpu.mainOffset = toppaddinglayers * elementsPerLayer;
        gpu.bottomPaddingOffset = gpu.mainOffset + gpu.mainLayers *
    elementsPerLayer;

        currentLayer += gpu.mainLayers;

        gpuErrchk(cudaMalloc(&gpu.data1, (gpu.mainLayers + toppaddinglayers +
     bottompaddinglayers) * bytesPerLayer));
        gpuErrchk(cudaMalloc(&gpu.data2, (gpu.mainLayers + toppaddinglayers +
     bottompaddinglayers) * bytesPerLayer));
        gpuStructs.push_back(gpu);
    }

    initData(); // Data initialization removed.

    syncStreams();
}
```

Programmausschnitt 3.17: Funktion `initSimulation` der nativen Referenzimplementierung

[8] siehe Anhang, Programmausschnitt A.5 ab Z. 279.

Zur Durchführung eines Simulationsschritts werden in der Funktion `simulateStep` (Programmausschnitt 3.18) zunächst, ähnlich wie in der Muesli-Funktion `syncPLCubes`, die oberen und unteren Puffer synchronisiert (Z. 2-17). Als nächstes wird, vor dem Aufrufen der Kernelfunktion `update`, für jede GPU die Variable `time_split` gesetzt, um die Zeit des reinen Kernelaufrufs zu messen.

```
1  void simulateStep() {
2      for (int i = 1; i < gpuStructs.size(); i++) {
3          // Copy bottom padding for i - 1
4          gpuErrchk(cudaMemcpyAsync(
5          &gpuStructs[i - 1].data1[gpuStructs[i - 1].bottomPaddingOffset],
6          &gpuStructs[i].data1[gpuStructs[i].mainOffset],
7          bytesPerLayer, cudaMemcpyDefault, streams[i - 1]
8          ));
9          // Copy top padding for i
10         gpuErrchk(cudaMemcpyAsync(
11         gpuStructs[i].data1,
12         &gpuStructs[i - 1].data1[gpuStructs[i - 1].bottomPaddingOffset -
       elementsPerLayer],
13         bytesPerLayer, cudaMemcpyDefault, streams[i]
14         ));
15     }
16
17     syncStreams();
18     time_split = timer.get();
19     for (auto &gpu : gpuStructs) {
20         cudaSetDevice(gpu.device);
21         dim3 threadsPerBlock(1024);
22         size_t worksize = size.x * size.y * gpu.mainLayers;
23         dim3 numBlocks(
24         (worksize + threadsPerBlock.x - 1) / threadsPerBlock.x
25         );
26         update<<<numBlocks, threadsPerBlock, 0, streams[gpu.device]>>>(
27         gpu.data2, gpu.data1, worksize, size, gpu.mainOffset /
       elementsPerLayer
28         );
29         std::swap(gpu.data1, gpu.data2); // data1 is always pointing to up-to
       -date buffer.
30     }
31     std::swap(u1, u2);
32     syncStreams();
33 }
```

Programmausschnitt 3.18: Funktion `simulateStep` der nativen Referenzimplementierung

In der Kernelfunktion `update` (Programmausschnitt 3.19) muss zunächst sichergegangen werden, dass der aktuelle Thread sich an einem validen Index befindet (Z. 3-5). Außerdem müssen die Koordinaten der Zelle (Z. 6-8) und der relative Index der Zelle innerhalb der auf der GPU liegenden Arrays `dst` und `src` berechnet werden. Der folgende Code ist größtenteils analog zur Muesli-Umsetzung, abgesehen davon, dass im

Strömungsschritt in Z. 21-23 manuell geprüft werden muss, ob die Nachbarzellen tatsächlich existieren. Außerdem wird in Z. 24 direkt in das Zielarray geschrieben und mithilfe der eigens implementierten Hilfsfunktion `pack`[9], welche den Index für angegebene Koordinaten ausrechnet, auf das Quellarray zugegriffen. Der Kollisionsschritt ist als eigene Methode `collisionStep`[10] analog zu Muesli implementiert. Der Funktion wird in Z. 26 die Zelle `dst[index]` per Referenz übergeben, sodass diese in der Methode bearbeitet werden kann.

```
1  __global__ void update(cell_t *dst, cell_t *src, const size_t worksize, const
       vec3<size_t> globalsize, const size_t zoffset) {
2      size_t i = blockIdx.x * blockDim.x + threadIdx.x;
3      if (i >= worksize) {
4          return;
5      }
6      size_t x = i % globalsize.x;
7      size_t y = (i / globalsize.x) % globalsize.y;
8      size_t z = (i / (globalsize.x * globalsize.y)) + zoffset;
9      size_t index = i + zoffset * globalsize.x * globalsize.y;
10     floatparts* parts = (floatparts*) &src[index][0];
11     if (parts->exponent == 255) {
12         if (parts->mantissa & FLAG_KEEP_VELOCITY) {
13             dst[index] = src[index];
14             return;
15         }
16     }
17     for (int i = 0; i < Q; i++) {
18         int sx = x + (int) offsets[i].x;
19         int sy = y + (int) offsets[i].y;
20         int sz = z + (int) offsets[i].z;
21         if (sx < 0 || sy < 0 || sz < 0 || sx >= globalsize.x || sy >=
       globalsize.y || sz >= globalsize.z) {
22             continue;
23         }
24         dst[index][i] = src[pack(globalsize.x, globalsize.y, globalsize.z, sx
       , sy, sz)][i];
25     }
26     collisionStep(dst[index]);
27 }
```

Programmausschnitt 3.19: Kernelfunktion `update` der nativen Referenzimplementierung

Nach dem Kernelaufruf wird in `simulateStep` (Programmausschnitt 3.18) in Z. 29 und 31 sowohl die Pointer der GPU-Speicherbereiche, als auch die Pointer der Arrays im Arbeitsspeicher getauscht.

[9] siehe Anhang, Programmausschnitt A.5, ab Z. 103
[10] siehe Anhang, Programmausschnitt A.5, ab Z. 114

3.6. Optimierung

3.6.1. Muesli

Der `operator()` des `PLCube<T>` kann, statt einem Objekt vom Typ `T`, auch eine konstante Referenz zum Objekt zurückgeben. Mit dieser Optimierung muss das übergebene Objekt nicht kopiert werden, was insbesondere bei Objekten mit vergrößertem Speicherbedarf zu einer Leistungsverbesserung führen kann. Zur Implementierung der Optimierung muss lediglich die Methodensignatur von Z. 1 zu Z. 2 des Programmausschnitts 3.20 geändert werden.

```
1 MSL_USERFUNC T operator() (int x, int y, int z) const           // vorher
2 MSL_USERFUNC const T& operator() (int x, int y, int z) const    // nachher
```

Programmausschnitt 3.20: Veränderte der Methodensignatur des `operator()` des `PLCube<T>`

3.6.2. Nativ

In der `update`-Funktion der nativen Umsetzung der LBM-Simulation wird vielmalig in das `dst`-Array geschrieben; sowohl in Z. 24 in der `update`-Funktion selbst, als auch in der Funktion `collisionStep`, werden jeweils Komponentenweise die Werte der Zelle gesetzt. Hierbei eignet es sich, stattdessen in eine lokale Variable zu schreiben, welche in einem Register mit geringerer Latenz liegt und erst am Ende einmalig den Wert in das `dst`-Array zu kopieren. Also wird eine lokale Variable `cell_t dest = src[index]` definiert und auf diese, statt auf `dst[index]` zugegriffen. Da `dest[0]` nun schon den Wert von `src[index][0]` besitzt, kann der Strömungsschritt mit `i = 1` starten. Am Ende werden mithilfe von `dst[index] = dest` die Änderungen in das Zielarray geschrieben (vgl. Anhang, Programmausschnitt A.5 Z. 148-170).

4. Leistungsanalyse

Zur Analyse der Performance von Muesli und der nativen Referenzimplementierung wird die Laufzeit von 200 Iterationen der LBM-Simulation gemessen. Die Messungen wurden in dem Hochleistungscluster PALMA II[11] der Universität Münster auf drei unterschiedlichen Rechnerknoten ausgeführt:

- AMD EPYC 7513 CPU mit 8x Nvidia GeForce RTX 2080 Ti mit jeweils 11 GB GDDR6
- AMD EPYC 7343 CPU mit 4x Nvidia Titan RTX mit jeweils 24 GB GDDR6
- AMD EPYC 7343 CPU mit 8x Nvidia A100 mit jeweils 80 GB HBM

Dabei wird die Anzahl der GPUs, die zur Berechnung benutzt werden, sowie die Größe der Datenstruktur, mit der die Berechnungen ausgeführt werden, variiert. Als Größe der Datenstruktur wird immer ein Würfel gewählt, sodass die Seitenlängen des `DistributedCube` in allen drei Dimensionen gleich sind. Dementsprechend wird im Folgenden die Größe der Datenstrukturen immer durch die Seitenlänge angegeben. Die maximal mögliche Größe ist dabei begrenzt durch den Speicher der Grafikkarten. Jede Zelle `cell_t` nimmt 19 32-Bit-Gleitkommazahlen, also 76 Bytes in Anspruch. Da eine Quell- und eine Zieldatenstruktur benötigt werden, können also, abhängig von der Speichergröße in GB, theoretisch Datenstrukturen mit einer Seitenlänge von bis zu

$$d(\text{gb}) := \sqrt[3]{\text{gb} \cdot \frac{2^{30}}{2 \cdot 76}}$$

behandelt werden. So ergibt sich für eine Grafikkarte als theoretische maximale Seitenlänge für die RTX 2080 Ti **426**, für die Titan RTX **553** und für die A100 **826**. Die Werte werden in der Realität nicht genau den theoretischen entsprechen; sie schaffen aber trotzdem einen guten Überblick über die Größenordnung der Grafikkarten. Wenn mehrere Grafikkarten zur Berechnung eingesetzt werden, können natürlich auch größere Datenstrukturen behandelt werden. Alle gemessenen Laufzeitdaten sind in Anhang B zu finden.

4.1. Vor der Optimierung

Bei dem Vergleich der Laufzeiten der LBM-Simulation mithilfe des `mapStencil`-Skeletts in Muesli und der nativen Referenzimplementierung mit einer Grafikkarte fällt zunächst auf, dass sich die Laufzeiten wie angenommen etwa kubisch zur Seitenlänge der Datenstrukturen entwickeln. Der Ausführungszeitraum des Muesli-Skeletts beträgt auf der Nvidia A100 das rund 1,34-Fache der nativen Implementierung. Entgegen der Erwartungen ist die Referenzimplementierung auf der Nvidia GeForce RTX 2080 Ti und der

[11] `https://www.uni-muenster.de/IT.Technik/Server/HPC.html`

Nvidia Titan RTX allerdings sogar minimal langsamer als die Muesli-Alternative (siehe Abbildung 4.1).

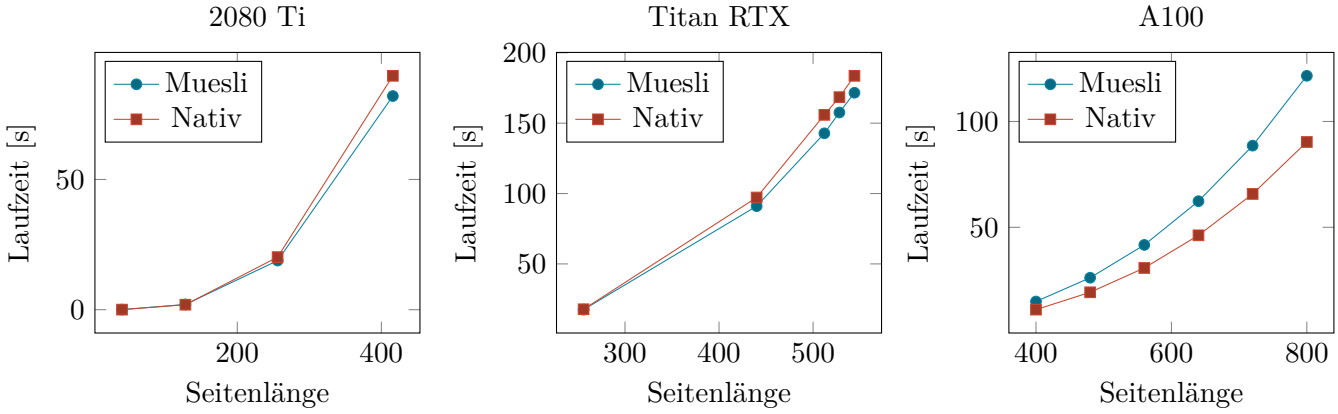


Abbildung 4.1: Laufzeit des `mapStencil`-Skeletts und der Referenzimplementierung ohne Optimierung nach Datenstrukturgröße auf jeweils einer GPU

4.2. Nach der Optimierung

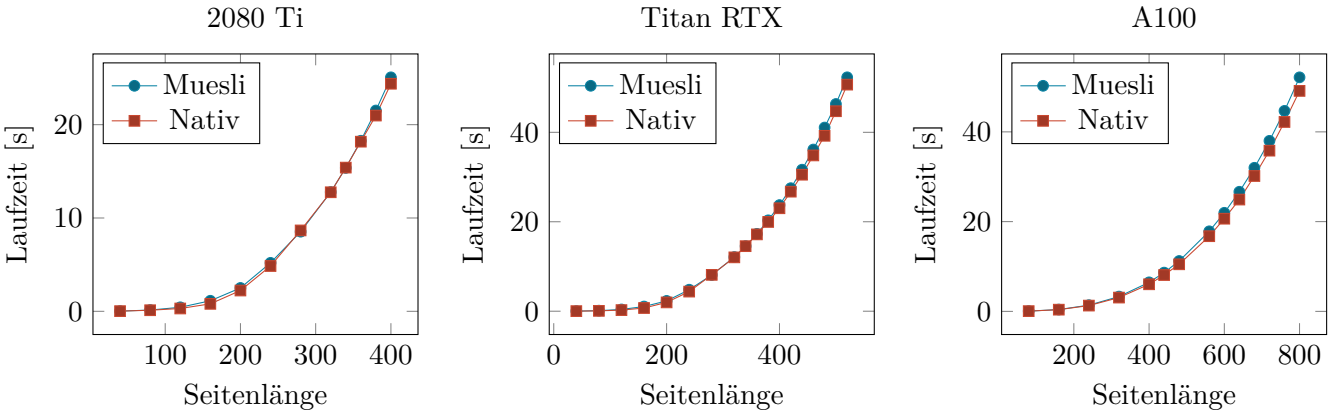


Abbildung 4.2: Laufzeit des `mapStencil`-Skeletts und der Referenzimplementierung mit Optimierung nach Datenstrukturgröße auf jeweils einer GPU

Beim Messen der Laufzeiten der optimierten Anwendungen (Abbildung 4.2) fällt bei allen drei Grafikkarten eine deutliche Verbesserung sowohl bei Muesli als auch bei der nativen Implementierung auf: Bei dem Beispiel eines Würfels mit Seitenlänge 440 auf der Nvidia Titan RTX reduziert sich die Laufzeit von Muesli von 91.03s um 65.24% auf 31.64s und bei der nativen Referenzimplementierung von 97.19s um 68.60% auf 30.52s. Bemerkenswert ist, dass nun auf allen Grafikkarten die Referenzimplementierung etwas

schneller als Muesli ausgeführt wird. Mit diesem Wissen lassen sich die Unterschiede zwischen den Grafikkarten bei den Messungen vor der Optimierung erklären: Die Verbesserung bei der nativen Implementierung entstand durch die Reduktion von Schreibzugriffen auf den GPU-Speicher. Die Nvidia A100 zeichnet sich durch einen sehr schnelle Speicheranbindung aus [20], sodass die vielen Schreibvorgänge der unoptimierten nativen Implementierung die Laufzeit auf dieser Grafikkarte weniger stark beeinflusst hat, als es bei den anderen beiden Grafikkarten der Fall war.

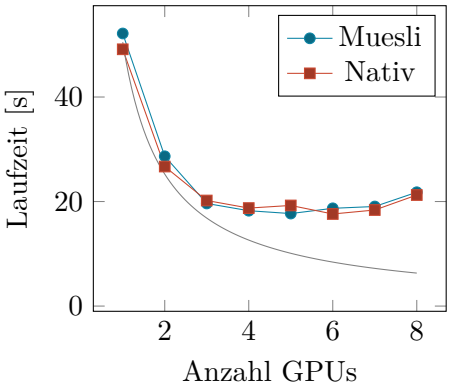


Abbildung 4.3: Laufzeit des `mapStencil`-Skeletts und der Referenzimplementierung mit Optimierung nach GPU-Anzahl mit einer Würfellänge von 800 auf der Nvidia A100

Zur Analyse der Leistung der Implementierungen auf mehreren GPUs sind in Abbildung 4.3 die Laufzeiten von Muesli und der nativen Implementierung in Abhängigkeit zur Anzahl der benutzten GPUs abgebildet. Theoretisch ist es möglich, dass bei der Verwendung von n GPUs die Laufzeiten auf das $1/n$-fache der Laufzeit einer GPU reduziert werden. Diese optimale Laufzeit wird in den Abbildung als graue Linie dargestellt.

Bei einer Würfellänge von 800 zeigt die Ausnutzung von bis zu drei Nvidia A100 deutliche Beschleunigungen: So wird die Laufzeit mit zwei GPUs fast halbiert ($\times 0.55$ bei Muesli, $\times 0.54$ bei der Referenzimplementierung). Mit drei GPUs beträgt die Laufzeit bei Muesli das 0.38-Fache, bei der Referenzimplementierung das 0.41-Fache der jeweiligen Laufzeit mit einer GPU. Mit vier GPUs wird bei beiden Anwendungen nur noch eine minimale Beschleunigung gegenüber drei GPUs erreicht. Ab sechs GPUs steigt die Laufzeit mit zunehmender GPU-Anzahl wieder an.

In Abbildung 4.4 sind zusätzlich zu den Laufzeiten aus Abbildung 4.3 auch die Dauer des der initialen Speichersynchronisation und die Dauer des reinen Kernelaufrufs aufgezeigt. Dabei ist ersichtlich, dass die Laufzeit des Kernels, sowohl bei Muesli als auch bei der Referenzimplementierung, nur minimal von der theoretisch optimalen Laufzeit abweicht. Die Zeit der Speichersynchronisation steigt mit zunehmender Anzahl an GPUs in etwa linear an. Dabei ist der Anstieg nicht konsistent, was auch eine Erklärung für die Tatsache ist, dass in Abbildung 4.3 die native Referenzimplementierung bei drei, vier und fünf

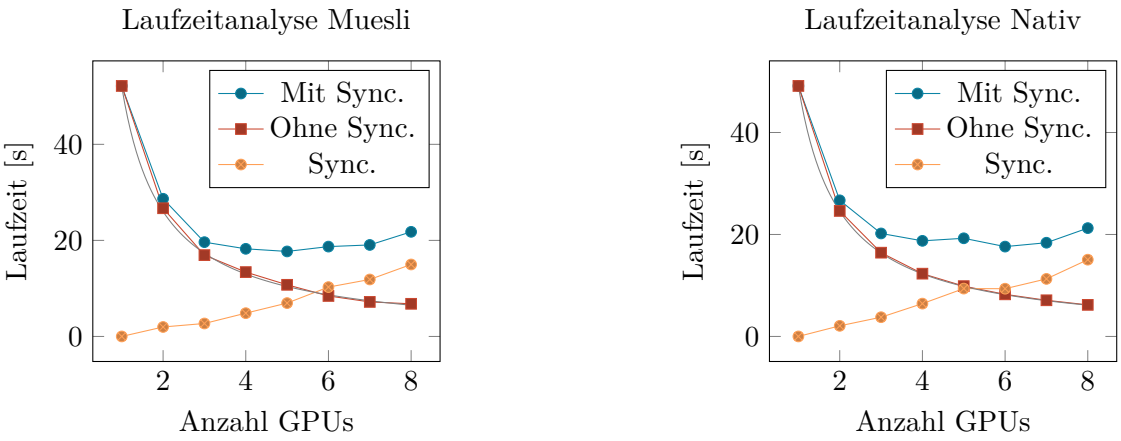


Abbildung 4.4: Aufspaltung der Laufzeiten aus Abbildung 4.3

GPUs etwas langsamer als die Implementierung mit Muesli läuft. Da sich die Laufzeit beider Kernel sich proportional zu $1/nGPUs$ verhält, also mit zunehmender Anzahl an GPUs immer weniger Laufzeit eingespart wird, und gleichzeitig mit jeder zusätzlichen GPU der Speicheraustausch in etwa konstant länger wird, gibt es für jede Kombination aus Grafikkarte und Datengröße ein Optimum, um die Berechnung schnellstmöglich auszuführen.

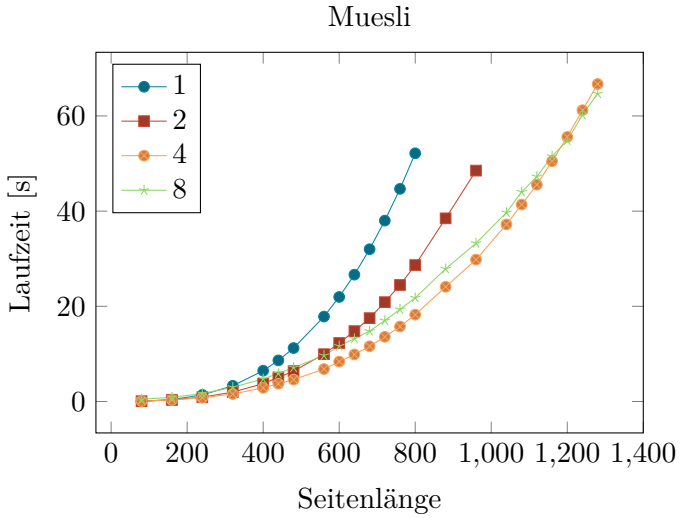


Abbildung 4.5: Laufzeit von Muesli abhängig von der Seitenlänge des Würfels mit verschiedenen Anzahlen an GPUs (Nvidia A100)

Dies ist auch in Abbildung 4.5 zu sehen, in denen die Laufzeiten der Muesli-Implementierung mit einer, zwei, vier und acht Nvidia A100 mit jeweils verschiedenen Datengrößen abgebildet ist. Die Berechnung mit vier GPUs ist für den Großteil der Seitenlängen der

Datenstrukturen die schnellste Variante. Die Berechnung mit acht GPUs ist bis zu einer Seitenlänge von 320 langsamer als die Berechnung mit einer GPUs und bis zu einer Seitenlänge von 560 langsamer als die Berechnung mit zwei GPUs. Ab einer Seitenlänge von 1200 ist sie aber schneller als die Berechnung mit vier GPUs.

5. Fazit

Das Ziel dieser Bachelorarbeit ist die Implementierung und Optimierung eines effizienten dreidimensionalen `mapStencil`-Skeletts in der Muenster Skeleton Library (Muesli) zur Entwicklung von Iterative Stencil Loop (ISL). Wie in Kapitel 3 gesehen, erleichtert die Benutzung des erstellten Skeletts die Entwicklung der Strömungssimulation mithilfe der Lattice-Boltzmann-Methode deutlich. Außerdem ist bei der Verwendung des Muesli-Skeletts kaum spezifisches Wissen der parallelen Programmierung nötig. Ebenfalls werden durch Anwendung des Skeletts viele mögliche Fehlerquellen eliminiert, sodass zum Beispiel der bei der initialen Version der nativen Referenzimplementierung durch häufige Speicherzugriffe begründete Leistungsverlust bei Benutzung von Muesli komplett ausgeschlossen werden kann.

Die Analyse in Kapitel 4 ergab, dass nach der Optimierung beider Anwendungen die in Muesli implementierte Simulation eine minimal längere Laufzeit gegenüber der nativen Referenzimplementierung besitzt. Dies ist durch den in Muesli gegebenen Mehraufwand des Aufrufs vom `operator()` des `PLCube` und durch die dortige, auf Grund der Partitionierung des Speichers in den oberen Puffer, Hauptspeicher und unteren Puffer benötigte, zusätzliche Verzweigung des Codes zu erklären.

In Zukunft könnte geforscht werden, wie die Kombinierung der drei Speicherbereiche auf jeder GPU technisch möglich wäre, sodass im `operator()` des `PLCube` weniger Unterscheidungen nötig sind, ohne dabei jedoch die Funktion der anderen Skelette von Muesli zu beeinflussen, für die jeweils nur der Hauptspeicherbereich wichtig ist. Insbesondere ist es interessant, ob diese oder andere eventuell mögliche Optimierungen an der Implementierung des `mapStencil`-Skeletts in Muesli dessen minimal längere Laufzeit gegenüber der nativen Referenzimplementierung noch weiter verkürzen können. Zusätzlich könnte versucht werden, das dreidimensionale `mapStencil`-Skelett auf andere Dimensionen zu verallgemeinern. Außerdem liegt der Fokus dieser Arbeit auf der Parallelisierung mittels CUDA. Eine zusätzliche Beschleunigung könnte erreicht werden, indem Teile der Berechnungen in mehreren Threads auf der CPU ausgeführt werden. Ebenfalls kann das Skelett weiterhin mittels MPI parallelisiert werden, wodurch die Berechnung auf viele Rechner verteilt werden kann.

Abschließend lässt sich sagen, dass, insbesondere für Entwickler mit wenig Erfahrung im Bereich der parallelen Programmierung, die Benutzung von Muesli eine gute Alternative bei der Implementierung von ISLs darstellt, ohne dabei große Leistungsverluste einbüßen zu müssen.

Literatur

[1] Riccardo Cattaneo u. a. „On How to Accelerate Iterative Stencil Loops: A Scalable Streaming-Based Approach“. In: *ACM Trans. Archit. Code Optim.* 12.4 (Dez. 2015) (siehe S. 1).

[2] Fabien Sanglard. *A history of Nvidia Stream Multiprocessor*. 2. Mai 2020. URL: `https://fabiensanglard.net/cuda/` (besucht am 11. 01. 2023) (siehe S. 2, 3).

[3] Nolan Goodnight. „GPU Computing“. In: *GPU Gems 3*. Addison-Wesley, Aug. 2007 (siehe S. 2).

[4] Michael Uelschen. *Software Engineering Paralleler Systeme: Grundlagen, Algorithmen, Programmierung*. Springer Vieweg, Juni 2019 (siehe S. 2, 3).

[5] Michael J. Flynn. „Very high-speed computing systems“. In: *Proceedings of the IEEE* 54.12 (Dez. 1966), S. 1901–1909 (siehe S. 3).

[6] TechPowerUp. *GeForce RTX 3060 Specs*. URL: `https://www.techpowerup.com/gpu-specs/geforce-rtx-3060.c3682` (besucht am 14. 01. 2023) (siehe S. 3).

[7] Intel. *Export Compliance Metrics*. 26. Okt. 2022. URL: `https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Core-Processors.pdf` (besucht am 14. 01. 2023) (siehe S. 3).

[8] Nvidia. *CUDA C++ Programming Guide*. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/` (besucht am 11. 01. 2023) (siehe S. 3, 5).

[9] Nvidia. *CUDA Runtime API - CUDA Toolkit Documentation*. URL: `https://docs.nvidia.com/cuda/archive/11.8.0/cuda-runtime-api/index.html` (besucht am 20. 01. 2023) (siehe S. 4).

[10] Murray Cole. „Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming“. In: *Parallel Computing* 30.3 (März 2004), S. 389–406 (siehe S. 5).

[11] Sergei Gorlatch und Herbert Kuchen. „Guest Editorial: High-Level Parallel Programming with Algorithmic Skeletons“. In: *International Journal of Parallel Programming* 46 (Mai 2017), S. 1–3 (siehe S. 5).

[12] Philipp Ciechanowicz. *Datenparallele algorithmische Skelette Erweiterungen und Anwendungen der Münster Skelettbibliothek Muesli*. 2010 (siehe S. 6).

[13] Steven Bell u. a. „Image Processing with Stencil Pipelines“. In: *Compiling Algorithms for Heterogeneous Systems*. Cham: Springer International Publishing, 2018, S. 27–31 (siehe S. 8).

[14] Robert Fisher u. a. *Mean Filter*. URL: `https://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm` (besucht am 13. 02. 2023) (siehe S. 9).

[15] Robert Fisher u. a. *Laplacian/Laplacian of Gaussian*. URL: `https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm` (besucht am 13. 02. 2023) (siehe S. 9).

[16] Timm Krüger u. a. *The Lattice Boltzmann Method - Principles and Practice*. Okt. 2016 (siehe S. 10, 21).

[17] Herbert Kuchen und Jörg Striegnitz. „Higher-Order Functions and Partial Applications for a C++ Skeleton Library“. In: *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*. Association for Computing Machinery, 2002, S. 122–130 (siehe S. 12).

[18] Dieter A. Wolf-Gladrow. *Lattice Gas Cellular Automata and Lattice Boltzmann Models*. Springer Berlin Heidelberg, 2000 (siehe S. 21).

[19] „IEEE Standard for Floating-Point Arithmetic“. In: *IEEE Std 754-2019* (2019) (siehe S. 21).

[20] Nvidia. *NVIDIA A100 Tensor Core-GPU*. URL: `https://www.nvidia.com/de-de/data-center/a100/` (besucht am 13. 02. 2023) (siehe S. 32).

A. Programmdateien

Das gesamte Zusatzmaterial (Der Code von Muesli und der nativen Implementierung, sowie die Laufzeitdaten) ist auch unter `https://justusdieckmann.de/bachelor/` zu finden.

Programmausschnitt A.1: `PLCube<T>`

```
#pragma once

#include "argtype.h"
#include "muesli.h"
#include <array>

#ifndef __CUDACC__
typedef struct {
    int x, y, z;
} int3;
#endif

namespace msl {

/**
 * \brief Class PLCube represents a padded local cube (partition). It serves
 *        as input for the mapStencil skeleton and actually is a shallow copy that
 *        only stores the pointers to the data. The data itself is managed by the
 *        mapStencil skeleton. For the user, the only important part is the \em get
 *        function.
 *
 * @tparam T The element type.
 */
    template <typename T>
    class PLCube {
    public:
        int width;
        int height;
        int depth;

        int stencilSize;
        T neutralValue;
        int dataStartIndex = 0;
        int dataEndIndex = 0;
        int topPaddingStartIndex = 0;
        int bottomPaddingEndIndex = 0;
        T *data;
        T *topPadding;
        T *bottomPadding;

        /**
         * \brief Constructor: creates a PLCube.
         */
```

```
44          PLCube(int width, int height, int depth, std::array<int, 3> start, std::
    array<int, 3> end, int device,
45                 int stencilSize, T neutralValue, T *data)
46              : width(width), height(height), depth(depth),
47              stencilSize(stencilSize), neutralValue(neutralValue),
48              dataStartIndex(msl::PLCube<T>::coordinateToIndex(start)),
49              dataEndIndex(msl::PLCube<T>::coordinateToIndex(end)),
50              topPaddingStartIndex(msl::PLCube<T>::coordinateToIndex(
51                      std::max(start[0] - stencilSize, 0),
52                      std::max(start[1] - stencilSize, 0),
53                      std::max(start[2] - stencilSize, 0)
54              )),
55              bottomPaddingEndIndex(msl::PLCube<T>::coordinateToIndex(
56                      std::min(end[0] + stencilSize, width - 1),
57                      std::min(end[1] + stencilSize, height - 1),
58                      std::min(end[2] + stencilSize, depth - 1)
59              )),
60              data(data) {
61  #ifdef __CUDACC__
62              cudaSetDevice(device);
63              cudaMalloc(&topPadding, (dataStartIndex - topPaddingStartIndex) *
    sizeof(T));
64              cudaMalloc(&bottomPadding, (bottomPaddingEndIndex - dataEndIndex) *
    sizeof(T));
65  #endif
66          }
67
68          MSL_USERFUNC const T& operator() (int x, int y, int z) const {
69              if (x < 0 || y < 0 || z < 0 || x >= width || y >= height || z >= depth
    ) {
70                  return neutralValue;
71              }
72              int index = coordinateToIndex(x, y, z);
73              if (index >= dataStartIndex) {
74                  if (index > dataEndIndex) {
75                      return bottomPadding[index - dataEndIndex - 1];
76                  } else {
77                      return data[index - dataStartIndex];
78                  }
79              } else {
80                  return topPadding[index - topPaddingStartIndex];
81              }
82          }
83
84          MSL_USERFUNC inline int coordinateToIndex(int x, int y, int z) const {
85              return (z * height + y) * width + x;
86          }
87
88          [[nodiscard]] inline int coordinateToIndex(const std::array<int, 3> &
    coords) const {
89              return coordinateToIndex(coords[0], coords[1], coords[2]);
90          }
91
```

```
92          MSL_USERFUNC int3 indexToCoordinate(int i) const {
93              int x = i % width;
94              i /= width;
95              int y = i % height;
96              int z = i / height;
97              return {x, y, z};
98          }
99
100         [[nodiscard]] inline int getTopPaddingElements() const {
101             return dataStartIndex - topPaddingStartIndex;
102         }
103
104         [[nodiscard]] inline int getBottomPaddingElements() const {
105             return bottomPaddingEndIndex - dataEndIndex;
106         }
107     };
108 }
```

Programmausschnitt A.2: `array<T, size>`

```
#ifndef GASSIMULATION_ARRAY_H
#define GASSIMULATION_ARRAY_H

template<typename T, unsigned int N>
struct array {
    T data[N];

    MSL_USERFUNC T operator[](size_t n) const {
        return data[n];
    }

    MSL_USERFUNC T& operator[](size_t n) {
        return data[n];
    }
};

#endif //GASSIMULATION_ARRAY_H
```

Programmausschnitt A.3: vec3<T>

```
#ifndef GASSIMULATION_VEC3_H
#define GASSIMULATION_VEC3_H

template<typename T>
struct vec3 {

    T x, y, z;

    MSL_USERFUNC void operator+=(const vec3<T>& other) {
        x += other.x;
        y += other.y;
        z += other.z;
    }

    MSL_USERFUNC friend vec3<T> operator+(vec3<T> v, const vec3<T>& other) {
        v += other;
        return v;
    }

    MSL_USERFUNC void operator*=(const T& val) {
        x *= val;
        y *= val;
        z *= val;
    }

    MSL_USERFUNC friend vec3<T> operator*(vec3<T> v, const T& val) {
        v *= val;
        return v;
    }

    MSL_USERFUNC friend T operator*(const vec3<T>& v1, const vec3<T>& v2) {
        return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
    }
};

#endif //GASSIMULATION_VEC3_H
```

Programmausschnitt A.4: Beispielanwendung LBM-Simulation mithilfe von Muesli

```
#include <iostream>
#include <cmath>
#include <vector>
#include <csignal>

#include "muesli.h"
#include "dc.h"
#include "functors.h"
#include "array.h"
#include "vec3.h"

const int FLAG_OBSTACLE = 1 << 0;
const int FLAG_KEEP_VELOCITY = 1 << 1;

#ifdef __CUDACC__
#define MSL_MANAGED __managed__
#define MSL_CONSTANT __constant__
#else
#define MSL_MANAGED
#define MSL_CONSTANT
#endif

typedef struct {
    unsigned int mantissa : 23;
    unsigned int exponent : 8;
    unsigned int sign : 1;
} floatparts;

const size_t Q = 19;
typedef array<float, Q> cell_t;
typedef vec3<float> vec3f;

std::ostream& operator<< (std::ostream& os, const cell_t f) {
    os << "(" << f[0] << ", " << f[1] << ", " << f[2] << "...)";
    return os;
}

int CHECK = 0;
int OUTPUT = 1;
namespace msl::gassimulation {

    MSL_MANAGED vec3<int> size;

    MSL_MANAGED float deltaT = 1.f;

    MSL_MANAGED float tau = 0.65;
    MSL_MANAGED float cellwidth = 1.f;

    MSL_CONSTANT const array<vec3f, Q> offsets {
        0, 0, 0,   // 0
        -1, 0, 0,  // 1
        1, 0, 0,   // 2
```

```
        0, -1, 0,  // 3
        0, 1, 0,   // 4
        0, 0, -1,  // 5
        0, 0, 1,   // 6
        -1, -1, 0, // 7
        -1, 1, 0,  // 8
        1, -1, 0,  // 9
        1, 1, 0,   // 10
        -1, 0, -1, // 11
        -1, 0, 1,  // 12
        1, 0, -1,  // 13
        1, 0, 1,   // 14
        0, -1, -1, // 15
        0, -1, 1,  // 16
        0, 1, -1,  // 17
        0, 1, 1,   // 18
    };

    MSL_CONSTANT const array<unsigned char, Q> opposite = {
            0,
            2, 1, 4, 3, 6, 5,
            10, 9, 8, 7, 14, 13, 12, 11, 18, 17, 16, 15
    };

    MSL_CONSTANT const array<float, Q> wis {
        1.f / 3,
        1.f / 18,
        1.f / 18,
        1.f / 18,
        1.f / 18,
        1.f / 18,
        1.f / 18,
        1.f / 36,
        1.f / 36,
        1.f / 36,
        1.f / 36,
        1.f / 36,
        1.f / 36,
        1.f / 36,
        1.f / 36,
        1.f / 36,
        1.f / 36,
        1.f / 36,
    };

    MSL_USERFUNC inline float feq(size_t i, float p, const vec3f& v) {
        float wi = wis[i];
        float c = cellwidth;
        float dot = offsets[i] * c * v;
        return wi * p * (1 + (1 / (c * c)) * (3 * dot + (9 / (2 * c * c)) * dot *
    dot - (3.f / 2) * (v * v)));
    }
```

```
105
106     MSL_USERFUNC cell_t update(const PLCube<cell_t> &plCube, int x, int y, int z)
        {
107         cell_t cell = plCube(x, y, z);
108
109         auto* parts = (floatparts*) &cell[0];
110
111         if (parts->exponent == 255) {
112             if (parts->mantissa & FLAG_KEEP_VELOCITY) {
113                 return cell;
114             }
115         }
116
117         // Streaming.
118         for (int i = 1; i < Q; i++) {
119             int sx = x + (int) offsets[i].x;
120             int sy = y + (int) offsets[i].y;
121             int sz = z + (int) offsets[i].z;
122             cell[i] = plCube(sx, sy, sz)[i];
123         }
124
125         // Collision.
126         if (parts->exponent == 255) {
127             if (parts->mantissa & FLAG_OBSTACLE) {
128                 cell_t cell2 = cell;
129                 for (size_t i = 1; i < Q; i++) {
130                     cell[i] = cell2[opposite[i]];
131                 }
132             }
133             return cell;
134         }
135         float p = 0;
136         vec3f vp {0, 0, 0};
137         for (size_t i = 0; i < Q; i++) {
138             p += cell[i];
139             vp += offsets[i] * cellwidth * cell[i];
140         }
141         vec3f v = p == 0 ? vp : vp * (1 / p);
142
143         for (size_t i = 0; i < Q; i++) {
144             cell[i] = cell[i] + deltaT / tau * (feq(i, p, v) - cell[i]);
145         }
146         return cell;
147     }
148
149     class Initialize : public Functor4<int, int, int, cell_t, cell_t> {
150     public:
151         MSL_USERFUNC cell_t operator()(int x, int y, int z, cell_t c) const
        override {
152             for (int i = 0; i < Q; i++) {
153                 c[i] = feq(i, 1.f, {.1f, 0, 0});
154             }
155
```

```
156             if (x <= 1 || y <= 1 || z <= 1 || x >= size.x - 2 || y >= size.y - 2
    || z >= size.z - 2 ||
157                 std::pow(x - 50, 2) + std::pow(y - 50, 2) + std::pow(z - 8, 2) <=
    225) {
158                 auto* parts = (floatparts*) &c[0];
159                 parts->sign = 0;
160                 parts->exponent = 255;
161                 if (x <= 1 || x >= size.x - 2 || y <= 1 || y >= size.y - 2 || z <=
     1 || z >= size.z - 2) {
162                     parts->mantissa = 1 << 22 | FLAG_KEEP_VELOCITY;
163                 } else {
164                     parts->mantissa = 1 << 22 | FLAG_OBSTACLE;
165                 }
166             }
167             return c;
168         }
169     };
170
171     void gassimulation_test(vec3<int> dimension, int iterations, const std::string
     &importFile, const std::string &exportFile) {
172         size = dimension;
173
174         DC<cell_t> dc(size.x, size.y, size.z);
175
176         if (importFile.empty()) {
177             Initialize initialize;
178             dc.mapIndexInPlace(initialize);
179         } else {
180             std::ifstream infile(importFile, std::ios_base::binary);
181
182             std::vector<char> buffer((std::istreambuf_iterator<char>(infile)),
183                                      std::istreambuf_iterator<char>());
184
185             if (buffer.size() != size.x * size.y * size.z * sizeof(cell_t)) {
186                 std::cerr << "Inputfile is " << buffer.size() << " bytes big, but
    needs to be " << size.x * size.y * size.z * sizeof(cell_t) << " to match the
    given dimensions!" << std::endl;
187                 exit(-1);
188             }
189
190             auto* b = (cell_t*) buffer.data();
191
192             infile.close();
193
194             for (int i = 0; i < dc.getSize(); i++) {
195                 dc.localPartition[i] = b[i];
196             }
197             dc.setCpuMemoryInSync(true);
198             dc.updateDevice();
199         }
200
201         DC<cell_t> dc2(size.x, size.y, size.z);
202
```

```
203         // Pointers for swapping.
204         DC<cell_t> *dcp1 = &dc;
205         DC<cell_t> *dcp2 = &dc2;
206 
207         for (int i = 0; i < iterations; i++) {
208 
209             double time = MPI_Wtime();
210 
211             dcp1->mapStencil<update>(*dcp2, 1, {});
212 
213             double endTime = MPI_Wtime();
214             double onlyKernelTime = endTime - Muesli::start_time;
215             double totalTime = endTime - time;
216 
217             std::cout << totalTime << " / " << onlyKernelTime << std::endl;
218 
219             std::swap(dcp1, dcp2);
220         }
221 
222         if (!exportFile.empty()) {
223             dcp1->updateHost();
224             std::ofstream outfile(exportFile, std::ios_base::binary);
225             outfile.write((char*) dcp1->localPartition, dcp1->getSize() * sizeof(
    cell_t));
226             outfile.close();
227         }
228     }
229 }
230 
231 void exitWithUsage() {
232     std::cerr << "Usage: ./gassimulation_test [-d <xdim> <ydim> <zdim>] [-g <nGPUs
    >] [-n <iterations>] [-i <importFile>] [-e <exportFile>]" << std::endl;
233     exit(-1);
234 }
235 
236 int getIntArg(char* s, bool allowZero = false) {
237     int i = std::atoi(s);
238     if (i < 0 || (i == 0 && !allowZero)) {
239         exitWithUsage();
240     }
241     return i;
242 }
243 
244 int main(int argc, char** argv){
245     vec3<int> size {100, 100, 100};
246     int gpus = 1;
247     int iterations = 1;
248     std::string importFile, exportFile;
249     for (int i = 1; i < argc; i++) {
250         if (argv[i][0] != '-') {
251             exitWithUsage();
252         }
253         switch(argv[i++][1]) {
```

```
254             case 'd':
255                 if (argc < i + 3) {
256                     exitWithUsage();
257                 }
258                 size.x = getIntArg(argv[i++]);
259                 size.y = getIntArg(argv[i++]);
260                 size.z = getIntArg(argv[i]);
261                 break;
262             case 'g':
263                 gpus = getIntArg(argv[i]);
264                 break;
265             case 'n':
266                 iterations = getIntArg(argv[i], true);
267                 break;
268             case 'i':
269                 importFile = std::string(argv[i]);
270                 break;
271             case 'e':
272                 exportFile = std::string(argv[i]);
273                 break;
274             default:
275                 exitWithUsage();
276         }
277     }
278 
279     msl::setNumRuns(1);
280     msl::initSkeletons(argc, argv);
281     msl::Muesli::cpu_fraction = 0;
282     msl::Muesli::num_gpus = gpus;
283     msl::gassimulation::gassimulation_test(size, iterations, importFile,
    exportFile);
284     msl::terminateSkeletons();
285     return EXIT_SUCCESS;
286 }
```

Programmausschnitt A.5: Native Referenzimplementierung der LBM-Simulation

```
#include "cuda.cuh"
#include "array.h"
#include <cmath>
#include <fstream>
#include <vector>

Timer timer = Timer();
double time_split;

const int FLAG_OBSTACLE = 1 << 0;
const int FLAG_KEEP_VELOCITY = 1 << 1;

typedef struct {
    unsigned int mantissa : 23;
    unsigned int exponent : 8;
    unsigned int sign : 1;
} floatparts;

const size_t Q = 19;
typedef array<float, Q> cell_t;
typedef vec3<float> vec3f;

struct gpu_t {
    int device;
    size_t mainGlobalIndex;
    size_t mainLayers;
    cell_t *data1;
    cell_t *data2;
    size_t mainOffset;
    size_t bottomPaddingOffset;
};

vec3<size_t> size;

size_t cells;
size_t bytesPerLayer;
size_t elementsPerLayer;

std::vector<gpu_t> gpuStructs;

cudaStream_t *streams;

__managed__ float deltaT = 1.f;

__managed__ float tau = 0.65;
__managed__ float cellwidth = 1.f;

bool pause = false;

__constant__ const array<vec3f, Q> offsets {
        0, 0, 0,   // 0
        -1, 0, 0,  // 1
```

```
        1, 0, 0,   // 2
        0, -1, 0,  // 3
        0, 1, 0,   // 4
        0, 0, -1,  // 5
        0, 0, 1,   // 6
        -1, -1, 0, // 7
        -1, 1, 0,  // 8
        1, -1, 0,  // 9
        1, 1, 0,   // 10
        -1, 0, -1, // 11
        -1, 0, 1,  // 12
        1, 0, -1,  // 13
        1, 0, 1,   // 14
        0, -1, -1, // 15
        0, -1, 1,  // 16
        0, 1, -1,  // 17
        0, 1, 1,   // 18
};

__constant__ const array<unsigned char, Q> opposite = {
        0,
        2, 1, 4, 3, 6, 5,
        10, 9, 8, 7, 14, 13, 12, 11, 18, 17, 16, 15
};

__constant__ const array<float, Q> wis {
    1.f / 3,
    1.f / 18,
    1.f / 18,
    1.f / 18,
    1.f / 18,
    1.f / 18,
    1.f / 18,
    1.f / 36,
    1.f / 36,
    1.f / 36,
    1.f / 36,
    1.f / 36,
    1.f / 36,
    1.f / 36,
    1.f / 36,
    1.f / 36,
    1.f / 36,
    1.f / 36,
    1.f / 36,
};

cell_t *u1;
cell_t *u2;

__device__ __host__ inline size_t pack(size_t w, size_t h, size_t d, size_t x,
    size_t y, size_t z) {
    return (z * h + y) * w + x;
```

```
}

__device__ __host__ inline float feq(const size_t i, const float p, const vec3f& v) {
    float wi = wis[i];
    float c = cellwidth;
    float dot = offsets[i] * c * v;
    return wi * p * (1 + (1 / (c * c)) * (3 * dot + (9 / (2 * c * c)) * dot * dot - (3.f / 2) * (v * v)));
}

__device__ inline void collisionStep(cell_t &cell) {
    floatparts* parts = (floatparts*) &cell[0];
    if (parts->exponent == 255) {
        if (parts->mantissa & FLAG_OBSTACLE) {
            cell_t cell2 = cell;
            for (size_t i = 1; i < Q; i++) {
                cell[i] = cell2[opposite[i]];
            }
        }
        return;
    }
    float p = 0;
    vec3f vp {0, 0, 0};
    for (size_t i = 0; i < Q; i++) {
        p += cell[i];
        vp += offsets[i] * cellwidth * cell[i];
    }
    vec3f v = p == 0 ? vp : vp * (1 / p);

    for (size_t i = 0; i < Q; i++) {
        cell[i] = cell[i] + deltaT / tau * (feq(i, p, v) - cell[i]);
    }
}

__global__ void update(cell_t *dst, cell_t *src, const size_t worksize, const vec3<size_t> globalsize, const size_t zoffset) {
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= worksize) {
        return;
    }
    size_t x = i % globalsize.x;
    size_t y = (i / globalsize.x) % globalsize.y;
    size_t z = (i / (globalsize.x * globalsize.y)) + zoffset;
    size_t index = i + zoffset * globalsize.x * globalsize.y;

    cell_t dest = src[index];
    floatparts* parts = (floatparts*) &dest;

    if (parts->exponent == 255) {
        if (parts->mantissa & FLAG_KEEP_VELOCITY) {
            dst[index] = src[index];
            return;
```

```
155         }
156     }
157 
158     for (int i = 0; i < Q; i++) {
159         int sx = x + (int) offsets[i].x;
160         int sy = y + (int) offsets[i].y;
161         int sz = z + (int) offsets[i].z;
162         if (sx < 0 || sy < 0 || sz < 0 || sx >= globalsize.x || sy >= globalsize.y
     || sz >= globalsize.z) {
163             continue;
164         }
165         dest[i] = src[pack(globalsize.x, globalsize.y, globalsize.z, sx, sy, sz)][
    i];
166     }
167 
168     collisionStep(dest);
169     dst[index] = dest;
170 }
171 
172 __device__ unsigned char floatToChar(float f) {
173     return (unsigned char) min(max((f * 10.f + 1.f) * 127.f, 0.f), 255.f);
174 }
175 
176 void syncStreams() {
177     for (auto gpu : gpuStructs) {
178         gpuErrchk(cudaStreamSynchronize(streams[gpu.device]));
179     }
180 }
181 
182 __global__ void renderToBuffer(uchar4 *destImg, cell_t *srcU, vec3<size_t> size) {
183     size_t x = blockIdx.x * blockDim.x + threadIdx.x; // Not calculating border
    cells.
184     size_t y = blockIdx.y * blockDim.y + threadIdx.y;
185     size_t z = 8;
186 
187     size_t iP = pack(size.x, size.y, size.z, x, y, z);
188     size_t iI = (size.y - y - 1) * size.x + x; // Invert opengl image.
189     vec3f p{};
190     cell_t cell = srcU[iP];
191     for (int i = 0; i < Q; i++) {
192         p += offsets[i] * cell[i];
193     }
194     destImg[iI] = {
195             floatToChar(p.x), floatToChar(p.y), floatToChar(p.z), 255
196     };
197 
198 }
199 
200 void render(uchar4 *img, const int width, const int height) {
201     simulateStep();
202     dim3 threadsPerBlock(1, 1);
203     dim3 numBlocks(size.x, size.y);
204     renderToBuffer<<<numBlocks, threadsPerBlock>>>(img, gpuStructs[0].data1, size)
```

```
    ;
    cudaDeviceSynchronize();
}

__device__ __host__ inline void generate(cell_t &cell, int x, int y, int z, const vec3<size_t> globalSize) {
    for (int i = 0; i < Q; i++) {
        float f = feq(i, 1.f, {.1f, 0, 0});
        cell[i] = f;
    }

    if (x <= 1 || y <= 1 || z <= 1 || x >= globalSize.x - 2 || y >= globalSize.y - 2 || z >= globalSize.z - 2 ||
        std::pow(x - 50, 2) + std::pow(y - 50, 2) + std::pow(z - 8, 2) <= 225) {
        auto *parts = (floatparts *) &cell[0];
        parts->sign = 0;
        parts->exponent = 255;
        if (x <= 1 || x >= globalSize.x - 2 || y <= 1 || y >= globalSize.y - 2 || z <= 1 || z >= globalSize.z - 2) {
            parts->mantissa = 1 << 22 | FLAG_KEEP_VELOCITY;
        } else {
            parts->mantissa = 1 << 22 | FLAG_OBSTACLE;
        }
    }
}


__global__ void init(cell_t *dst, size_t worksize, const vec3<size_t> globalsize, size_t zpaddingtop, size_t zoffset) {
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= worksize) {
        return;
    }

    size_t x = i % globalsize.x;
    size_t y = (i / globalsize.x) % globalsize.y;
    size_t z = (i / (globalsize.x * globalsize.y)) + zoffset;
    size_t index = i + zpaddingtop * globalsize.x * globalsize.y;

    generate(dst[index], x, y, z, globalsize);
}

void initSimulation(size_t xdim, size_t ydim, size_t zdim, size_t gpus, const std::string &importFile) {
    size = {xdim, ydim, zdim};
    cells = xdim * ydim * zdim;

    size_t layersPerGpu = zdim / gpus;
    size_t remainder = zdim - layersPerGpu * gpus;
    elementsPerLayer = xdim * ydim;
    bytesPerLayer = elementsPerLayer * sizeof(cell_t);

    u1 = new cell_t[cells];
```

```
252     u2 = new cell_t[cells];
253 
254     streams = new cudaStream_t[gpus];
255 
256     size_t currentLayer = 0;
257     gpuStructs = std::vector<gpu_t>();
258     gpuStructs.reserve(gpus);
259 
260     for (int i = 0; i < gpus; i++) {
261         gpuErrchk(cudaSetDevice(i));
262         gpuErrchk(cudaStreamCreate(&streams[i]));
263         gpu_t gpu{};
264         gpu.device = i;
265         int toppaddinglayers = i > 0 ? 1 : 0;
266         int bottompaddinglayers = i < gpus - 1 ? 1 : 0;
267         gpu.mainGlobalIndex = currentLayer * elementsPerLayer;
268         gpu.mainLayers = layersPerGpu + (i < remainder ? 1 : 0);
269         gpu.mainOffset = toppaddinglayers * elementsPerLayer;
270         gpu.bottomPaddingOffset = gpu.mainOffset + gpu.mainLayers *
    elementsPerLayer;
271 
272         currentLayer += gpu.mainLayers;
273 
274         gpuErrchk(cudaMalloc(&gpu.data1, (gpu.mainLayers + toppaddinglayers +
    bottompaddinglayers) * bytesPerLayer));
275         gpuErrchk(cudaMalloc(&gpu.data2, (gpu.mainLayers + toppaddinglayers +
    bottompaddinglayers) * bytesPerLayer));
276         gpuStructs.push_back(gpu);
277     }
278 
279     if (!importFile.empty()) {
280         importFrame(importFile);
281         for (auto gpu : gpuStructs) {
282             gpuErrchk(cudaMemcpyAsync(&gpu.data1[gpu.mainOffset], &u1[gpu.
    mainGlobalIndex], gpu.mainLayers * bytesPerLayer, cudaMemcpyDefault, streams[
    gpu.device]));
283         }
284     } else {
285         for (auto &gpu : gpuStructs) {
286             cudaSetDevice(gpu.device);
287             dim3 threadsPerBlock(512);
288             size_t worksize = size.x * size.y * gpu.mainLayers;
289             dim3 numBlocks(
290                     (worksize + threadsPerBlock.x - 1) / threadsPerBlock.x
291             );
292             init<<<numBlocks, threadsPerBlock, 0, streams[gpu.device]>>>(
293                     gpu.data1, worksize, size, gpu.mainOffset / elementsPerLayer,
    gpu.mainGlobalIndex / elementsPerLayer
294             );
295         }
296     }
297     syncStreams();
298 }
```

```
299
300 void togglePause() {
301     pause = !pause;
302 }
303
304 void updateHost() {
305     for (auto gpu : gpuStructs) {
306         gpuErrchk(cudaMemcpy(&u1[gpu.mainGlobalIndex], &gpu.data1[gpu.mainOffset],
     gpu.mainLayers * bytesPerLayer,
307                              cudaMemcpyDeviceToHost));
308     }
309 }
310
311 void updateDevice() {
312     for (auto gpu: gpuStructs) {
313         gpuErrchk(cudaMemcpy(&gpu.data1[gpu.mainOffset], &u1[gpu.mainGlobalIndex],
     gpu.mainLayers * bytesPerLayer,
314                 cudaMemcpyHostToDevice));
315     }
316 }
317
318 void simulateStep() {
319     if (pause) {
320         return;
321     }
322
323     for (int i = 1; i < gpuStructs.size(); i++) {
324         // Copy bottom padding for i - 1
325         gpuErrchk(cudaMemcpyAsync(
326                 &gpuStructs[i - 1].data1[gpuStructs[i - 1].bottomPaddingOffset],
327                 &gpuStructs[i].data1[gpuStructs[i].mainOffset],
328                 bytesPerLayer, cudaMemcpyDefault, streams[i - 1]
329         ));
330         // Copy top padding for i
331         gpuErrchk(cudaMemcpyAsync(
332                 gpuStructs[i].data1,
333                 &gpuStructs[i - 1].data1[gpuStructs[i - 1].bottomPaddingOffset -
    elementsPerLayer],
334                 bytesPerLayer, cudaMemcpyDefault, streams[i]
335         ));
336     }
337
338     syncStreams();
339     time_split = timer.get();
340     for (auto &gpu : gpuStructs) {
341         cudaSetDevice(gpu.device);
342         dim3 threadsPerBlock(1024);
343         size_t worksize = size.x * size.y * gpu.mainLayers;
344         dim3 numBlocks(
345                 (worksize + threadsPerBlock.x - 1) / threadsPerBlock.x
346         );
347         update<<<numBlocks, threadsPerBlock, 0, streams[gpu.device]>>>(
348                 gpu.data2, gpu.data1, worksize, size, gpu.mainOffset /
```

```
    elementsPerLayer
349         );
350         std::swap(gpu.data1, gpu.data2); // data1 is always pointing to up-to-date
     buffer.
351     }
352     std::swap(u1, u2);
353     syncStreams();
354 }
355 
356 void exportFrame(const std::string& filename) {
357 
358     updateHost();
359 
360     std::ofstream out;
361     out.open(filename, std::ios::out | std::ios::binary);
362 
363     for (auto gpu : gpuStructs) {
364         gpuErrchk(cudaMemcpy(&u1[gpu.mainGlobalIndex], &gpu.data1[gpu.mainOffset],
     gpu.mainLayers * bytesPerLayer, cudaMemcpyDeviceToHost));
365     }
366 
367     out.write(reinterpret_cast<const char *>(u1), cells * sizeof(cell_t));
368 
369     out.close();
370 }
371 
372 void importFrame(const std::string& importFile) {
373     std::ifstream infile(importFile, std::ios_base::binary);
374     infile.read(reinterpret_cast<char *>(u1), cells * sizeof(cell_t));
375     updateDevice();
376 }
```

B. Laufzeitdaten

In den folgenden Tabellen bezeichnet die Spalte „Dim^3“ die Seitenlänge des Würfels. Die Laufzeiten sind jeweils die Angaben von 200 Iterationen in Sekunden; die in Klammern gedruckte Zahl gibt die Laufzeit ohne die Speicheraktualisierung zwischen den GPUs an.

B.1. Laufzeitdaten vor der Optimierung

Muesli auf Stand von Commit `86d4133`, die native Anwendung auf Commit `21f7744`.

Tabelle B.1: Laufzeitdaten mit Nvidia A100 SXM 80 GB GPUs

Dim^3	GPUs	Muesli		Native	
400	1	14.9385	(14.9376)	11.1112	(11.1107)
400	2	7.9544	(7.5037)	6.0519	(5.5632)
400	3	4.5021	(3.8407)	5.4520	(3.7311)
400	4	4.9829	(3.7848)	3.9759	(2.8092)
400	5	4.8024	(3.0321)	4.8187	(2.2538)
400	6	4.4638	(1.9564)	4.9438	(1.8926)
400	7	4.6411	(1.7080)	4.5683	(1.6184)
400	8	5.5946	(1.9119)	5.1865	(1.4258)
480	1	26.1673	(26.1664)	19.2963	(19.2958)
480	2	13.8879	(13.1470)	11.1443	(9.6650)
480	3	9.7965	(8.8028)	8.2468	(6.4797)
480	4	8.3702	(6.6140)	7.7375	(4.8686)
480	5	7.9657	(5.3000)	6.4529	(3.9033)
480	6	8.1592	(4.4256)	7.0368	(3.2561)
480	7	7.0082	(2.9424)	6.6885	(2.8136)
480	8	8.5945	(3.3344)	7.3123	(2.4522)
560	1	41.6738	(41.6727)	30.7944	(30.7939)
560	2	21.8380	(20.9316)	16.4072	(15.4184)
560	3	12.0957	(10.8306)	12.7101	(10.3448)
560	4	12.7695	(10.5176)	10.0382	(7.7487)
560	5	11.9287	(8.4223)	10.3378	(6.2116)
560	6	10.3508	(5.4469)	11.2479	(5.2215)
560	7	11.6303	(6.0372)	9.9172	(4.4425)
560	8	12.0724	(5.2955)	10.5239	(3.9037)
640	1	62.3326	(62.3313)	46.2171	(46.2165)
640	2	32.6314	(31.3006)	27.7482	(23.1253)
640	3	17.8479	(16.0570)	17.2092	(15.4465)
640	4	18.8260	(15.7218)	16.3048	(11.6190)
640	5	17.2179	(12.5893)	15.3265	(9.3002)
640	6	14.6830	(8.3324)	16.0149	(7.7918)
640	7	14.4525	(7.0594)	13.9408	(6.6849)
640	8	16.5371	(7.9082)	13.0976	(5.8315)
720	1	88.5611	(88.5595)	65.7521	(65.7514)
720	2	46.0609	(44.4716)	34.4978	(32.9078)

Fortführung auf nächster Seite

Fortführung von „Laufzeitdaten mit Nvidia A100 SXM 80 GB GPUs“

Dim^3	GPUs	Muesli		Native	
720	3	31.8659	(29.7299)	24.0928	(22.0133)
720	4	26.1034	(22.3192)	20.5554	(16.5296)
720	5	24.1364	(17.8724)	18.6896	(13.2239)
720	6	23.3110	(14.9106)	19.1800	(11.0345)
720	7	20.6835	(12.7196)	18.5333	(9.4857)
720	8	22.4957	(11.2200)	20.9311	(8.3003)
800	1	121.5570	(121.5552)	90.2953	(90.2947)
800	2	62.9538	(61.0179)	48.7286	(45.1858)
800	3	36.6943	(34.0116)	34.8845	(30.2379)
800	4	35.3847	(30.6404)	30.3761	(22.6676)
800	5	31.8589	(24.5184)	27.1673	(18.1338)
800	6	25.9567	(16.1020)	26.8666	(15.1906)
800	7	25.3687	(14.1324)	24.4713	(13.0518)
800	8	29.2375	(15.3865)	27.0402	(11.3643)
880	2	83.6428	(81.1812)	62.7277	(60.3180)
880	4	46.5908	(40.7390)	36.1090	(30.2375)
880	8	37.1653	(20.4611)	31.0845	(15.1935)
960	2	108.3259	(105.4636)	81.5065	(78.3920)
960	4	59.7951	(52.8922)	46.1967	(39.2771)
960	8	45.6877	(26.5313)	43.1515	(19.7132)
1040	4	75.2658	(67.1482)	62.7509	(50.0442)
1040	8	55.7067	(33.7038)	51.8838	(25.0639)
1080	4	83.8485	(75.0962)	65.0860	(56.1063)
1080	8	61.3629	(37.7091)	51.4356	(28.1020)
1120	4	93.1402	(83.8674)	74.9217	(62.4394)
1120	8	67.0218	(42.1071)	60.7846	(31.2584)
1160	4	103.0882	(93.0613)	79.7896	(69.5427)
1160	8	72.6492	(46.6767)	56.9245	(34.7839)
1200	4	113.8870	(103.2413)	99.2813	(76.9754)
1200	8	79.8106	(51.7710)	66.3823	(38.5531)
1240	4	125.0698	(113.6080)	99.5433	(85.0219)
1240	8	86.2345	(56.9805)	72.5818	(42.5757)
1280	4	137.4176	(125.2300)	116.6246	(93.1826)
1280	8	94.3141	(62.8048)	78.6891	(46.6682)
1320	8	-	(-)	84.5165	(51.3588)

Tabelle B.2: Laufzeitdaten mit Nvidia GeForce RTX 2080 Ti 11GB GPUs

Dim^3	GPUs	Muesli		Native	
10	1	0.0074	(0.0073)	0.0044	(0.0042)
10	2	0.0197	(0.0148)	0.0187	(0.0138)
10	4	0.0223	(0.0114)	0.0214	(0.0096)
10	8	0.0278	(0.0065)	0.0318	(0.0100)
40	1	0.0564	(0.0563)	0.0306	(0.0305)
40	2	0.0426	(0.0296)	0.0288	(0.0166)
40	4	0.0499	(0.0199)	0.0381	(0.0127)
40	8	0.0772	(0.0188)	0.0743	(0.0113)
128	1	2.0062	(2.0060)	1.9204	(1.9203)
128	2	1.1744	(1.0358)	1.1642	(1.0183)
128	4	0.8383	(0.5291)	0.8154	(0.5225)
128	8	0.8585	(0.2745)	0.8584	(0.2942)
256	1	18.8560	(18.8558)	20.1854	(20.1853)
256	2	10.2581	(9.7000)	11.0495	(10.4889)
256	4	6.0111	(4.8744)	6.3941	(5.2513)
256	8	4.7636	(2.4415)	5.0440	(2.6933)
416	1	82.0089	(82.0086)	89.7745	(89.7743)
416	2	43.4474	(42.1385)	47.6662	(46.3579)
416	4	23.9598	(21.0918)	25.9784	(23.1450)
416	8	16.5905	(10.5573)	17.6964	(11.6097)
440	2	50.8060	(49.6539)	57.9893	(56.6562)
440	4	27.9723	(24.9232)	31.6740	(28.3049)
440	8	19.3568	(12.4638)	21.3575	(14.2439)
480	2	65.9410	(64.4023)	74.4443	(72.8893)
480	4	35.8158	(32.3099)	39.9565	(36.2959)
480	8	23.9010	(16.1613)	26.1534	(18.1711)
520	2	84.0970	(81.9217)	96.2426	(94.0852)
520	4	45.5919	(41.0867)	51.5726	(46.9358)
520	8	29.9742	(20.5636)	33.0317	(23.4343)
560	4	56.8684	(51.3769)	64.1523	(58.6654)
560	8	36.9578	(25.8336)	40.7915	(29.2388)
600	4	69.5609	(63.0553)	79.9614	(73.1920)
600	8	45.1801	(31.5501)	50.7909	(36.5661)
640	4	83.5231	(76.2503)	95.8841	(88.4506)
640	8	52.9216	(38.1098)	59.5506	(44.2881)
680	8	62.8411	(45.8587)	71.3348	(53.6052)
720	8	73.4547	(54.3626)	83.3602	(63.6851)
760	8	84.4833	(63.8370)	97.4672	(75.6814)

Tabelle B.3: Laufzeitdaten mit Nvidia Titan RTX 24 GB GPUs

Dim^3	GPUs	Muesli		Native	
256	1	17.5239	(17.5237)	17.7916	(17.7914)
256	2	8.9668	(8.7908)	9.1375	(8.9572)
256	3	5.7092	(4.7848)	7.3207	(6.2369)
256	4	6.7918	(4.5393)	5.9092	(4.6587)
440	1	91.0329	(91.0324)	97.1918	(97.1916)
440	2	46.4250	(45.8742)	49.3835	(48.8322)
440	3	23.6812	(20.9535)	35.9296	(33.5097)
440	4	29.6237	(23.4541)	27.9827	(25.1206)
512	1	142.7831	(142.7822)	155.9195	(155.9191)
512	2	72.8214	(72.0723)	78.8924	(78.1812)
512	3	36.7900	(32.9734)	57.1420	(53.8251)
512	4	45.3554	(36.9625)	44.0834	(40.2527)
528	1	157.5765	(157.5756)	168.4733	(168.4726)
528	2	80.3268	(79.5392)	85.3284	(84.5546)
528	3	57.9885	(54.0012)	61.2182	(57.9513)
528	4	49.3700	(40.5313)	47.8680	(43.4143)
544	1	171.6023	(171.6013)	183.6478	(183.6475)
544	2	87.4270	(86.6002)	92.9961	(92.1918)
544	3	44.9422	(40.6661)	67.7561	(63.0429)
544	4	54.2330	(44.1921)	55.0679	(47.3124)
554	2	91.7328	(90.8747)	101.2579	(100.4186)
554	3	46.6268	(42.3646)	73.3173	(69.1014)
554	4	41.4294	(31.4977)	56.4971	(51.7358)
560	2	95.1351	(94.2586)	102.3303	(101.4787)
560	3	48.6467	(44.0819)	73.5872	(69.6770)
560	4	58.9552	(48.3318)	58.9344	(52.2286)
600	2	116.7182	(115.7482)	127.8168	(126.8535)
600	3	83.9352	(78.9689)	90.6497	(86.7078)
600	4	70.6804	(59.1221)	69.9102	(65.0122)
640	2	141.1783	(140.0943)	154.3935	(153.2756)
640	3	71.3256	(65.6238)	110.8543	(104.8459)
640	4	85.5545	(71.6389)	86.3488	(78.9366)
680	2	169.4098	(168.1920)	188.2035	(186.9860)
680	3	88.8856	(82.2248)	134.7587	(128.0365)
680	4	101.5412	(85.9765)	103.9867	(95.6887)
720	3	143.4570	(136.0904)	159.0241	(151.4162)
720	4	119.3627	(102.1388)	122.5177	(113.8210)
760	3	118.1335	(110.1361)	186.6387	(180.1134)
760	4	138.3713	(119.6720)	143.6766	(135.1846)
800	4	160.4786	(139.5736)	167.0690	(156.9471)

B.2. Laufzeitdaten nach der Optimierung

Muesli auf Stand von Commit `12b2160`, die native Anwendung auf Commit `cf79742`.

Tabelle B.4: Laufzeitdaten mit Nvidia A100 SXM 80 GB GPUs

Dim^3	GPUs	Muesli		Native	
80	1	0.0560	(0.0553)	0.0529	(0.0525)
80	2	0.0637	(0.0328)	0.1154	(0.0318)
80	3	0.0713	(0.0223)	0.2699	(0.0245)
80	4	0.0934	(0.0231)	0.3474	(0.0206)
80	5	0.1573	(0.0179)	0.2549	(0.0163)
80	6	0.2731	(0.0138)	0.2610	(0.0145)
80	7	0.3723	(0.0134)	0.3399	(0.0152)
80	8	0.4808	(0.0188)	0.4922	(0.0175)
160	1	0.4050	(0.4042)	0.3798	(0.3794)
160	2	0.3193	(0.2121)	0.3001	(0.1945)
160	3	0.2774	(0.1345)	1.0204	(0.1350)
160	4	0.3102	(0.1129)	0.6098	(0.1034)
160	5	0.5356	(0.0936)	0.4972	(0.0852)
160	6	0.6942	(0.0771)	0.8597	(0.0759)
160	7	0.8781	(0.0681)	1.4653	(0.0698)
160	8	0.8818	(0.0668)	0.8647	(0.0611)
240	1	1.3821	(1.3815)	1.2888	(1.2884)
240	2	0.9043	(0.7033)	1.4449	(0.6519)
240	3	0.7501	(0.4731)	0.7095	(0.4375)
240	4	0.7990	(0.3601)	0.7746	(0.3319)
240	5	1.2146	(0.2926)	1.3527	(0.2698)
240	6	1.2000	(0.2471)	1.4520	(0.2285)
240	7	1.3203	(0.1992)	1.8642	(0.2041)
240	8	1.6694	(0.1939)	2.1156	(0.1788)
320	1	3.2972	(3.2965)	3.0736	(3.0731)
320	2	1.9626	(1.6773)	1.8610	(1.5470)
320	3	1.4434	(1.0359)	1.4576	(1.0365)
320	4	1.5675	(0.8523)	1.5069	(0.7800)
320	5	2.3196	(0.6907)	2.0980	(0.6285)
320	6	2.1843	(0.5375)	2.2847	(0.5335)
320	7	2.3264	(0.4653)	2.7732	(0.4606)
320	8	2.9930	(0.4488)	2.9695	(0.4071)
400	1	6.4701	(6.4694)	6.0364	(6.0352)
400	2	3.7563	(3.2798)	4.2981	(3.0332)
400	3	2.6734	(1.9962)	2.7081	(2.0284)
400	4	2.8660	(1.6597)	2.6996	(1.5260)
400	5	3.0835	(1.3377)	3.5046	(1.2278)
400	6	3.5700	(1.0286)	4.7530	(1.0351)
400	7	3.9241	(0.8788)	4.5774	(0.8929)
400	8	4.7313	(0.8628)	4.6263	(0.7840)
440	1	8.6369	(8.6362)	8.1005	(8.0999)
440	2	5.0434	(4.3680)	4.7445	(4.0698)

Fortführung auf nächster Seite

Fortführung von „Laufzeitdaten mit Nvidia A100 SXM 80 GB GPUs“

Dim^3	GPUs	Muesli		Native	
440	3	3.6011	(2.7299)	4.0494	(2.7222)
440	4	3.7339	(2.2127)	3.5584	(2.0444)
440	5	4.0054	(1.7827)	4.2286	(1.6406)
440	6	4.6949	(1.3889)	3.8189	(1.3824)
440	7	4.7119	(1.2759)	4.8334	(1.1851)
440	8	5.8879	(1.1477)	4.7014	(1.0407)
480	1	11.2279	(11.2271)	10.4911	(10.4906)
480	2	6.3859	(5.6753)	6.0214	(5.2632)
480	3	4.7685	(3.7934)	5.6341	(3.5142)
480	4	4.5981	(2.8619)	5.2091	(2.6415)
480	5	4.8229	(2.3002)	5.8011	(2.1206)
480	6	5.7305	(1.9310)	6.0375	(1.7737)
480	7	5.8228	(1.5473)	5.8145	(1.5334)
480	8	7.1341	(1.4720)	6.3876	(1.3449)
560	1	17.8431	(17.8422)	16.7499	(16.7493)
560	2	9.9389	(9.0191)	9.8333	(8.4023)
560	3	6.9827	(5.6973)	7.6473	(5.6158)
560	4	6.8152	(4.5343)	7.6880	(4.2170)
560	5	7.1533	(3.6462)	7.7713	(3.3815)
560	6	7.9986	(2.8610)	7.7759	(2.8380)
560	7	8.4956	(2.6334)	8.1941	(2.4322)
560	8	9.6660	(2.3232)	9.9776	(2.1369)
600	1	21.9671	(21.9662)	20.6503	(20.6498)
600	2	12.2863	(11.0909)	11.5399	(10.3505)
600	3	9.0112	(7.4133)	9.1955	(6.9105)
600	4	8.3857	(5.5739)	9.0777	(5.1899)
600	5	8.5365	(4.4755)	8.1465	(4.1601)
600	6	9.6777	(3.7576)	8.9006	(3.4719)
600	7	9.3061	(3.0672)	7.8371	(2.9879)
600	8	11.6557	(2.8491)	9.8960	(2.6222)
640	1	26.6543	(26.6533)	24.9157	(24.9150)
640	2	14.7847	(13.4554)	15.8628	(12.4826)
640	3	10.1648	(8.4129)	10.0920	(8.3415)
640	4	9.8814	(6.7586)	10.5587	(6.2583)
640	5	10.0127	(5.4289)	9.8575	(5.0121)
640	6	11.1028	(4.3571)	10.4600	(4.1997)
640	7	11.3887	(3.6672)	10.7052	(3.6184)
640	8	13.1953	(3.4402)	14.1121	(3.1681)
680	1	31.9975	(31.9964)	30.1665	(30.1658)
680	2	17.5133	(16.1538)	17.4527	(15.1252)
680	3	12.2097	(10.2963)	12.0704	(10.0874)
680	4	11.6028	(8.1113)	12.9295	(7.5700)
680	5	11.6733	(6.5040)	12.8429	(6.0631)
680	6	12.7824	(5.1469)	12.5846	(5.0813)
680	7	13.0185	(4.3631)	13.4166	(4.3618)
680	8	14.7910	(4.1407)	13.5215	(3.8149)
720	1	38.0035	(38.0022)	35.8136	(35.8130)

Fortführung auf nächster Seite

Fortführung von „Laufzeitdaten mit Nvidia A100 SXM 80 GB GPUs“

Dim^3	GPUs	Muesli		Native	
720	2	20.8701	(19.2830)	20.3055	(17.9669)
720	3	15.0515	(12.9098)	17.5063	(11.9831)
720	4	13.5862	(9.7301)	12.7923	(8.9875)
720	5	13.4806	(7.8132)	12.8759	(7.1975)
720	6	14.9291	(6.5613)	14.1704	(6.0106)
720	7	13.1705	(5.6751)	14.4577	(5.1728)
720	8	17.0092	(4.9981)	18.9525	(4.5374)
760	1	44.6793	(44.6779)	42.2083	(42.2077)
760	2	24.4792	(22.5776)	26.9907	(21.1603)
760	3	16.7401	(14.2441)	16.6205	(14.0984)
760	4	15.7534	(11.3108)	16.5654	(10.5882)
760	5	15.5188	(9.0744)	14.8918	(8.4771)
760	6	16.3368	(7.3672)	18.4440	(7.0869)
760	7	16.6290	(6.1715)	14.0248	(6.0910)
760	8	19.3511	(5.7391)	16.9830	(5.3375)
800	1	52.1563	(52.1547)	49.1322	(49.1316)
800	2	28.6634	(26.6920)	26.6970	(24.6158)
800	3	19.6313	(16.9330)	20.1999	(16.4274)
800	4	18.2452	(13.3990)	18.7502	(12.3113)
800	5	17.7005	(10.7519)	19.2517	(9.8662)
800	6	18.7068	(8.4323)	17.6208	(8.2586)
800	7	19.0526	(7.1902)	18.3796	(7.0933)
800	8	21.7812	(6.7965)	21.2477	(6.1938)
880	2	38.4843	(35.9247)	35.4732	(32.9464)
880	4	24.0942	(18.1500)	25.9059	(16.4807)
880	8	27.8753	(9.2477)	20.7229	(8.2812)
960	2	48.5150	(45.4674)	47.0620	(42.7995)
960	4	29.8129	(22.7825)	31.0011	(21.4156)
960	8	33.2635	(11.5021)	34.2904	(10.7646)
1040	4	37.2133	(28.8565)	38.1879	(27.3418)
1040	8	39.6955	(14.5654)	34.4907	(13.7432)
1080	4	41.3894	(32.5564)	45.8923	(30.6500)
1080	8	44.0387	(16.5233)	42.6000	(15.3827)
1120	4	45.5613	(36.3801)	43.5251	(34.0640)
1120	8	47.2638	(18.3170)	47.3870	(17.0989)
1160	4	50.4871	(40.2286)	60.5884	(37.9796)
1160	8	51.5342	(20.2491)	50.9255	(19.0508)
1200	4	55.5976	(44.5028)	58.6800	(42.0213)
1200	8	54.8289	(22.3653)	48.1304	(21.0718)
1240	4	61.2149	(49.4272)	57.9644	(46.3712)
1240	8	60.2327	(24.9127)	59.3051	(23.2547)
1280	4	66.7108	(54.3833)	65.9004	(50.8985)
1280	8	64.6841	(27.3561)	67.1510	(25.5243)

Tabelle B.5: Laufzeitdaten mit Nvidia GeForce RTX 2080 Ti 11GB GPUs

Dim^3	GPUs	Muesli		Native	
40	1	0.0209	(0.0208)	0.0181	(0.0180)
40	2	0.0313	(0.0184)	0.0313	(0.0185)
40	3	0.0337	(0.0128)	0.0354	(0.0148)
40	4	0.0427	(0.0123)	0.0407	(0.0135)
40	5	0.0560	(0.0109)	0.0537	(0.0146)
40	6	0.0534	(0.0092)	0.0640	(0.0138)
40	7	0.0699	(0.0107)	0.0710	(0.0123)
40	8	0.0771	(0.0111)	0.0752	(0.0128)
80	1	0.1347	(0.1346)	0.1146	(0.1145)
80	2	0.1374	(0.0772)	0.1214	(0.0695)
80	3	0.1264	(0.0497)	0.1340	(0.0557)
80	4	0.1521	(0.0459)	0.1661	(0.0494)
80	5	0.1686	(0.0400)	0.1856	(0.0491)
80	6	0.1953	(0.0309)	0.2162	(0.0520)
80	7	0.2293	(0.0301)	0.2514	(0.0500)
80	8	0.2692	(0.0355)	0.2828	(0.0452)
120	1	0.4293	(0.4292)	0.3016	(0.3015)
120	2	0.3678	(0.2372)	0.3042	(0.1894)
120	3	0.3423	(0.1645)	0.3331	(0.1587)
120	4	0.3988	(0.1302)	0.4169	(0.1619)
120	5	0.3998	(0.1121)	0.4592	(0.1709)
120	6	0.5097	(0.1064)	0.5643	(0.1662)
120	7	0.5345	(0.0715)	0.5897	(0.1647)
120	8	0.6044	(0.0932)	0.6627	(0.1670)
160	1	1.1299	(1.1298)	0.7995	(0.7994)
160	2	0.8210	(0.5920)	0.6193	(0.4140)
160	3	0.5736	(0.2900)	0.6148	(0.3184)
160	4	0.7641	(0.3067)	0.7561	(0.3253)
160	5	0.7615	(0.2527)	0.8364	(0.3274)
160	6	0.8371	(0.1712)	0.9857	(0.3182)
160	7	0.9105	(0.1562)	1.0747	(0.3156)
160	8	1.0770	(0.1996)	1.1971	(0.3359)
200	1	2.5028	(2.5026)	2.2262	(2.2261)
200	2	1.6525	(1.3085)	1.5180	(1.1600)
200	3	1.0563	(0.6199)	1.2249	(0.7782)
200	4	1.3293	(0.6689)	1.3021	(0.6550)
200	5	1.3090	(0.5373)	1.4990	(0.7222)
200	6	1.3551	(0.3197)	1.6728	(0.6042)
200	7	1.5345	(0.2885)	1.7711	(0.6153)
200	8	1.7833	(0.3539)	2.0204	(0.5844)
240	1	5.1936	(5.1934)	4.8548	(4.8547)
240	2	3.1794	(2.6989)	2.9618	(2.5061)
240	3	2.4142	(1.8038)	2.2618	(1.6719)
240	4	2.3193	(1.3568)	2.2471	(1.2602)
240	5	2.1723	(1.0905)	2.5006	(1.4046)
240	6	2.4679	(0.9116)	2.7236	(1.1860)
240	7	2.2322	(0.5396)	2.8417	(1.1809)

Fortführung auf nächster Seite

Fortführung von „Laufzeitdaten mit Nvidia GeForce RTX 2080 Ti 11GB GPUs“

Dim^3	GPUs	Muesli		Native	
240	8	2.7641	(0.7001)	3.1712	(1.1174)
280	1	8.5335	(8.5333)	8.6673	(8.6672)
280	2	5.0410	(4.4140)	5.1072	(4.4657)
280	3	2.7395	(1.9053)	3.8127	(2.9751)
280	4	3.5089	(2.2138)	3.5724	(2.2356)
280	5	3.3048	(1.7813)	3.6824	(2.1726)
280	6	3.0956	(1.0068)	4.0199	(1.9564)
280	7	3.6384	(1.2862)	4.1154	(1.8000)
280	8	3.9414	(1.1393)	4.5845	(1.8159)
320	1	12.7780	(12.7778)	12.7569	(12.7568)
320	2	7.4040	(6.5931)	7.3777	(6.5673)
320	3	4.0656	(3.0159)	5.4583	(4.3945)
320	4	4.9811	(3.3097)	4.9619	(3.2804)
320	5	4.5825	(2.6502)	4.9169	(3.0057)
320	6	4.2111	(1.4838)	5.2912	(2.6058)
320	7	4.3021	(1.3195)	5.6273	(2.7044)
320	8	5.3112	(1.6886)	6.1221	(2.5094)
340	1	15.3707	(15.3706)	15.4001	(15.3999)
340	2	8.8219	(7.9386)	8.8403	(7.9309)
340	3	4.6711	(3.5131)	6.4495	(5.2818)
340	4	5.8848	(3.9780)	5.8822	(3.9708)
340	5	5.3467	(3.1776)	5.8294	(3.6422)
340	6	4.9261	(1.8519)	6.3732	(3.2887)
340	7	4.9576	(1.5397)	6.6034	(3.2782)
340	8	5.4989	(1.3682)	6.9572	(2.8795)
360	1	18.2778	(18.2776)	18.1789	(18.1788)
360	2	10.4474	(9.4265)	10.3913	(9.3572)
360	3	7.5958	(6.2905)	7.5486	(6.2361)
360	4	6.8227	(4.7173)	6.7896	(4.6651)
360	5	6.2570	(3.7830)	6.4985	(4.0896)
360	6	6.5788	(3.1527)	7.3238	(3.9335)
360	7	5.5772	(1.8427)	7.7383	(4.0074)
360	8	6.9602	(2.3937)	8.2906	(3.7352)
380	1	21.5199	(21.5197)	20.9764	(20.9763)
380	2	12.1934	(11.0917)	11.9465	(10.8279)
380	3	6.6004	(5.1488)	8.6829	(7.2159)
380	4	7.8905	(5.5448)	7.7910	(5.4002)
380	5	7.1170	(4.4466)	7.4248	(4.6761)
380	6	6.3494	(2.5233)	8.4579	(4.6582)
380	7	6.4862	(2.1708)	8.7582	(4.5878)
380	8	7.0597	(1.9034)	9.4021	(4.3747)
400	1	25.0758	(25.0756)	24.3875	(24.3873)
400	2	14.1496	(12.9320)	13.7979	(12.5570)
400	3	7.4168	(5.8107)	9.9728	(8.3577)
400	4	9.0259	(6.4575)	8.8749	(6.2577)
400	5	8.1603	(5.1780)	9.2588	(6.2927)
400	6	7.3525	(3.0690)	9.4649	(5.3511)

Fortführung auf nächster Seite

Fortführung von „Laufzeitdaten mit Nvidia GeForce RTX 2080 Ti 11GB GPUs“

Dim^3	GPUs	Muesli		Native	
400	7	7.1682	(2.4797)	9.6868	(5.0572)
400	8	8.8751	(3.2735)	10.8641	(5.2956)
420	1	-	(-)	28.2909	(28.2907)
420	2	16.3208	(14.9972)	15.8569	(14.5638)
420	3	11.7760	(9.9949)	11.4889	(9.6918)
420	4	10.3328	(7.4772)	10.1668	(7.2728)
420	5	9.2890	(6.0047)	10.3741	(7.0205)
420	6	9.6440	(5.0018)	10.7611	(6.0091)
420	7	9.3786	(4.3060)	11.2766	(6.0651)
420	8	8.8698	(2.6341)	11.7035	(5.4531)
440	2	18.4179	(17.2310)	18.0156	(16.6690)
440	4	11.7421	(8.5991)	11.4127	(8.3001)
440	8	11.3842	(4.3569)	13.4024	(6.6074)
480	2	23.8524	(22.3548)	23.0899	(21.4561)
480	4	14.7038	(11.1506)	14.3127	(10.6635)
480	8	13.3872	(5.6346)	16.1418	(8.3163)
520	2	30.5899	(28.4691)	29.6268	(27.6193)
520	4	18.5367	(14.1855)	17.9596	(13.7307)
520	8	16.4211	(7.1678)	20.2374	(11.1722)
560	4	23.0034	(17.6941)	22.9814	(17.5966)
560	8	19.8568	(8.8947)	24.6569	(13.7474)
600	4	28.3776	(21.7837)	28.4819	(22.0196)
600	8	24.8434	(11.0015)	29.9340	(16.4696)
640	4	33.6159	(26.3754)	34.2046	(26.9767)
640	8	28.3424	(13.3178)	33.5997	(19.0755)
680	8	33.4742	(16.0240)	38.7912	(22.0305)
720	8	38.3281	(18.8858)	44.6724	(26.0709)
760	8	43.3334	(22.1673)	51.7406	(31.2113)

Tabelle B.6: Laufzeitdaten mit Nvidia Titan RTX 24 GB GPUs

Dim^3	GPUs	Muesli		Native	
40	1	0.0197	(0.0196)	0.0169	(0.0168)
40	2	0.0202	(0.0120)	0.0190	(0.0109)
40	3	0.0419	(0.0143)	0.0485	(0.0146)
40	4	0.0670	(0.0109)	0.0533	(0.0156)
80	1	0.1225	(0.1224)	0.0798	(0.0797)
80	2	0.0894	(0.0697)	0.0789	(0.0590)
80	3	0.1394	(0.0516)	0.1722	(0.0614)
80	4	0.2616	(0.0460)	0.1752	(0.0533)
120	1	0.3897	(0.3896)	0.2454	(0.2453)
120	2	0.2498	(0.2071)	0.1719	(0.1294)
120	3	0.3605	(0.1431)	0.3970	(0.1706)
120	4	0.6834	(0.1255)	0.4409	(0.1870)
160	1	1.0471	(1.0470)	0.7135	(0.7134)
160	2	0.5991	(0.5300)	0.4210	(0.3524)
160	3	0.6302	(0.2696)	0.7880	(0.3386)
160	4	1.2116	(0.2927)	0.8609	(0.3700)
200	1	2.3029	(2.3027)	1.9697	(1.9695)
200	2	1.2665	(1.1603)	1.1029	(0.9949)
200	3	1.1063	(0.5724)	1.3818	(0.7042)
200	4	2.0555	(0.6280)	1.5506	(0.7887)
240	1	4.8070	(4.8068)	4.3848	(4.3847)
240	2	2.5699	(2.4132)	2.3609	(2.2017)
240	3	2.5214	(1.6764)	2.4161	(1.5226)
240	4	3.3274	(1.2755)	2.6812	(1.6258)
280	1	8.1033	(8.1031)	8.1310	(8.1308)
280	2	4.2767	(4.0647)	4.2938	(4.0795)
280	3	2.8919	(1.7991)	4.1856	(2.8853)
280	4	4.8560	(2.1311)	3.6012	(2.1139)
320	1	12.1024	(12.1022)	12.0394	(12.0392)
320	2	6.3448	(6.0674)	6.3135	(6.0299)
320	3	4.3739	(2.8282)	5.8203	(4.2179)
320	4	6.7457	(3.1620)	5.0380	(3.1255)
340	1	14.5480	(14.5478)	14.5406	(14.5404)
340	2	7.6090	(7.2936)	7.6070	(7.2874)
340	3	5.0232	(3.3134)	6.6888	(5.0021)
340	4	7.8463	(3.7999)	5.7488	(3.7473)
360	1	17.2911	(17.2909)	17.1771	(17.1770)
360	2	9.0245	(8.6656)	8.9713	(8.6111)
360	3	7.9060	(5.9699)	7.8072	(5.9078)
360	4	9.1190	(4.5055)	6.6980	(4.4687)
380	1	20.3201	(20.3199)	19.9602	(19.9601)
380	2	10.5840	(10.1867)	10.4402	(10.0194)
380	3	6.9686	(4.8424)	9.0840	(6.9206)
380	4	10.3077	(5.2881)	7.7499	(5.1011)
400	1	23.7264	(23.7261)	23.0172	(23.0170)
400	2	12.3355	(11.8957)	11.9895	(11.5297)
400	3	7.8633	(5.4990)	10.5074	(7.9060)

Fortführung auf nächster Seite

Fortführung von „Laufzeitdaten mit Nvidia Titan RTX 24 GB GPUs“

Dim^3	GPUs	Muesli		Native	
400	4	11.9356	(6.1533)	10.3006	(6.0102)
420	1	27.5082	(27.5079)	26.7288	(26.7287)
420	2	14.2769	(13.7867)	13.9059	(13.3949)
420	3	11.9668	(9.4695)	11.9872	(9.1851)
420	4	13.5007	(7.1393)	10.1832	(6.8708)
440	1	31.6415	(31.6413)	30.5201	(30.5200)
440	2	16.4044	(15.8584)	15.8519	(15.2945)
440	3	10.4679	(7.6708)	12.4129	(10.4876)
440	4	14.5690	(8.1959)	11.2312	(7.8401)
460	1	36.1479	(36.1476)	34.8296	(34.8294)
460	2	18.7151	(18.1184)	18.0409	(17.4394)
460	3	11.4977	(8.4673)	14.0246	(11.9741)
460	4	16.8499	(9.3437)	11.8952	(9.0183)
480	1	41.0624	(41.0620)	39.2220	(39.2217)
480	2	21.2489	(20.5901)	20.3005	(19.6520)
480	3	17.4049	(14.1060)	17.0717	(13.4208)
480	4	18.4360	(10.5795)	16.0591	(10.0329)
500	1	46.3301	(46.3297)	44.7124	(44.7119)
500	2	23.9531	(23.2373)	23.1356	(22.4170)
500	3	15.0645	(11.4587)	18.3897	(15.4134)
500	4	20.4656	(11.9763)	15.3928	(11.4734)
520	1	52.2976	(52.2972)	50.6627	(50.6623)
520	2	27.0015	(26.2223)	26.1400	(25.3770)
520	3	16.6202	(12.5921)	21.6159	(17.3397)
520	4	23.2163	(13.4772)	17.9571	(12.9837)
560	2	33.6155	(32.7442)	33.3507	(32.4852)
560	3	21.2184	(16.5668)	26.9787	(22.2172)
560	4	27.7951	(16.8545)	22.0904	(16.5971)
600	2	41.2729	(40.2961)	41.8142	(40.8514)
600	3	32.5936	(27.4868)	31.8640	(27.9678)
600	4	31.8870	(20.6750)	25.9524	(20.8671)
640	2	50.0126	(48.9020)	51.1585	(50.0779)
640	3	29.8036	(23.9813)	39.8124	(34.2492)
640	4	38.8909	(24.9904)	31.9869	(25.6656)
680	2	60.1835	(58.9640)	61.3336	(60.1189)
680	3	37.8651	(31.1287)	47.3392	(41.0242)
680	4	46.2703	(30.0343)	38.7218	(30.6218)
720	3	55.0270	(47.4887)	54.9244	(48.4227)
720	4	53.8722	(35.6522)	44.7757	(36.2445)
760	3	49.9641	(41.5874)	62.9604	(56.9091)
760	4	61.8806	(42.0466)	51.7020	(42.7666)
800	4	70.9441	(48.9988)	58.7492	(48.7731)

Eidesstattliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit über „Implementierung und Optimierung von dreidimensionalen Stencil-Skeletten mithilfe paralleler Verarbeitung“ selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Steinfurt, den 14. Februar 2023,

_____

Justus Dieckmann