Effiziente Konstruktion von Gridlets auf der GPU zur Visualisierung von AMR Daten

Masterarbeit

vorgelegt von **Maria Zhumabaeva**

am 28. September 2023

Department Mathematik/Informatik
Mathematisch-Naturwissenschaftliche Fakultät
Universität zu Köln

Erstgutachter: PD Dr. Stefan Zellmann

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Die Leistungssteigerungen der Hardware führen zu einer zentralen Herausforderung: Wegen des rasanten Wachstums der Rechenleistung, das die Kapazitäten der Speicherbandbreite und die Verringerung der Latenz übersteigt, ist eine besondere Strategie erforderlich. Diese Strategie soll sicherstellen, dass das erweiterte Potenzial dennoch effizient ausgeschöpft werden kann. Eine mögliche Vorgehensweise besteht darin, die zu übertragende Datenmenge proportional zu reduzieren.

So beschäftigt sich der Ansatz von Berger et al. [4] mit der Frage, wie Berechnungen sich nur auf die in einem Kontext relevanten Daten beschränkt werden können. Dieser Kontext kann zum Beispiel sein, dass Gebiete, die wenig neue Informationen beitragen, wie glatte oder homogene Gebiete, keine feinmaschige Verfügbarkeit erfordern. Damit kann sich sowohl der Speicher- als auch der Rechenbedarf erheblich reduzieren.

Der Einsatz von dieser Art von Daten, der AMR (Adaptive Mesh Refinement) Daten, ist vielfältig. Ursprünglich haben Berger et al. die adaptive finite Differenzen Methode zum Lösen von hyperbolischen partiellen Differentialgleichungssystemen entwickelt. Seitdem hat sich der Anwendungsbereich erweitert. Beispielsweise beinhalten kosmologische Daten (Norman et al. [13]) viel leeren Raum und gleichzeitig Strukturen mit einer sehr hohen Informationsdichte wie Planeten oder Asteroidengürtel. Aus diesem Grund ist es sinnvoll, sowohl zur Visualisierung als auch für andere Berechnungen die Auflösung variieren zu können. Weitere Einsatzbereiche sind Wettermodellierung (Ferguson et al. [5]) und die Visualisierung anderer physikalischer Größen.

Das Ziel dieser Arbeit besteht darin, auf den Forschungsergebnissen aus dem Paper von Zellmann et al. [25] aufbauend, eine Datenstruktur zur Verarbeitung von AMR-Daten zu beschleunigen. Dazu werden wir, wie in dem oben genannten Paper vorgeschlagen, Teile des zur Verfügung gestellten Codes von der CPU auf die GPU portieren. Hierbei streben wir an, durch die besondere Architektur von GPUs, insbesondere der Parallelität, den Aufbau von Gitterstrukturen effizienter zu gestalten.

Zunächst führen wir in Kapitel 2 die Grundbegriffe ein. Dann gehen wir im Kapitel 3 auf den derzeitigen Forschungsstand ein und betrachten, welche alternativen Ansätze es gibt, um AMR Daten zu verarbeiten.
In Kapitel 4 stellen wir Methoden vor, um Teile des Algorithmus, im Speziellen das Erzeugen der Gridlets auf der GPU auszuführen.
In Kapitel 5 gehen wir kurz auf die Testumgebung ein. Wir beschreiben die Datensätze, die wir für die Auswertung verwenden, in Kapitel 6. Anschließend vergleichen wir in Kapitel 7 die Performanz der Implementierungen in [25] mit den in Kapitel 4.3 vorgestellten Modifikationen.
Abschließend diskutieren wir die Ergebnisse und gehen darauf ein, welche weiteren Verbesserungen noch in Zukunft möglich wären.

Im Anhang befindet sich eine Liste mit den Quellcodedateinamen, die im Laufe dieser Arbeit entstehen.

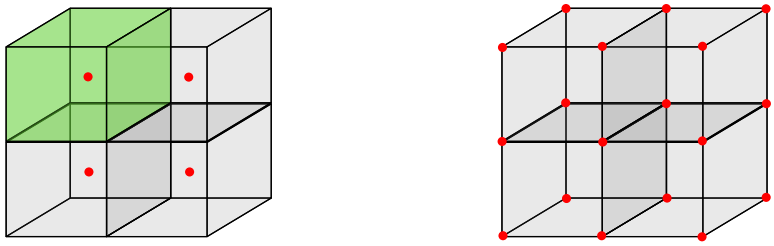
2 Grundlagen

Im folgenden Kapitel führen wir die zentralen Begriffe ein. Die Daten, um die es in dieser Arbeit geht, sind nach Zellen strukturiert. In der nachfolgenden Definition halten wir die Notation fest.

Definition 1 (Zellen und Würfel)
Sei $C_{i,j,k} := [i, i + 2^\ell] \times [j, j + 2^\ell] \times [k, k + 2^\ell]$ *ein Würfel mit Seitenlängen* 2^ℓ *und Level* $\ell \in \mathbb{N}_0$ *eine* Zelle. *Die Ecke* $C_{lower} = (i, j, k)$ *wird als die Ecke mit den kleinsten Koordinaten definiert. Dabei können die Datenpunkte an den Ecken des Würfels liegen. Diese Daten werden knotenzentrische Daten genannt. Alternativ kann sich der Datenpunkt im* Zellkern C_p *an den Koordinaten* $(i + 2^{\ell-1}, j + 2^{\ell-1}, k + 2^{\ell-1})$ *befinden. Dessen Inhalt ist als* $C_v \in \mathbb{R}$ *notiert. Diese Art von Daten werden* zellzentrisch *genannt.*

Den Unterschied zwischen knotenzentrischen und zellzentrischen Daten verdeutlichen wir in der Abbildung 2.1.

Die Daten können Skalare, Vektoren oder Tensoren sein. Sie könnten beispielsweise Geschwindigkeit und Wirbelstärke wie bei Simulation eines Flugzeugs (Zellmann et al. [25]) oder Gasdichte und Teilchen im Weltall (Norman et al. [13]) beschreiben. Wir nehmen im Folgenden an, dass der Input aus Skalaren besteht, da der Schwerpunkt in der Anwendung der Daten im Kapitel 4 dort liegt.



(a) (b)

Abbildung 2.1: Ein $2 \times 1 \times 2$ Gitter mit Datenpunkten als rote Kreise.
(a) Zellzentrische Daten: Eine Zelle ist grün hervorgehoben.
(b) Knotenzentrische Daten. (Eigene Darstellung)

Im dreidimensionalen Raum beschreibt der triviale Fall ein kartesisches Gitter. Die Daten an den Gitterpunkten angeordnet und die dazugehörigen Zellen haben alle das gleiche Level. Das Sampeln, also das stichprobenartige Berechnen eines Datenpunktwertes innerhalb einer Zelle aus gegebenen Gitterpunkten, ist in so einem Fall mithilfe der trilinearen Interpolation möglich.

2.1 AMR Datentyp

Für diese Arbeit gehen wir davon aus, dass die Inputdaten zellzentrisch sind, da die modellierten Daten in der Praxis oftmals zellzentrisch sind.

Im Jahr 1983 stellen Berger und Oliger 1983 AMR (Adaptive Mesh Refinement) [4] vor, um hyperbolische partielle Differentialgleichungen mit finiten Differenzen lösen zu können. Sie definieren eine Gebietszerlegung mit Hilfe einer Gitterstruktur, die nach Bedarf rekursiv verfeinert wird. Der Vorteil, auf den die Autoren hinweisen, ist, dass an den Stellen, an denen die Lösung schwer zu approximieren ist, ein feineres Gitter verwendet werden kann. Solche Stellen können zum Beispiel Unstetigkeiten oder Randbereiche sein.

Auf die Lösung im feineren Gitterbereich wird dann das finite Differenzen Verfahren angewandt. Eine Besonderheit ist, dass die Subgitter rotiert und rechteckig (nicht quadratisch) sein dürfen. Dabei werden die Gitter nicht in die gröberen Gitter gestaffelt und sind somit nicht verbunden. Diese Trennung dient dazu, über die einzelnen Subgitter fast unabhängig voneinander integrieren zu können. Die Rotation lassen sie zu, um beispielsweise bei einer Schockwelle die Tangentialeigenschaft auszunutzen. Da die Lösungsvektoren jeweils getrennt für jedes Subgitter berechnet werden, kann die Lage des Gitters geschickt gewählt werden.
In einem späteren Paper [3] verzichten Berger et al. auf diese Eigenschaft, um den Rechenaufwand zu verringern.

Zusammenfassend ergibt sich die folgende Definition, die als Grundlage für viele späteren Arbeiten dient:

Definition 2 (AMR Datentyp nach [3])
Sei G eine Gebietszerlegung in Rechtecke beziehungsweise Quader. Ein Gitter G_ℓ mit Level $\ell \in \mathbb{N}_0$ und einer festen Gitterweite h_ℓ, $h_\ell < h_{\ell-1}$ ist eine Vereinigung aus endlich vielen Subgittern. Die Vereinigung aus allen G_ℓ entspricht wiederum dem ganzen Gebiet. Mit anderen Worten:

$$G = \bigcup_l G_l = \bigcup_l \bigcup_k G_{l,k}.$$

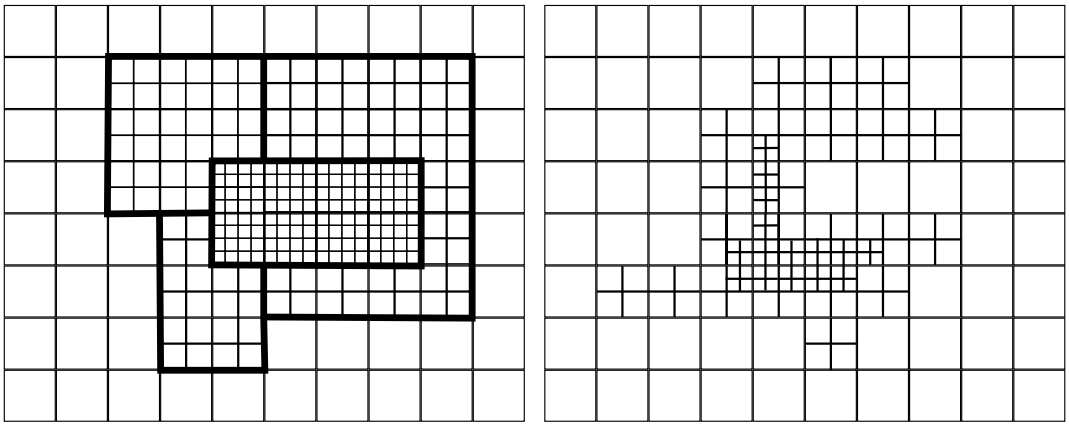
Außerdem müssen folgende Bedingungen erfüllt sein:

1. *Die Subgitter sind so ausgerichtet, dass die obersten und untersten Ecken an den Zellecken des gröberen Gitters liegen.*

2. *Für jede Zelle darf sich das Level der benachbarten Subgitter maximal um eins unterscheiden.*

Der Verfeinerungsfaktor $r \in N$, ein Verhältnis welches den Unterschied zwischen Gitterweiten zweiter Level beschreibt, sei konstant.

Diese Art von AMR wird als *block-structured AMR* bezeichnet, da die Subgitter als zusammenhängende Blöcke betrachtet werden können. Eine Alternative dazu bildet zum Beispiel eine Baumstruktur. Abbildung 2.2 zeigt Beispiele für die beiden Typen.



(a) (b)

Abbildung 2.2: Zwei Arten von AMR Daten in zwei Dimensionen. (a) Blockstruktur aus einem Level 0 Block und drei Level 1 Blöcke in einem Level 2 Block. (b) Baumstruktur aus drei ineinander verschachtelten Leveln. (In Anlehnung an Wang et al. [21])

Dabei unterscheiden sich die benachbarten Zellen maximal um ein Level. Eine Einschränkung, die die Forscher noch über einige Zeit beschäftigt, ist der Levelübegang von mehr als einer Stufe. In späteren Verfahren kann schließlich diese Bedingung gelockert werden, worauf wir im Kapitel 3 näher eingehen werden.
Die Gitter können unstrukturiert sein und aus nicht rechtwinkligen Elementen wie Dreiecken bestehen.

Es kann sinnvoll sein, die Subgitter levelweise zu betrachten. Das heißt, anstelle von einzelnen $G_{\ell,k}$ betrachten wir jedes Level als ein strukturiertes Gitter. Nicht zusammenhängende Bereiche führen dann dazu, dass nicht alle Zellen, die durch dieses Gitter definiert sind, tatsächlich existieren.

Definition 3 (reale und virtuelle Zellen)
Falls eine Zelle $C_{i,j,k} \in \bigcup_\ell G_{\ell,k}$*, so nennen wir diese eine* reale *Zelle, sonst nennen wir* $C_{i,j,k}$ *eine* virtuelle *Zelle.*

Die als am nächsten liegende existierende Zelle $\hat{C}$ *von einer virtuellen Zelle* C *bezeichnen wir entweder die Zelle selbst, sofern sie real ist, oder diejenige Zelle, die an dieser Stelle existiert, das feinste Level besitzt und in der eine Ecke von* C *liegt.*

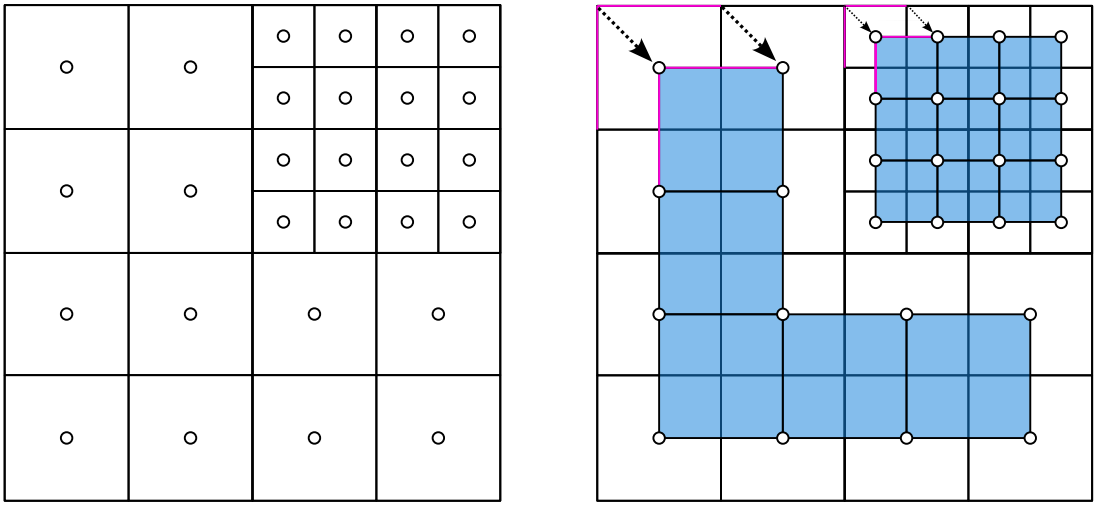
Eine reale Zelle mit dem niedrigsten Level in der P *liegt, bezeichnen wir als eine Blattzelle.*

2.2 Von AMR Daten zur Visualisierung

Die gesammelten Daten dienen als Input für Software zur wissenschaftlichen Visualisierung wie zum Beispiel VTK [14] oder ParaView [2]. Diese bereiten den Input so auf, dass die Berechnungen auf der Hardware ausgeführt werden können. Die Datenstruktur als Input (beziehungsweise als Output der Simulation) wird implizit durch die Wahl des Renderingalgorithmus bestimmt.

Bei der Rekonstruktion der Daten gibt es unterschiedliche Ansätze. Die einfachste Möglichkeit wäre es die Blattzelle C von dem gesuchten Wert von Punkt P zu finden und dessen C_v zu übernehmen. Dann würden wir jedoch eine unstetige Abbildung bekommen, da die Übergänge sogar innerhalb eines Levels an den Zellgrenzen Stufen bilden.
Eine gängige Methode innerhalb eines Würfels zu interpolieren ist die trilineare Interpolation. Diese ist jedoch nur anwendbar, wenn die Werte der Eckpunkte bekannt sind, also bei einem knotenzentrischen Gitter. Verschieben wir das zellzentrische Gitter, so führen wir das Problem auf ein knotenzentirsches zurück:

Definition 4 (duales Gitter)
Ein duales Gitter *ist ein derart um eine Konstante verschobenes Gitter, dass die alten Koordinaten der* C_p *nun die Eckpunkte der Zellen des dualen Gitters sind. Analog zu der Definition 3 definieren wir die Begriffe für duale Zellen* $D^{(\ell)}_{i,j,k}$*. Den Operator* $D^{\ell}(P)$ *erweitern wir um die Fähigkeit, die am nächsten liegende existierende Zellen der Ecken von der dualen Zelle zu bestimmen.*



(a) (b)

Abbildung 2.3: Aus einem zellzentrischen Gitter mit zwei Levels (a) entsteht durch das Verschieben um eine halbe Zellenbreite des jeweiligen Levels ein duales Gitter (b), in blau hervorgehoben. (In Anlehnung an Zellmann et al. [25])

Betrachten wir jedes Level als ein Gitter und interpolieren jede Zelle, so haben wir eine bessere Methode zur Rekonstruktion. Um zu ermöglichen, dass das Verfahren beim Interpolieren des dualen Gitters adaptiv bleibt, müssen wir nicht nur die Ecken einer dualen Zelle berücksichtigen, sondern auch die benachbarten Zellen der anderen Level. Dazu können wir die Werte an den Levelübergängen verblenden Wald et al. [18].

Das Erlauben einer Leveldifferenz in den benachbarten Zellen führt hier zu dem sogenannten *t-junction Problem* Weber et al. [22]. Es beschreibt Unstetigkeiten an den Levelübergängen, die die Visualisierungsqualität durch Knicke und Risse verschlechtern, ebenso die Berechnungen in anderen Anwendungen ungenau machen. In der Abbildung 2.4 sehen wir dafür einen Beispiel im zweidimensionalen Raum. Während für Punkte P_1 im feineren und P_2 im gröberen Gitter eine gewöhnliche bilineare Interpolation möglich ist, ist es für den Wert von P_3 unklar, welche Datenpunkte wir für die Berechnungen mit einbeziehen sollen.

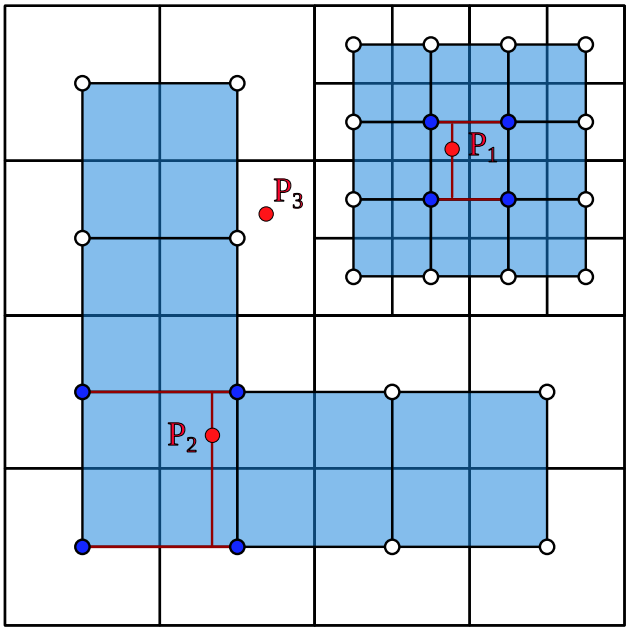


Abbildung 2.4: Für die Punkte P_1 und P_2 (rote Kreise) ist eine Interpolation mit Hilfe der Eckpunkte (dunkelblaue Kreise) im dualen Gitter (hellblaue Rechtecke) möglich. P_3 kann nicht analog interpoliert werden. (In Anlehnung an Zellmann et al. [25])

Wir werden mehrere Ansätze sehen, dieses Problem zu lösen: Zum einen durch den Einsatz von speziellen Verbindungselementen - den *stitching cells*, zum anderen kann es ganz umgangen werden, wenn wir das duale Gitter nicht benutzen, sondern mit dem zellzentrischen Gitter arbeiten.

3 Forschungsstand

In diesem Kapitel schauen wir uns vier Ansätze an, um AMR Daten zu verarbeiten und welche Herausforderungen diese Lösungen mit sich bringen. Am Ende jedes Unterkapitels sind weitere Verfahren aufgeführt, die auf den jeweiligen Ansätzen basieren oder ähnliche Grundgedanken haben.

3.1 Basisfunktionen

Die Idee einer Rekonstruktion mit Basisfunktionen ist, die Eckpunkte von der dualen Zelle eines gegebenen Punktes P als eine Menge von gestreuten Datenpunkten zu betrachten und mit der Methode von Franke und Nielson [6] zu interpolieren. Die Basisfunktion ist dabei frei wählbar. In [18] benutzen Wald et al. Zeltbasisfunktionen aufgrund der einfachen Umsetzbarkeit und der Tatsache, dass die Ergebnisse für viele P mit der normalen trilinearen Interpolation übereinstimmen.

Definition 5 (Zeltbasisfunktionen)
Sei D eine duale Zelle, $P \in D$ und C eine beliebige benachbarte Zelle aus Ursprungsgitter mit Zellkern $C_p \in \mathbb{R}^3$. Eine Zeltbasisfunktion *ist definiert durch*

$$\hat{H}_C(P) = \hat{h}\left(\frac{|C_{p,x} - P_x|}{C_w}\right) \cdot \hat{h}\left(\frac{|C_{p,y} - P_y|}{C_w}\right) \cdot \hat{h}\left(\frac{|C_{p,z} - P_z|}{C_w}\right) \tag{3.1}$$

mit C_w der Zellenbreite von C und

$$\hat{h}(t) = \max(1 - t, 0). \tag{3.2}$$

Die Basisfunktionen liegen somit mittig über das Zellinnere und haben den Träger, der um eine halbe Zellenbreite über die eigentlichen Zellgrenzen hinausragt.

Die Autoren verblenden die Werte mit Hilfe der Zellen C aus dem ursprünglichen Gitter. Dazu wählen sie diejenigen Zellen, die für eine gegebene duale Zelle D an dessen Ecken liegen. Sie definieren die entsprechende Funktion wie folgt:

$$\text{lerp}(P, D) = \sum_{C \in \text{Ecken von } D} \hat{H}_C(P) C_v. \tag{3.3}$$

Anschließend berechnen sie den Mittelwert daraus und somit ergibt sich die folgende Gleichung, um den Wert an der Stelle P zu berechnen:

$$\text{AMR}_{basis}(P) = \frac{\sum_{C_i} \hat{H}_{C_i}(P) C_{v_i}}{\sum_{C_i} \hat{H}_{C_i}(P)}.$$

Die Basisfunktionen sind Funktionen, die die Anzahl der zu betrachtenden Datenpunkte je nach Abstand zu P beschränken und gleichzeitig die Gewichte, mit denen diese in die Gesamtauswertung eingehen, beschreiben. So ist die Gewichtung des Zellwerts größer, je näher sich der Punkt an dem entsprechenden Zellkern befindet, also je kleiner der Abstand $|C_p - P|$ ist. Das Zusammenspiel der Basisfunktionen stellen wir in der Abbildung 3.1 dar.

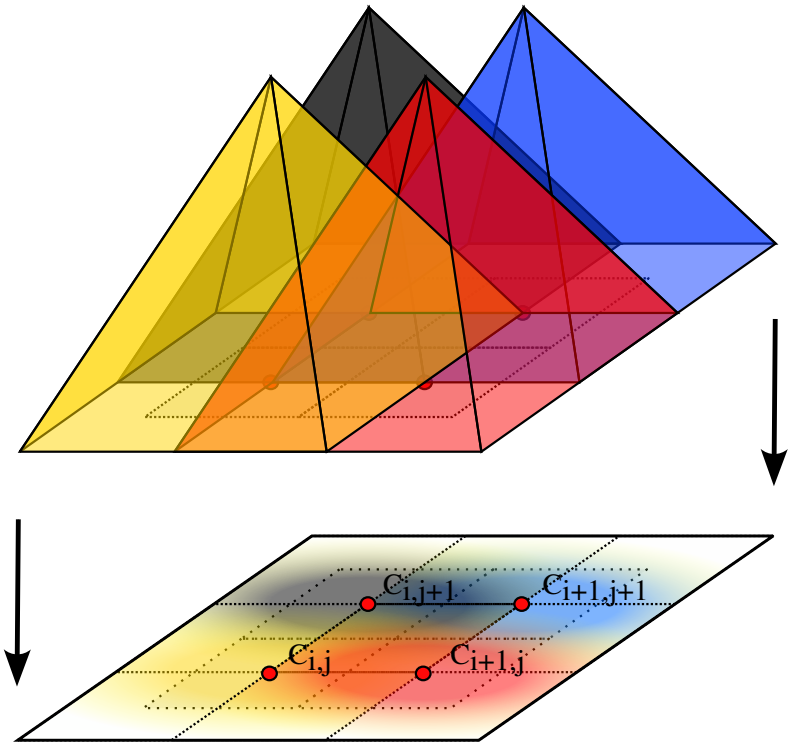


Abbildung 3.1: Ein Beispiel für Zeltbasisfunktionen. Die Punkte $C_{i,j}$, $C_{i+1,j}$, $C_{i,j+1}$ und $C_{i+1,j+1}$, als rote Kreise dargestellt, spannen eine duale Zelle auf. Über jeder dieser Punkte ist eine Basisfunktion definiert - hier Pyramiden. Für einen beliebigen Punkt aus dem Gitter (schwarz gestrichelt) definieren die Höhen der Pyramiden, wie viel Einfluss sie jeweils auf den Punkt haben. Der Einfluss ist hier als eine Projektion dargestellt: Je intensiver die Farbe des Schattens der Pyramide, desto mehr Einfluss hat diese in einem bestimmten Punkt. (Eigene Darstellung)

Im folgenden Algorithmus 1 fassen wir das Vorgehen zusammen:

Algorithm 1: Interpolation mit Zeltbasisfunktionen nach Wald et al. [18]

```
1  float AMRbasis(P)
2      gewichteSumme = 0
3      gewichteteWerteSumme = 0
4      for ℓ = 0, ... do
5          D = findeDualeZelle(P)
6          foreach Zelle C an den Ecken von D do
7              if C ist eine Blattzelle then
8                  gewichteSumme+ = Ĥ(P,C) // Ĥ(P,C) Zeltbasisfunktion
9                  gewichteteWerteSumme+ = Ĥ(P,C) * C.v
10             end
11         end
12         if keine der C in D ist ein innerer Knoten then
13             break
14         end
15         return gewichteteWerteSumme/gewichteSumme
16     end
```

Zum Schluss wird über die Struktur mit Algorithmus 2 ein k-d Baum gebildet, um durch die AMR Blöcke traversieren zu können:

Algorithm 2: k-d Baum über AMR Blöcke nach [18]

```
1  function partition(BlockBoundariesList)
2      if BlockBoundariesList ≠ ∅ then
3          wähle partitionPlane ∈ BlockBoundariesList als Ebene, die am nächsten
            zum räumlichen Median liegt
4          forall Block ∈ Blöcke do
5              if Block liegt links oder in der partitionPlane then
6                  leftList.append(Block)
7              end
8              if Block liegt rechts oder in der partitionPlane then
9                  rightList.append(Block)
10             end
11         end
12         partition(leftList)
13         partition(rightList)
14     end
15     else
16         return Blatt aus Blöcken
17     end
```

Damit ist die Stetigkeit, insbesondere an den Levelübergängen, gegeben, was zu einer glatten und knickfreien Darstellung führt.

Ein Nachteil dieser Methode ist, dass ohne eine effiziente Methode für die Bestimmung des gröbsten Levels, welches einen Einfluss auf P hat, unter Umständen viele teure Suchanfragen der dualen Zellen gemacht werden müssen. Ebenso bleibt die Interpolationseigenschaft nicht erhalten. Das bedeutet, dass an den Punkten, an denen die Werte bekannt sind, sich die rekonstruierten Werte von den echten Werten abweichen können.

Um die Zellabfragen schneller zu gestalten, implementieren die Autoren weitere Verbesserungen:

- Die Zellen werden mit den float Zenterpunktkoordinaten adressiert.
- N Anfragen werden parallel verarbeitet und gehen gemeinsam in die Baumstruktur rein.
- Abfragen für die dualen Zellen mit Hilfe von 6 Zustandsvariablen realisiert:
  Eine Zelle ist ein Schnitt aus 6 Ebenen. Die Zustandsvariablen geben an, welche Ebenen in einem Unterbaum aktiv sind. Sobald ein Blatt erreicht wurde, beschreiben diese Variablen, welche der Ecken der Zelle aktiv und mit den Blockinformationen aus dem Blatt zu befüllen sind.

Andere, rechenintensivere, Basisfunktionen können zum Beispiel inverse gewichtete kleinste Quadrate Funktionen sein. Denkbar wären ebenfalls alle typischen Funktionen, die bei der finiten Elemente Methode verwendet werden.

Leaf et al. [9] stellen eine Methode mit einem glatten Interpolanten für eine parallele Verarbeitung von großen AMR Daten auf mehreren GPUs vor. Sie verwenden die Idee von Ljung et al. [10] das ganze Gebiet in Blöcke zu zerteilen und fest zu halten, wie viele benachbarte Blöcke gebraucht werden, um einen Punkt mit Hilfe eines gewichteten Mittels zu samplen. Der Unterschied zu der Basisfunktionenmethode besteht darin, dass der Träger an den Grenzen der Blöcke endet.

3.2 Stitching

Das Verschieben des knotenzentrischen Gitters führt zu einem Spalt zwischen den Subgittern unterschiedlicher Level (vgl. Abbildung 2.4). In [23] wird von Weber et al. zum ersten Mal die Idee von *stitching cells* vorgestellt. Diese füllen die Lücke zwischen den dualen Gittern aus.

Da die Autoren sich auf die ursprüngliche Struktur von Berger und Colella [3] berufen, bleibt die Einschränkung des Levelunterschieds von eins noch erhalten.

Zunächst wird das duale Gitter mit den Stitchingelementen bestimmt und anschließend werden zu Visualisierung die Isoflächen aus den *marching cubes* Falltabellen von Lorensen und Cline [11] ausgewählt.

Die Basiselemente der Gitter im dreidimensionalen Raum sind Hexaeder, dreieckige Prismen, Pyramiden und Tetraeder. Die Verbindung der Gitter ist durch eine Falltabelle definiert. Dabei können sie mit einer Ecke, einem Segment einer Kante oder einer anderen Seitenfläche verbunden werden.
Diese zusätzlichen Elemente ermöglichen zwar eine glatte Interpolation, jedoch ist dessen Einsatz mit einem höheren Speicherverbrauch verbunden, da nicht mehr die implizite Topologie des regulären Gitters ausgenutzt werden kann: Die Eckpunkte müssen einzeln gespeichert werden.

Moran und Ellsworth [12] erweitern dieses Konzept, indem sie nicht mehr die 36 festen Fälle in einer Tabelle nachschlagen, sondern Verzerrungen einschließen und somit einen beliebigen Levelunterschied an den Gittergrenzen ermöglichen. Analog zum Paper gehen wir zunächst auf den Fall im zweidimensionalen Raum ein.

Das Bestimmen der Zelle, in der sich ein gegebener Punkt P befindet, geschieht mit der Punktesuchefunktion *point location*. Die Autoren suchen dazu das feinste Gitter $G_{l,k}$ mit $P \in G_{l,k}$ und geben die Zelle C in der P liegt zurück.

Zusätzlich verbessern sie die Punktesuche: Sie speichern das letzte betrachtete Subgitter und suchen zunächst dort nach dem P, da die aufeinanderfolgenden Zugriffe für die Punkte P mit hoher Wahrscheinlichkeit in der nahen Umgebung voneinander liegen werden. Falls P nicht in diesem Subgitter liegt, so gehen sie solange einen Level höher, bis sie das Subgitter gefunden haben oder schließen können, dass der Punkt außerhalb

des Gebiets liegt.

Eine Besonderheit ist die neue Methode, duale Zellen zu finden. Ein kleines Quadrat, ε-Quadrat, mit Kantenlänge $h_{\ell_{max}}$ wird mittig über eine Ecke v einer Zelle gelegt. Auf jede Ecke p_i vom ε-Quadrat wird *point location* angewandt. Diese Funktion sucht nach einer Zelle C des feinstmöglichen Levels, sodass $p_i \in C$. Die Mittelpunkte der vier Zellen sind dann genau die Ecken der dualen Zelle. Dabei werden sowohl gewöhnliche Zellen als auch die Stitchingelemente, *stitching cells*, erzeugt. Die Letzteren sind entweder Drei- oder Vierecke. Die Dreiecke entstehen dadurch, dass an den Levelübergängen v nur drei statt vier Nachbarzellen besitzt. Die degenerierten Kanten nennen Moran und Ellsworth dann *collapsed edges*.

Wir fassen es im Algorithmus 3 zusammen.

Algorithm 3: duale Zellen mit ε-Quadrat nach Moran und Ellsworth [12]

```
1  function findDualCell(v)
2      definiere einen ε-Quadrat mit v als dem Mittelpunkt
3      Menge der Zellen in der Nachbarschaft C_{1,...,4} = point location der Ecken vom
         ε-Quadrat
4      for alle Ecken vom ε-Quadrat do
5          p_{1,...,4} = pointLocation(Ecke)
6          D = Zelle mit den Mittelpunkten von C_{1,...,4} als Eckpunkte
7          if alle p_i sind unterschiedlich oder collapsed edges erlaubt then
8              return D
9          end
10         vergleiche Kanten von D
11         return D mit 3 unterschiedlichen Ecken
12     end
```

Wir betrachten wieder das Beispiel für ein duales Gitter aus Kapitel 2. In Abbildung 3.2 ergänzen wir die Stitchingelemente. Nun kann der Wert für P_3 mit Hilfe der Ecken interpoliert werden.

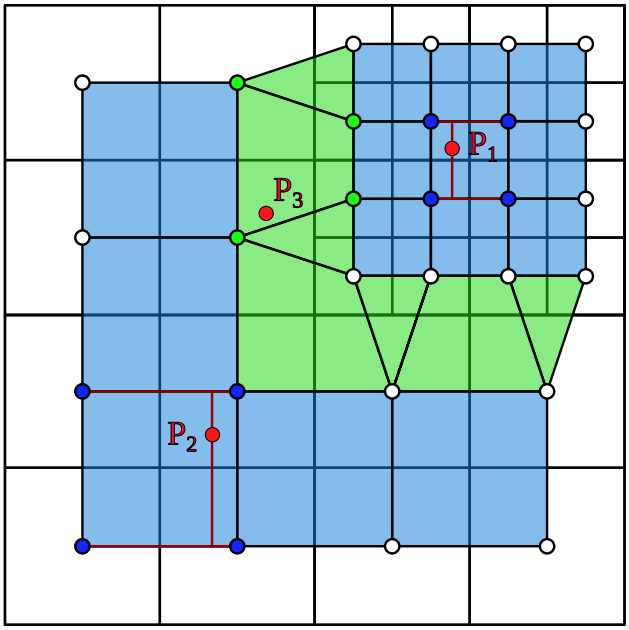


Abbildung 3.2: Beispiel für Stiching in 2D. Das duale Gitter in hellblau, Stitching in hellgrün. Der Punkt P_3 liegt nun in einer eindeutigen Zelle, hervorgehoben durch grüne Kreise. (In Anlehnung an Zellmann et al. [25])

Um es auf den dreidimensionalen Fall zu übertragen, müssen die Autoren zum einen statt einem ε-Quadrat einen ε-Würfel nehmen und zwischen mehr stitching cells Typen unterscheiden. Es gibt 7 Grundfälle:

- Typ H: 0 collapsed edges, alle 6 viereckigen Flächen bleiben erhalten (Hexaeder)
- Typ PW: 1 collapsed edge, 4 viereckige Flächen (Pyramide und Prisma)
- Typ W: 2 collapsed edges, 3 viereckige Flächen (dreckiges Prisma)
- Typ PP: 2 collapsed edges, 2 viereckige Flächen (zwei Pyramiden)
- Typ TT: 3 collapsed edges, 2 viereckige Flächen (zwei Tetraeder)

- Typ P: 4 collapsed edges, 1 viereckige Fläche (Pyramide)
- Typ T: 5 collapsed edges, keine viereckigen Flächen (Tetraeder)

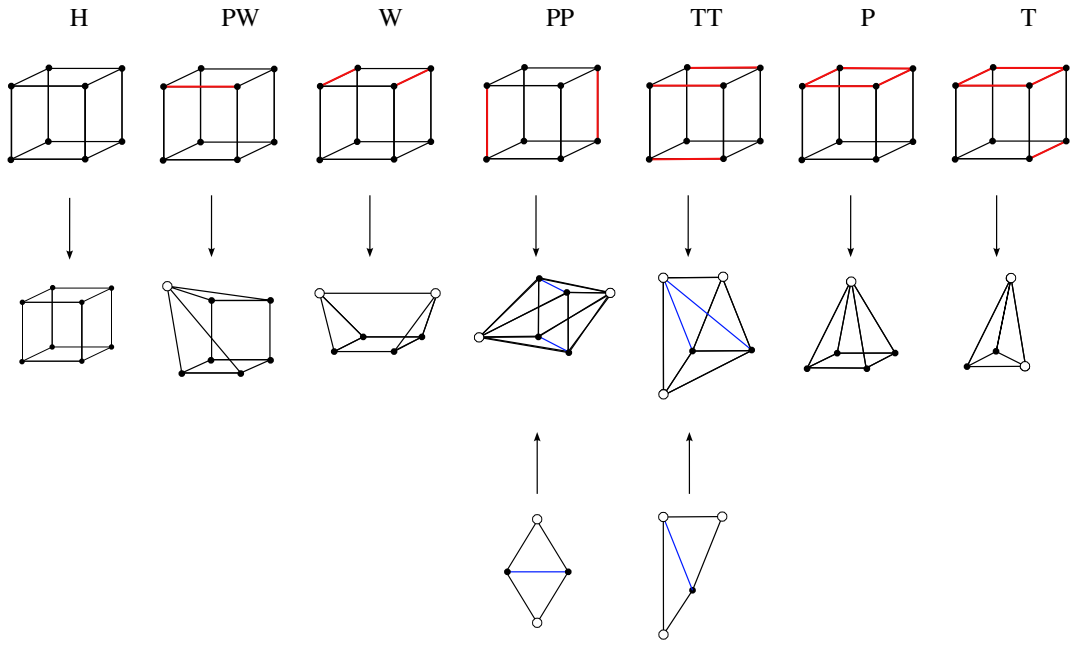


Abbildung 3.3: Alle möglichen Stitchingelemente (bis auf die nicht an den Achsen ausgerichteten und nichtplanaren Fälle sowie Drehungen). Die roten Kanten in der ersten Reihe sind degenerierte Kanten. Die zweite Reihe zeigt die neuen Elemente. Die weißen Kreise sind degenerierte Flächen und in blau sind die vom Algorithmus ausgewählten Diagonalen. Die dritte Reihe verdeutlicht die Wahl. (In Anlehnung an Moran und Ellsworth [12].)

Durch die zusätzliche Dimension können duale Zellen mit nichtplanaren Flächen entstehen. Die Entscheidung, Hexaeder weiter zu zerteilen, hängt von der weiteren Verarbeitung ab: Sind collapsed edges und nichtplanare Flächen nicht erlaubt, so müssen diese zerteilt werden. Dazu betrachten wir die Ecken der viereckigen Seitenflächen und definieren alle möglichen Diagonalen. Die Autoren wählen die Diagonale, die die Ecken enthält, die das größte r haben. Falls es mehrere solcher Diagonalen gibt, so entscheiden sie sich für diejenige, die gleichzeitig das größte Index in der Liste besitzt. Abbildung 3.3

zeigt die neu entstandenen Grundformen sowie die zu wählenden Diagonalen.

Durch das Ausprobieren aller Fälle haben die Autoren festgestellt, dass es immer Diagonalen gibt mit, dessen Hilfe eine Zerlegung in Tetraeder möglich ist. Anschließend codieren sie die duale Zelle und die ausgewählte Diagonale in eine Tabelle.

Gegeben einen Punkt P wollen sie nun die entsprechende duale Zelle finden. Mit der Funktion *point location* finden die Autoren eine solche Zelle C, sodass $P \in C$. Sie wählen eine Ecke v, die am nächsten zu P liegt. Dies bestimmen sie mit dem Parameter ξ_c, den sie mit der point location erhalten haben. Anschließend gehen sie sukzessive über alle dualen Zellen, bis sie eine duale Zelle D, sodass $P \in D$ gilt, gefunden haben. Wir fassen es im Algorithmus 4 zusammen.

Algorithm 4: Suche nach einer dualen Zelle nach Moran und Ellsworth [12]

```
1  function findDual(P, G)
2      C, ξ_c = pointLocation(P, G)
       // Wählen der initialen Ecke
3      Wähle Ecke v ∈ C mit Abstand von P zu v minimal mit Daten aus ξ_c
       // Finden der dualen Zelle
4      D = duale Zelle zu v mit der ε-Quadrat Methode
5      while P nicht in D do
6          for Flächen f = 0, ..., 5 von D do
7              berechne orient(P, D, f)
8              if P außerhalb von D then
9                  v = nächste Ecke in der Nachbarschaft entsprechend v, f, D
10                 D = duale Zelle von v
11             end
12         end
13     end
14     return D
```

Dabei ist *orient* eine Funktion, die mit Hilfe von Flächen f_i von D bestimmt, ob P in D liegt, indem es die Lage von P relativ zu f_i ermittelt. Veranschaulicht können wir die Fläche als eine Art Trennebene betrachten.

Es gibt drei mögliche Ausgaben von *orient*:

- -1: falls P nicht in D, sondern auf der anderen Seite der Ebene f_i liegt,
- 0: falls P dazwischen liegt. Es ist auch der Fall, wenn f_i komplett eingebrochen ist,
- 1: falls P in D liegt.

Anschließend bestimmen die Autoren das ε_d für die Interpolation, welches die Position von P in D beschreibt. Dazu minimieren sie den Abstand $|f(\varepsilon_d) - P|$ mit dem Newtonverfahren, wobei

$$\begin{aligned} f(\varepsilon) =& (1-\varepsilon_0)(1-\varepsilon_1)(1-\varepsilon_2)f_0 \\ &+ \varepsilon_0(1-\varepsilon_1)(1-\varepsilon_2)f_1 \\ &+ (1-\varepsilon_0)\varepsilon_1(1-\varepsilon_2)f_2 \\ &+ \varepsilon_0\varepsilon_1(1-\varepsilon_2)f_3 \\ &+ (1-\varepsilon_0)(1-\varepsilon_1)\varepsilon_2 f_4 \\ &+ \varepsilon_0(1-\varepsilon_1)\varepsilon_2 f_5 \\ &+ (1-\varepsilon_0)\varepsilon_1\varepsilon_2 f_6 \\ &+ \varepsilon_0\varepsilon_1\varepsilon_2 f_7 \end{aligned} \tag{3.4}$$

mit $f_j,\ j \in \{0, \ldots, 7\}$ den Koordinaten der Ecken von D.

Eine Fallunterscheidung zwischen dualen und Stitchinggebieten führt zu einem effizienteren Algorithmus:

Moran und Ellsworth ergänzen dazu das Markieren der Nachbarschaft von v bei der *point location*, wenn sie feststellen, dass die Hexaeder in der Nachbarschaft nicht in dem gleichen Gitter liegen. Das bedeutet, dass v im Stitchinggebiet liegen muss.

Sie fahren gleich, wie oben im Algorithmus 4 beschrieben, fort. Sonst liegt v im dualen Gitter und die Zeilen 5-13 können übersprungen werden.

Der wesentliche Vorteil ist, dass das ε_{d_i} ohne das teurere Newtonverfahren berechnet werden kann - sie addieren zum ε_{c_i} 0.5, falls $\varepsilon_{c_i} < 0.5$ und subtrahieren im sonstigen Fall 0.5.
Den Wert von P bekommen sie, indem sie mit Hilfe von ε_d die Werte an den Ecken von D berechnen und diese in 3.4 einsetzen.

In Wald [17] stellt ein vereinfachtes Verfahren vor, welches unabhängig entwickelt, jedoch ähnliche Grundgedanken wie die gerade vorgestellte Methode hat. Durch die neue Struktur wird der Algorithmus parallelisierbar. Eine entsprechende Implementierung auf der GPU ist in [17] beschrieben und verlinkt.

Der Autor führt den Begriff *snap* ein, welcher das Analogon zum Anheften der Ecken des ε-Würfels an die Mittelpunkte der Zellen, in denen die Zellen liegen, darstellt. Alternativ interpretiert, ist es die Methode *logische duale Zellen* an die *realen Zellen* an zuheften. Das Ergebnis ist die gleiche Zerlegung wie bei Moran und Ellsworth. Die Funktion *snap(C)* sucht nach der zugehörigen realen Zelle $\hat{C}$ zu gegebenen Koordinaten (i, j, k). Dazu sortiert der Autor zuerst die Liste mit allen Zellen nach ihren Integerkoordinaten und berechnet das maximale Level. Mit Hilfe der binären Suche findet er dann die reale Zelle (auf dem gleichen, feineren oder gröberen Level).

Für die Punktelokation kann dann wie folgt verfahren werden:
Er projiziert auf allen Leveln die Koordinaten (i, j, k) auf die Koordinaten der auf dem jeweiligen Level zugehörigen Level-ℓ Zelle. Dazu setzt er die ℓ letzten bits von (i, j, k) auf Null. Die binäre Suche gibt dann entweder die gesuchte Zelle zurück oder die Meldung, dass diese nicht existiert.

Das Erzeugen des dualen Gitters basiert auf der Beobachtung, dass die Mittelpunkte der dualen Zellen die Ecken der ursprünglichen realen Zellen sind. Das heißt, dass es reicht, ein Mal über die realen Zellen zu iterieren, um dessen duale Zellen zu bestimmen.

Es wird allerdings unmittelbar zu mehrfacher Erzeugung gleicher dualen Zellen kommen. Um dem entgegenzuwirken, führt der Autor drei Regeln zum Verwerfen der dualen Zellen ein:

- Findet sich keine reale Zelle an die eine Ecke einer dualen Zelle D angeheftet werden kann, dann besitzt D keine zugelassene Form. (nicht valide Elemente aussortieren)
- Gibt es eine Ecke von D, die sich an ein feineres Level anheftet, so muss diese nicht mehr beachtet werden, da sie von einem feineren Level erzeugt wird. (feinere Zellen sind bevorzugt)
- Heftet sich eine Ecke von D an eine Zelle, die das gleiche Level wie D hat, aber die Koordinaten niedriger sind, dann muss diese nicht betrachtet werden, da es von einer zuvor gegangener dualen Zelle erzeugt wird. (Reihenfolge)

Es fällt auf, dass zum Verwerfen der dualen Zellen keine Kommunikation mit den anderen dualen Zellen stattfinden muss. Da die Zellen nach dem Sortieren unabhängig voneinander bearbeitet werden können, ist dieser Algorithmus parallelisierbar - die Berechnungen können auf eine Grafikkarte übertragen werden. Dadurch ist es im Vergleich zu der vorher vorgestellten Methode von Moran und Ellsworth effizienter im Hinblick auf die Laufzeit. Ebenfalls positiv wirkt sich die Tatsache aus, dass auf die direkten Berechnungen mit floating points verzichtet werden kann und keine sonstigen Hierarchien benutzt werden müssen.

Der komplette Algorithmus:

Algorithm 5: Erzeugen des dualen Gitters nach Wald [17]

```
1  void snap(i, j, k)
2      binarySearch(i, j, k und Level ℓ, nach int Koordinaten sortiertes Inputarray)
3      if eine Zelle gefunden then
4          return Ĉ mit Koordinaten i', j', k' und Level ℓ
5      end
6      else
7          return ∅
8      end
9  void doCell(i, j, k, ℓ)
10     erzeuge Zelle C mit Koordinaten i, j, k und Level ℓ
11     for alle 8 Ecken do
12         doDualCell(C, i, j, k, ℓ)
13     end
14 void doDualCell(C, i, j, k, ℓ)
15     Cell vertex = [2][2][2][ for alle Ecken von C do
           // Erzeuge eine temporäre duale Zelle D aus der Ecke
16         D = snap(Ecke)
17         if eine der 3 Regeln ist für eine Ecke von C verletzt then
18             return
19         end
20         speichere Ecke in vertex
21     end
22     return duale Zelle D aus Ecken in vertex
```

In Kapitel 4 gehen wir darauf ein, wie die Verfahren aus diesem Unterkapitel erweitert werden können.

3.3 Oktantenmethode

Die Oktantenmethode von Wang et al. [20] verzichtet auf die Einführung von zusätzlichen nicht quadratischen Elementen, schafft es dennoch eine stetige und knickfreie Geometrie aus den blockartig strukturierten AMR Daten zu erzeugen. Das gelingt unter anderem dadurch, dass an den Levelübergängen die feineren Punkte mit Hilfe der gröberen Zellen bestimmt werden.

In den vorherigen Kapiteln haben wir gesehen, dass die Berechnung von dualen Zellen aufwendig ist. Die Interpolationsfilter bei dieser Methode verarbeiten nicht die dualen Zellen direkt, sondern verwenden diese implizit mit den Oktanten:

Definition 6 (Oktant)
Sei C eine Zelle. Für den Mittelpunkt von C mit Wert C_v schreiben wir $O^{(0)}$. Zerteilen wir diese in 8 gleich große Würfel O_i, $i \in \{1, \ldots, 8\}$ entlang der Hauptachsen, so nennen wir jeden dieser Würfel einen Oktanten*. Je nach Achsenrichtung der Ecken benennen wir von $O^{(0)}$ ausgehend die restlichen Ecken der Oktanten: Von $O^{(0)}$ in Richtung X liegt $O^{(X)}$, in Richtung XY liegt $O^{(XY)}$ und in Richtung XYZ liegt $O^{(XYZ)}$.*

Analog benennen wir die benachbarten Zellen mit beispielsweise $C^{(X)}$ den Nachbarn von $C^{(0)}$ in X Richtung.

Für diesen Abschnitt benutzen wir außerdem eine ähnliche Notation für duale Zellen. Dabei gelte $D^{(0)} = O^{(0)}$.

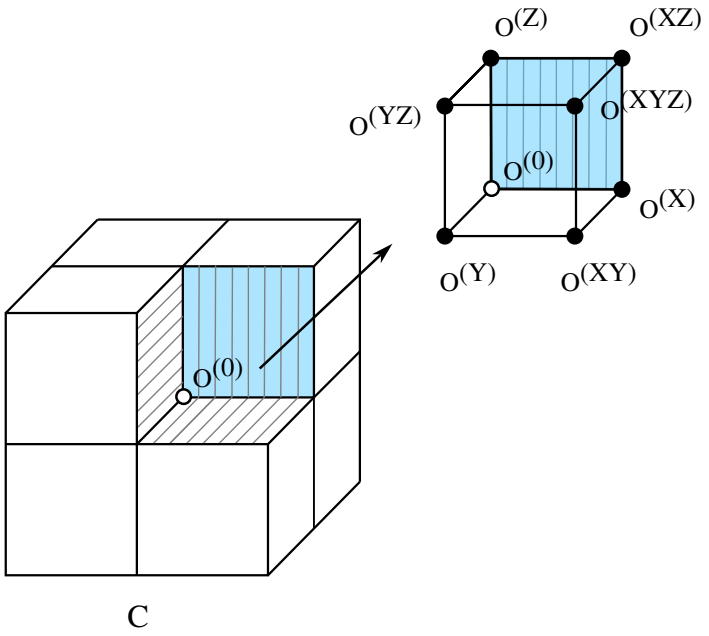


Abbildung 3.4: Eine Zelle C und die Notation für einen der 8 Oktanten als Ausschnitt aus der Zelle. Der Kern $O^{(0)}$ ist als ein weißer Kreis dargestellt. (Eigene Darstellung)

Die Autoren berechnen die Ecken eines Oktanten mit Hilfe einer dualen Zelle. Aus der Definition 6 ergeben sich folgende Beobachtungen, die gleichzeitig die Regeln für die Konstruktion des Interpolanten beschreiben:

1. $O^{(0)}$ hat den Wert des Zellkerns von C.

2. Es gilt $O_v^{(X)} = \frac{1}{2}(C_v^{(0)} + C_v^{(X)})$, da es genau zwischen $C^{(0)}$ und der benachbarten Zelle $C^{(X)}$ in X Richtung liegt und folglich $O_v^{(X)} = \frac{1}{2}(D_v^{(0)} + D_v^{(X)})$, da $O^{(0)}$ von $C^{(X)}$ dem $D^{(X)}$ entspricht.

3. $O^{(XY)}$ ist der Mittelwert von den Zellen $C^{(0)}, C^{(X)}, C^{(Y)}$ und $C^{(XY)}$, da es genau in der Mitte der von diesen Punkten aufgespannten Ebene liegt.

4. $O^{(XYZ)}$ ist der Mittelwert aller Ecken von D, da es genau im Kern von D liegt.

Die Regeln 2-4 verdeutlichen wir in der Abbildung 3.5.

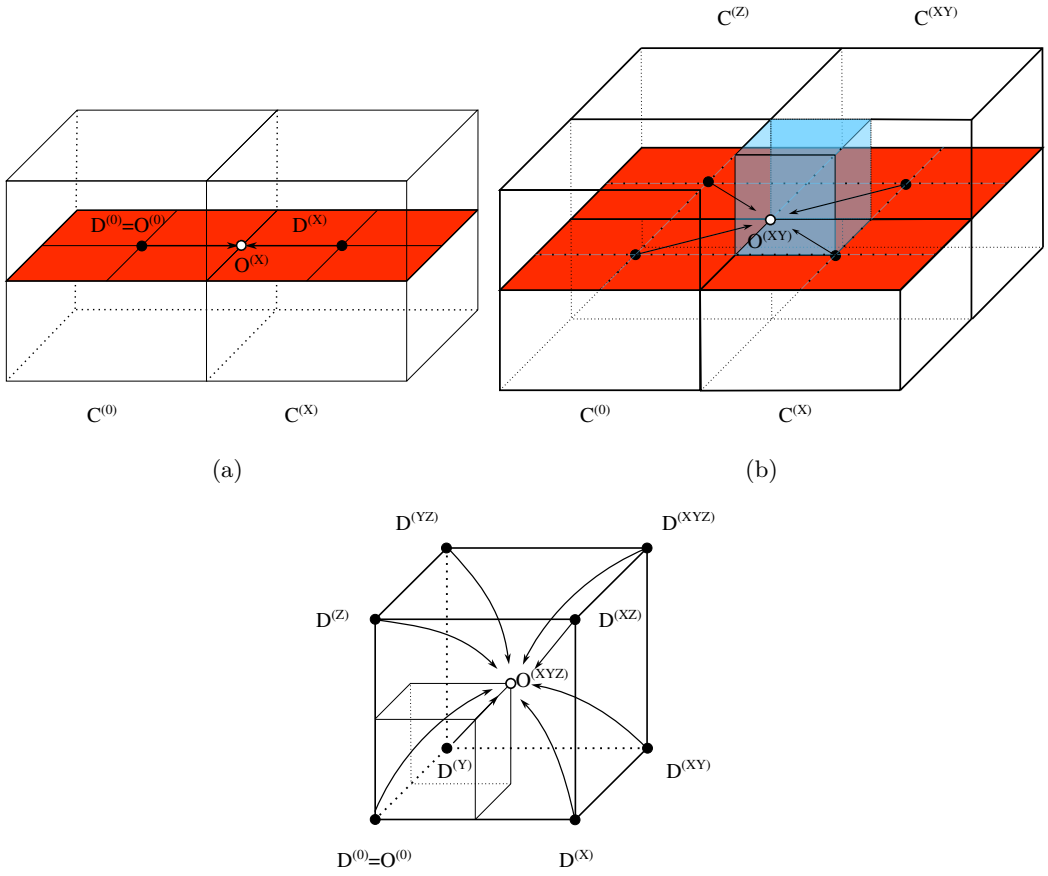


Abbildung 3.5: Der zu bestimmende Wert ist als weißer Kreis markiert. Die Punkte, die zur Berechnung beitragen, sind schwarze Kreise.
(a) $O^{(X)}$ liegt genau zwischen der Kerne von den Würfeln $C^{(0)}$ und $C^{(X)}$ und ist demnach der Mittelwert von $D^{(0)}$ und $D^{(X)}$. In rot ist der Schnitt in der mittleren Ebene hervorgehoben.
(b) $O^{(XY)}$ in dem Oktanten, hier in blau, kann aus den Kernen der Würfel $C^{(0)}, C^{(X)}, C^{(Y)}$ und $C^{(XY)}$ berechnet werden.
(c) $O^{(XYZ)}$ ist der Mittelwert aus den Ecken einer dualen Zelle in der der zugehörige Oktant liegt. (Eigene Darstellung)

Wang et al. fangen damit an, dass sie für einen gegebenen Punkt P dessen Blattzelle C und den zugehörigen Oktanten bestimmen und eine duale Zelle D drüber legen. Nach Regel 1 können sie direkt den Wert $O_v^{(0)} = C_v$ setzen. Für das Berechnen der restlichen Ecken des Oktanten muss erst herausgefunden werden, ob es sich um einen Punkt in einem Stitchinggebiet handelt. Dazu unterscheiden sie folgende Fälle:

1. Alle Ecken von D liegen auf dem gleichen Level wie C: P liegt nicht in einem Stitchinggebiet und sie können aus den oben aufgelisteten Regeln die Ecken des Oktanten ableiten.

2. Mindestens eine Ecke von D liegt auf der gröberen Seite: P liegt in einem Stitchinggebiet auf der feineren Seite. Sie definieren $P' = P + \varepsilon$ mit ε einem kleinen Wert so, dass P' auf der gröberen Seite liegt und bestimmen den neuen Oktanten für P'. Sie wiederholen das Verschieben rekursiv, bis sie auf der gröbsten Seite liegen. Sie bestimmen dann die fehlenden Ecken mit einer beliebigen Methode. (Näheres dazu später.)

3. Sonst liegt der Punkt in einem Stitchinggebiet auf der gröberen Seite und sie können selbst wählen, wie sie die Ecken bestimmen. (Näheres dazu später.)

Zum Schluss interpolieren sie den Wert von P aus den Werten der Ecken von O.

Für das Berechnen des Interpolanten gibt es keine feste Vorgaben, die Autoren schlagen vier Methoden vor: Verblenden mit dem feinsten, gröbsten oder dem aktuellen Level oder mit Hilfe der Basisfunktionen (siehe Kapitel 3.1).

Verblenden mit dem feinsten Level: Berechnen das gewichtete Mittel über alle Blattzellen, die den Punkt schneiden. Diese Methode wird von den Autoren als die beste und einfachste in der Praxis bezeichnet.

Verblenden mit dem gröbsten Level: Eine trilineare Interpolation auf dem gröbsten Level. Dabei kann das Verschieben um ε ausgelassen werden.

Verblenden mit dem aktuellen Level: Sie finden zuerst die Blattzelle und wenden darauf die trilineare Interpolation an.

Algorithm 6: Oktantenmethode nach [20]

float *octant(P)*

oct = Oktantenblattzelle mit $P \in oct$

D = Duale Zelle gegeben den Oktanten

`// Wert von` $O^{(0)}$ `des Oktanten -> Regel 1`

$oct[0].v = C_v$

`// Werte der Ecken an Kanten von` C

$\ell Xmin = \min(D[0].level, D[X].level)$

$\ell Xmax = \max(D[0].level, D[X].level)$

if $\ell Xmin == \ell Xmax$ **then**

`// nicht am Stitchinggebiet -> Regel 2`

$oct[X].v = \frac{1}{2}(D[0].v + D[X].v)$

end

else if $\ell Xmin < oct.level$ **then**

`// feineres Gebiet, verschiebe` P `um` ε `in die gröbere Richtung und berechne rekursiv`

$oct[X].v = octant(oct[X].p + \varepsilon * oct.dX)$

end

else

`// gröberes Gebiet`

Interpoliere $oct[X].v$ mit einer beliebigen Methode

end

`// Werte der Ecken an den Seitenflächen von` C `-> (ggf) Regel 3`

$\ell XYmin = \min(D[0].level, D[X].level, D[Y].level, D[XY].level)$

`// Rest analog`

`// Wert von` $O^{(XYZ)}$ `des Oktanten -> (ggf) Regel 4`

$\ell XYZmin =$
$\min(D[0].level, D[X].level, D[Y].level, D[XY].level, D[YZ].level,$
$D[XZ].level, D[XYZ].level)$

`// Rest analog`

Anmerkung: Das Level 0 ist in diesem Algorithmus das gröbste Level.

Anschließend wird ähnlich wie in Kapitel 3.1 ein k-d Baum erzeugt.

Ein Nachteil dieser Methode ist, dass das häufige Nachschauen der Zellpositionen relativ teuer ist. Die Autoren weisen ebenfalls auf das Problem der möglichen Rundungsfehler an den Levelübergängen hin, da der Zugriff nicht in derselben Reihenfolge erfolgt.

Wang et al. [21] haben die Idee von Oktanten benutzt, um neben den block structured AMR Daten auch tree-based AMR verarbeiten zu können. Dazu sortieren sie die Zellen in einen dünnbesetzen Octree (*sparse octree*).

3.4 ExaBrick

Die k-d Bäume aus dem letzten Kapitel haben den Nachteil, dass sie für jeden Sample diese erneut durchlaufen müssen. Die wiederholten Traversierungen zu minimieren, entwickeln Wald et al. [19] eine Methode, bei der die Nachbarschaftsbeziehungen effizienter benutzt werden können. Ihr Verfahren ist für verschiedene AMR Typen anwendbar, da die Skalare getrennt von der ursprünglichen Datenstruktur und neu sortiert in einem gemeinsamen Array gespeichert werden.

Das Verfahren lässt sich in folgende Schritte einteilen, die wir später im Detail erläutern werden:

1. Das Verbinden der Zellen vom gleichen Level in disjunkte *Bricks*.
2. Das Bestimmen der Umgebung *ABR* (Active Brick Regions), die die Rekonstruktionsfilter der einzelnen Bricks decken und einer Datenstruktur, die den Raum trennt und dessen zugehörige Blätter diejenigen Bricks enthalten, die einen Einfluss auf das jeweilige Gebiet haben.
3. Das Erzeugen einer BVH (bounding volume hierarchy) über die ABR.

Wir betrachten nun die einzelnen Datenstrukturen.

Definition 7 (Brick)
Ein Brick *ist ein dreidimensionales Gitter, welches die untersten beziehungsweise die kleinsten Eckkoordinaten, das Level und die Anzahl der Zellen in alle Richtungen beinhaltet. Das Maximum in jeder Dimension sei auf 32 gesetzt. Dabei seien alle Zellen in einem Brick vom gleichen Level.*

Die Grundidee, Blöcke an Daten beziehungsweise Bricks zu verwenden, ist ebenfalls in früheren Veröffentlichungen bei Kähler et al. [8] und Kähler und Abel [7] zu finden. Wie in der obigen Definition lassen sie innerhalb eines Bricks ausschließlich Zellen vom gleichen Level zu und verwerfen die ursprüngliche Struktur aus den Inputdaten. Dafür gehen sie ausgehend vom gröbsten Level (hier 0) wie folgt vor:

1. Die *bounding box* B_0, welche alle Gitter enthält, ist der Wurzelknoten.
2. Durch Trennebenen wird B_0 rekursiv bei jedem Schritt in zwei Knoten zerteilt, bis in einem Knoten nur noch Zellen eines Levels sind. Die Trennebene wird so gewählt, dass der Schnitt mit den bounding boxes der Untergitter dieses Levels minimal ist und das Verhältnis der Zellen links und rechts möglich symmetrisch ist.
3. Für jeden Blattknoten wird eine Liste aus allen Subgittern, die den jeweiligen Bereich überdecken, gebildet. Wenden Schritt 2 auf jeden der neuen Blätter an, indem die Trennebenen als Schnitte zwischen den Subgittern und den Blättern berechnet werden.
4. Zuletzt werden Schritte 2 und 3 für die restlichen Level wiederholt.

Das Erzeugen der Bricks bei Wald et al. [19] geschieht ebenfalls mit einem top-down k-d-Baum, aber mit einer einfacheren Trennregel: Es wird entlang der längsten Achse so lange innerhalb der Umbox gesplitted, bis in dem jeweiligen Blatt alle Zellen das gleiche Level haben und die Umbox vollständig ausfüllen.

Um das Zerteilen der Zellen zu verhindern, muss die Teilungsposition auf die Größe des aktuell gröbsten Levels gerundet werden.

Anschließend verwerfen die Autoren die Struktur und behalten ausschließlich die Blätter.

Als Rekonstruktionsmethode wählen die Autoren die Basismethode unter Verwendung der Zeltbasisfunktionen, die wir bereits in Kapitel 3.1 behandelt haben.

Sie stellen eine Verbesserung davon vor, indem sie auf das Problem der häufigen Baumtraversierungen eingehen: Im Allgemeinen kann es passieren, dass die Zellen, die einen Sample beeinflussen, in unterschiedlichen Bricks liegen können. Obwohl die Zellen dann benachbart sind, muss erst über die Baumstruktur in den nächsten Brick navigiert werden.

Die Autoren berechnen deshalb eine Hifsstruktur, die *ABR*. Diese beschreibt für jedes Gebiet, welche Bricks es beeinflussen. Haben sie eine Liste an solchen Bricks, so bleibt nur noch über diese zu iterieren und die den Sample beeinflussenden Zellen an die Basismethode zu übergeben.

Zuvor eine weitere Definition zur Konstruktion:

Definition 8 (Träger)
Sei C eine Zelle und $\hat{H}_C(P)$ ihre Basisfunktion. Der Träger *(auch* support*) von C ist ein Gebiet für den $\hat{H}_C(P) > 0$ gilt. Für ein Brick B definieren wir den Träger als die Vereinigung aller Träger von den Zellen, die in B liegen:*

$$\text{support}(B) = \bigcup_i \text{support}(C_i),\ C_i \in B$$

Aus der Definition der Zeltbasisfunktionen folgt, dass der Träger von Zellen Quader sind, die in jeder Dimension um eine halbe Zellenbreite über die Zelle hinausragen. Dementsprechend hat der Träger eines Bricks die gleiche Form. Notwendigerweise werden sich die Träger überschneiden. Die Autoren zerteilen die Bereiche in Quader soass diese für eine gegebene Zelle eine Liste von Kandidaten zurück geben können, in welchen Bricks die vorgegebene Zelle liegen könnte.

Definition 9 (ABR)

Ein ABR *ist eine Vereinigung aus Trägern:*

$$\mathrm{ABR}_i = \bigcup_{j=1}^{n} \mathrm{support}(B_j)$$

so, dass für alle Zellen $C \in \mathrm{ABR}_i$*,* C *ausschließlich in dem* ABR_i *mit den maximalen* n *liegt. Die entsprechenden IDs der Bricks für jedes* ABR_i *mit dessen bounding boxes sind in einer Liste gespeichert.*

Die Konstruktion von *ABR* fassen wir im Algorithmus 7 zusammen.

Ein Vorteil von der ExaBricks Methode ist, dass *empty space skipping* ohne eine zusätzliche Struktur möglich ist. Die Bereiche ohne Zellen werden dabei übersprungen, da für diese gar nicht erst Bricks gebildet werden. Zusätzlich werden Metadaten über den jeweiligen Bereich gespeichert: die minimalen und maximalen Skalarwerte innerhalb des *ABR*. Ist die daraus resultierende maximale Opazität gleich Null, so kann das ganze ABR verworfen werden.

Für eine *adaptive sampling rate* wird an dieser Stelle die feinste Zellenbreite gespeichert und im sampling Schritt verwendet.

Um schnell durch die Bricks iterieren zu können, erzeugen die Autoren einen BVH aus den durch die Blätter beschriebenen Gebiete. Sie verwenden die RTX Hardware auf der GPU, indem sie Primitive in der gleichen Anzahl wie ABR erzeugen und daraus die Hierarchie bauen. Da die Strahlen auf der Hardware billig sind, gehen sie Autoren bei der Traversierung wie folgt vor: Zuerst definierten sie einen Suchintervall $[0, \ldots, t_{max})$. Falls keine Region gefunden wurde, brechen sie ab, sonst berechnen sie den Intervall mit Ein- und Austritt $[t_{in}, t_{out}]$, schicken einen neuen Strahl ab $t_{out} + \varepsilon$ und wiederholen es so lange bis der Strahl opak wird oder keine Weitere gefunden wurde.

Algorithm 7: Erzeugen von ABR nach [19]

```
1  function makeABR(Bricks)
2      forall Brick ∈ Bricks do
3          fragment = (support(Brick), Brick.id)
4          fragmentList.append(fragment)
5      end
6      partition(fragmentList)
7  function partition(fragmentList)
8      partitionPlane = Grenzfläche aller fragments entlang der aktuell breitesten
         Dimension, die am nächsten an der Mitte der aktuellen Region liegt
9      if partitionPlane = ∅ then
10         erzeuge ein Blatt aus fragmentList.Brick.ids
11         return Blatt
12     end
13     else
14         trenne entlang partitionPlane in leftFragmentList und rightFragmentList
15         partition(leftFragmentList)
16         partition(rightFragmentList)
17     end
```

Die Abbildung 3.6 veranschaulicht das beschriebene Vorgehen grafisch.

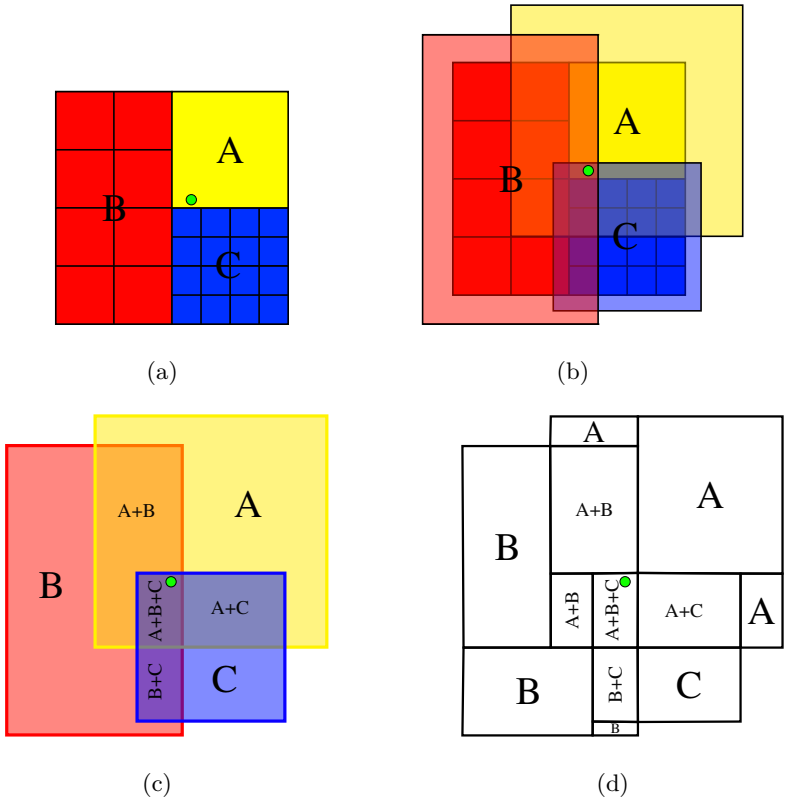


Abbildung 3.6: (a) Bricks A, B, C mit drei unterschiedlichen Levels. Ein Punkt, als grüner Kreis dargestellt, liegt in Brick A.

(b) Die Träger der einzelnen Bricks sind halbtransparent in den jeweiligen Farben dargestellt.

(c) Die Überschneidungen der Träger. Der grüne Punkt P liegt in allen drei Träger.

(d) Die einzelnen Gebiete werden noch weiter zerteilt, sodass daraus rechteckige Elemente entstehen - diese stellen die komplette Gebietszerlegung in ABR dar. P liegt im ABR $A + B + C$. Das heißt, es wird von allen drei Bricks beeinflusst. (In Anlehnung an Wald et al. [19])

Während das Ergebnis von ExaBricks eine glatte Interpolation liefert, ist ein Nachteil dieser Methode, dass unter Umständen viele Bricks erzeugt werden, die aus einer einzigen Zelle bestehen (Zellmann et al. [24]). Dies ist dann der Fall, wenn es starke Schwankungen im Level zwischen der naheliegenden Zellen gibt. Das führt dazu, dass die eigentlichen Vorteile der Bricks nicht ausgenutzt werden können. Im nächsten Kapitel 4 werden wir uns eine Methode anschauen, die es verhindert.

In [24] analysieren die Autoren die *ExaBricks*-Struktur und erweitern diese. Die Autoren gehen das Problem an, dass bei großen Datenmengen die Visualisierung von AMR teuer ist und der Speicher weder in den aktuellen GPUs noch in DDR dafür ausreicht, mehrere Zeitschritte interaktiv anzuzeigen. Sie verwenden einen *streaming* Ansatz und fangen mit der naiven Implementierung an: Jeder Zeitschritt wird als separates Eingabedatensatz angesehen. Die Schritte, die bei der naiven Implementierung jedes Mal neu ausgeführt werden, sind:

1. Input/Output der Daten
2. Umstrukturieren der mehrdimensionalen Zellennummern *cellID* ins Eindimensionale
3. Kopien zwischen host und device
4. Berechnen von maximalen und minimalen Skalare für space skipping
5. Generierten einer BVH Struktur
6. Rendern.

Diese Schritte optimieren die Autoren wie folgt:

1. Verstecken der I/O Latenszeit durch *double buffering* - Vorzeitiges laden der Skalare für $t+1$, während andere Schritte des vorausgegangenen Zeitschritts t auf der GPU ausgeführt werden.

2. Benutzen der *page-locked memory* für das Speichern der eindimensionalen cellIDs.

3. Benutzen der I/O, welches *page cache* vermeidet, um sie synchronen Operationen darauf zu eliminieren.

4. Umordnen der Skalarfelder nach den cellIDs. Eindeutigkeit ist dadurch gegeben, dass cellIDs eins zu eins auf die Skalare abgebildet werden.

5. Verändern des BVH anstelle einer kompletten neuen Generierung bei jedem Zeitschritt. Modifikationsgrund des einmalig erzeugten BVH ist dann die Veränderung in der Sichtbarkeit.

6. Verstecken der Datentransferzeit von der CPU zur GPU durch das parallele Ausführen des Renderingschrittes.

Allerdings unterstreichen Zellmann et al., dass viele der oben genannten Verbesserungen nur für Datensätze möglich sind, die einen ähnlichen Aufbau wie das von den Autoren betrachtete Modell haben: Die Topologie darf sich mit der Zeit nicht verändern.

Insgesamt ergibt sich ein Framework für große Datensätze, welches interaktiv ist und glatte Interpolationen sichert.

4 Gridlets - Erzeugung und Verarbeitung

Diese Arbeit basiert auf der Grundlage von Zellmann et al. [25]. Die Autoren optimieren die Idee von Moran und Ellsworth [12] (Kapitel 3.2) im Hinblick auf die Speichereffizienz. Ihre Strategie ist es, Stitching zu verwenden und im Bereich außerhalb der Verbindungselemente die würfelartigen Zellen mit Hilfe der *Gridlets* zu bündeln.

4.1 Grundlagen

Definition 10 (Gridlets)
Ein Gridlet ist ein kartesisches Gitter einer festen Größe $N \times M \times K$ *mit* $N, M, K \in \mathbb{N}$. *Alle Zellen in einem Gridlet haben das gleiche Level. Die Datenpunkte liegen an den Ecken, das heißt* $(N+1) * (M+1) * (K+1)$ *Skalarindizes werden in einem Gridlet gespeichert. Dabei können die Zellen auch virtuell sein - den Skalarindizes solcher leeren Zellen weisen wir dann den Wert -1 zu.*

Die Wahl von den Dimensionen ist ein Kompromiss zwischen einer feineren Auflösung und geringeren Kosten. Die Autoren berichten, dass für ihre Daten $N = M = K = 8$ die besten Ergebnisse erzielt hat.

Ein essenzieller Vorteil von Gridlets ist, dass die teuren Traversierungen von BVH bei nebeneinanderliegenden Samplepunkten in bestimmten Fällen umgehen werden können. Die Autoren speichern bei einer *null collision* die ID des Gridelts und überprüfen dann bei dem nächsten Sample, ob dieser zu dem gleichen Gridlet gehört. Ein Sample vom BVH ist in diesem Fall nicht mehr nötig.

Durch die feste (kleinere) Größe der Gridlets vermeiden wir ebenso die lang gezogenen sowie eine Mehrzahl an Bricks, die aus einer einzigen Zelle bestehen (Zellmann et al. [19]), und die Laufzeit wesentlich beeinflussen (Zellmann et al. [24]).

Die gesamte Methode von Daten bis zum Rendering teilen die Autoren in drei Etappen auf:

1. Erzeugen des dualen Gitters aus den AMR Daten und anschließende Generierung von Gridlets.
2. Erzeugen eines uniformen Gitters aus Makrozellen und einer BVH für die Hierarchie beziehungsweise die Nachbarschaftsbeziehungen.
3. Rendern.

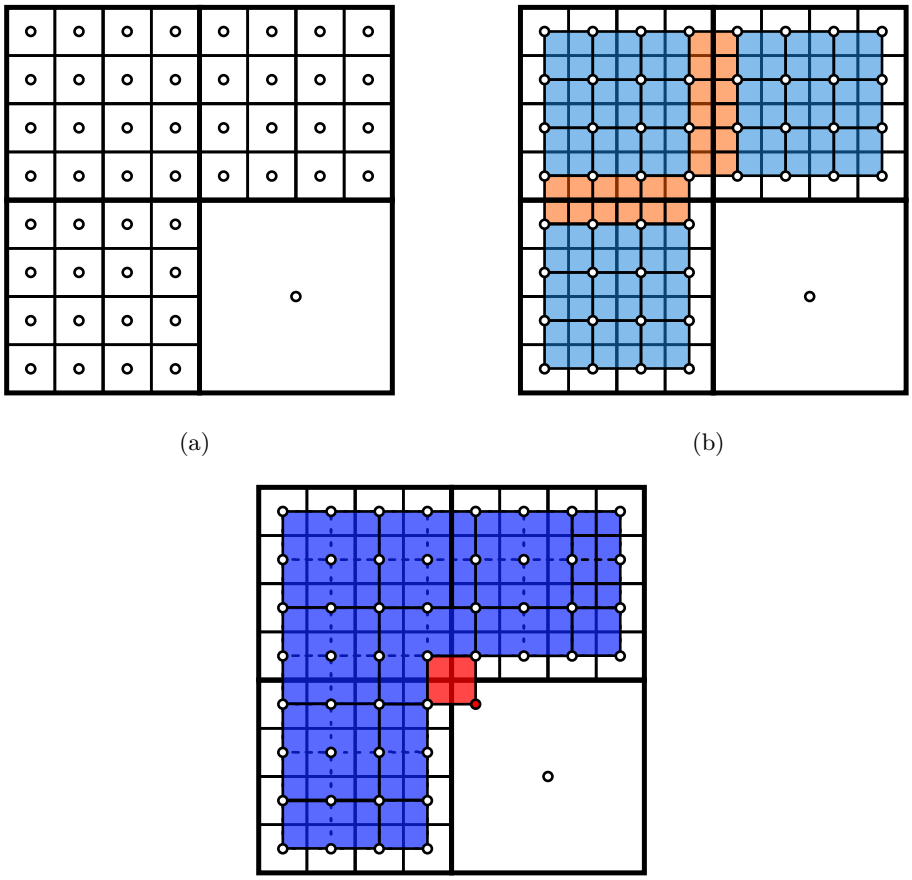
Im Rahmen dieser Arbeit werden wir uns vor allem auf den ersten Punkt konzentrieren.

Zum Erzeugen des Gitters verwenden die Autoren das Verfahren von Wald [17], welches wir bereits im Kapitel 3.2 behandelt haben. Sie verändern die `doDualCell` Methode und schreiben die dualen Zellen direkt. Zusätzlich ergänzen sie diese um eine Abfrage, ob alle Ecken der realen dualen Zellen den gleichen Verfeinerungsfaktor r haben. Damit trennen sie die richtigen Würfelzellen von den restlichen Elementen, denn nur eine Zelle, dessen r der Ecken gleich ist, kann ein Würfel sein.

Nun können die Würfelelemente levelweise zu Gridlets vereinigt werden. Dazu werden die Zellen zuerst in Voxel (wir nennen sie auch *Cubes*) umgewandelt und anschließend nach ihrem Level in Listen sortiert. Jede dieser Listen wird einzeln mit dem Algorithmus 8 verarbeitet.

Algorithm 8: Erzeugen von Gridlets nach Zellmann et al. [25]

```
1  function makeGrids(level, Liste Cubes)
2      definiere eine leere Liste Macrocells
3      foreach Cubes c ∈ Cubes do
           // projiziere C auf das virtuelle Gitter
4          mc = Macrocells[C]
5          (setze mc als aktiv)
6          erweitere Grenzen der Makrozelle mc um C
7          füge C der Makrozelle hinzu
8      end
9      foreach mc aus der Liste Macrocells do
10         if mc ist aktiv then
11             numScalars = mc.size() + (1, 1, 1)
12             mc.ScalarIDs.Resize(numScalars)
13             setze alle ScalarIDs auf -1
14             foreach C der Makrozelle mc do
15                 schreibe die Skalare in mc in VTK Reihenfolge rein
16             end
17         end
18     end
```



(a) (b)

(c)

Abbildung 4.1: (a) Drei Gitter vom Level 0, ein Gitter vom Level 2, hervorgehoben durch dickere schwarze Linien.
(b) An den Grenzen der dualen Gitter (hier in blau) vom gleichen Level können weitere Cubes ergänzt werden, hier in orange.
(c) Gridlets der Größe 2×2 sind als dunkelblaue Vierecke markiert. In rot ist eine leere Zelle, die dadurch entsteht, dass eine Ecke zum Erzeugen eines Gridlets fehlt. (In Anlehnung an [25])

Im ersten Schritt unterteilen die Autoren das ganze Level in Makrozellen, Gitter aus virtuellen Zellen der Größe der zukünftigen Gridlets.

In der ersten Schleife in Zeile 3 ordnen sie alle Cubes C der entsprechenden Makrozelle mc zu. Die Makrozellen, die mindestens einen Cube haben, betrachten sie als aktiv und erweitern die Grenzen der jeweiligen mc um die Grenzen von C.

In der zweiten Schleife gehen die Autoren über alle aktiven Makrozellen und setzen dessen Attribute. Sie berechnen die Anzahl der Skalare. Dann setzen sie alle Skalarindizes `scalarIDs` zunächst auf -1, die als Platzhalter für leere Zellen dienen. Die innere Schleife iteriert über alle Zellen in der Makrozelle. Falls eine Zelle tatsächlich existiert, werden die Skalarindizes mit den Skalarindizes dieser Zelle überschrieben. Die Reihenfolge der Skalarindizes ist einheitlich auf die Reihenfolge wie in VTK festgelegt:

$$v_0 = (0,0,0), \quad v_1 = (0,0,1), \quad v_2 = (0,1,0), \quad v_3 = (0,1,1)$$
$$v_4 = (1,0,0), \quad v_5 = (1,0,1), \quad v_6 = (1,1,0), \quad v_7 = (1,1,1)$$

mit $v_i,\ i \in \{0,\ldots,7\}$ die Ecken von einem Cube.

Dieses Vorgehen ermöglicht eine Kompressionsrate von etwa 1:8 von Cubes zu Gridlets bei den von den Autoren verwendeten Datensätzen.

Die Programmiersprache des Projekts ist C++ unter Verwendung von CUDA. Um die Vergleichbarkeit der Codes zu vereinfachen, bleiben wir bei der Notation des Ausgangscodes:

Ein *Cube* ist ein struct, der folgende Informationen beinhaltet:

- `vec3f lower` - das C_{lower} der Zelle
- `int level` - das Level auf dem sich die Zelle befindet
- `std::array<int,8> scalarIDs` - die Indizes der Skalare der Eckpunkte der Zelle.

Ein *Brick* ist ein struct, der ein Gridlet beschreibt:

- `box3i dbg_bounds` - die kleinste und größte Koordinate des Gridlets
- `vec3i lower` - die kleinste Koordinate des Gridlets
- `int level` - das Level
- `vec3i numCubes` - Anzahl der Cubes im Gridlet, leere Zellen mit eingeschlossen
- `std::vector<int> scalarIDs` - Skalare der Cubes.

Dabei sind `vec3i`, `vec3f` sowie `box3i` Elemente aus dem Submodul `umesh` und stellen entsprechend einen 3-Tupel aus Integerzahlen, einen 3-Tupel aus Fließkommazahlen und eine Box, die eine `vec3i lower` und `vec3i upper` Koordinate besitzt, dar.

4.2 Aufbau des Projekts

Wir bauen auf dem Quellcode von Zellmann et al. [25] auf. Die Daten aus einer Software zum Erzeugen der AMR Strukturen wie Chombo [1] dienen als Input für das Programm `amrMakeDualMesh`. Dieses gibt zwei Dateien zurück: die Datei mit der Endung `.umesh` beinhaltet unstrukturierte Elemente für das Stitching wie Pyramiden und `.cubes` die Würfel. Anschließend wandelt `amrMakeGrids` die Würfel in Gridlets `.grids` um. Die Skalare werden gesondert gespeichert. In den Gridlets befinden sich nur dessen Indizes. Das hat den Vorteil, dass die Skalare als Integer dargestellt und somit die Gridlets insgesamt kompakter gespeichert werden können.

Wir halten den Ablauf in der Abbildung 4.2 fest.

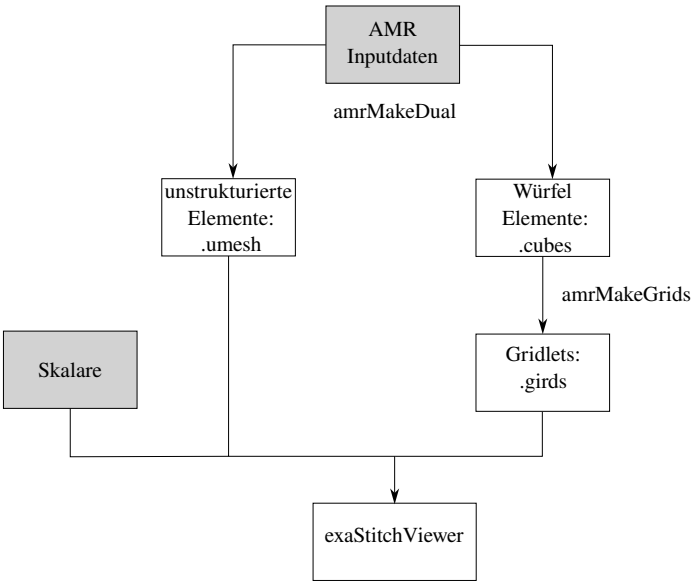


Abbildung 4.2: von AMR Daten zum Viewer (Eigene Darstellung)

4.3 Vorstellung der Methoden

Zu Beginn konzentrieren wir uns auf die ursprüngliche CPU Version der Methode `makeBricksForLevel()` in `makeGrids.cpp` des Projekts `amrMakeDualMesh`.

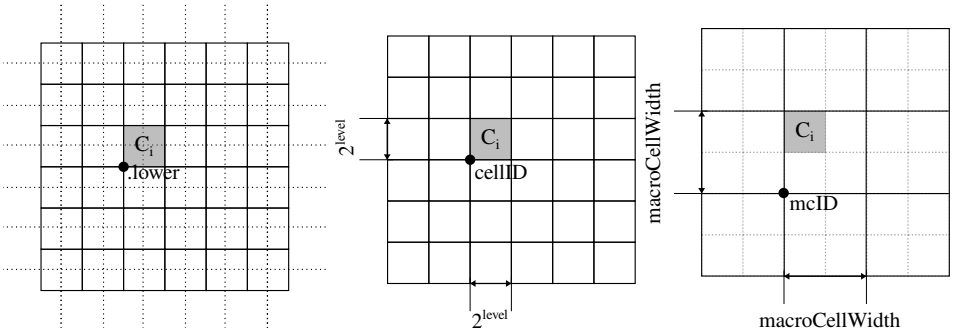
Die `main` Methode von `makeGrids.cpp` erwartet Dateien des Typs `.cubes`. Dieser beinhaltet structs vom Typ Cubes eines Levels. Grundsätzlich wird zwar in dem vorangehenden Verarbeitungsschritt `makeDual.cpp` gewährleistet, dass sich in einer Datei ausschließlich Cubes des gleichen Levels befinden, jedoch sind diese nicht sortiert.

Bei einem Funktionsaufruf von `makeBricksForLevel()` werden die *Cubes* einer `.cubes` Datei sowie dessen Level übergeben.

Zunächst wird aus den *Cubes* ein Gitter aus Makrozellen generiert. Dazu bekommt jeder *Cube* eine eindeutige ganzzahlige Zellnummer `cellID` zugeordnet. Sei $C_{lower} = (x, y, z),\ x, y, z \in \mathbb{R}$, dann ist die `cellID` durch $(x/2^\ell, y/2^\ell, z/2^\ell)$ gegeben. Es stellt eine Nummerierung in relativen Koordinaten dar.

Dabei ist die Zelle vom feinsten Level eine Einheit groß. Die Eindeutigkeit ist garantiert, da jeder *Cube* die Größe 2^{ℓ} besitzt.

Das anschließende Abbilden der `cellID` auf die Makrozellen ist analog durch das Teilen der drei Koordinaten durch die Makrozellbreite `macroCellWidth` definiert.



(a) (b) (c)

Abbildung 4.3: Projektion auf Makrozellen.

(a) Eine Zelle C_i ist zu Beginn durch ihre Fließkommazahl Koordinate `.lower` definiert.

(b) C_i wird durch ihre Levelbezogene Koordinate `cellID` definiert.

(c) C_i wird auf ihre entsprechende Makrozelle abgebildet. Einzelnen Makrozellen sind schwarz hervorgehoben. (Eigene Darstellung)

„Aktiviert“ wird eine Makrozelle, indem dessen `mcBounds` erweitert werden. `mcBounds` wiederum werden aus den der Makrozelle zugewiesenen Zellen berechnet Cubes. Das geschieht in einer Schleife über alle Cubes. Die zweite Schleife iteriert über die `mcBounds` und erzeugt dabei die Bricks aus den Makrozellen. Zum Schluss schreibt die dritte Schleife über alle Cubes diese in die entsprechenden Bricks herein.

Die Autoren verwenden bei jeder der drei Schleifen eine `std::map`. Dieses Vorgehen hat mehrere Nachteile. Zum einen werden alle Speicher- und Rechenoperationen seriell ausgeführt. Jedoch sind viele der Operationen identisch und voneinander unabhängig. Beispielsweise das Berechnen der `mcID` eines Cubes oder Schreiben der Attribute der Bricks

getrennt verlaufen, da es keine Informationen über die anderen Cubes oder Bricks erfordert. Das bietet eine geeignete Grundlage für eine parallele Ausführung. Die `std::map` ordnet die Daten in einem balancierten Binärbaum basierend auf den Schlüsseln an. Die Anzahl der Elemente beziehungsweise der Cubes und somit der Makrozellen ist im Allgemeinen sehr groß. Das Suchen und Einfügen von Elementen ist aus diesem Grund relativ langsam und teuer, was zu einem Bottleneck führt. In Kapitel 7 werden wir sehen, dass das Erzeugen der Gridlets durch diese Struktur erheblich beeinträchtigt wird.

Im Folgenden widmen wir uns der Übertragung der Berechnung der einzelnen Programmblöcke auf die GPU. Wir schauen uns zwei Programme an, wobei das zweite eine Verbesserung des ersten Programms im Hinblick auf den Speicherverbrauch ist.

4.3.1 makeGrids3Kernels.cu

Der Anfang des Programmablaufes ist wie bei der CPU Variante: Die main Methode bekommt `.cubes` Dateien mit einem Level pro Datei übergeben. In `makeGridsFor()` lesen wir dann die *Cubes* in std::vector ein und übergeben diese an `makebricksForLevel()` gemeinsam mit der zugehörigen Levelvariable.

Wir wollen zuerst die Umbox der Zellen auf dem jeweiligen Level ausrechnen. Dazu iterieren wir über alle Cubes und speichern jeweils die minimale und maximale `lower` Koordinate der Cubes. Es reicht nicht, nur den letzten und ersten Cube zu nehmen, da diese im Allgemeinen nicht sortiert sind. In der gleichen Schleife kopieren wir die Skalare und `lower` Koordinaten in separate std::vector Variablen. Im Folgenden nennen wir sie einfach Vektoren. Durch das Auseinandernehmen der Cubes in Listen umgehen wir eine *array of structures of arrays* Konstruktion, die wir an die GPU übergeben müssten. Damit können wir in den Kernels die spezielle Eigenschaft der der GPU ausnutzen, da die gebrauchten Daten nun kompakter nebeneinander liegen. Ebenso möchten wir das Padding vermeiden, welches durch den Einsatz von structs geschehen kann.

Aus obersten und untersten Koordinate berechnen wir die maximale Anzahl der Makrozellen in alle Richtungen. Diese brauchen wir, um später eine eindeutige Ordnung der Makrozellen zu gewährleisten, die wir in einen eindimensionalen Vektor umzuwandeln.

Mit der Formel:

$$mcID_x + mcID_y * levelSize_x + mcID_z * levelSize_x * levelSize_y \tag{4.1}$$

berechnen wir einen linearen Index, wobei $mcID$ die Makrozellen ID und $levelSize$ die zuvor ermittelte Größe des Levels in Makrozellen sind.

Im ersten Kernel `setBoundsAndCubes()` wird genau ein Cube pro thread parallel bearbeitet. Wir ordnen jedem Cube seine Makrozelle zu, indem wir zuerst die $cellID$ und daraus die $mcID$ berechnen. Diese liefert uns mit Hilfe von 4.1 den eindeutigen linearen Index. Im nächsten Schritt müssen wir die Grenzen der Makrozellen aktualisieren. Da mehrere Cubes in eine Makrozelle parallel geschrieben werden könnten, benutzen wir `atomicMin` und `atomicMax`. Aus ähnlichen Überlegungen arbeiten wir auch hier nicht mit dem struct *Brick* direkt, sondern legen mehrere eindimensionale Vektoren an. So speichern wir die Grenzen der Bricks in einem std::vector `mcBounds`.

Da wir in CUDA nicht effizient mit Listen einer variablen Länge arbeiten können, jedoch an dieser Stelle festhalten möchten, zu welcher Makrozelle der Cube im aktuellen thread gehört, erstellen wir einen Vektor von der Größe der maximalen Anzahl an Makrozellen `offsetsCubes` in der wir die Anzahl der Cubes für die entsprechende Makrozelle festhalten.

Das Aktualisieren geschieht wieder mit einer atomaren Funktion *atomicAdd*(). Anschließend schreiben wir die Nummer des aktuellen Cubes in den vector `listOfcubesIDXsforMC()` rein. Mit der Nummer des Cubes ist der thread, in dem diese verarbeitet wird, gemeint.

Dadurch, dass wir nun wissen, wie viele Elemente eine Makrozelle besitzt, können wir den Index, ab dem die Nummer der Cubes liegen, ausrechnen. Der erste Kernel entspricht der ersten Schleife im Algorithmus 8.

In den Abbildungen 4.4 und 4.5 verdeutlichen wir diesen Ablauf.

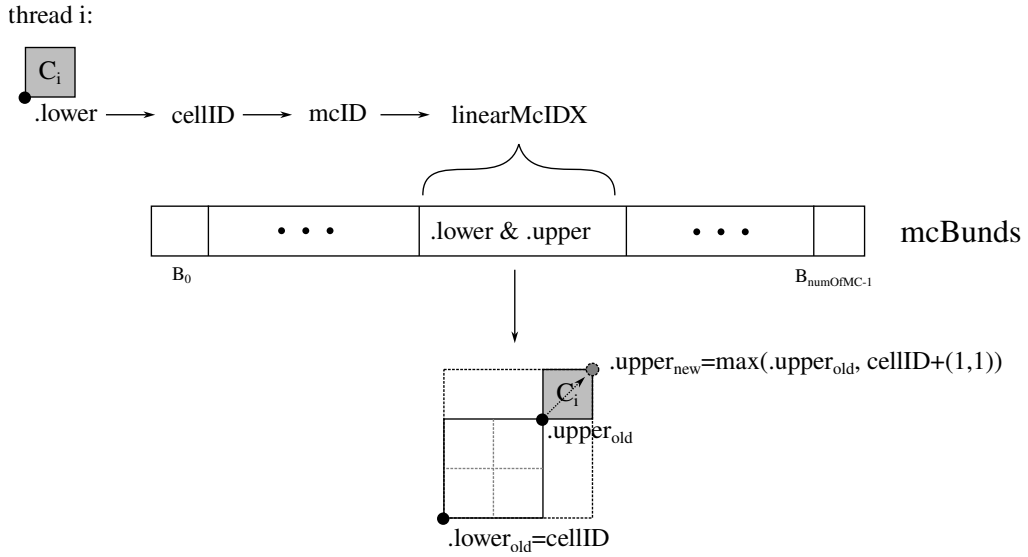


Abbildung 4.4: kernel 1 - Aktivierten einer Makrozelle: Vorgang für thread i: Aus der globalen Koordinate einer Zelle C_i berechnen wir den Index der Makrozelle in der sie liegt und damit die Stelle im Array `mcBounds` an der wir die Grenzen der Makrozelle speichern. Durchgezogene schwarze Linie zeigt die alten Grenzen, schwarz gestrichelt sind die neuen Grenzen nach dem die Makrozelle um C_i erweitert ist. Andere Fälle sind auch möglich: Erweiterung in weniger Dimensionen, „nach unten“ oder gar keine. Hier sehen wir auch, dass mehrere Zellen parallel auf die gleiche Stelle in `mcBounds` zugreifen könnten, da wir nach den Cubes und nicht den Makrozellen iterieren. (Eigene Darstellung)

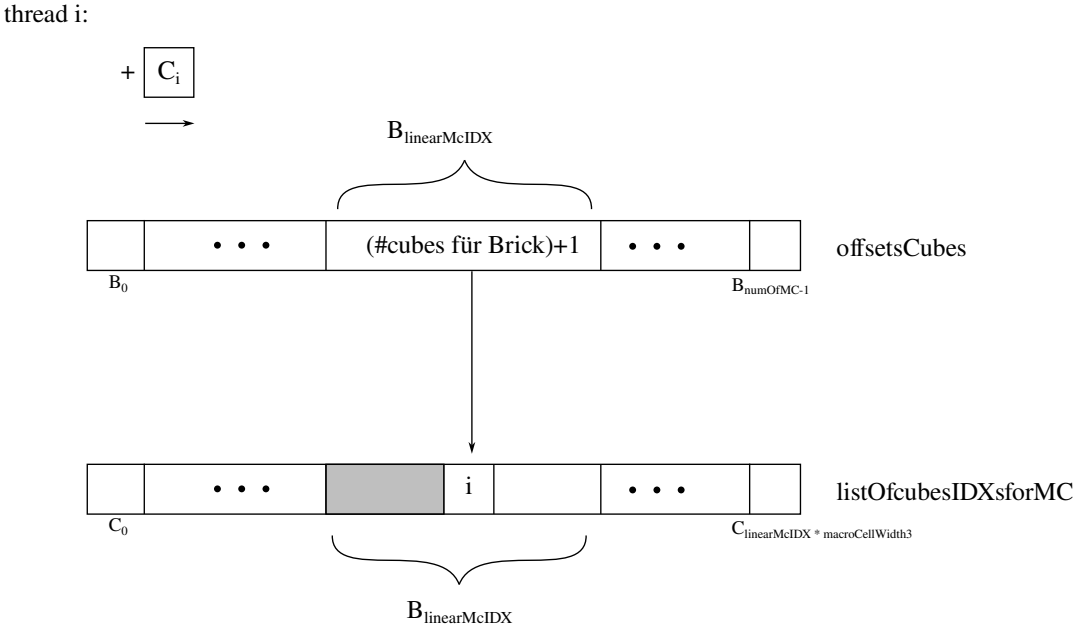


Abbildung 4.5: kernel 1 - Abbilden von Cubes nach MC B: Wenn wir den i-ten Cube betrachten, erhöhen wir den Zähler für die Anzahl der Cubes in einer Makrozelle im Array `offsetsCubes` und notieren die Nummer des Cubes, hier i, an die nächstfreie Stelle im Array `listOfcubesIDXsforMC` entsprechend der mcID. (Eigene Darstellung)

Mit Hilfe der nächsten Kernel implementieren wir die zweite Schleife aus dem Algorithmus 8.

Im zweiten Kernel `calcMaxNumOfScalars()` berechnen wir die Anzahl der Skalare, die wir ausgehend von der Größe des jeweiligen *Bricks* `mcBounds[linearer Brick Index]` ermitteln. Dabei schließen wir die Skalare der leeren Zellen mit ein.

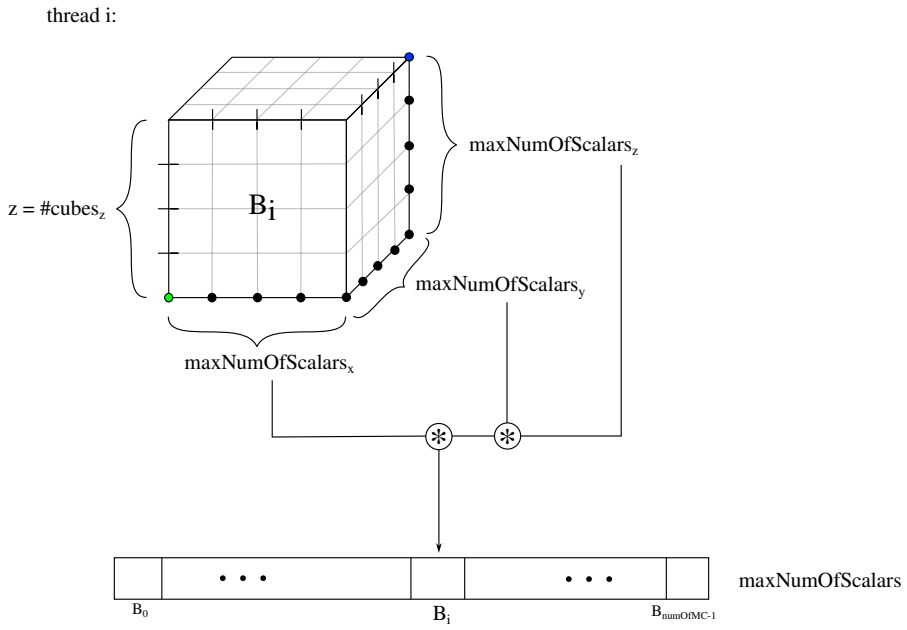


Abbildung 4.6: kernel 2: Aus `mcBounds.lower` (grüner Kreis) und `mcBounds.upper` (blauer Kreis) berechnen wir die Anzahl der Skalare für Brick B_i. Die Skalare in eine Richtung sind jeweils mit Kreisen markiert. (Eigene Darstellung)

Im nächsten Schritt benutzen wir die Ausgaben des zweiten Kernels, um mit der Funktion `exclusive_scan` von der thrust Bibliothek [16] Versätze zu berechnen. Diese Funktion addiert parallel die zuvor berechnete Anzahl an *Cubes* und speichert es als Präfixsumme. Ein Eintrag im Array beschreibt die Summe aller vorhergehender Anzahlen an Skalaren:

$$\texttt{maxNumOfScalars}[n] = \sum_{i=0}^{n-1} \text{Anzahl Skalare für Brick } i$$

mit `maxNumOfScalars`$[n]$ für $n > 0$, dem Versatz für den n-ten Brick. Dabei ist `maxNumOfScalars`$[0] = 0$. Das Ergebnis hilft uns dabei die Größe des Skalararrays, sowie die Indizes, ab denen die Skalare eines *Bricks* beginnen, zu bestimmen. Um die Gesamtanzahl an Skalarindizes für den `resultScalarArray` zu berechnen, erhöhen wir die Größe von `maxNumOfScalars` um einen imaginären Brick mit Skalaranzahl 0.

Der letzte Eintrag gibt uns dann die Summe über alle `numberOfMC` Bricks.

Im letzten Kernel `createAndFillBricks()` fügen wir alles zusammen. Mit einer Makrozelle pro thread überprüfen wir zunächst, ob diese aktiv ist, mit anderen Worten, ob die Grenzen von der Makrozelle schon mal erweitert wurden. Dies ist der Fall, wenn die `offsetCubes` für die aktuelle Makrozelle nicht Null sind. Alle Makrozellen, die es erfüllen, werden zu *Bricks* beziehungsweise in der Notation von [25] zu Gridlets.

Wir übergeben dem Kernel einen Vektor aus *Bricks*. Das Schreiben der Attribute der *Bricks* wie das Level und die Anzahl an *Cubes* in parallel ist effizient. Allerdings müssten wir bei dem Reservieren des Speichers bereits wissen, wie viele Skalare es in jedem *Brick* geben soll. Wir könnten einfach für alle *Bricks* die maximale Anzahl an Skalaren reservieren. Das würde dazu führen, dass wir zum einen bei *Bricks* mit vielen leeren *Cubes* zu viel Speicher verbrauchen und zum anderen müssten wir die Datenstruktur erweitern. Denn dann müssten wir wissen, an welcher Stelle wir die Indizes entsprechend der einzelnen `mcBounds` abschneiden müssen. Das Anpassen an die Größe würde ebenfalls viele Rechenkapazitäten verbrauchen.
Aus diesem Grund lagern wir das Speichern der Skalare in einen großen dynamischen Vektor aus, dessen Größe wir entsprechend den Berechnungen aus dem zweiten Kernel und der thrust Funktion anpassen. In den Bricks selbst speichern wir lediglich einen Pointer, der auf den ersten Skalar des Bricks in dem Ergebnisvektor `resultScalarsArr` zeigt.
Wir setzen nun alle Skalare im Ergebnisvektor auf -1.

Da die Gridlets keine Überschneidungen haben, an den Grenzen die Skalare duplizieren und wir über die Makrozellen iterieren, können keine *race conditions* oder sonstige Zugriffsfehler auftauchen. Mit der Methode `setAttributes()` setzen wir die Grenzen der Bricks `dbg_bounds`, den Ursprung `lower`, die Anzahl der *Cubes* `numCubes`, den Index im Skalarenvektor `offset` und die Anzahl an Skalaren `numScalars`. Diesen Ablauf verdeutlichen wir in der Abbildung 4.7.

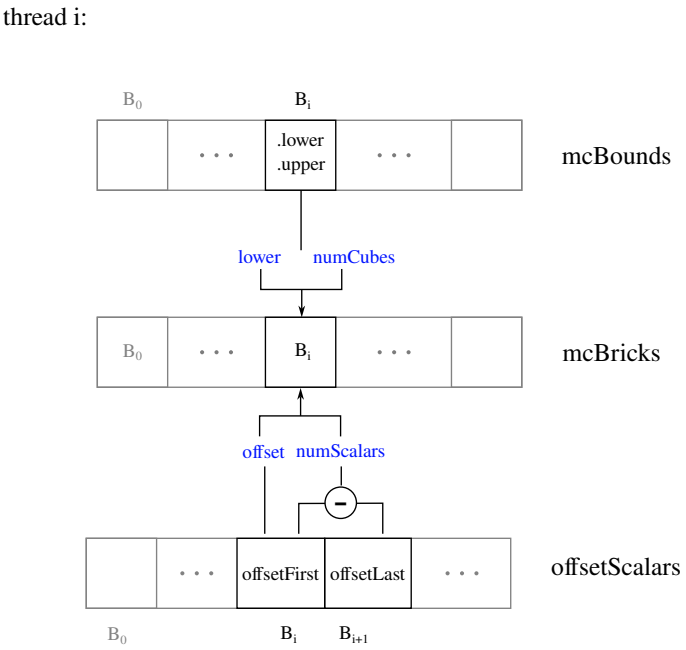


Abbildung 4.7: Methode `setAttributes()` im dritten Kernel. Für einen Brick per thread setzen wir dessen Attribute, die in der Abbildung blau markiert sind. (Eigene Darstellung)

Wir gehen die Liste der Cubes für den jeweiligen Brick durch und kopieren nacheinander die Skalarindizes zunächst in ein temporäres Array der Größe acht. Diesen übergeben wir an die Funktion `writeCube()`, die die Skalare in der VTK Reihenfolge in den Ergebnisvektor reinschreibt und benutzen dafür wieder die Formel für den linearen Index 4.1.

Die benachbarten Cubes haben gemeinsame Ecken und teilen somit Skalare. Wir übernehmen die Methode aus dem ursprünglichen Code, die keine Fallunterscheidung durchführt, sondern die mehrfach belegten Stellen immer wieder überschreibt.

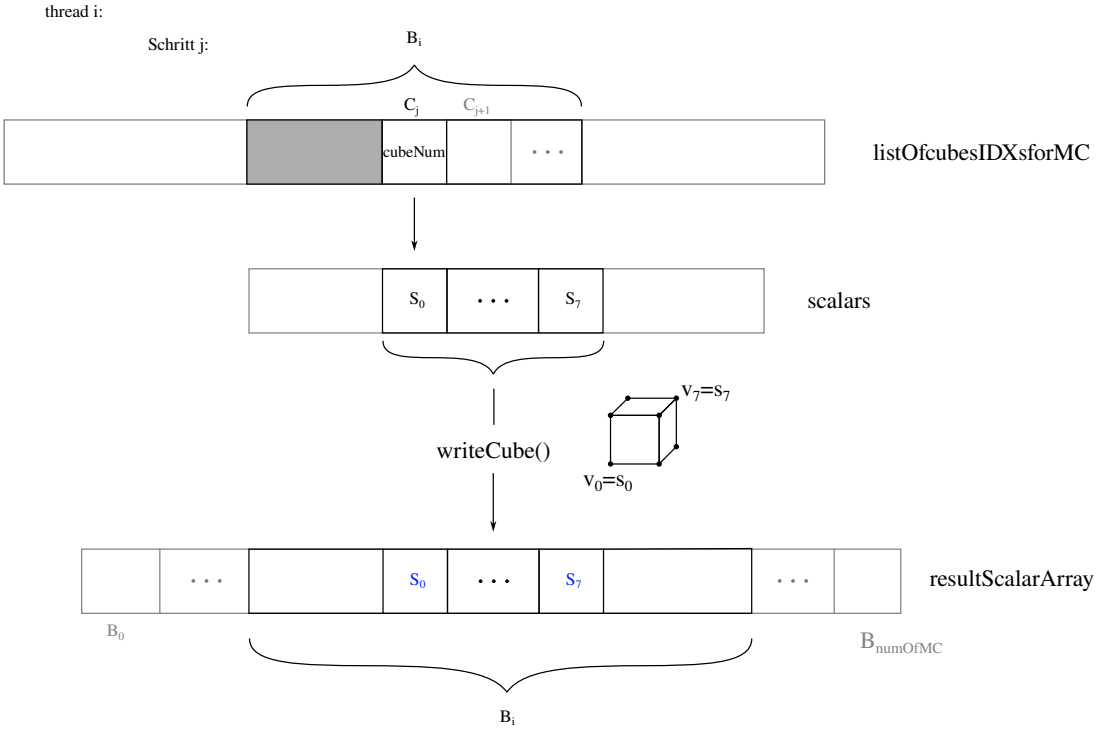


Abbildung 4.8: Schreiben der Cubes. Wir suchen für den j-ten Schritt den j-ten Cube heraus, kopieren dessen Skalarindizes aus dem langen Array `scalars` in dem Skalare aller Cubes gesondert gespeichert sind, und schreiben diese in den langen Ergebnisarray `resultScalarArray`. Die Skalarindizes (blau) müssen jedoch, anders als in der Abbildung, nicht hintereinander gespeichert sein. (Eigene Darstellung.)

Zum Schluss kopieren wir alle Daten mit `cudaMemcpy()` wieder auf die CPU zurück und setzen die Pointer für alle Skalarindizes der Bricks.

Beim Schreiben der Bricks in die `.grids` Datei achten wir darauf, dass wir nur diejenigen Bricks schreiben, die nicht leer sind.

4.3.2 makeGrids4Kernels.cu

Die Schleife im dritten Kernel läuft über alle Cubes des jeweiligen Bricks. Im schlimmstmöglichen Fall hat sie die Größe von `macroCellWidth`^{3} - gerade bei größeren Makrozellen könnte es unter Umständen zu Problemen führen. Inwieweit sich die Laufzeit tatsächlich verbessert, schauen wir uns im Kapitel 8 an.

Die Idee ist, dass wir nicht mehr über die aktiven Makrozellen, sondern über die aktiven Cubes iterieren. Zuvor haben wir das Problem, dass wir aus der Sicht des Bricks nicht wussten, welche Cubes rein gehören und haben dafür eine extra Variable `listOfcubesIDXsforMC()` eingeführt, die die Nummern der Cubes gespeichert hat. Anders herum, aus der Sicht der Cubes, können wir aus der `lower` Koordinate die eindeutige Zuordnung mit `mcID` von der Zelle zur Makrozelle berechnen. Das hat zur Folge, dass `listOfcubesIDXsforMC()` nicht mehr gebraucht wird.

Die ersten beiden Kernel sowie die Berechnung von der Präfixsumme bleibt gleich.

Wir sollten die Attribute der Bricks nur ein Mal schreiben. Würden wir den letzten Kernel des vorherigen Codes einfach nur über die Cubes laufen lassen, so müssten wir einen Weg finden um zu überprüfen, ob die Daten bereits von einem anderen thread schon geschrieben worden sind. Um Branching und Zugriffsfehler möglichst zu vermeiden, schieben wir einen weiteren Kernel dazwischen. Dieser geht über alle aktiven Makrozellen und setzt mit `setAttributes()` die Grenzen der Bricks `dbg_bounds`, den Ursprung `lower`, die Anzahl der *Cubes* `numCubes`, den Index im Skalarenvektor `offset` und die Anzahl an Skalaren `numScalars`. Ebenfalls bietet es sich an, schon an dieser Stelle die Skalare im Ergebnisvektor auf -1 zu setzen.

Im letzten Kernel schauen wir uns genau einen Cube pro thread an und berechnen zuerst seine `cellID` und daraus die `mcID`. Wir kopieren wieder die acht Skalare in ein temporäres Array und übergeben diesen an `writeCube()`. Da die Cubes gemeinsame Ecken teilen, werden die Einträge in dem Ergebnisvektor mehrfach überschrieben.

Das hat zur Folge, dass wir gleichzeitig auf die gleiche Adresse schreiben würden (Abbildung 4.9). Aus diesem Grund benutzen wir `atomicExch()` - eine atomare Funktion, die garantiert, dass auch unter paralleler Ausführung das Resultat wohldefiniert ist. Eine Ecke kann maximal von 8 Cubes genutzt werden, das heißt, es wird maximal acht Mal auf die gleiche Adresse geschrieben. Dies sollte sich nicht signifikant auf die Laufzeit auswirken.

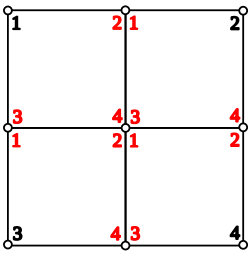


Abbildung 4.9: Mehrfachbelegung der Skalarindizes am Beispiel eines zweidimensionalen 2×2 Gridlets. Die schwarzen Zahlen an den Datenpunkten (Kreise) beschreiben diejenigen Stellen, die nur zu einer Zelle gehören. Rote Zahlen weisen auf die Ecken hin, die von mindestens zwei den Zellen innerhalb eines Gridlets geteilt werden. (Eigene Darstellung)

Die veränderten Methoden haben den Vorteil, dass der Speicher früher befreit werden kann: Nachdem die Attribute der Bricks in dem dritten Kernel geschrieben wurden, wird weder `mcBounds` noch `offsetsCubes` weiter gebraucht.

Um weitere Schritte zu sparen, können wir statt die Cubes erst einzulesen und dann in einem Vektor zu speichern, der wiederum in zwei weitere Vektoren mit den `lower` Koordinaten und Skalarindizes gespalten wird wie folgt vorgehen: Wir lesen einen Cube aus der `.cubes` Datei heraus und speichern gleich dessen beiden Attribute in getrennte Vektoren `cubesLower` und `scalarsArray`. Das hat den Vorteil, dass wir zum Kopieren nur ein mal über die ganzen Cubes iterieren müssen. Die Schleife zum Berechnen der minimalen und maximalen Koordinate über die alle Koordinaten der Cubes bleibt zwar bestehen, aber es ist zu erwarten, dass es nicht wesentlich zu der Gesamtlaufzeit beitragen

sollte.

Die Ausgabedateien der vorgestellten Methoden werden in dem Ordner `outputGrids` gespeichert. Dabei ist `orig_out_level_i.grid` der Output des originalen Programms, `cuda_k3_level_i.grids` und `cuda_k4_level_i.grids` entsprechend die Outputs von `makeGrids3Kernels.cu` und `makeGrids4Kernels.cu` für den i-ten Level.

4.3.3 Andere Strategien

Relativ zu Beginn der Entwicklungszeit wird klar, dass die Datentransferzeit eine der größten Engstellen sein werden. Aus diesem Grund ist es wichtig, so viel Kontrolle über diese Schritte wie möglich zu haben. Einen alternativen Ansatz mit der *unified memory* mit Hilfe von `cudaMallocManaged()` können wir aus diesem Grund verwerfen.

Eine andere Möglichkeit des Datenlayouts wäre es, die Vektoren aus Cubes und Bricks unverändert als Ganzes beizubehalten. Dies würde zwar zu einem kompakteren Code führen, jedoch sind solche Konstruktionen durch das mögliche Padding und nicht kohärente Datenzugriffe nicht effizient. Durch das Verzögern des Datenversands zur Grafikkarte bis zu dem Zeitpunkt, an dem sie benötigt werden, minimieren wir den maximal benötigten Speicher.

In [25] erwähnen Zellmann et al. eine potenzielle Umsetzung mit Hilfe von zwei Kernels. Das Planen der Datenzugriffe und Rechenschritte hat jedoch gezeigt, dass eine solche Variation komplizierter ist und in dieser Arbeit keine effiziente Implementierung gefunden wird. Als eine der Herausforderungen stellt sich die Divergenz der Threads und das Speichermanagement dar.

5 Testumgebung

Das Messinstrument für die CPU Zeit ist die `high_resolution_clock` aus der `chrono` Bibliothek.

Um die Korrektheit der Ausgaben zu überprüfen, bietet es sich an, die berechneten Daten der Bricks mit der Originalausgabe zu vergleichen. Für kleinere Datensätze können wir diese mit einem `print` Befehl auf die Konsole schreiben und per Hand vergleichen. In der Regel ist der Aufwand jedoch zu groß. Das Programm `testBricksOutput.cpp` verschafft Abhilfe:

Wir übergeben dem Programm über die Konsole als ersten Argument die Datei vom Datentyp `.grids` mit den originalen Bricks und an zweiter Stelle die zu vergleichende Datei. Nachdem die Eingaben in Vektoren der C++ standard library eingelesen sind, werden sie mithilfe von Mortoncodes nach den `lower` Koordinaten sortiert. Die Eindeutigkeit der Sortierung ist bis zu einer Anzahl von etwa 2^{31} Bricks gewährleistet, da wir einen 32 bit encoder verwenden. Anschließend vergleicht die Methode die Anzahl der Bricks in den Dateien und alle Attribute: die `lower` Koordinate, die Skalarindizes, das Level und die Anzahl an Cubes. Bei den Skalarindizes wird zusätzlich explizit auf die Reihenfolge geachtet.

Alle Ausgaben der Codes aus Kapitel 4.3 mit den weiter beschriebenen Datensätzen stimmen mit der Originalausgabe überein. Demnach können wir annehmen, dass die Berechnungen auf der GPU korrekt sind.

Wir testen die Daten auf einem Rechner mit den folgenden Spezifikationen durch:

```
Intel Core i7-9750H CPU (6 Kerne, 12 threads und 16 GB RAM)
NVIDIA GeForce GTX 1650 GPU mit 4 GB Speicher
Ubuntu Linux 22.04.3 LTS
Grafiktreiberversion 535.54.03
CUDA 12.2.
```

Listing 5.1: Computer 1

Des Weiteren führen wir Tests für einen Datensatz mit realen Daten (siehe Kapitel 6 Cloud Datensatz) auf dem genannten System durch:

```
Intel(R) Xeon(R) CPU E5-1620 v2 (4 Kerne, 8 threads und 16 GB RAM)
GeForce RTX 2080 mit 8 GB Speicher
Ubuntu 22.04.1 LTS
Grafiktreiberversion 525.85.12
CUDA 12.0.
```

Listing 5.2: Computer 2

Zur genaueren Laufzeit- und Speicheranalyse benutzen wir das Programm `cubesGeneration.cpp`. Dieses kann Cubes in Abhängigkeit von dem Level der Makrozellengröße und Anzahl generieren.

6 Datensätze

Für den ersten Datensatz erzeugen wir mit `cubesGeneration.cpp` einen Level aus $416 \times 416 \times 416$ dicht gepackten Cubes beziehungsweise $52 \times 52 \times 52$ Makrozellen mit 3.5 GB. Es sind keine leeren Zellen zugelassen. Die Größe kommt durch Ausprobieren der maximal möglichen Anzahl der Makrozellen, die gerade noch von allen drei Versionen ausgeführt werden können, zustande. Wir wählen es so, dass es durch 8 teilbar ist, da wir bei den Benchmarks die Makrozellengröße auf $8 \times 8 \times 8$ setzen.

Um die Skalierbarkeit zu untersuchen, variieren wir die Makrozellengröße und bekommen die folgende Datenmenge: $\{(i \times i \times i) |\ i \in 41, \ldots, 52\}$.

Der dritte Datensatz ist eine Sammlung aus 20 Leveln aus jeweils $280 \times 280 \times 280$ von dicht gepackten Cubes mit 1.1 GB. Die Anordnung der Level erfolgt so, dass das feinere Level an der obersten Ecke des gröberen Levels anfängt. Die Größe des Datensatzes resultiert aus dem Ziel, einen möglichst tiefen Baum zu erzeugen, während gleichzeitig die Anzahl der Elemente ausreichend groß sein soll, um Messfehler zu minimieren.

Der Datensatz „Cloud“ repräsentiert eine reale Modellierung einer molekularen Gaswolke mit acht Leveln, von denen wir ausschließlich die ersten fünf verwenden. Diese wurde mit der SILCC-Zoom Simulation (Life Cycle of Molecular Clouds [15]) erzeugt. Diese Simulation besteht aus Octree Strukturen.

	Level 0	Level 1	Level 2	Level 3	Level 4
Cubes	30643304	13207932	34162517	385900	98

Tabelle 6.1: Datensatz: Cloud - Anzahl der Cubes

7 Ergebnisse und Auswertung

Wir haben uns im Kapitel 4.3 darauf konzentriert, in `makeGrids.cpp` die Methode `makeBricksForLevel()` zu modifizieren. In diesem Kapitel vergleichen wir die Implementierungen miteinander. Das `CMake.txt` erzeugt aus den `makeGrids.cpp`, `makeGrids3Kernels.cu` und `makeGrids4Kernels.cu` die Dateien `amrMakeGrids`, `amrMakeGrids_cuda3` und `amrMakeGrids_cuda4`. Wir verändern und verkürzen in diesem Kapitel die Namen zu: `original`, `makeGrids_k3` und `makeGrids_k4` entsprechend.

7.1 Datensatz: dicht gepackte Zellen

Um einen ersten Überblick über die Laufzeit zu bekommen testen wir die drei Programme mit einem künstlichen Datensatz, der gerade noch auf von allen Codes ausgeführt werden kann. Dieser besteht aus einem Level mit $\ell = 0$ und $416 \times 416 \times 416$ dicht gepackten Zellen. Wie von Zellmann et al. [25] empfohlen, setzen wir die Größe der Gridlets auf $8 \times 8 \times 8$. Bei Experimenten mit der Makrozellengröße gab es keine signifikanten Unterschiede in der Laufzeit.

	Gesamtlaufzeit	makeBricksForLevel()	max. Speicherverbrauch (GPU)
original	26.33	19.37	—
makeGrids_k3	11.27	3.90	3770
makeGrids_k4	8.82	1.56	3490

Tabelle 7.1: Laufzeiten sind in Sekunden und auf zwei Nachkommastellen gerundet. Der Speicherverbrauch ist in MiB für einen Level mit `nvidia-smi` gemessen.

Die Gesamtlaufzeit bezieht sich auf die CPU Laufzeit ab dem Beginn der `main()` Methode bis einschließlich der Zeit zum Schreiben der Bricks.

Die Differenz der Gesamtlaufzeit und die CPU Laufzeit der `makeBricksForLevel()` gibt an, wie viel Zeit um das Berechnen der Bricks herum vergeht. Diese beträgt hier bei allen Funktionen in etwa 7 Sekunden, da sich die Schritte außerhalb der `makeBricksForLevel()` bis auf eine Schleife nicht unterscheiden. Diese Schleife ist für das Kopieren der Cubes aus der Inputdatei in einen Vektor verantwortlich - bei `makeGrids_k4` schreiben wir die Attribute der eingelesenen Cubes gleich in zwei getrennte Vektoren (vgl. 4.3.2).

Der Unterschied spiegelt sich in der Laufzeit wieder: Während das Kopieren aus der *.cubes* Datei bei beiden etwa gleich ist (6.2899s zu 6.39047s), kommt bei `makeGrids_k3` noch 1.61971s in der `makeBricksForLevel()` Methode hinzu. Das schließt zwar die Zeit mit ein, die für das Finden der minimalen/maximalen Koordinate benötigt wird, jedoch beträgt diese bei den Messungen wenige Millisekunden und ist aus diesem Grund im Vergleich vernachlässigbar. Für größere Datensätze wird der Einfluss der doppelten Kopierzeit noch deutlicher, da die Länge der kopierenden for-Schleife direkt von der Anzahl an Cubes abhängt.

Umgekehrt können wir daraus schließen, dass wenn die Daten schon auf der GPU sind, der Vorteil der neuen Methoden noch deutlicher wird.

Wir sehen, dass die `makeBricksForLevel()` Methode für diesen Datensatz etwa fünfmal, beziehungsweise 12 Mal, so schnell im Vergleich zu der CPU Version ist. Wir stellen im nächsten Schritt die zwei Implementierungen dieser Funktion gegenüber:

```
Time(%)     Time     Calls       Avg       Min       Max  Name
GPU activities:
59.05%  1.37582s         1  1.37582s  1.37582s  1.37582s  createAndFillBricks(...)
24.66%  574.43ms         4  143.61ms  529.46us  414.62ms  [CUDA memcpy HtoD]
13.43%  312.86ms         3  104.29ms  1.9190us  311.08ms  [CUDA memcpy DtoH]
2.86%   66.587ms         1  66.587ms  66.587ms  66.587ms  setBoundsAndCubes(...)
0.00%   37.951us         1  37.951us  37.951us  37.951us  calcMaxNumOfScalars(...)
0.00%   8.0000us         1  8.0000us  8.0000us  8.0000us  thrust ScanKernel
0.00%   1.2160us         1  1.2160us  1.2160us  1.2160us  thrust DeviceScanInitKernel
API calls:
93.20%  2.26464s         7  323.52ms  24.325us  1.37774s  cudaMemcpy
3.57%   86.641ms         2  43.320ms  6.8650us  86.634ms  cudaDeviceSynchronize
2.74%   66.515ms         1  66.515ms  66.515ms  66.515ms  cudaStreamSynchronize
0.23%   5.6306ms         9  625.62us  7.2490us  2.9197ms  cudaMalloc
0.21%   5.0616ms         9  562.41us  8.1980us  2.7231ms  cudaFree
0.03%   833.31us         5  166.66us  6.6970us  767.97us  cudaLaunchKernel
0.01%   359.66us       114  3.1540us     181ns  150.47us  cuDeviceGetAttribute
[...]
```

Listing 7.1: Ein Auszug aus den Ausgaben des Profilers nvprof für *makeGrids_k3* . Zur besseren Übersichtlichkeit ohne Argumente und API calls, die <0.01% Zeit beanspruchen.

```
Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:
39.47%  576.90ms         4    144.23ms  526.50us  414.28ms  [CUDA memcpy HtoD]
27.12%  396.44ms         1    396.44ms  396.44ms  396.44ms  writeScalars(...)
22.47%  328.45ms         3    109.48ms  2.1440us  326.75ms  [CUDA memcpy DtoH]
6.54%   95.635ms         1    95.635ms  95.635ms  95.635ms  createBricks(...)
4.39%   64.191ms         1    64.191ms  64.191ms  64.191ms  setBoundsAndCubes(...)
0.00%   37.248us         1    37.248us  37.248us  37.248us  calcMaxNumOfScalars(...)
0.00%   10.016us         1    10.016us  10.016us  10.016us  thrust ScanKernel
0.00%   1.5670us         1    1.5670us  1.5670us  1.5670us  thrust DeviceScanInitKernel
API calls:
83.88%  1.30344s     7    186.21ms  65.595us  414.38ms  cudaMemcpy
6.47%   100.47ms     8    12.559ms  12.083us  95.769ms  cudaFree
5.08%   78.871ms     1    78.871ms  78.871ms  78.871ms  cudaDeviceSynchronize
4.13%   64.121ms     1    64.121ms  64.121ms  64.121ms  cudaStreamSynchronize
0.35%   5.3625ms     8    670.31us  7.3860us  2.9818ms  cudaMalloc
0.09%   1.3629ms     6    227.15us  6.3550us  1.2598ms  cudaLaunchKernel
0.01%   231.54us   114    2.0310us     177ns  88.362us  cuDeviceGetAttribute
[...]
```

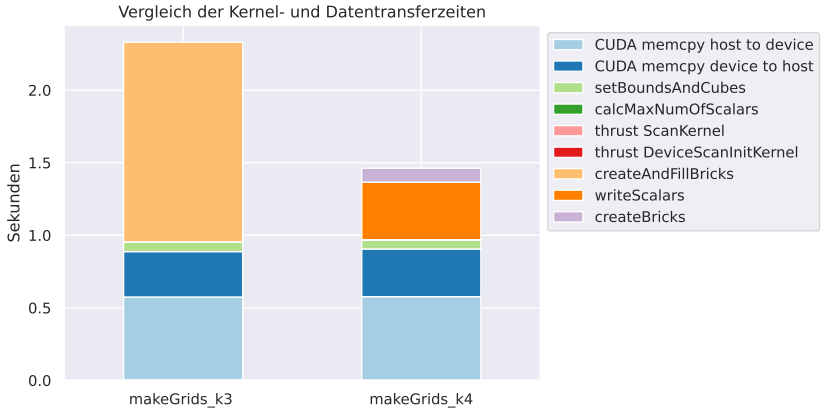
Listing 7.2: Ein Auszug aus den Ausgaben des Profilers nvprof für *makeGrids_k4*. Zur besseren Übersichtlichkeit ohne Argumente und API calls, die <0.01% Zeit beanspruchen.

Wir führen die beiden Codes unter Verwendung von `nvprof` nochmal aus, um die genauen GPU Zeiten zu erfassen. In Listing 7.1 und 7.2 sehen wir eine Auflistung der Funktionen der Methode `makeBricksForLevel()`, die mit der GPU in Verbindung stehen.

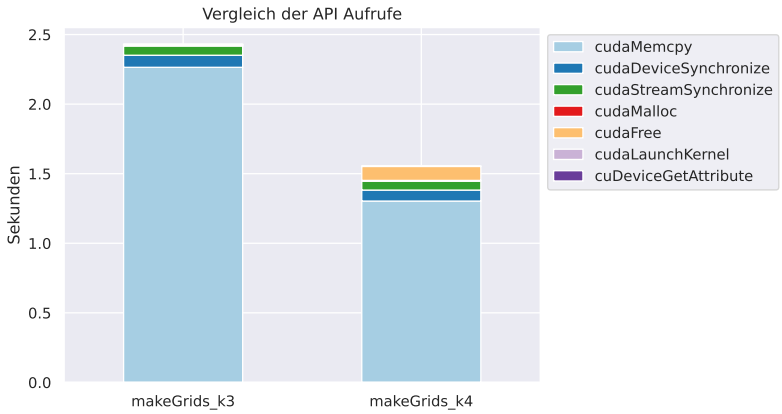
Sowohl die Datentransfers als auch Kernel 1 (`setBoundsAndCubes()`), 2 und thrust Aufrufe (`calcMaxNumOfScalars()`) beanspruchen bei beiden in etwa gleich viel Zeit. Der dritte Kernel (`createAndFillBricks()`), der eine innere Schleife über die Anzahl an Cubes hat, läuft deutlich langsamer als die vorhergehenden Kernel. Ebenfalls ist er im Vergleich zu der Summe der Laufzeiten der beiden Kernel `writeScalars()` und `createBricks()` langsamer: mit 1.38s zu 0.49s.

Wir nehmen deswegen an, dass `createAndFillBricks()` schlechter skaliert, wenn die Anzahl an Cubes in Bricks zunimmt. Das kann im Code bestätigt werden:

- `makeGrids_k3`: Parallelisieren über alle Makrozellen - geschachtelte Schleife zum Setzen der Skalare über Cubes der Makrozelle und dann über die Skalarindizes dieser
- `makeGrids_k4`: Parallelisieren über alle aktiven Cubes - eine Schleife über die 8 Skalarindizes des jeweiligen Cubes.



(a)



(b)

Abbildung 7.1: (a) Eine Gegenüberstellung der GPU Laufzeiten für `makeGrids_k3` und `makeGrids_k4`. Datenübertragungen vom host zum device und zurück in blau und hellblau. Die restlichen Anteile sind die Kernel.

Anmerkung: Die Methode `createAndFillBricks()` ist exklusiv für `makeGrids_k4` und die Methoden `writeScalars()` und `createBricks()` führt nur `makeGrids_k4` aus.

(b) Gegenüberstellung der API Aufrufe für `makeGrids_k3` und `makeGrids_k4`. (Eigene Darstellung)

In Abbildung 7.1(a) sehen wir, dass die Version mit drei Kernel im Gegensatz zu der anderen rechenintensiver ist. Der Kernel `createAndFillBricks()` nimmt einen überproportional größeren Anteil als die Datentransfers ein. Aus diesem Grund würde es sich an dieser Stelle anbieten, die Speicher- und Rechenoperationen mithilfe der asynchronen Funktionen zu überschneiden. Bei der Version mit vier Kernel ist das Verhältnis der Kopieroperationen zu Rechenoperationen ausgewogen.

Abbildung 7.1(b) verdeutlicht, dass die teuerste Operation bei beiden Varianten des Programms die Datenübertragungsfunktion `cudaMemcpy()` ist. Bei einer möglichen Verbesserung müssen wir demnach die Datenmengen durch einen besseren Layout reduzieren oder einen Teil der Datenübertragungen gleichzeitig mit den Rechenoperationen ausführen.

Wir erfassen den maximalen GPU-Speicherverbrauch mit `nvidia-smi`, wobei eine Differenz im Speicherverbrauch von 280 MiB feststellbar ist. Wenn wir die Anzahl der Cubes erhöhen, dann sehen wir, dass die Skalierbarkeit von `makeGrids_k3` schlechter ist. Während die Variante mit vier Kernel einen Datensatz von $432 \times 432 \times 432$ Cubes noch verarbeiten kann, reicht der Speicher für die Version mit drei Kernen nicht aus. Im nächsten Unterkapitel untersuchen wir diesen Punkt genauer.

	Gesamtlaufzeit	`makeBricksForLevel()`	max. Speicher (GPU)
original	47.27	35.79	—
`makeGrids_k4`	14.48	3.54	3900

Tabelle 7.2: Laufzeiten in Sekunden auf zwei Nachkommastellen gerundet und Speicherverbrauch in MiB für einen $432 \times 432 \times 432$ Level.

Die Cubes sind im Allgemeinen nicht sortiert. Wir schauen uns in Tabelle 7.3 an, wie für einen Extremfall, bei dem die Cubes in einer zufälligen Reihenfolge an die drei Programme übergeben werden, aussehen könnte. Dazu vermischen wir den Datensatz mit $416 \times 416 \times 416$ Cubes mit der Funktion `std::shuffle`.

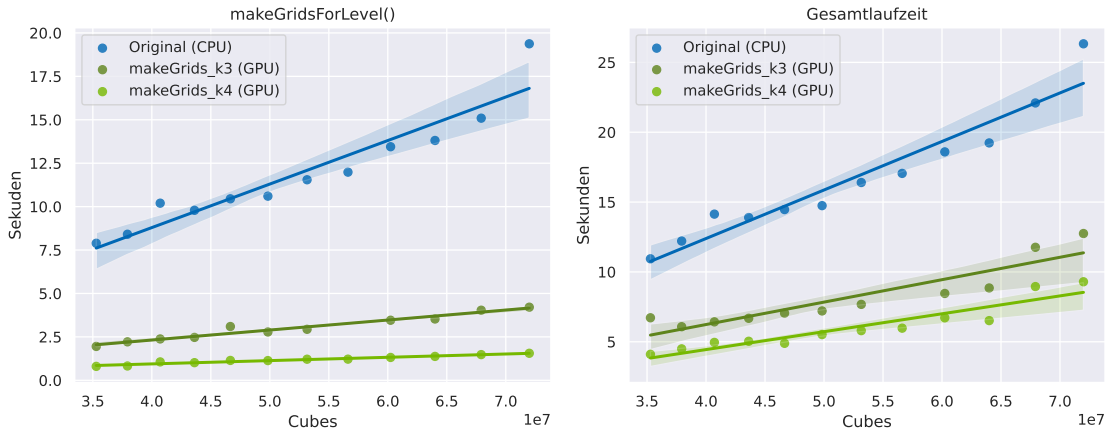
	Gesamtlaufzeit	`makeBricksForLevel()`	max. Speicher(GPU)
original	122.54	112.99	—
`makeGrids_k3`	16.24	6.27	3770
`makeGrids_k4`	13.70	4.25	3490

Tabelle 7.3: Laufzeiten in Sekunden und Speicherverbrauch in MiB für einen ungeordneten $416 \times 416 \times 416$ Level.

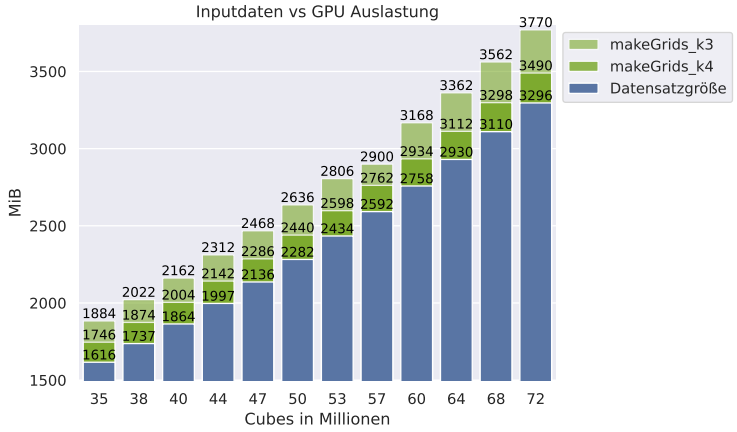
Es ist unwahrscheinlich, dass die Inputdaten in so einem Ausmaß durcheinander gespeichert sind. Dennoch wird anhand dieser Messung deutlich, dass falls eine gewisse Unordnung gegeben ist, die GPU Implementierungen im Vergleich zur CPU Version nur geringfügig an Laufzeit zunehmen. Dabei bleibt die Speicherauslastung, und somit auch die maximale mögliche Größe des Inputs, unverändert.

7.2 Datensatz: Variation der Anzahl der Gridlets

Um die Skalierbarkeit zu untersuchen, schauen wir uns den Zusammenhang zwischen der Anzahl der Gridlets beziehungsweise Cubes und der Laufzeit an. Obwohl wir die Größe der Gridlets auch hier auf $8 \times 8 \times 8$ setzen, kann sie auch anders gewählt werden. Für bessere Übersicht rechnen wir die Anzahl der Gridlets in Anzahl der Cubes um.



(a) (b)



(c)

Abbildung 7.2: (a) Ein Vergleich der Laufzeiten bei steigender Anzahl an Cubes für die `makeGridsForLevel()` Methode und der Gesamtlaufzeit in (b). Tatsächlich gemessene Zeitpunkte sind als Kreise dargestellt. Zusätzlich ergänzen wir die Darstellungen um eine lineare Regression.
(c)Ein Vergleich zwischen der Datensatzgröße und der zur Verarbeitung benötigten GPU Ressourcen. (Eigene Darstellung)

Wir können aus der Abbildung 7.2(a) und (b) entnehmen, dass die zuvor festgestellte Reihenfolge der Dauer für die `makeGridsForLevel()` Methode auch hier für die Gesamtdauer bestehen bleibt. Die originale Funktion ist am langsamsten, dann folgt die Implementierung mit drei Kernels und schließlich ist die Implementierung mit vier Kernels am schnellsten.

Es fällt auf, dass in den gemessenen Zahlen Ausreißer vorhanden sind. Wir können es zum einen darauf zurückführen, den wir in dem letzten Unterkapitel angesprochen haben: Die Cubes liegen in einer ungünstigen Reihenfolge. Zusätzlich spielen Messfehler und Hintergrundprozesse eine wichtige Rolle. Um diesen Effekt zu minimieren, führen wir die Messungen mehrmals durch und nehmen den durchschnittlichen Wert.

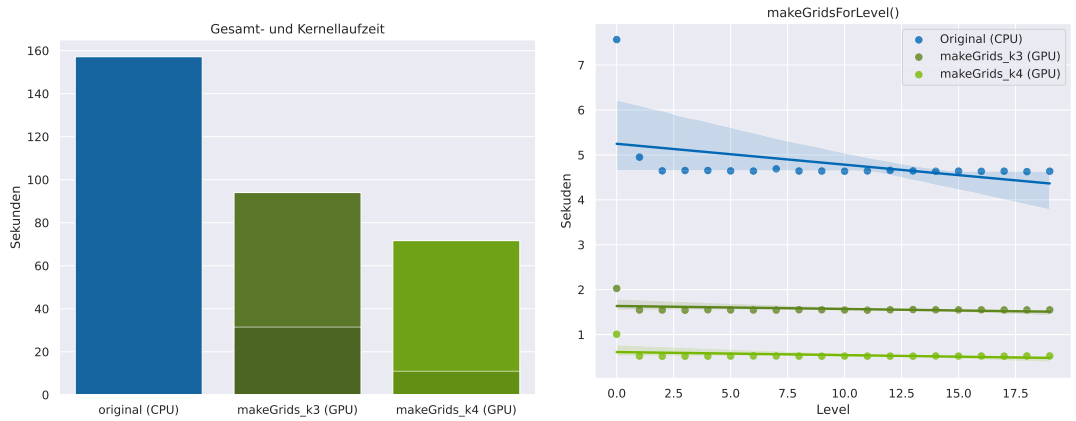
Dennoch lässt sich anhand der Daten ein Trend erkennen. Die CPU Variante steigt sowohl bei der Gesamtlaufzeit als auch der Funktion zum Berechnen der Gridlets `makeGridsForLevel()` deutlich stärker an als für die anderen beiden. Wir sehen außerdem, dass die Regressionsgerade eine größere Steigung bei der Gesamtlaufzeit im Vergleich zur `makeGridsForLevel()` hat. Für diesen Unterschied sind die Verarbeitungsschritte auf der CPU verantwortlich. Diese beinhalten das Einlesen der Cubes und das Schreiben der Gridlets.

Ein wesentlicher Nachteil vom Einsatz der GPU ist der kleinere Speicher. So tragen die zusätzlichen Variablen dazu bei, dass die höchstmögliche Datensatzgröße durch diese weiter eingeschränkt wird. Wie wir zuvor festgestellt haben, benötigt die `makeGrids_k3` mehr zusätzlichen Speicher als `makeGrids_k4`. Dieser steigt jedoch nur geringfügig mit steigender Datensatzgröße an.

7.3 Datensatz: tiefer Baum

Bei diesem Datensatz ist die Anzahl der Elemente für jedes der 20 Level gleich. Wie in Abbildung 7.3 deutlich wird, benötigt auch hier die CPU Version mehr Zeit. Dabei fällt in (a) auf, dass die reine GPU Zeit einen kleineren Prozentsatz ausmacht. Die Schritte auf der CPU, wie Datentransfer oder Schreiben der Gridlets, verlangsamen demnach

die Laufzeit der GPU Varianten um ein Vielfaches. Die Abbildung 7.3 (b) zeigt, dass im Schnitt die `makeGridsForLevel()` Methode eine relativ konstante Zeit beansprucht. Dabei fällt jedoch auf, dass die CPU Version tendenziell mehr Ausreißer hat. Alle drei Programme zeigen für das erste Level eine deutliche Verzögerung. Folglich könnte die Reihenfolge, in der die Level bearbeitet werden, von Bedeutung sein. Wenn sie sich, anders als hier in Anzahl der Cubes unterscheiden, so muss diese „warm-up Phase“ geschickt ausgenutzt werden.



(a) (b)

Abbildung 7.3: Datensatz mit einem tiefen Baum. CPU Version ist in blau, GPU Versionen sind in grün.

(a) Laufzeiten der drei Programme, in einer dunkleren Farbe ist bei den GPU Versionen jeweils die Summe an GPU Laufzeit hevorgehoben.

(b) Durchschnittliche Laufzeit für die `makeGridsForLevel()` Methode in Abhängigkeit von Leveln mit einer Regressionsgeraden. (Eigene Darstellung)

7.4 Datensatz: Cloud

Beim Ausführen von `makeGrids_k3` für das nullte Level reicht der Speicher beim Allokieren des Speichers für `listOfcubesIDXsforMC` nicht aus: CUDA Runtime API gibt beim `malloc` Befehl einen `cudaError_t` mit einem Fehlercode für `out of memory` zurück.

Im Gegensatz zu dem Datensatz aus dem vorherigen Kapitel können hier die Gridlets leere Zellen beinhalten. Wir tragen die Informationen über die Cubes, Gridelts und Makrozellen in Tabelle 7.4 zusammen.

	input C	C in Gridlets	leere C	# MC	# Gridlet	leere MC
$\ell = 0$	30643304	30664804	0.07%	4948560	63248	98.72%
$\ell = 1$	13207932	13234629	0.20%	640800	28122	95.61%
$\ell = 2$	34162517	34177166	0.04%	82800	70811	14.48%
$\ell = 3$	385900	386957	0.27%	10580	980	90.74%
$\ell = 4$	98	98	0%	1	1	0 %

Tabelle 7.4: Cloud Datensatz: Anzahl der Elemente nach Leveln ℓ und Cubes C.

Obwohl das zweite Level etwa 10.3% mehr Zellen hat, kann es ohne Fehlermeldungen korrekt verarbeitet werden (zur Korrektheit siehe Kapitel 5). Aus der geringen Anzahl der leeren Cubes in Gridlets und der relativ kleinen Anzahl an Gridlets an sich können wir ableiten, dass wir nicht viele dünn besetzte Gridlets haben. Was wir jedoch in der letzten Spalte sehen können, ist, dass der Anteil der leeren Makrozellen in Level 0 mit 98.72% wesentlich höher im Vergleich zu dem Anteil von 14.48% des zweiten Levels ist. Makrozellen stellen eine flächendeckende Zerlegung des ganzen Gebietes dar. Wenn die Cubes weit zerstreut sind, so sind auch die Gridlets weit zerstreut. Das führt zu einer Vielzahl an leeren Makrozellen. Das `listOfcubesIDXsforMC` ist ein Array, welches direkt mit der Anzahl an Makrozellen zusammenhängt, da es die Größe `numberOfMC * (macroCellWidth`^{3}`)` besitzt.

Wir benutzen es als eine Art Indexverzeichnis, welches den Makrozellen beziehungsweise den Gridlets ihre Cubes zuweist. Da `makeGrids_k3` nicht über die Makrozellen parallel läuft, braucht es diese zusätzlichen Informationen nicht.

Wir sehen in Abbildung 7.4, dass das Erzeugen der Gridlets sowohl einzeln als auch gemeinsam für alle Level bei der CPU-Variante um ein Vielfaches länger dauert. Wie erwartet sind die Laufzeiten proportional zu der Anzahl an Cubes.

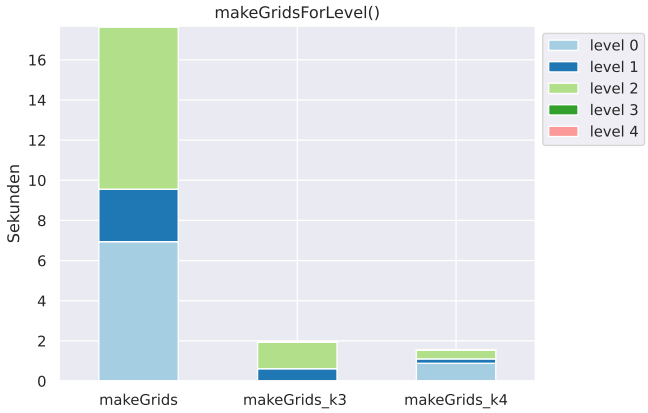


Abbildung 7.4: Laufzeiten des Cloud Datensatzes für die drei Varianten nach Leveln aufgelistet. Anmerkung: Bei `makeGrids_k3` fehlt der Level 0, da der Speicher der GPU nicht ausgereicht hat. (Eigene Darstellung)

In Abbildung 7.5 stellen wir zwei Computer gegenüber (vgl. Listing 5.1 und 5.2). PC 1 und PC 2 haben vergleichbare CPUs. Die Gesamtlauflaufzeit der ursprünglichen Variante ist signifikant länger im Vergleich zu den beiden neuen Varianten. PC 1 hat eine 1650 Grafikkarte mit 4GB Speicher (Listing 5.1) und PC 2 eine 2080 Grafikkarte mit 8GB Speicher (Listing 5.2).

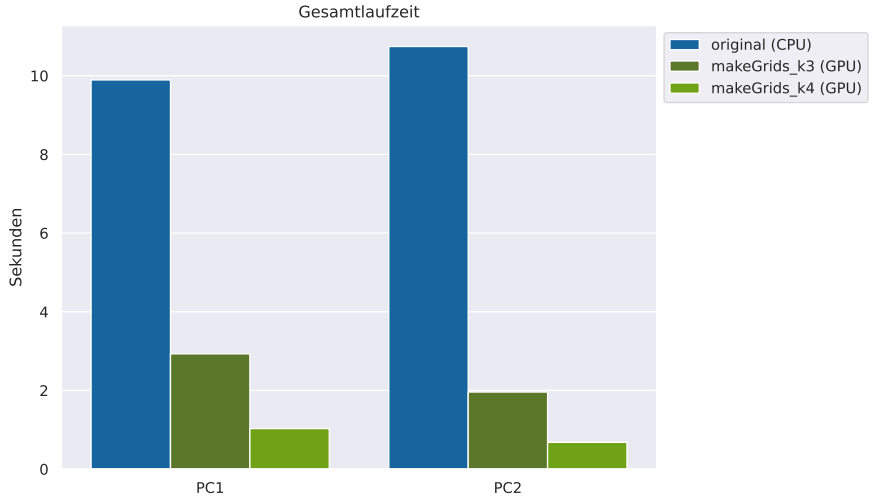


Abbildung 7.5: Gesamtlaufzeit für Level 1-4 des Cloud Datensatzes an zwei unterschiedlichen Systemen. Laufzeiten der originalen Implementierung in blau, neuen Implementierungen unter Einsatz der GPU in grün. (Eigene Darstellung)

Abbildung 7.6 bietet eine ausführliche Aufschlüsselung nach Level und vergleicht die Leistungen der beiden Computer im Detail. Im Schnitt ist PC 2 für die Methode `makeGridsForLevel()`, die ausschließlich die CPU benutzt, um etwa 10.07% langsamer. Bei der GPU macht sich der Unterschied der besseren Grafikkarte deutlich bemerkbar: Für `makeGrids_k3` ist PC 2 im Schnitt um 43.84% und bei `makeGrids_k4` um 26.47% schneller.

Ein detaillierter Vergleich der Kernellaufzeiten in der Tabelle 7.5 verdeutlicht den signifikanten Nutzen, den die Programme aus einer größeren GPU Speicherkapazität ziehen. Besonders auffällig ist dabei der vierte Kernel in `makeGrids_k4` mit einer Verbesserung von 85.74%.

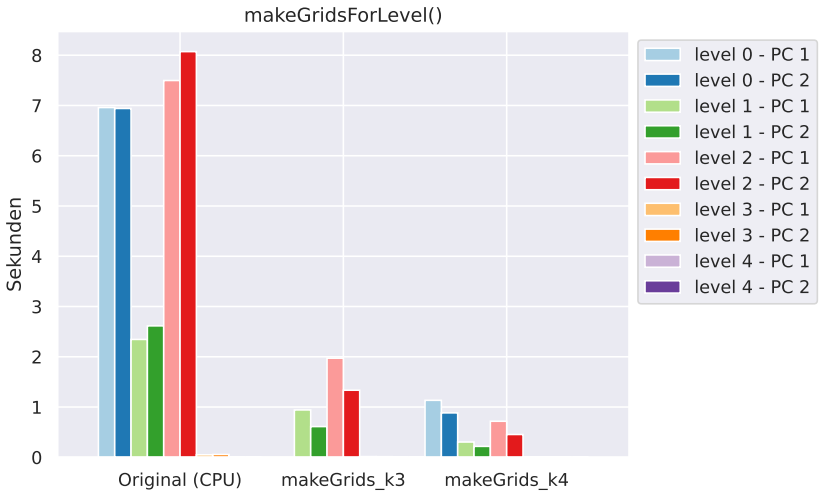


Abbildung 7.6: Vergleich zweier Grafikkarten für den Cloud Datensatz. Level 0 wird für die `makeGrids_k3` Variante aus Speichergründen nicht berücksichtigt. (Eigene Darstellung)

	makeGrids_k3			makeGrids_k4			
	kernel 1	kernel 2	kernel 3	kernel 1	kernel 2	kernel 3	kernel 4
PC 1	45.88	0.19	710.55	71.44	1.39	53.59	245.14
PC 2	13.72	0.06	280.11	21.62	0.41	20.09	34.95
Beschleunigung	70.10%	68.42%	60.58%	69.74%	70.50%	62.51%	85.74%

Tabelle 7.5: Laufzeiten der Kernel für Cloud auf PC 1 und PC 2 in Millisekunden, auf zwei Nachkommastellen gerundet. Gemessen mit `nvprof`.

8 Fazit

In dieser Arbeit haben wir unterschiedliche Arten gesehen, wie AMR Daten verarbeitet werden können. Im Prinzip lassen sich die untersuchten Methoden in zwei Kategorien einteilen: die, die auf den zellzentrischen Daten arbeiten und die, die ein duales Gitter verwenden.

Für eine Visualisierung mit guter Qualität sind Interpolanten nötig, die nicht nur auf homogenen Gebieten eines Levels C^0 stetig sind, sondern die Levelübergänge möglichst geschickt verbinden. Aus den unterschiedlichen Ansätzen resultieren unterschiedliche Herausforderungen und Probleme wie zum Beispiel das t-junction Problem. Zellmann et al. [25] lösen dieses Problem mithilfe von Stitchingelementen an den Levelgrenzen und den Einsatz eines dualen Gitters im Inneren. Eine besondere Rolle spielen hier die Gridlets, die die Zellen beziehungsweise die Würfelelemente des dualen Gitters bündeln. Die Autoren der Methode zeigen, dass dieser Ansatz effizient ist. Jedoch findet das Erzeugen der Gridlets in ihrer Umsetzung auf der CPU mit einer `std::map` statt. Sie schlagen vor, die Parallelität einer GPU auszunutzen, um die Berechnungen zu beschleunigen.

Wir haben diese Idee verfolgt und anhand mehrerer Datensätze gesehen, dass die Portierung zum einen gelungen ist und zum anderen tatsächlich eine Verbesserung in der Laufzeit aufzeigt. Dabei haben wir im Laufe der Entwicklung eine optimierte Version implementiert und mit der ersten verglichen. Die Veränderung hat zu einer besseren Laufzeit und geringerem GPU-Speicherverbrauch geführt. Wir haben ebenfalls eine Verbesserung in der Skalierbarkeit erreicht.

Insbesondere zeigt die zweite modifizierte Version gute Ergebnisse.

Weitere Verbesserungen der Umsetzung sind denkbar. Vor allem bleibt die Frage, ob die Anzahl der zahlreichen Hilfsarrays nicht durch eine andere Datenstruktur verringert werden kann.

Die Effizienz von Gridlets zeigt, dass diese in Zukunft auch weiterhin verwendet werden sollten. Insbesondere wenn es um große Datenmengen geht, ist es essenziel, diese Art von Datenstruktur in Betracht zu ziehen, da sie speichereffizient sind. Wenn wir die Gridelts als einen zusätzlichen Datentyp ansehen, so ist es möglich, diese in ein beliebiges Renderingprogramm, welches mit unstrukturierten Elementen arbeiten kann, einzubauen.

Wir haben die Konstruktion dieser beschleunigt, sodass in Zukunft eine Integration in solche Programme eine weitere Verbesserung im Vergleich zu den aktuellen Methoden verspricht.

Anhang

Die folgenden Quellcodedateien sind im Laufe der Arbeit entstanden:

- `cubesGeneration.cpp` - Erzeugen unterschiedlicher Testdatensätze
- `makeGrids3Kernels.cu` - Erzeugen der Gridlets, Modifikation von `makeGrids.cpp` (vgl. Kapitel 4.3.1)
- `makeGrids4Kernels.cu` - Erzeugen der Gridlets, Modifikation von `makeGrids.cpp` (vgl. Kapitel 4.3.2)
- `mem.sh` - Messen des GPU-Speichers
- `testBricksOutput.cpp` - paarweises Vergleichen der Gridletdateien

Literatur

[1] M. Adams, P. Colella, D. T. Graves, J. Johnson, N. Keen, T. J. Ligocki., D. F. M. P. McCorquodale, D. M. P. Schwartz, T. Sternberg und B. V. Straalen, „Chombo Software Package for AMR Applications Design Document“, Lawrence Berkeley National Laboratory Technical Report LBNL-6616E, Technical Report (siehe S. 44).

[2] J. Ahrens, B. Geveci und C. Law, „Visualization Handbook“, in C. D. Hansen und C. R. Johnson, Hrsg. Burlington, MA, USA: Elsevier Inc., 2005, Kap. ParaView: An End-User Tool for Large Data Visualization, S. 717–731. Adresse: https://www.sciencedirect.com/book/9780123875822/visualization-handbook (besucht am 01. 09. 2023) (siehe S. 7).

[3] M. Berger und P. Colella, „Local adaptive mesh refinement for shock hydrodynamics“, *Journal of Computational Physics*, Jg. 82, Nr. 1, S. 64–84, 1989, ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(89)90035-1. Adresse: https://www.sciencedirect.com/science/article/pii/0021999189900351 (besucht am 01. 09. 2023) (siehe S. 5, 17).

[4] M. J. Berger und J. Oliger, „Adaptive mesh refinement for hyperbolic partial differential equations“, *Journal of Computational Physics*, Jg. 53, Nr. 3, S. 484–512, 1984, ISSN: 0021-9991. DOI: https://doi.org/10.1016/0021-9991(84)90073-1. Adresse: https://www.sciencedirect.com/science/article/pii/0021999184900731 (besucht am 01. 09. 2023) (siehe S. 1, 4).

[5] J. O. Ferguson, C. Jablonowski und H. Johansen, „Assessing Adaptive Mesh Refinement (AMR) in a Forced Shallow-Water Model with Moisture“, *Monthly Weather Review*, Jg. 147, Nr. 10, S. 3673–3692, 2019. DOI: https://doi.org/

10.1175/MWR-D-18-0392.1. Adresse: https://journals.ametsoc.org/view/journals/mwre/147/10/mwr-d-18-0392.1.xml (besucht am 01. 09. 2023) (siehe S. 1).

[6] R. Franke und G. Nielson, „Smooth interpolation of large sets of scattered data“, *International Journal for Numerical Methods in Engineering*, Jg. 15, Nr. 11, S. 1691–1704, 1980. DOI: https://doi.org/10.1002/nme.1620151110. Adresse: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.1620151110 (besucht am 01. 09. 2023) (siehe S. 11).

[7] R. Kaehler und T. Abel, „Single-pass GPU-raycasting for structured adaptive mesh refinement data“, in *SPIE Proceedings*, P. C. Wong, D. L. Kao, M. C. Hao, C. Chen und C. G. Healey, Hrsg., SPIE, Feb. 2013. DOI: 10.1117/12.2008552. Adresse: https://doi.org/10.1117%2F12.2008552 (besucht am 01. 09. 2023) (siehe S. 32).

[8] R. Kaehler, J. Wise, T. Abel und H.-C. Hege, „GPU-Assisted Raycasting for Cosmological Adaptive Mesh Refinement Simulations“, in *Volume Graphics*, R. Machiraju und T. Moeller, Hrsg., The Eurographics Association, 2006, ISBN: 3-905673-41-X. DOI: 10.2312/VG/VG06/103-110 (siehe S. 32).

[9] N. Leaf, V. Vishwanath, J. Insley, M. Hereld, M. E. Papka und K. Ma, „Efficient parallel volume rendering of large-scale adaptive mesh refinement data“, in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, Los Alamitos, CA, USA: IEEE Computer Society, Okt. 2013, S. 35–42. DOI: 10.1109/LDAV.2013.6675156. Adresse: https://doi.ieeecomputersociety.org/10.1109/LDAV.2013.6675156 (besucht am 01. 09. 2023) (siehe S. 16).

[10] P. Ljung, C. F. Lundström und A. Ynnerman, „Multiresolution Interblock Interpolation in Direct Volume Rendering“, in *Eurographics Conference on Visualization*, 2006. Adresse: https://api.semanticscholar.org/CorpusID:16026643 (besucht am 01. 09. 2023) (siehe S. 16).

[11] W. E. Lorensen und H. E. Cline, „Marching Cubes: A High Resolution 3D Surface Construction Algorithm“, *SIGGRAPH Comput. Graph.*, Jg. 21, Nr. 4, S. 163–

169, Aug. 1987, ISSN: 0097-8930. DOI: 10.1145/37402.37422. Adresse: https://doi.org/10.1145/37402.37422 (besucht am 01. 09. 2023) (siehe S. 17).

[12] P. Moran und D. Ellsworth, „Visualization of AMR Data With Multi-Level Dual-Mesh Interpolation“, *IEEE Transactions on Visualization and Computer Graphics*, Jg. 17, Nr. 12, S. 1862–1871, Dez. 2011, ISSN: 1941-0506. DOI: 10.1109/TVCG.2011.252 (siehe S. 17, 18, 20, 21, 39).

[13] M. L. Norman, G. L. Bryan, R. Harkness, J. Bordner, D. Reynolds, B. O'Shea und R. Wagner, *Simulating Cosmological Evolution with Enzo*, 2007. arXiv: 0705.1556 [astro-ph] (siehe S. 1, 3).

[14] W. Schroeder, K. Martin und B. Lorensen, *The Visualization Toolkit (4th ed.)* Kitware, 2006, ISBN: 978-1-930934-19-1 (siehe S. 7).

[15] D. Seifried, S. Walch, P. Girichidis, T. Naab, R. Wünsch, R. S. Klessen, S. C. O. Glover, T. Peters und P. Clark, „SILCC-Zoom: the dynamic and chemical evolution of molecular clouds“, *Monthly Notices of the Royal Astronomical Society*, Jg. 472, Nr. 4, S. 4797–4818, Sep. 2017. DOI: 10.1093/mnras/stx2343. Adresse: https://doi.org/10.1093%2Fmnras%2Fstx2343 (besucht am 01. 09. 2023) (siehe S. 61).

[16] „Thrust Quick Start Guide“. Version 12.2. (2023), Adresse: https://docs.nvidia.com/cuda/pdf/Thrust_Quick_Start_Guide.pdf (besucht am 01. 09. 2023) (siehe S. 51).

[17] I. Wald, „A Simple, General, and GPU Friendly Method for Computing Dual Mesh and Iso-Surfaces of Adaptive Mesh Refinement (AMR) Data“, *CoRR*, Jg. abs/2004.08475, 2020. arXiv: 2004.08475. Adresse: https://arxiv.org/abs/2004.08475 (besucht am 01. 09. 2023) (siehe S. 23, 25, 40).

[18] I. Wald, C. Brownlee, W. Usher und A. Knoll, „CPU Volume Rendering of Adaptive Mesh Refinement Data“, in *SIGGRAPH Asia 2017 Symposium on Visualization*, Ser. SA '17, Bangkok, Thailand: ACM, 2017, 9:1–9:8, ISBN: 978-1-4503-5411-0. DOI: 10.1145/3139295.3139305. Adresse: http://doi.acm.org/10.1145/3139295.3139305 (besucht am 01. 09. 2023) (siehe S. 8, 11, 14, 15).

[19] I. Wald, S. Zellmann, W. Usher, N. Morrical, U. Lang und V. Pascucci, „Ray Tracing Structured AMR Data Using ExaBricks“, *CoRR*, Jg. abs/2009.03076, 2020. arXiv: 2009.03076. Adresse: https://arxiv.org/abs/2009.03076 (besucht am 01. 09. 2023) (siehe S. 31, 32, 35, 36, 40).

[20] F. Wang, I. Wald, Q. Wu, W. Usher und C. R. Johnson, „CPU Isosurface Ray Tracing of Adaptive Mesh Refinement Data“, *IEEE Transactions on Visualization and Computer Graphics*, 2019. DOI: 10.1109/TVCG.2018.2864850 (siehe S. 26, 30).

[21] F. Wang, N. Marshak, W. Usher, C. Burstedde, A. Knoll, T. Heister und C. R. Johnson, „CPU Ray Tracing of Tree-Based Adaptive Mesh Refinement Data“, *Computer Graphics Forum*, Jg. 39, Nr. 3, S. 1–12, 2020. DOI: https://doi.org/10.1111/cgf.13958. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13958. (besucht am 01. 09. 2023) (siehe S. 6, 31).

[22] G. H. Weber, H. Childs und J. S. Meredith, „Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (AMR) data“, in *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2012, S. 31–38. DOI: 10.1109/LDAV.2012.6378973 (siehe S. 8).

[23] G. H. Weber, O. Kreylos, T. J. Ligocki, J. Shalf, H. Hagen, B. Hamann und K. I. Joy, „Extraction of Crack-free Isosurfaces from Adaptive Mesh Refinement Data“, in *VisSym*, 2001. Adresse: https://api.semanticscholar.org/CorpusID:907552 (besucht am 01. 09. 2023) (siehe S. 16).

[24] S. Zellmann, I. Wald, A. Sahistan, M. Hellmann und W. Usher, „Design and Evaluation of a GPU Streaming Framework for Visualizing Time-Varying AMR Data“, in *Eurographics Symposium on Parallel Graphics and Visualization*, R. Bujack, J. Tierny und F. Sadlo, Hrsg., The Eurographics Association, 2022, ISBN: 978-3-03868-175-5. DOI: 10.2312/pgv.20221066 (siehe S. 37, 40).

[25] S. Zellmann, Q. Wu, K.-L. Ma und I. Wald, „Memory-Efficient GPU Volume Path Tracing of AMR Data Using the Dual Mesh“, *Computer Graphics Forum*, 2023,

ISSN: 1467-8659. DOI: 10.1111/cgf.14811 (siehe S. 2, 3, 8, 9, 19, 39, 41, 42, 44, 52, 57, 63, 79).

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Ort, Datum Unterschrift