IMPLEMENTIERUNG VON PARALLELEN REDUKTIONEN IN RISE UNTER VERWENDUNG VON SHUFFLE INSTRUCTIONS

PIET BJÖRN ADICK

Bachelorarbeit

im Fach Informatik

Westfälische Wilhelms Universität Münster

Oktober 2020

Piet Björn Adick: *Implementierung von parallelen Reduktionen in RISE unter Verwendung von Shuffle Instructions,* Bachelorarbeit

BETREUER:
Prof. Dr. Sergei Gorlatch
Bastian Köpcke

ORT:
Münster

ZEITRAUM:
Oktober 2020

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

QUELLTEXTVERZEICHNIS

1 EINLEITUNG

Reduktionen formen ein häufig auftretendes algorithmisches Muster, das für viele Anwendungen einen Grundbaustein bildet. Mithilfe von Summen-Reduktionen lassen sich beispielsweise Matrix-Vektor-Multiplikationen berechnen [14]. In der Graphentheorie gibt es zudem Anwendungen für Min-Reduktionen. Diese lassen sich nutzen, um den kürzesten Pfad zwischen zwei Knoten zu bestimmen [16] oder um mithilfe des Algorithmus von Prim einen minimalen Spannbaum zu konstruieren [19]. Bei der Konstruktion von k-d-Bäumen kommen zudem sowohl Min-Reduktionen, Max-Reduktionen, als auch Sum-Reduktionen zum Einsatz [20].

Um Reduktionen möglichst effizient zu implementieren, muss für die Berechnungen das Potenzial der Hardware im Zielcomputersystem voll ausgeschöpft werden. Die maximale Leistung moderner Computersysteme lässt sich, aufgrund der oftmals großen Anzahl an Prozessorkernen, nur durch parallel ausgeführte Berechnungen annähernd vollständig abrufen. Reduktionen auf Graphics Processing Units (kurz: GPUs) sind hier von besonders großem Interesse, da sie aufgrund ihrer großen Anzahl von Prozessorkernen oftmals eine deutlich höhere theoretische Maximalleistung als CPUs bieten.

Die Programmierung von GPUs findet meist über Schnittstellen wie OpenCL [12] oder die NVIDIA-spezifische Plattform CUDA [2] statt. In diesen Programmiermodellen wird, verglichen mit CPU-Programmierung, eine sehr viel größere Anzahl an Threads verwendet. Außerdem müssen sich die Programmierenden eigenständig um Details wie die Speicherverwaltung kümmern. Durch diesen Zuwachs an Komplexität wird die Fehleranfälligkeit bei der Programmierung deutlich erhöht. Da sich GPUs in ihrer Architektur teilweise stark voneinander unterscheiden und verschiedene architekturspezifische Hardwarefeatures existieren, die explizit programmiert werden müssen, erfordert das Erreichen einer möglichst hohen Leistung zudem gerätespezifische Optimierungen.

Eine interessante Eigenschaft von GPU-Programmen ist, dass Threads von der Hardware in Gruppen der Größe 32 aufgeteilt werden. Diese Gruppen von Threads werden *Warps* genannt. Jeder Thread hat Zugriff auf eine Menge von ihm zugewiesenen Hardwareregistern. Diese sind sehr viel schneller als andere Arten von Speicher und daher von großem Interesse für den Austausch von Daten mit anderen Threads. Bis vor Kurzem konnte jedoch nur innerhalb eines Threads auf die eigenen Hardwareregister zugegriffen werden. Um dieses Problem zu umgehen, stellt CUDA für neuartige NVIDIA-GPUs sogenann-

te *Shuffle Instructions* zur Verfügung. Threads können hierdurch die Register anderer Threads innerhalb des selben Warps auslesen. An dieser Stelle kann der Datenaustausch über langsameren, geteilten Speicher vermieden werden. Mit Shuffle Instructions lassen sich unter anderem Berechnungen von Präfixsummen [4], Convolutions [17] und der Levenshtein-Distanz [10] optimieren. Auch für Reduktionen bieten Shuffle Instructions großes Potenzial. Es gibt zum Beispiel eine von NVIDIA veröffentlichte CUDA-Implementierung einer auf Shuffle Instructions basierenden parallelen Reduktion [15]. Zudem existiert mit CUB [6] eine Bibliothek von NVIDIA, welche unter anderem mit Shuffle Instructions optimierte Implementierungen von parallelen Reduktionen beinhaltet [15].

RISE [18] ist eine high-level funktionale Sprache mit dem Ziel, die Fehleranfälligkeit und den Aufwand bei der Implementierung von hochperformantem GPU-Code zu verringern. Die Grundlage der Sprache sind sogenannte *Patterns*. Diese abstrahieren häufig in GPU-Programmen auftretende Programmiermuster. Unterschiedliche Patterns können als Funktionen zu einem komplexeren GPU-Programm zusammengefügt werden. Bei diesen GPU-Programmen werden dann automatisiert gerätespezifische Optimierungen vorgenommen, um daraus für unterschiedliche Hardwarearchitekturen hochperformanten OpenCL- oder CUDA-Code zu generieren.

In dieser Arbeit präsentieren wir neu entwickelte Patterns, welche die Nutzung von Shuffle Instructions in RISE ermöglichen. Diese abstrahieren die Zugriffsmuster der jeweiligen Shuffle Instructions und werden vom RISE-Compiler Shine zur Generierung von hochperformantem CUDA-Code genutzt. Unter Verwendung dieser Patterns werden parallele Reduktionen als Beispielanwendung in low-level RISE entwickelt. Für diese Beispielanwendungen wird dann CUDA-Code generiert, der Shuffle Instructions verwendet.

Wir evaluieren unsere Erweiterung von RISE und die auf Shuffle Instructions basierende Implementierung einer Reduktion auf GPUs, indem wir die Laufzeiten unseres generierten Codes mit einer von NVIDIA veröffentlichten CUDA-Implementierung und mit CUB, einer Programmbibliothek von NVIDIA, vergleichen. Wir nehmen außerdem Vergleiche mit Implementierungen vor, welche auf die Nutzung von Shuffle Instructions verzichten und stattdessen auf den Shared Memory zurückgreifen.

1.1 HINTERGRUND

In diesem Kapitel erklären wir zunächst das Prinzip hinter parallelen Reduktionen. Daraufhin zeigen wir den Aufbau von Graphics Processing Units und deren Programmierung über CUDA. Dabei gehen näher auf die Verwendung von Shuffle Instructions in CUDA ein, da diese Optimierungen bei parallelen Reduktionen auf GPUs ermögli-

chen. Schlussendlich präsentieren wir die Grundlagen der Sprache RISE, da wir in dieser Sprache Shuffle Instructions nutzen wollen, um damit parallele Reduktionen zu implementieren.

1.1.1 *Parallele Reduktionen*

Gegeben einem Array $A = [a_1, a_2, ..., a_n]$ und einem binären assoziativen Operator $\oplus$ lässt sich das Ergebnis r einer Reduktion wie folgt berechnen:

$$r = (...(a_1 \oplus a_2) \oplus a_3)...) \oplus a_n)$$

Reduktionen lassen sich unter den diesen Bedingungen jedoch nur sequenziell berechnen.

Fordert man jedoch zusätzlich zu diesen Bedingungen, dass der Operator $\oplus$ kommutativ sein muss, so lassen sich Reduktionen auch parallel berechnen. Durch einen kommutativen Operator wird das Ergebnis der Reduktion nämlich, abseits von möglicherweise im anderen Maße auftretenden Genauigkeitsverlusten bei Gleitkommazahlen, unabhängig von der Reihenfolge, in welcher die Daten verarbeitet werden.

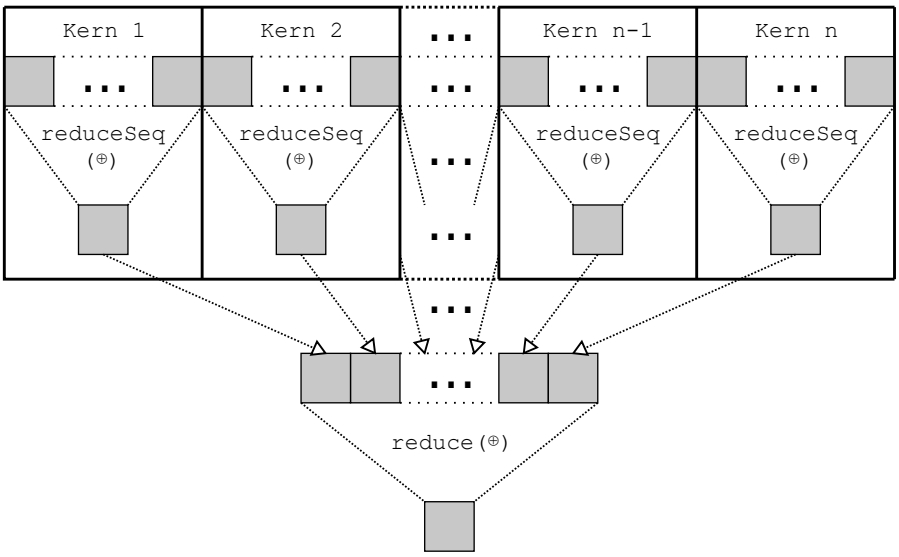

Abbildung 1.1: Grundlegender Aufbau einer parallelen Reduktion in zwei Schritten

Wie in Abbildung 1.1 illustriert, lässt sich eine solches Array auf mehrere Arrays aufteilen, um die Ergebnisse der Reduktionen für diese Teilarrays auf jeweils einem Prozessorkern sequenziell durchzuführen. Reduziert man diese Teilergebnisse nun mit dem Reduktionsoperator, so hat man das Ergebnis der Reduktion für das gesamte Array berechnet und dafür die Rechenleistung mehrerer Prozessorkerne genutzt.

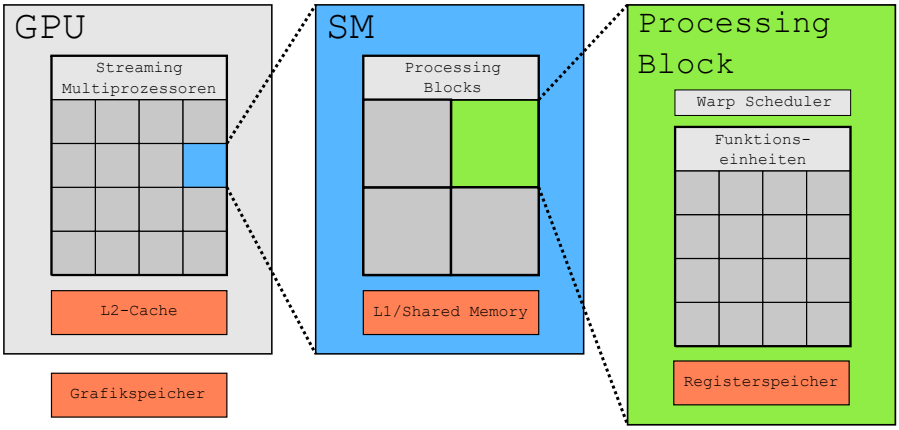

Abbildung 1.2: Vereinfachte Darstellung des Aufbaus einer GPU

1.1.2 *Graphics Processing Units*

Graphics Processing Units (kurz: GPUs) sind auf grafische Berechnungen spezialisierte Prozessoren, welche seit einigen Jahren vermehrt auch für general-purpose-Berechnungen eingesetzt werden. GPUs zeichnen sich durch ihre im Vergleich zu CPUs große Anzahl an Prozessorkernen aus. Die Kerne einer GPU sind zwar deutlich langsamer als CPU-Kerne, eine GPU weist jedoch, aufgrund der sehr viel größeren Anzahl an Kernen, eine höhere theoretische Maximalleistung auf.

Der Aufbau einer GPU wird in Abbildung 1.2 vereinfacht dargestellt. Eine GPU besteht aus mehreren Streaming Multiprozessoren. Diese können gemeinsam den Grafikspeicher und den L2-Cache nutzen. Der Grafikspeicher ist sowohl der größte, als auch der langsamste Speicher, der auf einer GPU zur Verfügung steht. Ein direkter Zugriff auf den RAM und den Festplattenspeicher ist von der GPU aus nicht möglich. Ein einzelner Streaming Multiprozessor hat zudem einen eigenen Speicherbereich, welcher für den L1-Cache und den Shared Memory benutzt wird. Der Shared Memory besteht aus mehreren Memory Banks, auf welche jeweils nur sequenzielle Zugriffe möglich sind. Die Größe dieser Memory Banks unterscheidet sich je nach Architektur der entsprechenden GPU.

Bei einigen modernen GPUs bestehen die Streaming Multiprozessoren aus mehreren *Processing Blocks* [3]. Jeder dieser Processing Blocks verfügt über mehrere Funktionseinheiten und einen Registerspeicher. Ausführungen werden innerhalb eines Processing Blocks über den sogenannten *Warp Scheduler* verwaltet.

Die Programmierung von GPUs findet meist über Schnittstellen wie OpenCL [12] oder das NVIDIA-spezifische CUDA [2] statt.

1.1.3 *Das CUDA-Programmiermodell*

CUDA ist eine von NVIDIA veröffentlichte Schnittstelle [2] zur *general purpose*-Programmierung von NVIDIA-GPUs. Hierfür werden Programmiersprachenerweiterungen, unter anderem für C++, angeboten. Sprechen wir folgend von CUDA-Code, so ist damit CUDA C++ gemeint. CUDA-Programme, auch Kernels genannt, sind wegen der großen Anzahl an Prozessorkernen von GPUs für eine große Anzahl an Threads ausgelegt.

Aufgrund der komplexen Architektur von GPUs gibt es verschiedene Strukturen, in welchen diese Threads organisiert werden. Threads werden in CUDA explizit von den Programmierenden in *Blocks* gruppiert. Die Threads innerhalb eines Blocks lassen sich in bis zu drei Dimensionen anordnen. Die Struktur, in welcher die Blocks organsiert werden, wird *Grid* genannt. Hier ist ebenfalls eine Anordnung in bis zu drei Dimensionen möglich.

Die Grids werden jeweils einer GPU zur Ausführung zugewiesen. Die Blocks innerhalb eines Grids werden hingegen jeweils einem Multiprozessor auf einer GPU zur Ausführung zugewiesen. Die Threads innerhalb eines Blocks werden von der GPU außerdem implizit in *Warps* aufgeteilt. Diese bestehen aus 32 Threads, welche im Kontext eines Warps auch als *Lanes* bezeichnet werden.

CUDA-Programme beschreiben, welche Instruktionen von einem einzelnen Thread ausgeführt werden. Damit einzelne Threads auf jeweils verschiedenen Daten arbeiten und untereinander kommunizieren können, kann innerhalb eines CUDA-Programms die Größe des Grids und der Blocks, die ID des jeweiligen Blocks innerhalb des Grids und die ID des jeweiligen Threads innerhalb des jeweiligen Blocks abgefragt werden. Da die Struktur, in welcher die Warps organisiert werden, fest vorgegeben ist, lässt sich zudem die Lane-ID und die Warp-ID aus diesen Informationen berechnen.

Die Threads innerhalb eines Blocks und die Lanes innerhalb eines Warps können jeweils Barrierensynchronisierungen durchführen, was für die Kommunikation und Kooperation auf der jeweiligen Hierarchiestufe notwendig ist.

Die Nutzung des Speichers einer GPU fußt auf dieser hierarchischen Ausführungsstruktur. Hierbei steht im Kontext der gesamten GPU der Global Memory zur Verfügung. Dieser entspricht auf der Hardwareebene dem Grafikspeicher. Im Kontext eines Blocks lässt sich zudem der Shared Memory nutzen, welcher dem Speicher eines Multiprozessors entspricht. Dieser ist aufgrund der Nähe zum jeweiligen Multiprozessor und einer anderen Konstruktionsweise deutlich schneller als der Global Memory, dafür jedoch auch deutlich kleiner.

Je nach GPU-Architektur haben die jeweiligen Speicherbereiche gewisse Eigenschaften. Bei Zugriffen auf den Global Memory ist es beispielsweise möglich, mithilfe von gewissen Zugriffsmustern die

Speicherzugriffe mehrerer Threads in eine Ladeoperation zusammenzufassen und so den Gesamtaufwand für Speicherzugriffe zu verringern [9]. Der Shared Memory besteht aus mehreren Memory Banks, auf welche jeweils nur sequenzielle Zugriffe möglich sind. Wollen mehrere Threads gleichzeitig auf eine Memory Bank zugreifen, so spricht man von einem Bank Conflict, welcher aufgrund des negativen Einflusses auf die Performance, soweit möglich, vermieden werden sollte.

Threads können außerdem Daten in Registern hinterlegen. Diese Register sind deutlich schneller als die zuvor erwähnten Formen von Speicher und somit von besonderem Interesse für den Austausch von Daten zwischen Threads. Der Zugriff auf diese Register ist jedoch üblicherweise nur für den jeweiligen Thread möglich. In CUDA gibt es, um den effizienten Datenaustausch unter Verwendung von Registern zu ermöglichen, *Shuffle Instructions*.

Die Speicherverwaltung in CUDA ist, abseits vom Caching im L1- und im L2-Cache, Aufgabe der Programmierenden. Dies erhöht durch die dadurch steigende Komplexität der Programme die Fehleranfälligkeit bei der Programmierung. Durch die teils sehr großen Unterschiede zwischen verschiedenen GPU-Architekturen müssen zudem viele gerätespezifische Optimierungen vorgenommen werden, wodurch der Aufwand bei der Programmierung weiter steigt und spezifisches Wissen über die jeweiligen Architekturen von den Programmierenden erfordert wird.

1.1.4 *Shuffle Instructions in CUDA*

In diesem Kapitel zeigen wir, wie Shuffle Instructions in CUDA verwendet werden und gehen dabei auf die besonderen Eigenschaften der jeweiligen Instruktionen ein. Hierbei beziehen wir uns auf den CUDA C++ Programming Guide von NVIDIA [1].

Shuffle Instructions ermöglichen seit der *Kepler*-Generation (Compute Capability 3.0) von NVIDIA eine effiziente Form des Datenaustauschs zwischen den Lanes innerhalb eines Warps. Jede Lane kann hier im Zuge einer Shuffle Instruction einen Wert übermitteln. Welche Lanes dann diesen Wert als Rückgabe erhalten, ist abhängig von der jeweiligen Shuffle Instruction und den Parametern, die hierbei angegeben werden.

Die Datentypen, welche für die zu übermittelnden Variablen unterstützt werden, sind folgende:

Ganzzahlen	Gleitkommazahlen	Vektortypen
int	float	__half2
unsigned int	double	__nv_bfloat162
long	__nv_bfloat16	
unsigned long		
long long		
unsigned long long		
__half		

Wir werden für diese Datentypen folgend *T* als Platzhalter nutzen. Die CUDA-spezifischen Datentypen __half, __half2, __nv_bfloat16 und __nv_bfloat162 lassen sich dabei nur auf manchen aktuellen NVIDIA-GPUs nutzen. Die Rückgabewerte aller Shuffle Instructions sind vom Typ *T*.

__shfl_sync

Parameter:

- unsigned mask
- T var
- int srcLane
- int width = warpSize

Die allgemeinste Shuffle Instruction ist *__shfl_sync*. Hier wird die Lane-ID der Lane, von welcher der Rückgabewert stammen soll, explizit als Parameter *srcLane* der Instruktion angegeben. Da die *srcLane* in jeder einzelnen Lane individuell berechnet werden kann, ist es möglich über eigens definierte Muster Variablen auszutauschen.

Der Parameter *mask* gibt an, welche Lanes an der Shuffle Instruction beteiligt sind. Sollte dabei ein Zugriff auf eine nicht aktive Lane erfolgen, führt dies zu undefiniertem Verhalten.

Mithilfe des *width*-Parameters lässt sich der Warp in mehrere logische Einheiten unterteilen. Hierbei sind Größen von 2, 4, 8, 16 und der *warpSize* von 32 möglich. Damit keine Zugriffe außerhalb dieser Einheiten erfolgen, wird die Lane-ID der Lane, aus welcher der Wert gelesen werden soll, durch $srcLane \mod width$ berechnet.

__shfl_down_sync

Parameter:

- unsigned mask
- T var

- unsigned int delta
- int width = warpSize

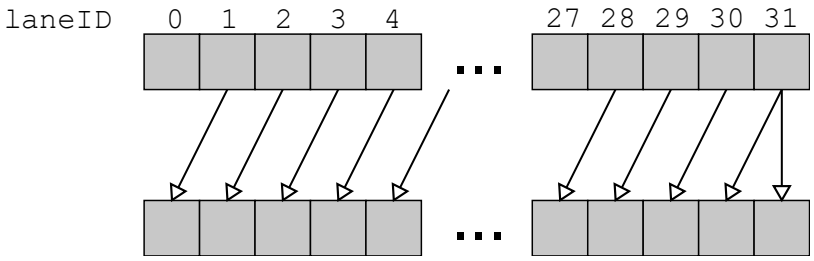

Abbildung 1.3: __shfl_down_sync(oxffffffff, val, 1)

Die Instruktion *__shfl_down_sync* lässt sich nutzen um die Werte von Lanes mit größerer Lane-ID als der eigenen zu erhalten. Der Parameter *delta* gibt dabei an, um wie viel die entsprechende Lane-ID größer sein soll. In Abbildung 1.3 wird das Verhalten eines beispielhaften Aufrufs mit *delta* = 1 illustriert.

Die Funktionsweise von *mask* ist identisch zu der bei *__shfl_sync*. Die Sektionierung über *width* hat zur Folge, dass die Lanes, die aus einer Lane außerhalb der Grenzen ihrer Sektion lesen wollen, ihren eigenen Wert als Rückgabe erhalten. Dies betrifft in Abbildung 1.3 die Lane mit der Lane-ID 31.

__shfl_up_sync

Parameter:

- unsigned mask
- T var
- unsigned int delta
- int width = warpSize

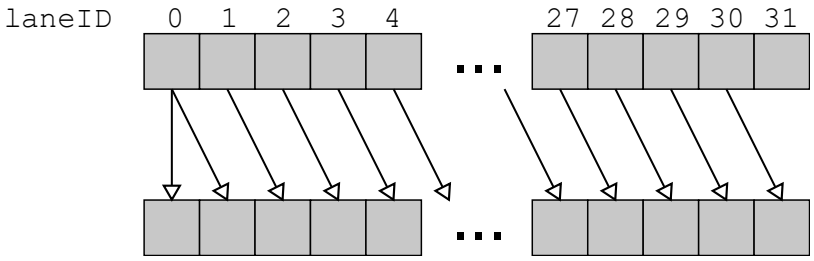

Abbildung 1.4: __shfl_up_sync(oxffffffff, val, 1)

Die Instruktion *__shfl_up_sync* verhält sich symmetrisch zu *__shfl_down_sync* und ist mit dieser Instruktion in ihren Eigenschaften ansonsten identisch. In Abbildung 1.4 ist der Datentransfer zwischen den Lanes bei einem exemplarischen Aufruf illustriert.

__shfl_xor_sync

Parameter:

- unsigned mask
- T var
- int laneMask
- int width = warpSize

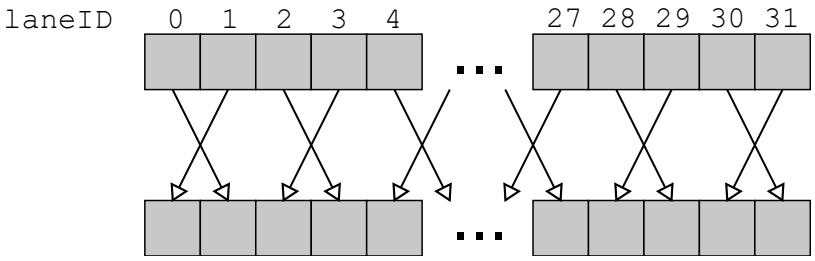

Abbildung 1.5: __shfl_xor_sync(oxffffffff, val, 1)

Um bei *__shfl_xor_sync* die Lane zu bestimmen, von welcher der Rückgabewert kommt, wird ein bitweises *xor* zwischen der eigenen Lane-ID und dem Parameter *laneMask* durchgeführt. Die Funktionsweise von *mask* ist hierbei identisch zu der bei den anderen Shuffle Instructions, bei *width* gibt es jedoch Unterschiede. Werte aus Sektionen mit kleineren Lane-IDs können nämlich gelesen werden, während dies andersherum nicht möglich ist. Das Verhalten bei einem exemplarischen Aufruf ist in Abbildung 1.5 illustriert.

1.1.5 *RISE*

RISE [18] ist eine high-level funktionale Sprache mit dem Ziel, die Fehleranfälligkeit und den Aufwand bei der Implementierung von hochperformantem GPU-Code zu verringern. Die Grundlage der Sprache sind sogenannte *Patterns*. Diese abstrahieren häufig in GPU-Programmen auftretende Programmiermuster. Unterschiedliche Patterns können als Funktionen zu einem komplexeren GPU-Programm zusammengefügt werden. Dabei beschreiben Programme *was*, aber nicht *wie* etwas zu berechnen ist.

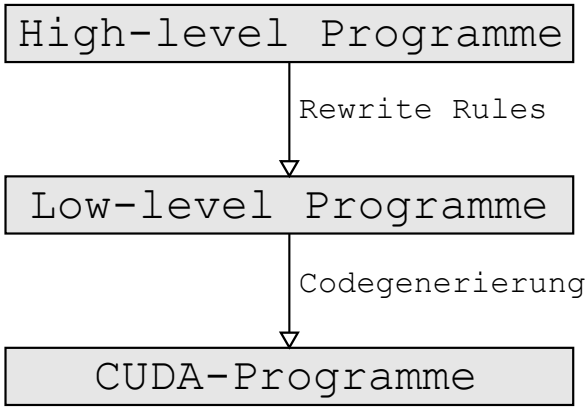

Abbildung 1.6: Übersetzung von RISE-Programmen

Wie in Abbildung 1.6 dargestellt, werden diese high-level Programme vom RISE-Compiler Shine mithilfe von semantikerhaltenden *Rewrite Rules* in gerätespezifisch optimierte low-level Programme transformiert. Anhand eines solchen transformierten Programms wird daraufhin hochperformanter OpenCL- oder CUDA-Code generiert. Dies ermöglicht es, mit einer einzelnen high-level Implementierung in RISE hochperformante Programme für unterschiedliche Hardwarearchitekturen zu generieren. Der Fokus dieser Arbeit liegt dabei auf low-level Programmen und deren Übersetzung in CUDA-Code. Die Transformierung von high-level Programmen wird hierbei nicht betrachtet.

Das CUDA-Backend und der CUDA-Executor für RISE, welche die Generierung und Ausführung von CUDA-Code über Shine ermöglichen, wurden erst kürzlich im Rahmen eines Projektseminars entwickelt [11]. Die Verwendung einiger CUDA-spezifischer Hardwarefeatures, darunter auch Shuffle Instructions, wurde hier jedoch noch nicht unterstützt.

1.2 VERWANDTE ARBEITEN

CUB CUB [6] ist eine Programmbibliothek von NVIDIA, welche Implementierungen mehrerer Primitive, darunter unter anderem ein Reduktions-Primitiv, für die verschiedenen Stufen der CUDA-Ausführungshierarchie beinhaltet. Die Implementierungen auf Warp-Ebene werden hierbei, sofern dies auf der verwendeten GPU möglich ist, mit Shuffle Instructions beschleunigt.

1.3 BEITRÄGE

LOW-LEVEL-PRIMITIVES Definition und Implementierung der low-level Primitive *shflWarp*, *shflDownWarp*, *shflUpWarp* und *shflXorWarp* zur Abstraktion der Funktionsweisen von Shuffle Instructions in RISE

CODEGENERIERUNG ÜBER SHINE Implementierung der Generierung von CUDA-Code für die low-level Primitive *shflWarp*, *shflDownWarp*, *shflUpWarp* und *shflXorWarp* zur Ermöglichung der Nutzung von Shuffle Instructions in RISE

REDUCE-KERNEL MIT SHUFFLE-PRIMITIVEN Implementierung eines low-level Programms zur parallelen Reduktion mit Shuffle-Primitiven in RISE, um vom Vorteil der Nutzung von Shuffle Instructions bei einer praktischen Anwendung Gebrauch zu machen

2 PARALLELE REDUKTIONEN AUF GPUS

In diesem Kapitel erklären wir, wie sich parallele Reduktionen auf GPUs durchführen lassen. Wir gehen dabei auf verschiedene Herangehensweisen für Reduktionen auf den jeweiligen Ebenen der CUDA-Ausführungshierarchie ein.

2.1 REDUKTION AUF DEVICE-EBENE

Es gibt mehrere Herangehensweisen zur Berechnung von parallelen Reduktionen auf Device-Ebene. Eine einfache Herangehensweise ist es beispielsweise, die Daten zunächst global auf alle Threads zu verteilen. Sind mehr Werte als Threads vorhanden, dann reduziere die einzelnen Threads zunächst sequenziell mehrere Werte. Dabei ist es besonders wichtig, die Zugriffe auf den Global Memory so zu optimieren, dass möglichst viele Speicherzugriffe zusammengefasst werden können.

Hierfür eignen sich in CUDA sogenannte *Grid-Stride Loops* [8]. Im Quelltext 2.1 wird eine solche Grid-Stride-Loop genutzt, um eine sequenzielle Reduktion, verteilt auf alle Threads, durchzuführen. Diese sorgen dafür, dass zusammenliegende Threads auf zusammenliegende Speicherbereiche zugreifen und maximieren so die Anzahl der zusammengefassten Zugriffe auf den Global Memory, was einen deutlichen Performancevorteil mit sich bringt. Außerdem bieten sie eine einfache Möglichkeit, das Programm mit einer beliebigen Anzahl an Threads ausführen zu können.

Die anfängliche Verteilung der Daten kann auf Device-Ebene auch so durchgeführt werden, dass die Daten zunächst auf die Blocks verteilt werden. Dies führt dann, wie bei einer globalen Verteilung, unter Umständen dazu, dass die Threads auf Block-Ebene sequenziell reduzieren müssen.

Sind die sequenziellen Reduktionsschritte abgeschlossen, so hat jeder Thread ein Teilergebnis der vollständigen Reduktion berechnet.

```
int sum = 0;
for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N;
 ↪  i += blockDim.x * gridDim.x) {
  sum += in[i];
}
```

Quelltext 2.1: Grid-Stride-Loop zur sequenziellen Reduktion in CUDA C++ [15]

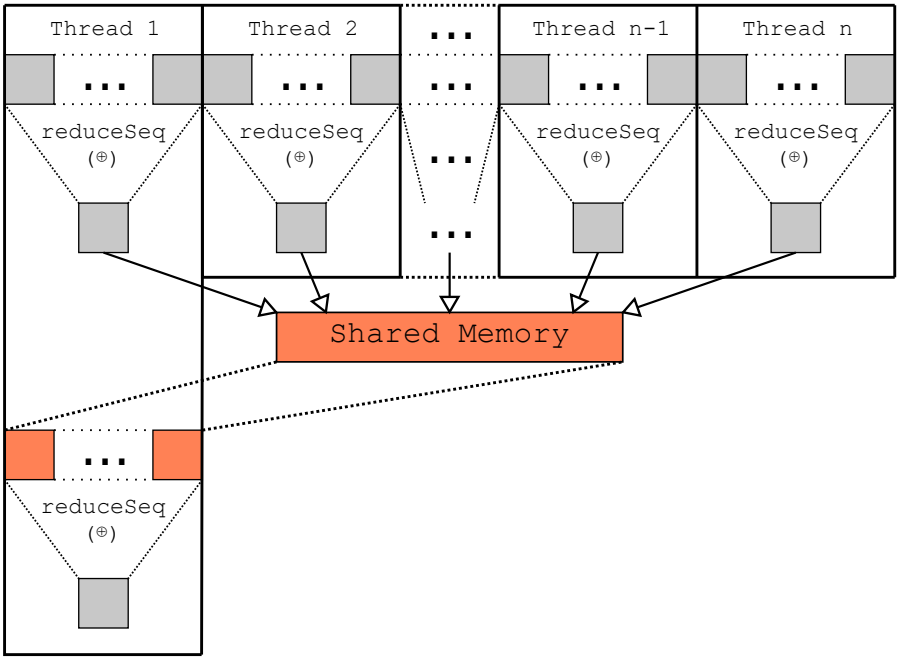

Abbildung 2.1: Parallele Reduktion mit Shared Memory (naiver Ansatz)

Da diese Teilergebnisse nun ebenfalls reduziert werden müssen, ist nun der Datenaustausch zwischen mehreren Threads erforderlich. Da GPUs keine Synchronisierung über einem gesamten Device ermöglichen, muss diese Reduktion zunächst auf der darunter liegenden Block-Ebene durchgeführt werden. Aus dem selben Grund muss, bei der Verwendung mehrerer Blocks, eine zweite Reduktion mit nur einem Block durchgeführt werden.

2.2 REDUKTION AUF BLOCK-EBENE

Sollte man auf Device-Ebene eine Herangehensweise gewählt haben, bei welcher die Daten ohne vorherige sequenzielle Reduktion auf die Blocks verteilt werden, muss in den einzelnen Threads innerhalb eines Blocks zunächst eine sequenzielle Reduktion durchgeführt werden. Die durch die sequenzielle Reduktion entstandenen Teilergebnisse müssen nun ebenfalls reduziert werden. Da einzelne Threads nun nur noch jeweils einen Wert halten, ist der Austausch von Daten zwischen den Threads notwendig. Auf Block-Ebene haben die Threads gemeinsam auf den Shared Memory, über den sie Daten austauschen können, Zugriff.

In Abbildung 2.1 wird ein naiver Ansatz dargestellt, bei welchem alle Threads zunächst einige Werte sequenziell reduzieren, um dann ihr berechnetes Teilergebnis in den Shared Memory zu schreiben. Diese Teilergebnisse werden dann vom ersten Thread aus dem Shared Memory gelesen, um erneut sequenziell zu reduzieren. Dies sorgt zwar wie gewünscht dafür, dass das Ergebnis der Reduktion auf

Block-Ebene vollständig berechnet wird, führt jedoch auch zu dem Problem, dass die GPU während dieser sequenziellen Reduktion nicht gut ausgelastet wird.

Eine Verbesserung der Performance ist in vielerlei Hinsicht möglich. In einem Vortrag von Mark Harris [7] wird eine CUDA-Implementierung vorgestellt, die einen Baum-basierten Ansatz zur parallelen Reduktion innerhalb eines Blocks wählt. Durch diesen können mehrere Threads gleichzeitig an der weiteren Reduktion teilnehmen und so die GPU besser auslasten. Es wurden hier zudem beispielsweise Optimierungen im Bezug auf die Speicherzugriffe vorgenommen.

2.3 REDUKTION AUF WARP-EBENE UNTER VERWENDUNG VON SHUFFLE INSTRUCTIONS

Eine weitere Möglichkeit, Reduktionen auf GPUs zu optimieren, sind *Shuffle Instructions*. Im NVIDIA Developer Blog wurden verschiedene Implementierungen vorgestellt, die Shuffle Instructions nutzen, um Reduktionen auf Warp-Ebene durchzuführen [15]. Diese werden dann genutzt, um Reduktionen auf Block- und Device-Ebene zu beschleunigen.

Im Verlauf dieser Arbeit werden RISE-Primitive für Shuffle Instructions entwickelt. Diese werden dann zur Berechnung von Reduktionen auf der Warp-Ebene genutzt. Um die Entwicklung von Primitiven für Shuffle Instructions zu erläutern, betrachten wir zunächst wie parallele Reduktionen in RISE mit vorhandenen Patterns durchgeführt werden.

3

PARALLELE REDUKTIONEN IN RISE

In diesem Kapitel erklären wir, wie sich parallele Reduktionen in RISE umsetzen lassen. Dabei beziehen wir uns auf die Ansätze aus Kapitel 2. Wir stellen zunächst einige dafür notwendige low-level Primitive vor. Wir zeigen daraufhin, wie sich unter Verwendung dieser Primitive eine Reduktion auf Device-Ebene durchführen lässt. Diese führt eine Reduktion auf Block-Ebene durch, welche wir ebenfalls näher thematisieren werden.

3.1 LOW-LEVEL-PRIMITIVE IN RISE

In RISE stehen low-level Primitive zur Verfügung, welche zu komplexeren Programmen kombiniert werden können. Einige für Reduktionen notwendige Primitive werden wir in diesem Abschnitt definieren. Hierfür geben wir die jeweiligen Typen dieser Primitive an.

Der Typ eines Primitivs setzt sich aus den Typen der Parameter und dem Rückgabetyp zusammen. Die Primitive nehmen ihre Parameter über Currying sequenziell an. Befindet sich ein Parameter in geschweiften Klammern, so kann dieser aus den anderen Parametern hergeleitet werden und muss nicht explizit angegeben werden. Der letzte auf die Parameter folgende Wert ist der Rückgabetyp des Primitivs.

Hierfür wichtige Typen sind *nat* und *data*. Ersteres ist eine natürliche Zahl und letzteres eine Sammlung aller Datentypen, darunter auch sämtliche Zahlentypen. Ein n vom Typ *nat* und ein t vom Typ *data* lassen sich zu $n.t$, einem Array der Länge n mit Elementen vom Typ t, kombinieren. Solche Arrays sind ebenfalls vom Typ *data*. Das Element eines Arrays *arr* an der Stelle i ergibt sich durch *arr* @ i.

CUDA-spezifische map-Primitive

Um Daten aus einem Array auf die verschiedenen Ebenen der CUDA-Ausführungshierarchie zu verteilen, gibt es spezielle map-Primitive. Mit diesen kann man eine Funktion auf die einzelnen Elemente dieses Arrays anwenden. Die Funktion wird hierbei von genau einer Instanz der jeweiligen Ebene bearbeitet. Die jeweiligen Primitive haben folgenden Typ:

$$\begin{aligned} mapP(i) :&\{n : nat\} \to \{s\ t : data\} \to (s \to t) \to n.s \to n.t \\ &\forall P \in \{Global, Block, Threads, Warp, Lane\} \wedge \\ &\forall i \in \{'x', 'y', 'z'\} \end{aligned}$$

Das Primitiv *mapGlobal* verteilt die Elemente des Eingabearrays über das gesamte Grid auf alle Threads. Will man ein Array auf verschiedene Blocks in einem Grid aufteilen, kann man dafür das *mapBlock*-Primitiv nutzen. Innerhalb eines Blocks kann man die Elemente eines Arrays mithilfe von *mapThreads* auf die Threads verteilen und mithilfe von *mapWarp* auf die Warps verteilen. Innerhalb eines Warps kann man ein Array zudem mithilfe von *mapLane* auf die Lanes verteilen. Der Parameter i gibt hierbei an, in welcher Dimension die Elemente verteilt werden sollen. Gibt man diesen Parameter nicht explizit an, wird die Verteilung eindimensional in x-Richtung vorgenommen.

Sollten bei der Ausführung mehr Arrayelemente als Instanzen der jeweiligen Hierarchiestufe vorhanden sein, so wird die Last im generierten Code mithilfe von Grid-Stride-Loops auf diese Instanzen aufgeteilt. Zur Vermeidung von undefiniertem Verhalten wird im Anschluss an ein CUDA-spezifisches *map*-Primitiv, sofern möglich, auf der jeweils darüber liegenden Ebene eine Barrierensynchronisierung durchgeführt.

split

Mithilfe des *split*-Primitivs lässt sich ein Array in ein Array aus Arrays der Länge n umformen. Dieses Primitiv hat folgenden Typ:

$$split : (n : nat) \to \{m : nat\} \to \{t : data\} \to nm.t \to m.n.t$$

Das Ergebnis von *split* ergibt sich wie folgt:

$$((split\ \ n\ arr)\ @\ y)\ @\ x = arr\ @\ (x + n * y)$$

Es gilt zu beachten, dass, falls n m nicht teilt, die letzten $m \mod n$ Elemente des Eingabearrays verworfen werden. Bei der Nutzung von *map*-Primitiven ist *split* ein besonders hilfreiches Primitiv, da man über eine Kombination der beiden Primitive Teilarrays auf die jeweiligen Einheiten verteilen kann.

join

Mithilfe des *join*-Primitivs lässt sich ein mehrdimensionales Array in seiner Dimension um 1 reduzieren. Hiermit hat *join* die entgegengesetzte Wirkung zum *split*-Primitiv. Der Typ des Primitivs ist folgender:

$$join : \{nm : nat\} \to \{t : data\} \to n.m.t \to nm.t$$

zip

Mithilfe des *zip*-Primitivs lassen sich zwei Arrays so miteinander zu einem Array kombinieren, dass die einzelnen Elemente zu Tupeln zusammengefasst werden. Dieses Primitiv hat folgenden Typ:

$$zip : \{n : nat\} \to \{s\ t : data\} \to n.s \to n.t \to n.(s \times t)$$

Das Ergebnis von *zip* ergibt sich dabei wie folgt:

$$(zip\ \ x\ \ y)\ @\ i\ =\ (x\ @\ i,\ y\ @\ i)$$

toMem

Mithilfe des *toMem*-Primitivs lassen sich Daten in einen Speicherbereich übertragen:

$$toMem : \{t : data\} \to (a : addrSpace) \to t \to t$$

Die Speicherbereiche, die hierfür zur Verfügung stehen, sind *Global*, *Shared* und *Private*. Es gibt zudem instanziierte Formen des *toMem*-Primitivs, wie beispielsweise *toPrivate* und *toShared*. Bei diesen muss kein Speicherbereich angegeben werden, da sich dieser schon aus dem jeweiligen Namen des Primitivs ergibt.

reduceSeq

Das *reduceSeq*-Primitiv führt eine sequenzielle Reduktion durch.

$$reduceSeq : (a : addrSpace) \to \{n : nat\} \to \{s\ t : data\} \to \\ (t \to s \to t) \to t \to n.s \to t$$

Dabei wird zunächst angegeben, in welchem Adressraum diese Reduktion stattfinden soll und welcher Reduktionsoperator verwendet werden soll. Außerdem wird ein Initialwert und das Array, welches reduziert werden soll, angegeben.

padCst

Mithilfe des *padCst*-Primitivs lässt sich ein Array mit einer Konstante am Anfang und am Ende des Arrays padden:

$$padCst : \{n : nat\} \to (l\ q : nat) \to \{t : data\} \to t \to n.t \to \\ (l + n + q).t$$

Hierbei werden l-viele Werte dieser Konstante an den Anfang des Arrays und q-viele Werte an das Ende des Arrays hinzugefügt. Dieses Primitiv ist besonders hilfreich, wenn ein Array mit *split* aufgeteilt werden soll, die Teilung jedoch nicht gleichmäßig möglich ist.

take

Mithilfe des *take*-Primitivs lässt sich ein Array verkleinern:

$$take : (n : nat) \to \{m : nat\} \to \{t : data\} \to (n + m).t \to n.t$$

Die ersten n Elemente werden hierbei beibehalten und die darauf folgenden m Elemente verworfen.

drop

Mithilfe des *drop*-Primitivs lässt sich ein Array verkleinern:

$$drop : (n : nat) \to \{m : nat\} \to \{t : data\} \to (n + m).t \to m.t$$

Die ersten m Elemente werden hierbei beibehalten und die darauf folgenden n Elemente verworfen.

transpose

Mithilfe des *transpose*-Primitivs lässt sich ein zweidimensionales Array transponieren:

$$transpose : \{n\ m : nat\} \to \{t : data\} \to n.m.t \to m.n.t$$

Dieses Primitiv ist insbesondere bei der Optimierung von Speicherzugriffen von großer Bedeutung.

3.2 REDUKTION AUF DEVICE-EBENE

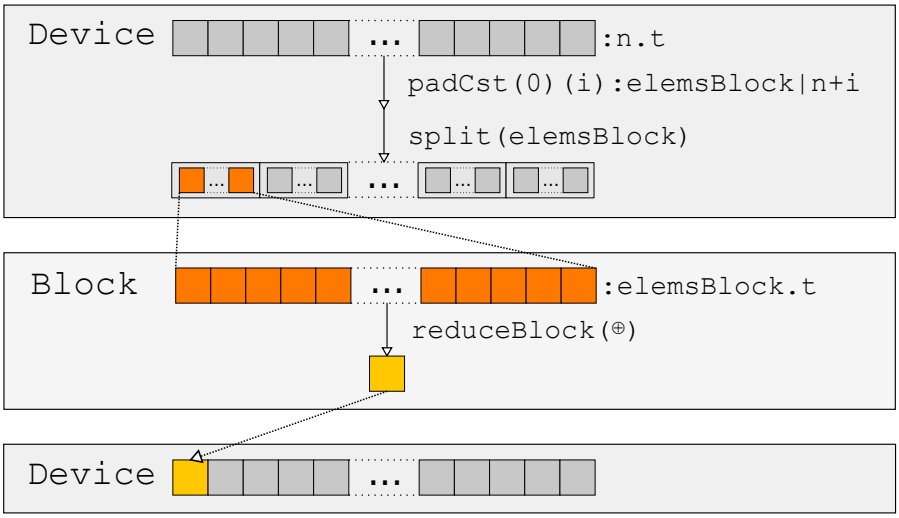

Abbildung 3.1: Reduktion auf Device-Ebene in RISE

Eine Reduktion auf Device-Ebene lässt sich in RISE einfach durchführen. In Abbildung 3.1 werden die dafür notwendigen Schritte gezeigt. Ein Array der Länge n wird hierbei über *mapBlock* auf Blocks verteilt. Die Länge der je Block verarbeiteten Teilarrays wird durch den Parameter *elemsBlock* festgelegt.

Sollte eine gleichmäßige Aufteilung des Arrays nicht möglich sein, so muss das Array zunächst mit dem *padCst*-Primitiv auf das nächste Vielfache von *elemsBlock* gepaddet werden. Der Wert, mit welchem

```
// Fehlerhaft generiertes Padding
for (int block_id_1357 = blockIdx.x; (block_id_1357 < ((n2
↪  / 32) + (((32 + (-1 * (n2 % 32))) % 32) / 32)));
↪  block_id_1357 = (block_id_1357 + gridDim.x))

// Korrektes Padding
for (int block_id_1357 = blockIdx.x; (block_id_1357 < (n2 +
↪  ((32 - (n2 % 32)) % 32)) / 32); block_id_1357 =
↪  (block_id_1357 + gridDim.x))
```

Quelltext 3.1: Fehlerhaft generiertes Padding und die entsprechende Korrektur

```
nFun((elemsBlock, n) =>
  fun(n `.` f32)(arr =>
    arr |>
    padCst(0)((elemsBlock-(n%elemsBlock))%elemsBlock)
    ↪  (l(0f)) |>
    split(elemsBlock) |>
    mapBlock(reduceBlock(op))
  )
)
```

Quelltext 3.2: RISE-Code für Abbildung 3.1

hier gepaddet wird, ist das neutrale Element des jeweiligen Reduktionsoperators. Die Anzahl an Elementen, die über *padCst* hinzugefügt werden müssen, ergibt sich durch die folgende Berechnungsvorschrift:

$$(elemsBlock - (n \mod elemsBlock)) \mod elemsBlock$$

Wie aus Quelltext 3.1 ersichtlich, ergibt sich, durch die Kombination aus dem *split* und dem *mapBlock*-Primitiv, bei der Codegenerierung das Problem, dass die Berechnung durch eine ungünstige Reihenfolge bei den notwendigen Ganzzahl-Divisionen verfälscht wird. Vor Verwendung des generierten Codes wurde dies per Hand korrigiert.

Innerhalb eines Blocks müssen daraufhin *elemsBlock* viele Elemente reduziert werden. Ist die Reduktion auf Block-Ebene abgeschlossen und $n > elemsBlock$, so sind danach auf Device-Ebene noch mehrere Teilergebnisse der gesamten Reduktion vorhanden. Da keine globale Synchronisierung möglich ist, muss für die Reduktion dieser Werte eine weitere Reduktion auf Device-Ebene durchgeführt werden.

In Quelltext 3.2 sieht man den RISE-Code für die Reduktion auf Device-Ebene aus Abbildung 3.1. Wir verzichten bei den folgenden Implementierungen darauf, diesen explizit zu zeigen, da sich der RISE-

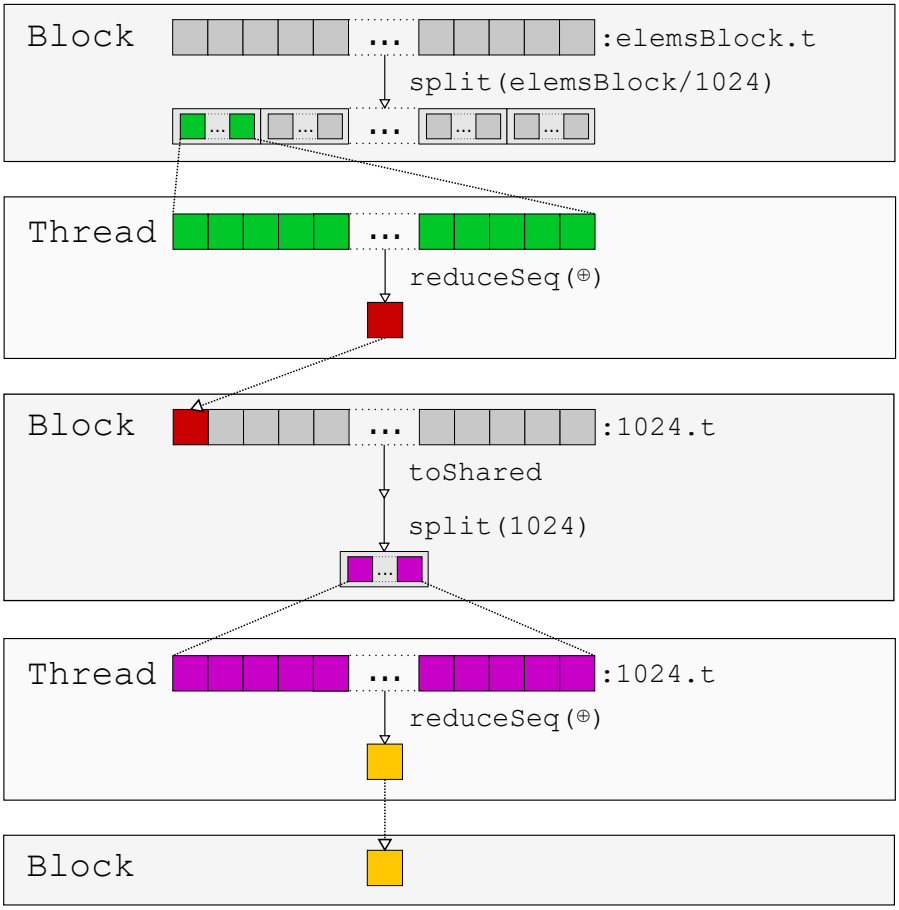

Abbildung 3.2: Reduktion auf Block-Ebene in RISE mit Shared Memory (naiver Ansatz)

Code bereits auf eine anschauliche Art und Weise aus den jeweiligen Abbildungen ergibt.

3.3 REDUKTION AUF BLOCK-EBENE

Das Verhalten des naiven Ansatzes aus Kapitel 2.2 lässt sich in RISE leicht umsetzen. Dies wird in Abbildung 3.2 illustriert.

Hierbei werden die pro Block zu verarbeitenden Werte zunächst mithilfe von *split* in Segmente aufgeteilt, welche dann mit *mapThreads* auf die Threads verteilt werden. Diese führen dann eine sequenzielle Reduktion mit dem Reduktionsoperator durch. Daraufhin werden alle in den Threads berechneten Teilergebnisse mit *toShared* in den Shared Memory geschrieben, damit ein Thread diese dann sequenziell reduzieren kann. Nach dieser zweiten Reduktion ist die Reduktion auf Block-Ebene vollständig abgeschlossen.

Das Verhalten auf Block-Ebene der in Kapitel 2.2 erwähnten Implementierung von Mark Harris [7] lässt sich in RISE grundsätzlich imitieren. Die sequenzielle Reduktion wird hier jedoch nicht wie im

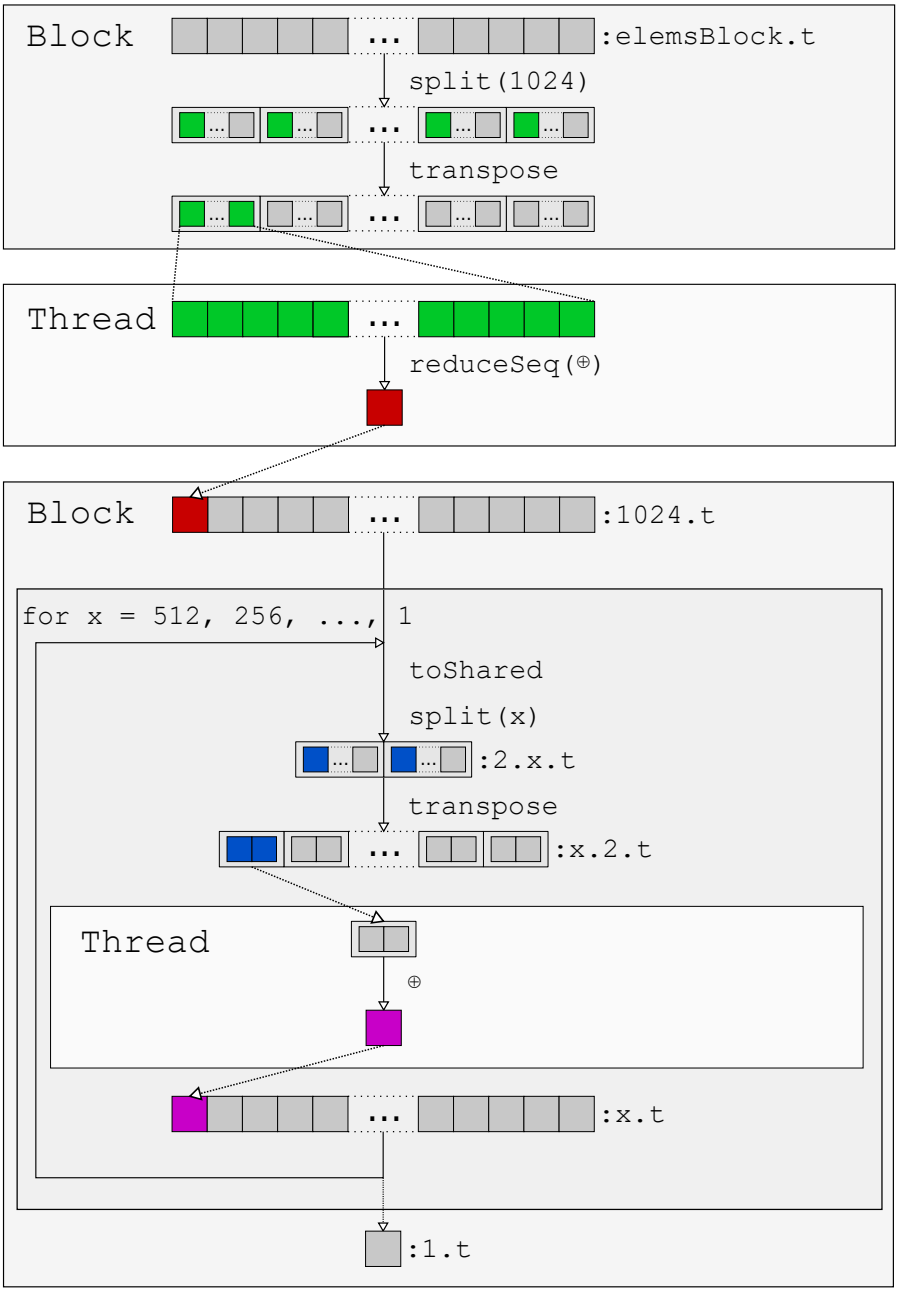

Abbildung 3.3: Reduktion auf Block-Ebene in RISE mit Shared Memory

originalen Kernel, sondern abhängig vom Parameter *elemsBlock* durchgeführt. Die Anzahl der sequenziell zu reduzierenden Elemente je Thread ergibt sich so nicht variabel anhand der Größe des Eingabearrays und der Anzahl der ausführenden Threads, was die Fleixibilität des generierten Codes reduziert. Eine Grid-Stride-Loop zur sequenziellen Reduktion, wie in Abbildung 2.1, lässt sich in RISE nicht umsetzen. Diese RISE-Implementierung ist zudem für eine Blockgröße von 1024 Threads ausgelegt.

Über das *split*-Primitiv wird das Array zunächst in 1024-elementige Teilarrays aufgeteilt. Durch das *transpose*-Primitiv wird dieses Array aus Teilarrays dann in ein 1024-elementiges Array aus Arrays umgeformt. Diese 1024 Arrays werden dann auf die Threads verteilt. Diese führen dann jeweils eine sequenzielle Reduktion durch.

Nach Abschluss der sequenziellen Reduktionen wird eine Baumreduktion im Shared Memory durchgeführt. Über *toShared* werden die Teilergebnisse zunächst in den Shared Memory geladen. Daraufhin wird das Array über das *split*-Primitiv in zwei Hälften aufgeteilt. Durch das *transpose*-Primitiv wird daraus ein Array aus zweielementigen Arrays. Dies bewirkt, dass die Zugriffe auf den Shared Memory, wie in der Implementierung von Mark Harris, über sequenzielle Addressierung stattfindet. Diese zweielementigen Arrays werden daraufhin auf Threadebene reduziert. Dieses verhalten wird wiederholt bis nur noch ein Wert vorhanden ist. Die Reduktion auf Block-Ebene ist damit abgeschlossen.

4

OPTIMIERUNGEN AUF WARP-EBENE IN RISE

In diesem Kapitel stellen wir die neu entwickelten Shuffle-Primitive in RISE vor. Unter Verwendung dieser neuen Primitive implementieren wir eine Reduktion auf Warp-Ebene in RISE. Damit diese Optimierungen im Kontext einer gesamten GPU genutzt werden können, zeigen wir außerdem, welche Änderungen dafür bei der Reduktion auf Block-Ebene notwendig sind. Diese Implementierungen orientieren sich an der bereits in Kapitel 2.3 erwähnten Implementierung von NVIDIA.

4.1 SHUFFLE-PRIMITIVE IN RISE

Die folgend näher betrachteten Low-Level-Primitive ermöglichen eine effiziente Form des Datenaustauschs zwischen den Lanes innerhalb eines Warps. Hierbei ist es erforderlich, dass jede der 32 Lanes jeweils einen skalaren Wert aus einem Array der Größe 32 im Private Memory hinterlegt hat. Dieser entspricht in RISE dem Registerspeicher in CUDA. Jedes dieser Primitive führt eine Synchronisierung auf Warp-Ebene durch.

Die GPU, auf welcher diese Primitive genutzt werden sollen, muss eine NVIDIA-GPU mit einer Compute Capability von mindestens 3.0 sein.

4.1.1 *shflDownWarp*

Das *shflDownWarp*-Primitiv sorgt dafür, dass alle Lanes innerhalb eines Warps, sofern möglich, ihren im Private Memory hinterlegten Wert an eine Lane mit einer kleineren Lane-ID übermitteln. Hierbei ist kein Umweg über den vergleichsweise langsamen Shared Memory notwendig.

Der Typ des shflDownWarp-Primitivs ist wie folgt definiert:

$$shflDownWarp : (dt : scalarType) \to (delta : nat) \to 32.dt \to 32.dt$$

Im Kontext einer Lane ergibt sich die Lane-ID der Lane, von der ein Wert erhalten wird, durch die Summe der eigenen Lane-ID und dem Parameter *delta*. Sollte diese größer als 31 sein, so erhält diese Lane ihren eigenen Wert zurück. Demnach ergibt sich das Ergebnis des *shflDownWarp*-Primitivs wie folgt:

$$\begin{aligned}(shflDownWarp\ \ t\ \ d\ \ arr)\ @\ i = \\ if(i + d \leq 31) : arr\ @\ (i + d) \\ else : arr\ @\ i\end{aligned}$$

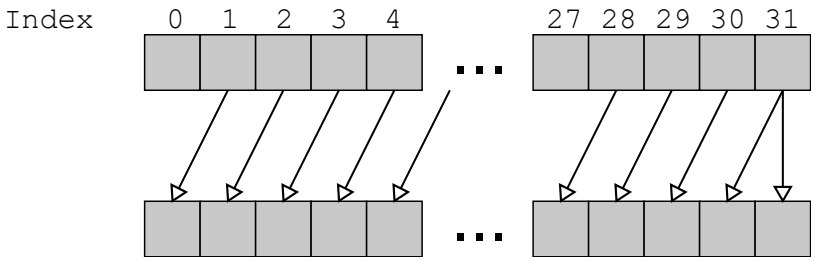

Abbildung 4.1: shflDownWarp mit delta=1

In Abbildung 4.1 wird beispielhaft illustriert, wie das shflDownWarp-Primitiv die Arrayelemente zwischen den Lanes bei *delta* = 1 austauscht. Dieses Austauschmuster entspricht dem der Shuffle Instruction *__shfl_down_sync*.

```
nFun(n =>
  fun(n `.` f32)(arr =>
    arr |> split(32) |>
      mapWarp('x')(fun(warpChunk =>
        warpChunk |>
          toPrivateFun(mapLane('x')(id)) |>
          shflDownWarp(1) |>
          mapLane('x')(id)
      ))
  )
)
```

Quelltext 4.1: Beispiel für shflDownWarp in RISE

Im Quelltext 4.2 wird gezeigt, welcher CUDA-Code für den RISE-Code aus Quelltext 4.1 generiert wird. Die Eingabedaten werden dabei zunächst in den Private Memory der jeweiligen Lanes geladen. Daraufhin wird das *shflDownWarp*-Primitiv auf Warp-Ebene angewendet. Da das *shflDownWarp*-Primitiv in RISE auf Warp-Ebene angewendet wird, das entsprechende CUDA-Gegenstück *__shfl_down_sync* jedoch von den einzelnen Lanes ausgeführt werden muss, ist es erforderlich, dass nach einem *shflDownWarp*-Aufruf eine Verteilung auf die Lanes erfolgt. Dies passiert über das *mapLane*-Primitiv, weswegen hier noch eine Funktion angegeben werden muss, die dann auf die einzelnen Elemente angewendet wird.

Im Beispielcode wurde dafür der Einfachheit halber die identische Abbildung gewählt. Dies ist in der Praxis jedoch keine sinnvolle Verwendung eines Shuffle-Primitivs, da eine Umordnung auf logischer Ebene in diesem Fall effizienter sein würde. Aus diesem Grund wurde keine Übersetzung implementiert, welche eine direkte Ausgabe der Daten nach Nutzung eines Shuffle-Primitivs ermöglicht.

```
/* mapWarp */
for (int warp_id_107 = (threadIdx.x / 32);(warp_id_107 <
 ↪  (n0 / 32));warp_id_107 = (warp_id_107 + (blockDim.x /
 ↪  32))) {
  {
    float x65[1];
    /* mapLane */
    /* iteration count is exactly 1, no loop emitted */
    int lane_id_110 = (threadIdx.x % 32);
    x65[0] = x0[(lane_id_110 + (32 * warp_id_107))];
    __syncwarp();
    /* mapLane */
    /* iteration count is exactly 1, no loop emitted */
    int lane_id_111 = (threadIdx.x % 32);
    output[(lane_id_111 + (32 * warp_id_107))] =
     ↪  __shfl_down_sync(0xFFFFFFFF, x65[0], 1);
    __syncwarp();
  }
}
__syncthreads();
```

Quelltext 4.2: Generierter CUDA-Code für den RISE-Code aus Quelltext 4.1

4.1.2 *shflUpWarp*

Das *shflUpWarp*-Primitiv unterscheidet sich lediglich in der Richtung, in welcher die Werte zwischen den Lanes ausgetauscht werden, vom *shflDownWarp*-Primitiv.

Der Typ des *shflUpWarp*-Primitivs ist wie folgt definiert:

$$shflUpWarp : (dt : scalarType) \to (delta : nat) \to 32.dt \to 32.dt$$

Das Ergebnis des shflUpWarp-Primitivs ergibt sich wie folgt:

$$(shflUpWarp\ \ t\ \ d\ \ arr)\ @\ i = \\ if(i - d \geq 0) : arr\ @\ (i - d) \\ else : arr\ @\ i$$

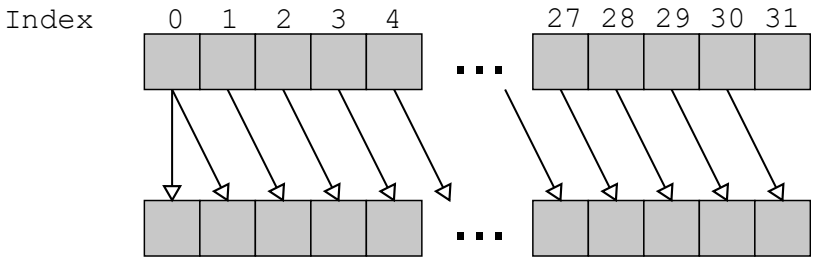

Abbildung 4.2: shflUpWarp mit delta=1

In Abbildung 4.2 wird beispielhaft illustriert, wie das *shflUpWarp*-Primitiv die Arrayelemente zwischen den Lanes bei *delta* = 1 austauscht. Dieses Austauschmuster entspricht dem der Shuffle Instruction *__shfl_up_sync*. Das *shflUpWarp*-Primitiv wird analog zu *shflDownWarp* in den jeweiligen Lanes zur Shuffle Instruction *__shfl_up_sync* übersetzt.

4.1.3 *shflXorWarp*

Das *shflXorWarp*-Primitiv unterscheidet sich von *shflDownWarp* und *shflUpWarp* dadurch, dass sich die Lane-ID der Lane, von der eine Lane einen Wert erhält, durch ein bitweises *xor* von *laneMask* und der eigenen Lane-ID ergibt.

Der Typ des shflXorWarp-Primitivs ist wie folgt definiert:

$$shflXorWarp : (dt : scalarType) \to (laneMask : nat) \to 32.dt \to 32.dt$$

Das Ergebnis des shflXorWarp-Primitivs ergibt sich wie folgt:

$$(shflXorWarp\ \ t\ \ m\ \ arr)\ @\ i = arr\ @\ (i\ xor\ m)$$

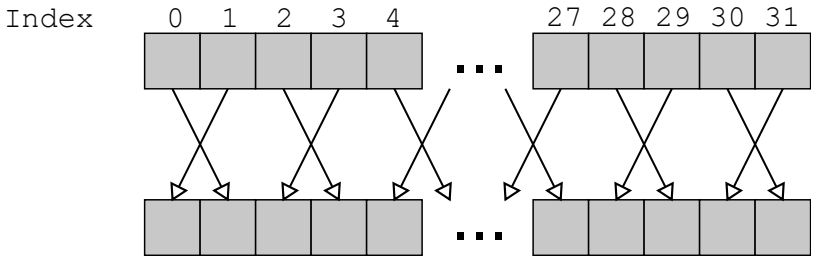

Abbildung 4.3: shflXorWarp mit laneMask=1

In Abbildung 4.3 wird beispielhaft illustriert, wie das *shflXorWarp*-Primitiv die Arrayelemente zwischen den Lanes bei *laneMask* = 1 austauscht. Dieses Austauschmuster entspricht dem der Shuffle Instruction *__shfl_xor_sync*. Das *shflXorWarp*-Primitiv wird analog zu *shflDownWarp* in den jeweiligen Lanes zur Shuffle Instruction *__shfl_up_sync* übersetzt.

4.1.4 *shflWarp*

Das *shflWarp*-Primitiv ermöglicht es, ein Zugriffsmuster frei zu definieren, indem ein 32-elementiges Array an Indizes als Parameter übergeben wird. Hierdurch bestimmt sich für jede Lane einzeln, von welcher Lane diese den Wert erhält.

Der Typ des shflWarp-Primitivs ist wie folgt definiert:

$$shflWarp : (dt : scalarType) \to 32.idx[32] \to 32.dt \to 32.dt$$

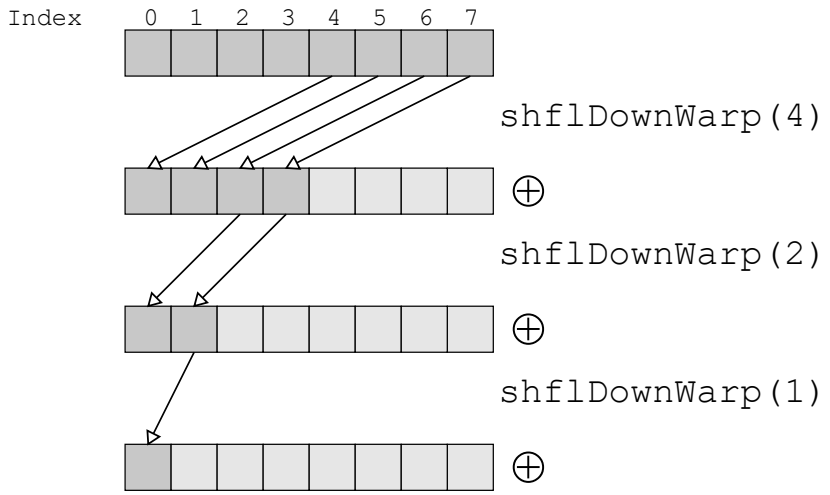

Abbildung 4.4: Reduktion auf Warp-Ebene mit shflDownWarp anhand von acht Lanes

Das Ergebnis des shflWarp-Primitivs ergibt sich wie folgt:

$$(shflWarp\ \ t\ \ l\ \ arr)\ @\ i = arr\ @\ (l\ @\ i)$$

Dadurch, dass die dynamische Berechnung von Indizes in RISE nicht wie gewünscht funktioniert, kann noch keine allgemeine Übersetzung in CUDA-Code stattfinden. Eine Übersetzung in einen Broadcast ist zwar möglich, dieser ist jedoch nicht relevant für die Durchführung von Reduktionen.

4.2 REDUKTION AUF WARP-EBENE IN RISE MIT SHUFFLE-PRIMITIVEN

Reduktionen auf Warp-Ebene lassen sich grundsätzlich über jedes Shuffle-Primitiv implementieren. In Abbildung 4.4 ist exemplarisch anhand von acht Lanes illustriert, wie eine Reduktion mithilfe des shflDownWarp-Primitivs funktioniert. Die obere Hälfte der Lanes übermittelt hierbei in jeder Iteration die eigenen Werte an die untere Hälfte der Lanes, in welcher jede einzelne Lane dann den eigenen Wert mit dem erhaltenen Wert kombiniert. Die obere Hälfte der Lanes wird daraufhin in der folgenden Iteration nicht mehr betrachtet. Nach der letzten Iteration befindet sich das Ergebnis der Reduktion im Private Memory der ersten Lane.

Aus Abbildung 4.5 wird ersichtlich, wie sich diese Herangehensweise in low-level-RISE umsetzen lässt. Zunächst wird mit *toPrivateFun* und einer Aufteilung über *mapLane* sichergestellt, dass jede Lane genau einen Wert des Eingabearrays im Registerspeicher hält. Da *mapLane* zwingend in jeder Lane eine Funktion ausführen muss, jedoch noch keine Änderungen gewünscht sind, wird hier die identische Abbildung *id* übergeben.

Daraufhin folgt eine Schleife, welche die in Abbildung 4.4 illustrierten Schritte für mehrere *delta*-Werte wiederholt. Hierfür wird in

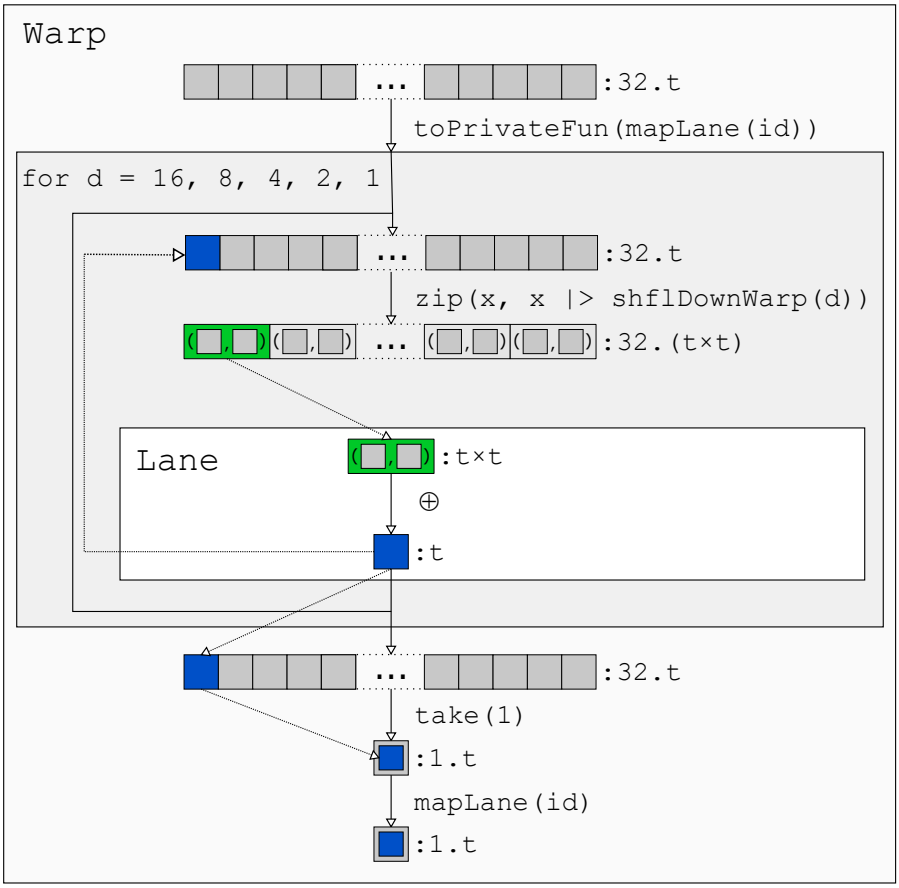

Abbildung 4.5: Reduktion auf Warp-Ebene in RISE mit shflDownWarp

```
float x1012[1];
{
  float x1032[1];
  {
    ...
  }
  /* mapLane */
  /* iteration count is exactly 1, no loop emitted */
  int lane_id_1432 = (threadIdx.x % 32);
  x1012[0] = (x1032[0] + __shfl_down_sync(0xFFFFFFFF,
   ↪  x1032[0], 16));
  __syncwarp();
}
```

Quelltext 4.3: Generierter Code für die erste Iteration der Schleife aus Abbildung 4.5

jeder Iteration zunächst auf Warp-Ebene das Eingabearray mithilfe des *zip*-Primitivs mit dem Ergebnis des *shflDownWarp*-Primitivs für dieses Array kombiniert. Auf die daraus entstehenden Tupel wird daraufhin über *mapLane* in den jeweiligen Lanes der Reduktionsoperator $\oplus$ angewendet.

Wie aus Quelltext 4.3 ersichtlich, sorgt die Kombination aus *mapLane* und dem Shuffle-Primitiv dafür, dass eine nicht notwendige Synchronisierung auf Warp-Ebene generiert wird.

Obwohl nur dieser Wert als Rückgabewert gewünscht ist, kann die Rückgabe der Reduktion auf Warp-Ebene nur über das entsprechende, mithilfe von *take* herausgezogene, einelementige Teilarray erfolgen. Dies begründet sich dadurch, dass der Zugriff auf dieses einzelne Element auf Lane-Ebene statt auf Warp-Ebene durchgeführt werden muss. Um zu bewirken, dass die Rückgabe durch eine einzelne Lane erfolgt, muss zuletzt noch ein *mapLane* mit der identischen Abbildung durchgeführt werden.

Dies führt, wie aus Quelltext 4.4 ersichtlich, im generierten Code dazu, dass eine eigentlich nicht notwendige Kopie durchgeführt wird. Hierbei erfolgt die Rückgabe durch eine Kopie in den Shared Memory, da dieser Codeausschnitt im Kontext eines größeren Programms generiert wurde. Die Rückgabe von einelementigen Arrays hat zur Folge, dass nach einem Aufruf der Reduktion auf Warp-Ebene eine *join*-Operation durchgeführt werden muss.

Eine Reduktion mithilfe von *shflUpWarp* ließe sich grundsätzlich analog zur Implementierung aus Abbildung 4.5 in RISE implementieren. Aufgrund der Symmetrie befindet sich das Ergebnis dabei schlussendlich im privaten Speicher der letzten Lane. Die direkte Rückgabe dieses Wertes ist jedoch bislang nicht möglich, da sowohl

```
float* x19587 =
↪  ((float*)(&(dynamicSharedMemory20137[0])));
/* mapWarp */
for (int warp_id_20051 = (threadIdx.x / 32);
↪  (warp_id_20051 < 4); warp_id_20051 = (warp_id_20051 +
↪  (blockDim.x / 32))) {
  {
    float x19600[1];
    {
      float x19620[1];
      {
        ...
      }
      /* mapLane */
      /* iteration count is exactly 1, no loop emitted */
      int lane_id_20082 = (threadIdx.x % 32);
      x19600[0] = (x19620[0] + __shfl_xor_sync(0xFFFFFFFF,
       ↪  x19620[0], 16));
      __syncwarp();
    }
    /* mapLane */
    for (int lane_id_20075 = (threadIdx.x % 32);
     ↪  (lane_id_20075 < 1); lane_id_20075 = (32 +
     ↪  lane_id_20075)) {
      x19587[(lane_id_20075 + warp_id_20051)] = x19600[0];
    }
  __syncwarp();
  }
}
```

Quelltext 4.4: Generierter Code für die Ergebnisrückgabe der Reduktion auf Warp-Ebene

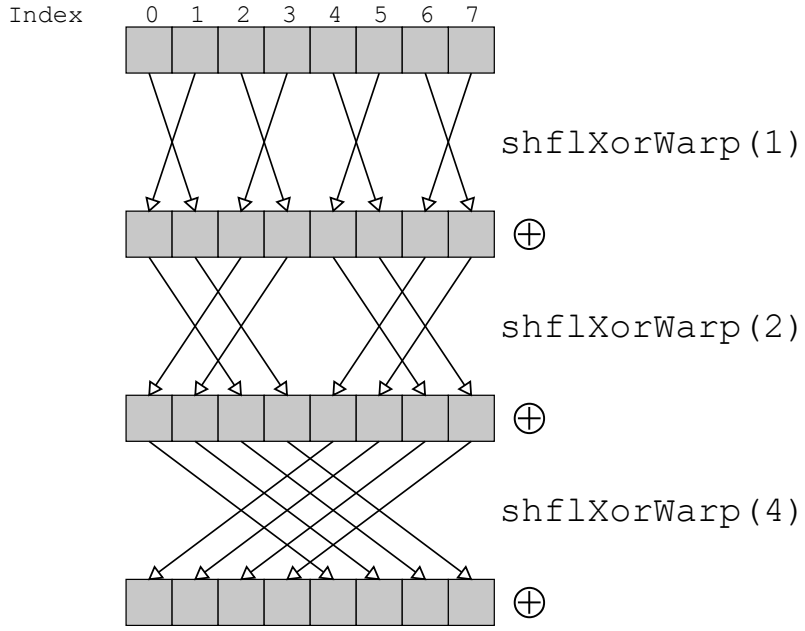

Abbildung 4.6: Reduktion auf Warp-Ebene mit shflXor anhand von acht Lanes

take, als auch *drop* keine Elemente vom Anfang eines Arrays entfernen können. Eine vorherige Umordnung ist, aufgrund der bereits in Kapitel 4.1.4 erwähnten Probleme bei Indexberechnungen, ebenfalls nicht möglich.

Sollte es notwendig sein, dass jede Lane das Ergebnis der Reduktion erhält, lässt sich, wie in Abbildung 4.6 illustriert, hierfür eine Reduktion mit shflXorWarp nutzen. Da die Rückgabe dann über eine beliebige Lane erfolgen kann, ist eine Rückgabe über die erste Lane wie bei einer Reduktion mit *ShflDownWarp* möglich.

Da sich die Austauschmuster von shflDownWarp, shflUpWarp und shflXorWarp über shflWarp imitieren lassen, sind Reduktionen mithilfe von shflWarp grundsätzlich ebenfalls leicht zu implementieren. Aufgrund der bereits erwähnten Probleme bei Indexberechnungen ist dies jedoch noch nicht möglich.

4.3 NOTWENDIGE ANPASSUNGEN AUF BLOCK-EBENE

Um bei einer Reduktion auf Block-Ebene die Optimierungen auf Warp-Ebene nutzen zu können, müssen einige Änderungen vorgenommen werden. Die folgend erläuterte Implementierung in low-level-RISE wird in Abbildung 4.7 dargestellt.

Mithilfe des *split*-Primitivs und einem darauf folgenden *mapBlock* wird das Eingabearray auf Device-Ebene zuvor auf die Blocks verteilt. Die Länge *elemsBlock* der Teilarrays, die je Block verarbeitet werden, ist ein Parameter, der bei der Codegenerierung angegeben werden muss. Unter Umständen muss hier auch wieder vor dem Aufteilen

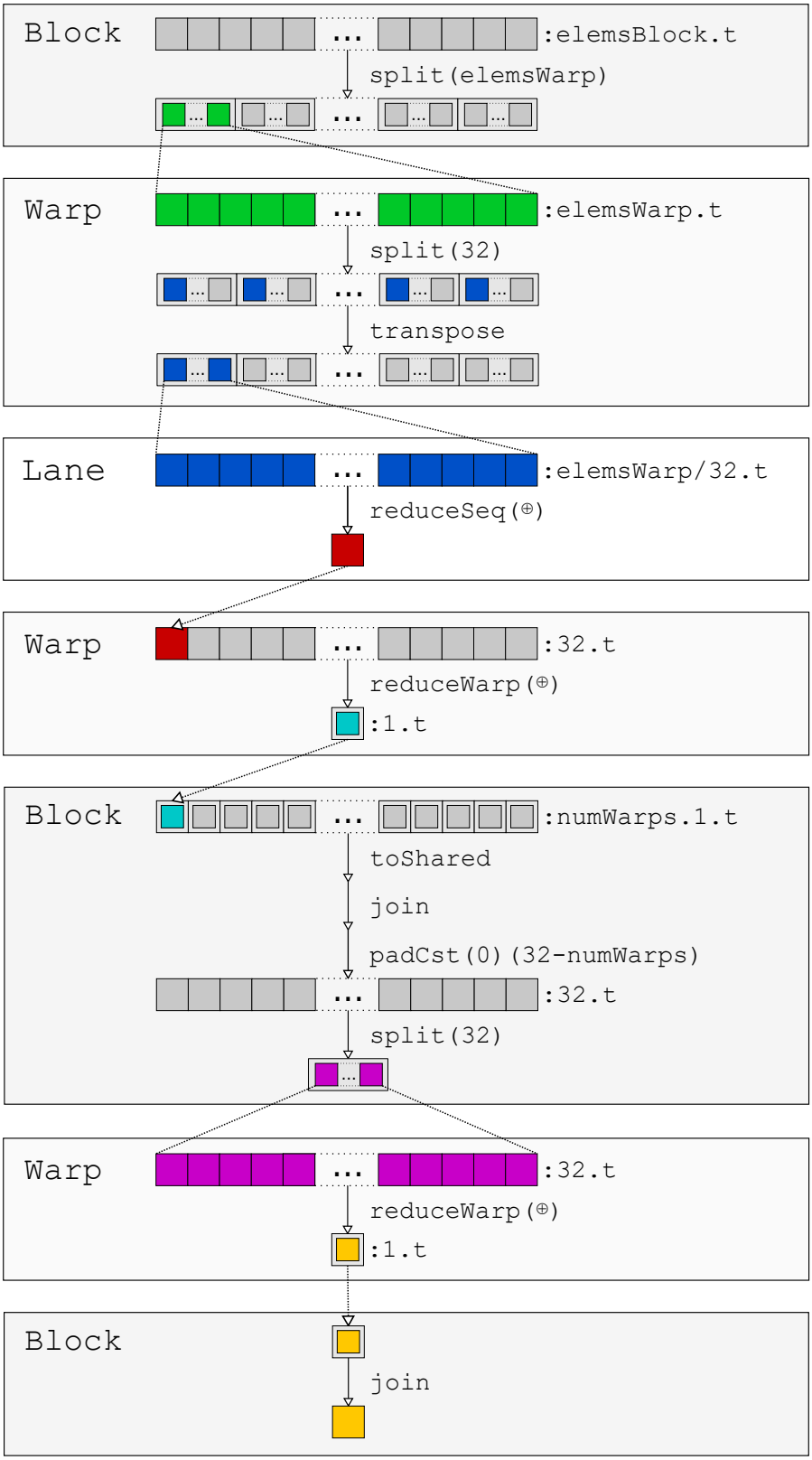

Abbildung 4.7: Reduktion auf Block-Ebene in RISE mit Shuffle-Primitiven

des Arrays noch ein Padding auf das nächste Vielfache von *elemsBlock* vorgenommen werden, damit das Array vollständig aufgeteilt werden kann.

Innerhalb eines Blocks werden die Daten ebenfalls mithilfe von *split* aufgeteilt. Daraufhin werden diese nicht direkt auf die Threads, sondern über *mapWarp* auf die Warps verteilt. Die Länge der Teilarrays, welche je Warp verarbeitet werden, wird über den Parameter *elemsWarp* ebenfalls bei der Codegenerierung bestimmt. Es ist erforderlich, dass *elemsWarp* ein Vielfaches von 32 ist, damit die Aufteilung auf die 32 Lanes gleichmäßig möglich ist. Außerdem muss, für die gleichmäßige Aufteilung auf die Warps, *elemsBlock* ein Vielfaches von *elemsWarp* sein. Da bereits auf Device-Ebene ein Padding auf das nächste Vielfache von *elemsBlock* vorgenommen wird, wird dadurch nicht die Unabhängigkeit von der Größe des Eingabearrays beeinträchtigt. Die Zahl der Warps pro Block ergibt sich dann durch $\frac{numElemsBlock}{numElemsWarp}$.

Innerhalb eines Warps wird das Teilarray zunächst mithilfe von *split* auf Arrays der Länge 32 aufgeteilt. Über das Primitiv *transpose* wird das so entstandene zweidimensionale Array transponiert. Dies hat zur Folge, dass das sich so ergebene Array aus 32 Arrays besteht. Diese werden dann mithilfe von *mapLane* auf die 32 Lanes innerhalb jedes Warps zur sequenziellen Reduktion im Private Memory über das *reduceSeq*-Primitiv verteilt. Durch die Kombination aus *split* und *transpose* wird bewerkstelligt, dass die dabei notwendigen Lesezugriffe eines Warps auf den Global Memory zusammengefasst werden können.

Nachdem die sequenzielle Reduktion innerhalb jeder Lane abgeschlossen ist, hat das Array auf Warp-Ebene nur noch 32 Elemente. Diese werden dann über eine Reduktion auf Warp-Ebene mithilfe der Shuffle-Primitive reduziert. Somit ist danach pro Warp noch ein Element vorhanden und die Reduktion auf Warp-Ebene abgeschlossen.

Auf Block-Ebene sind danach noch so viele Elemente im Array, wie es Warps in diesem Block gibt. Diese befinden sich jedoch noch in den Registern der jeweiligen Threads, auf welche man auf Block-Ebene keinen direkten Zugriff hat. Daher muss für die weiteren Schritte mithilfe von *toShared* eine Kopie in den Shared Memory vorgenommen werden. Diese Elemente sind, wie bereits in Kapitel 4.2 erwähnt, einelementige Arrays. Diese müssen daher noch über das *join*-Primitiv in ihrer Dimension reduziert werden.

Da eine weitere Reduktion auf Warp-Ebene aufgrund der Performancevorteile durch die Shfl-Primitive interessant ist, muss über das *padCst*-Primitiv ein Padding auf 32 Elemente vorgenommen werden. Mithilfe von *split* und *mapWarp* werden diese 32 Elemente dann einem einzigen Warp zugewiesen. Hier wird dann erneut eine Reduktion auf Warp-Ebene ausgeführt.

Nachdem diese zweite Reduktion auf Warp-Ebene abgeschlossen ist, gibt es je Block nur noch einen Wert im Array und die Reduktion auf Block-Ebene ist somit abgeschlossen. Dieser einzelne Wert befindet

sich, wie auch zuvor, in einem einelementigen Array, weswegen erneut das *join*-Primitiv zum Einsatz kommt.

5 EVALUATION

In diesem Kapitel evaluieren wir die Leistungsfähigkeit der Shuffle-Primitive, indem wir die Laufzeiten verschiedener Variationen des RISE-Programms aus Kapitel 4 untereinander und mit denen anderer Implementierungen vergleichen.

5.1 HARDWARE- UND SOFTWAREEIGENSCHAFTEN

Die Messungen wurden auf einem ArchLinux-Betriebssystem durchgeführt. Der GPU-Treiber hatte die Version 455.23.04. Als CUDA-Version wurde die Version 11.1 verwendet. Die verwendete GPU ist eine NVIDIA Titan RTX mit einer festgelegten Taktrate von 1890 MHz. Die Taktrate des Speichers der GPU wurde auf 6501 MHz festgelegt.

Jedes GPU-Programm wurde 110 mal ausgeführt. Die Ausführungszeiten der ersten zehn Ausführungen wurden hier jeweils verworfen, da diese nur dazu dienen, die GPU aufzuwärmen. Die 100 darauf folgenden Ausführungen wurden dann jeweils zur Evaluation genutzt.

Die Messung der Ausführungszeiten erfolgte über CUDA-Events. Da die Reduktions-Kernel mehrmals ausgeführt werden müssen, um ein Array auf einen einzelnen Wert zu reduzieren, entspricht die Ausführungszeit der Differenz zwischen dem Zeitpunkt beim ersten Kernel-Aufruf und dem Ende des letzten Kernel-Aufrufs. Als Reduktionsoperator wurde bei jedem Test der Additionsoperator genutzt. Der verwendete Datentyp für die Arrayelemente war eine 32-Bit-Gleitkommazahl. Als Arraygrößen wurden 2^{16}, 2^{17}, 2^{18}, 2^{19} und 2^{20} Elemente gewählt.

5.2 VARIATIONEN DER REDUKTIONS-KERNEL

Um eventuelle Leistungsunterschiede zwischen den verschiedenen Shuffle-Primitiven festzustellen, haben wir verschiedene Variationen der Reduktion mit Shuffle-Primitiven aus Kapitel 4 in den Vergleich einbezogen. Diese Versionen nutzen jeweils die Primitive *shflDownWarp* oder *shflXorWarp*.

Um zu bestimmen, welchen Einfluss die Parameter *elemsBlock* und *elemsWarp* auf die Leistung unseres Codes haben, haben wir hierfür ebenfalls verschiedene Variationen in den Vergleich einbezogen. Der Parameter *elemsWarp* wurde jeweils auf Werten von 32, 64, 128, 256 und 512 gesetzt, was bedeutet dass die einzelnen Lanes jeweils keinen, 2, 4, 8 oder 16 Werte sequenziell reduzieren. Der Parameter *elemsBlock* wurde dann jeweils auf das 1-, 2-, 4-, 8-, 16- oder 32-

fache von *elemsWarp* gesetzt. Die Anzahl an Threads, mit welchen die jeweiligen Kernel ausgeführt wurden, ergibt sich jeweils durch $(elemsBlock/elemsWarp) * 32$.

Um die Qualität des generierten Codes zu evaluieren, betrachten wir auch die von NVIDIA im Developer Blog veröffentlichte Implementierung ohne Atomics [15] und die *DeviceReduce*-Funktion aus Version 1.8.0 von CUB [6]. Erstere wurde dabei mit 32, 64, 128, 256, 512 und 1024 Threads pro Block getestet, während bei letzterer keine Threadanzahl spezifiziert werden muss.

Um den Einfluss der Nutzung von Shuffle Instructions zu messen, haben wir die vorgestellten Shared Memory-Implementierungen ebenfalls mit in diesen Vergleich einbezogen. Die Implementierung von Mark Harris [7] wurde zum Zwecke der Vergleichbarkeit so angepasst, dass die sequenzielle Reduktion exakt wie in der Implementierung mit Shuffle Instructions durchgeführt wird. Außerdem wurden in den letzten Iterationsschritten Synchronisierungen auf Warp-Ebene eingefügt, da diese bei Zugriffen auf den Shared Memory mittlerweile notwendig sind, um Wettlaufsituationen zu vermeiden [13]. Diese Implementierung wurde mit den gleichen Blockanzahlen wie die Implementierung mit Shuffle Instructions aus dem Developer Blog ausgeführt. Die Implementierung auf Block-Ebene aus Kapitel 3.3 wurde mit 1024, 2048, 4096, 8192 und 16384 Elementen pro Block getestet. Diese Implementierung wurde mit 1024 Threads pro Block ausgeführt, da sie in ihrer Struktur für exakt diesen Wert ausgelegt wurde.

5.3 EINFLUSS DER BLOCKANZAHL

Um den Einfluss der Blockanzahl auf die Ausführungszeiten der jeweiligen Kernel festzustellen und jeweils die optimale Anzahl zu finden, haben wir die Kernel mit verschiedenen Blockanzahlen ausgeführt. Da die Titan RTX 72 SMs hat und die Blocks auf die SMs gemappt werden, haben wir dafür die ersten acht Vielfachen von 72 und die jeweils darunter liegenden zwei und die darüber liegenden zwei Werte gewählt. Da die Blockanzahl die einzige nicht bei der Generierung festgelegte Variable ist, wurde dieser Ansatz anstelle von automatisiertem Tuning gewählt.

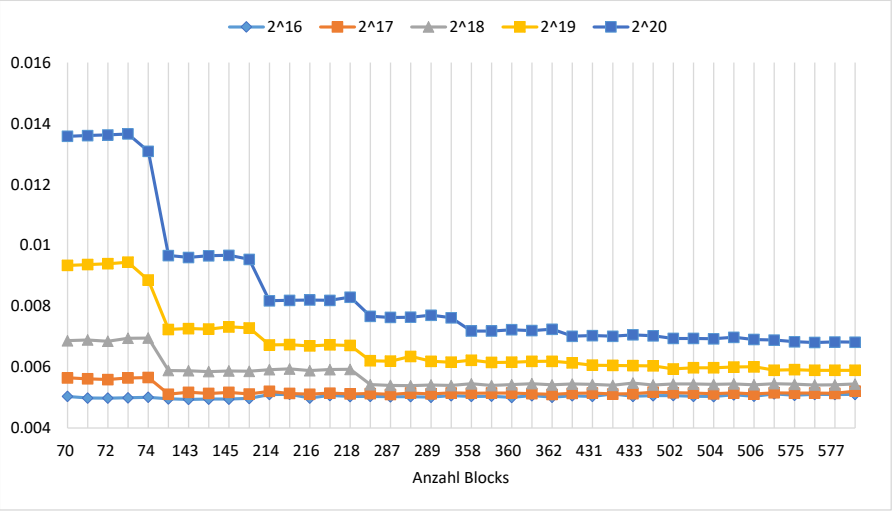

Abbildung 5.1: Laufzeit in ms abhängig von der Blockanzahl (generierter Code mit shflXorWarp, elemsBlock = 1024 und elemsWarp = 256)

In Abbildung 5.1 sieht man exemplarisch, wie sich die Laufzeit des generierten Codes mit *shflXorWarp*, 1024 Elementen pro Block und 256 Elementen pro Warp abhängig von der Blockanzahl entwickelt. Interessant ist, dass die für eine kurze Laufzeit notwendigen Blockanzahlen, je nach Größe des Eingabearrays, voneinander abweichen. Bei steigender Größe des Eingabearrays steigt hier auch die notwendige Anzahl an Blocks.

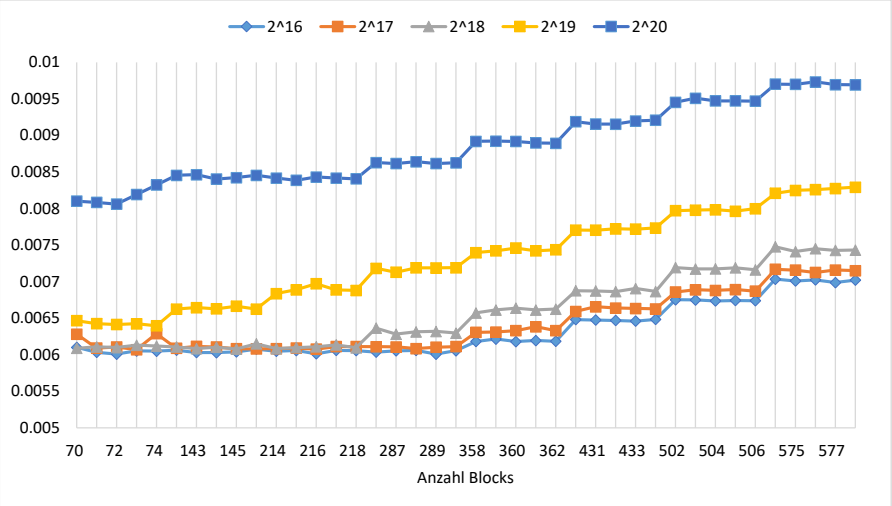

Abbildung 5.2: Laufzeit in ms abhängig von der Blockanzahl (generierter Code mit shflXorWarp, elemsBlock = 1024 und elemsWarp = 256)

In Abbildung 5.1 sieht man, wie sich die Laufzeit des generierten Codes verhält, wenn man bei gleichbleibendem *elemsWarp* eine größere Zahl für *elemsBlock* wählt. Interessant ist hierbei, dass sich die Ent-

wicklung der Laufzeit in Abhängigkeit von der Anzahl an gestarteten Blocks entgegengesetzt zur Version aus Abbildung 5.1 verhält.

5.4 VERGLEICH DER VERSCHIEDENEN IMPLEMENTIERUNGEN

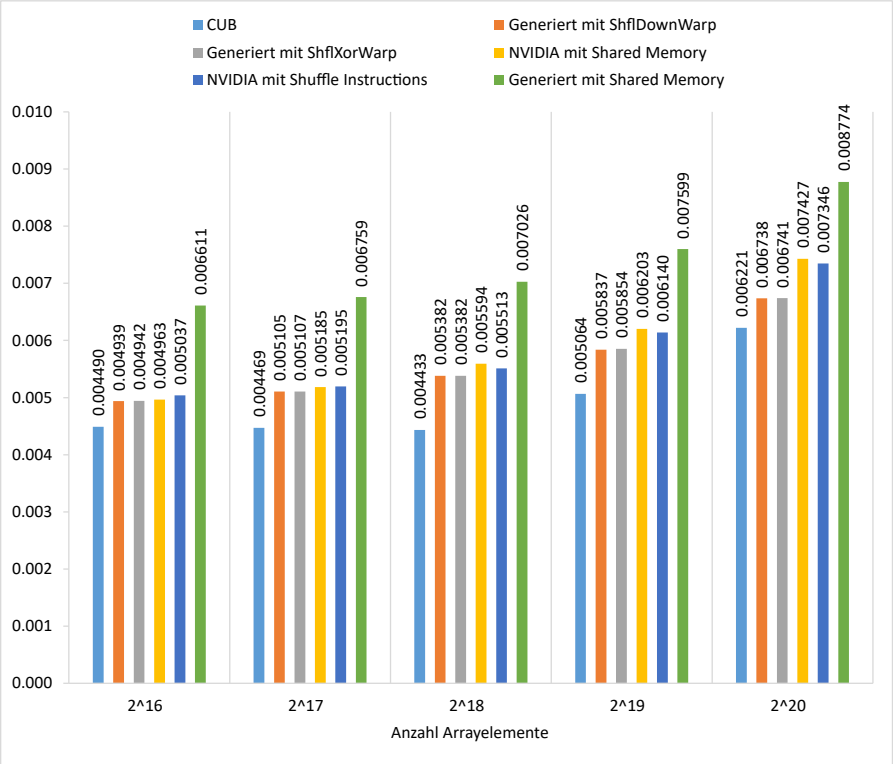

Abbildung 5.3: Laufzeit der verschiedenen Implementierungen in ms

In Abbildung 5.3 sind die Laufzeiten in ms für die Implementierungen zu sehen. Dabei wurde die für die jeweilige Arraygröße effizienteste getestete Variante einer Implementierung verwendet.

Interessant ist hierbei, dass der generierte Code mit Shuffle-Primitiven bei jeder getesteten Arraygröße schneller als die Implementierung von NVIDIA mit Shuffle Instructions ist. Zwischen den beiden gestesteten Shuffle-Primitiven gibt es dabei keine nennenswerten Leistungsunterschiede. Außerdem ist die Implementierung von NVIDIA mit Shuffle Instructions ab einer Arraygröße von 2^{17} schneller als die mit Shared Memory.

Die RISE-Implementierung mit Shared Memory ist deutlich langsamer als die anderen Implementierungen. Eine mögliche Ursache dafür ist, dass dieser Kernel für exakt 1024 Threads pro Block ausgelegt ist und demnach nur mit dieser Threadanzahl getestet wurde. Die CUDA-Implementierung mit Shared Memory hingegen ermöglicht beliebige Threadanzahlen und hat bei jeder getesteten Arraygröße mit 256 Threads pro Block die besten Ergebnisse gebracht.

Die Implementierung aus der CUB-Bibliothek erreicht bei allen getesteten Arraygrößen die kürzesten Ausführungszeiten unter den getesteten Implementierungen.

5.5 EINFLUSS DURCH OPTIMIERUNGEN IM GENERIERTEN CODE

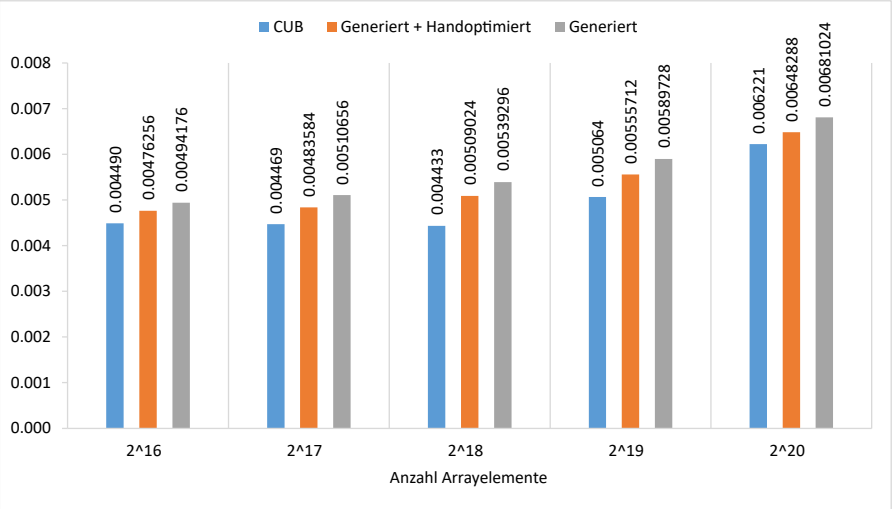

Abbildung 5.4: Laufzeit in ms für Vergleich mit handoptimiertem Code

In Abbildung 5.4 sieht man, anhand des generierten Codes mit *shflXorWarp*, *elemsBlock* = 1024 und *elemsWarp* = 256, welchen Einfluss mögliche Optimierungen haben. Hierbei wurden die nicht notwendigen Synchronisierungen auf Warp-Ebene, die nicht notwendigen Berechnungen der Lane-ID und die für diese Anzahl Threads pro Block nicht notwendigen Schritte bei der zweiten Reduktion auf Warp-Ebene entfernt. Außerdem wurde der Private Memory wiederverwendet.

6 FAZIT

In dieser Arbeit wurden neu entwickelte Patterns präsentiert, welche die Nutzung von Shuffle Instructions in RISE ermöglichen. Aus diesen Patterns lässt sich CUDA-Code generieren, in welchem Daten auf Warp-Ebene ausgetauscht werden können, ohne dabei auf den langsameren Shared Memory zurückgreifen zu müssen.

Unter Verwendung dieser Primitive wurden parallele Reduktionen auf Warp-Ebene in RISE implementiert. Es wurde zudem eine Reduktion auf Block- und Device-Ebene implementiert, welche in ihrer Struktur auf die Anpassungen auf Warp-Ebene zugeschnitten ist. Für verschiedene Varianten der Reduktion auf Device-Ebene wurde CUDA-Code generiert, der Shuffle Instructions verwendet.

Die Performanz dieses generierten CUDA-Codes wurde evaluiert, indem Laufzeitvergleiche mit anderen Implementierungen vorgenommen wurden. Dabei ergab sich ein Vorteil gegenüber der Implementierungen, welche keine Shuffle Instructions nutzen. Zudem sind die Laufzeiten des generierten Codes kürzer, als die einer Implementierung von NVIDIA, welche ebenfalls Shuffle Instructions nutzt. Die Laufzeiten von CUB, einer Programmbibliothek von NVIDIA, welche mit Shuffle Instructions optimierte Reduktionen bereitstellt, konnten jedoch nicht erreicht werden.

Weitere Optimierungen sind dadurch möglich, dass bei der Codegenerierung doppelte Synchronisierungen auf Warp-Ebene vermieden werden und die Wiederverwendung von Speicher in einem größeren Maße ermöglicht wird. Außerdem sollte die Generierung von Code auf Lane-Ebene so angepasst werden, dass keine Berechnung der Lane-ID stattfindet, falls diese für die weiteren Berechnungen nicht benötigt wird. Zudem sind Verbesserungen bei der Vereinfachung arithmetischer Ausdrücke notwendig, um semantisch korrekten Code für Padding auf das nächste Vielfache einer Zahl zu generieren. Da mit der Ampere-Generation von NVIDIA eigenständige Instruktionen eingeführt wurden, welche hardwarebeschleunigte Reduktionen auf Warp-Ebene mit verschiedenen Reduktionsoperatoren ermöglichen [5], sind Leistungsverbesserungen durch die Benutzung dieser Instruktionen denkbar.

LITERATUR

[1] NVIDIA Corporation. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions.

[2] NVIDIA Corporation. *CUDA Zone*. URL: https://developer.nvidia.com/cuda-zone.

[3] NVIDIA Corporation. *NVIDIA TESLA V100 GPU Architecture*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[4] NVIDIA Corporation. *Scan using shfl*. 2019. URL: https://github.com/NVIDIA/cuda-samples/blob/master/Samples/shfl_scan/shfl_scan.cu.

[5] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[6] NVIDIA Research Group. *CUB*. URL: https://nvlabs.github.io/cub/index.html.

[7] Mark Harris und NVIDIA Corporation. *Optimizing Parallel Reduction in CUDA*. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[8] Mark Harris und NVIDIA Corporation. *CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops*. 2013. URL: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

[9] Mark Harris und NVIDIA Corporation. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. 2013. URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/.

[10] ThienLuan Ho, Seung-Rohk Oh und HyunJin Kim. „A parallel approximate string matching under Levenshtein distance on graphics processing units using warp-shuffle operations“. In: *PLOS ONE* 12.10 (Okt. 2017), S. 1–15. DOI: 10.1371/journal.pone.0186251. URL: https://doi.org/10.1371/journal.pone.0186251.

[11] *Implementierung eines CUDA-Backends für den RISE-Compiler*. 2020.

[12] The Khronos Group Inc. *OpenCL overview*. URL: https://www.khronos.org/opencl/.

[13] Yuan Lin, Vinod Grover und NVIDIA Corporation. *Using CUDA Warp-Level Primitives*. 2018. URL: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[14] Y. Liu und B. Schmidt. *LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs*. 2015.

[15] Justin Luitjens und NVIDIA Corporation. *Faster Parallel Reductions on Kepler*. 2014. URL: https://devblogs.nvidia.com/faster-parallel-reductions-kepler/.

[16] Pedro Martín, Roberto Torres und Antonio Gavilanes. „CUDA Solutions for the SSSP Problem". In: Mai 2009, S. 904–913. DOI: 10.1007/978-3-642-01970-8_91.

[17] Chen Peng, Wahib Mohamed, Takizawa Shinichiro und Matsuoka Satoshi. *Pushing the Limits for 2D Convolution Computation On CUDA-enabled GPUs*. 2018.

[18] RISE-Arbeitsgruppe. *RISE language*. 2019. URL: https://rise-lang.org/.

[19] W. Wang, Y. Huang und S. Guo. „Design and Implementation of GPU-Based Prim's Algorithm". In: *International Journal of Modern Education and Computer Science (IJMECS)* 3.4 (2011), S. 55.

[20] Kun Zhou, Qiming Hou, Rui Wang und Baining Guo. „Real-Time KD-Tree Construction on Graphics Hardware". In: *ACM Trans. Graph.* 27 (Dez. 2008), S. 126. DOI: 10.1145/1457515.1409079.

PLAGIATSERKLÄRUNG DES STUDIERENDEN

Hiermit versichere ich, dass die vorliegende Arbeit über die „Implementierung von parallelen Reduktionen in RISE unter Verwendung von Shuffle Instructions “ selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Münster, den

Piet Björn Adick

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

Münster, den

Piet Björn Adick