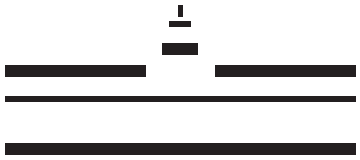
WWU Münster – Fachbereich für Mathematik und Informatik



Bachelorarbeit

Evaluation und Vergleich der Skelettbibliotheken SkelCL und SkePU

Matthias Droste

Matrikelnummer: 371998

Themensteller: Prof. Dr. habil. Sergei Gorlatch
Betreuer: Michel Steuwer
Institut für Informatik
Parallele und Verteilte Systeme

Inhaltsverzeichnis

1 Einleitung

In dieser Arbeit werden die beiden Skelettbibliotheken SkelCL und SkePU evaluiert und verglichen. Das Ziel beider Bibliotheken ist, die datenparallele Programmierung insbesondere auf Grafikprozessoren (GPUs) zu vereinfachen. Dazu kapseln sie existierende Parallelisierungstechnologien wie OpenCL mit einfacher zu nutzenden Schnittstellen. Die Motivation dahinter ist, dass GPUs bei datenparallelen Aufgaben häufig eine wesentlich höhere Leistung als CPUs erreichen, während die Programmierung verhältnismäßig aufwändig ist.

Im Laufe der Arbeit werden SkelCL und SkePU getestet, indem eine bestehende Implementierung des Radixsort-Algorithmus in OpenCL mit Hilfe der beiden Bibliotheken neu implementiert wird. Dabei sind die Testkriterien die Leistung der neuen Implementierungen, die Schwierigkeit der Nutzung der Bibliotheken sowie deren Mächtigkeit.

Im weiteren Verlauf des Textes werden zunächst die technischen und algorithmischen Hintergründe des Themas behandelt (Kapitel 2), dabei geht es um die Parallelisierungstechnologien OpenCL, CUDA und OpenMP sowie um Funktionsskelette und den Radixsort-Algorithmus. Danach wird in Kapitel 3 die in dieser Arbeit behandelte Software beschrieben, das sind die Bibliotheken SkelCL und SkePU und die Implementierung des Radixsort von AMD. Kapitel 4 behandelt die Implementierungen des Radixsort mit SkelCL und SkePU, in Kapitel 5 folgt die Auswertung der Ergebnisse und der Vergleich der Bibliotheken in Bezug auf Leistung, Mächtigkeit und Schwierigkeit der Benutzung. Zuletzt werden die Ergebnisse in Kapitel 6 zusammengefasst.

2 Hintergründe

In diesem Kapitel geht es um technische und algorithmische Hintergründe des Themas dieser Arbeit. Zunächst werden die von SkePU und SkelCL verwendeten Parallelisierungstechnologien OpenCL, CUDA und OpenMP behandelt. Dabei wird der Fokus auf OpenCL gelegt, da CUDA und OpenMP nur von SkePU benutzt werden. Danach geht es um die Funktionsskelette, die ein grundlegendes Prinzip von SkelCL und SkePU darstellen, und den zu implementierenden Radixsort-Algorithmus im Allgemeinen.

2.1 OpenCL

Dieser Abschnitt befasst sich mit OpenCL, einem Parallelisierungsstandard, der von SkelCL und SkePU verwendet wird. Dabei wird das Hardware-, Ausführungs- und Speichermodell besprochen. Außerdem geht es um Möglichkeiten, parallele Funktionen in OpenCL für GPUs zu optimieren.

OpenCL wurde 2008 als herstellerunabhängiger Standard zur datenparallelen Programmierung heterogener Systeme veröffentlicht, der von verschiedenen Treibern für unterschiedliche Hardware implementiert wird. Heterogene Systeme bezeichnen dabei Computersysteme, die über mehrere unterschiedliche Recheneinheiten verfügen, die unter Umständen keinen gemeinsamen Speicher und unterschiedliche Befehlssätze haben. Die Hauptmotivation war und ist, die Nutzung von GPUs für nicht-grafische Berechnungen (*General Purpose Computing on GPUs*, GPGPU) zu vereinfachen.

GPUs sind dabei besonders für Anwendungen interessant, bei denen große Datenmengen auf möglichst gleichmäßige Art und Weise verarbeitet werden. Das liegt daran, dass die theoretische maximale Rechenleistung von GPUs insbesondere bei Gleitkommazahlen häufig ein Mehrfaches der Maximalleistung von CPUs beträgt, aber nur von datenparallelen Programmen effizient genutzt werden kann. Diese Einschränkung liegt an der SIMD-Architektur der einzelnen *Cores* (unabhängigen Prozessoreinheiten) einer GPU: SIMD steht für *single instruction, multiple data* und bedeutet, dass in einem Takt mehrere Daten verarbeitet werden können, dabei aber die selbe Operation ausgeführt werden muss.

Hardwaremodell OpenCL abstrahiert zur Ausführung paraller Funktionen genutzte Hardware als *Devices*, die aus einer oder mehreren *Compute Units* bestehen, die wiederum ein oder mehrere *Processing Elements* enthalten. Dieser Aufbau wird zusammen mit dem Speichermodell (s.u.) in Abbildung 1 visualisiert. Dabei kann ein physisches Gerät auch in mehrere logische *Devices* aufgeteilt werden, wenn seine interne Struktur heterogen ist,

was z.B. beim Cell-Prozessor der Fall ist. Die SIMD-Architektur in typischen GPUs (Die SIMD-Cores einer GPU bestehen jeweils aus mehreren Stream Processors) kann mit Hilfe der zweistufigen Hierarchie von *Compute Units* und *Processing Elements* abgebildet werden. Bei CPUs wird ein Core als eine Compute Unit mit einem Processing Element aufgefasst.

Zusätzlich enthält das Hardwaremodell auch das Hardwaresystem zu Ausführung gewöhnlicher Programme, auf dem auch das OpenCL nutzende Programm ausgeführt wird. Dieses Hardwaresystem wird als Host bezeichnet und ist auch in Abbildung 1 dargestellt.

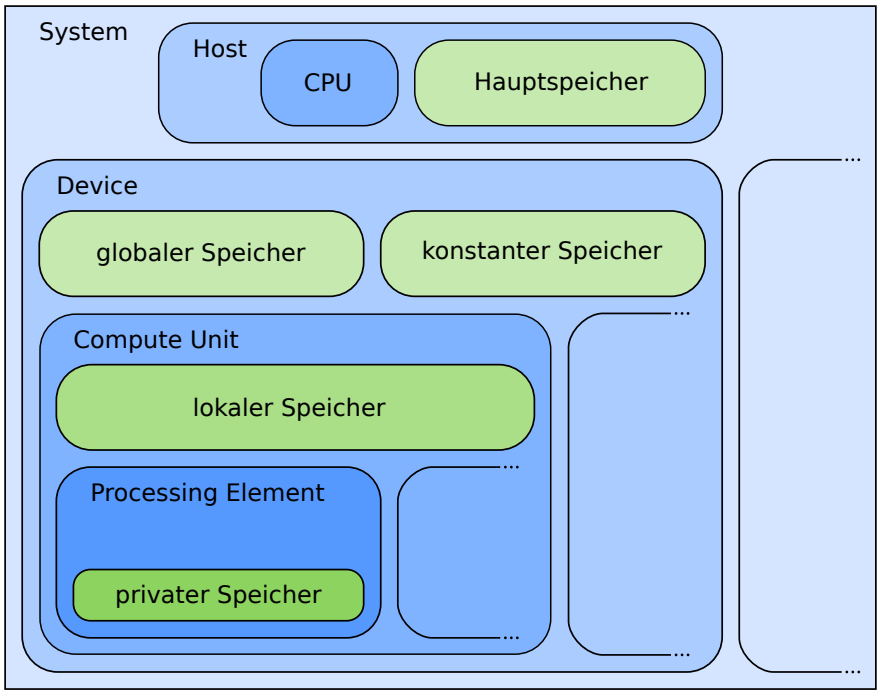


Abbildung 1: OpenCL-Hardwaremodell

Ausführungsmodell Die von OpenCL parallelisierten Funktionen (*Kernel*) geben an, was eine Instanz (*Work-Item*) dieser Funktion tut. Bei der Ausführung eines Kernels wird neben dem zu verwendenden Device die Anzahl der zu startenden Work-Items festgelegt, zusätzlich werden die Work-Items in sogenannte *Work-Groups* einer ebenfalls festzulegenden Größe eingeteilt. Bei der Ausführung erhalten alle Instanzen eines Kernels die selben Argumente, besitzen aber individuelle globale und gruppeninterne IDs, die ihr Verhalten individualisieren. Zusätzlich besitzt jede Work-Group eine ID.

Diese IDs können zum Beispiel angeben, welches Element aus einem Eingabearray verarbeitet wird.

Diese Work-Groups haben zwei Funktionen: Zum Einen dienen sie der Zusammenarbeit zwischen ihren Work-Items, indem sie Synchronisation ermöglichen und über lokalen Speicher (s.u.) verfügen. Zum Anderen bilden sie das Hardwaremodell ab, da eine Work-Group immer auf einer einzigen Compute Unit ausgeführt wird, was zur hardwarespezifischen Optimierung genutzt werden kann.

Um die Nutzung verschiedener heterogener Systeme zu ermöglichen, deren Details bei der Kompilierung eines OpenCL nutzenden Programms noch nicht bekannt sind, sind zur Ausführung einer Funktion mit OpenCL besondere Schritte nötig. Zunächst muss der Kernel zur Laufzeit als Quellcode in der auf C basierenden OpenCL-Sprache vorliegen. Dieser Quellcode wird dann von einer OpenCL-Implementierung für ein Device kompiliert und steht ab da zur Nutzung durch das Hauptprogramm (in OpenCL *Host-Programm* genannt) bereit. Zusätzlich müssen Ein- und Ausgaben eines Kernels explizit zwischen Host und Device transferiert werden. Die Datentransfers und Kernelausführungen funktionieren dabei asynchron über eine *Command Queue*, in die das Host-Programm Aufträge einreiht.

Speichermodell Auf einem Device wird zwischen vier Speichertypen unterschieden, global, konstant, lokal und privat. Diese Unterscheidung dient einerseits dazu, programmlogisch das Teilen von Daten zwischen Work-Items zu regulieren, andererseits sollen ggf. unterschiedliche Hardware-Speichertypen auswählbar sein, um hardwarespezifische Optimierungen durchführen zu können. Der Host kann dabei nur den globalen und konstanten Speicher lesen und schreiben.

Der globale Speicher ist allen Work-Items (auf einem Device) gemeinsam, diese können lesend und schreibend zugreifen. Er stellt zum einen einen universellen Arbeitsspeicher dar, zum anderen ist er nötig, um Daten mit dem Host auszutauschen. Auf GPUs entspricht der globale Speicher dem Grafikspeicher und ist dementsprechend wie der Arbeitsspeicher für die CPU relativ langsam.

Der konstante Speicher ist ebenfalls allen Work-Items gemeinsam, diese haben aber nur lesenden Zugriff. Je nach Hardware und Implementierung können Zugriffe auf den konstanten Speicher schneller als Zugriffe auf den globalen sein.

Der lokale Speicher existiert für jede Work-Group getrennt, er wird zum Datenaustausch innerhalb einer Work-Group und als Cache für den globalen Speicher genutzt. Er ist auf GPUs in Hardware vorhanden und deutlich

schneller als der globale Speicher.

Der private Speicher ist pro Work-Item privat und enthält die thread-lokalen Variablen. Auf GPUs liegt der lokale Speicher in den Registern der Stream Processors und ist damit am schnellsten.

Das Speichermodell und seine Verbindung mit dem Hardwaremodell wird in Abbildung 1 dargestellt, dabei werden die möglichen physischen Zuordnungen dargestellt. Die Zugriffsregeln bedeuten, dass ein Processing Element auf alle Speicherbereiche zugreifen kann, in deren Containern es enthalten ist.

Optimierungen Als Konsequenz aus den Leistungsmerkmalen der Speichertypen auf GPUs ist die wichtigste Maßnahme zur Optimierung von Kerneln, die Zugriffe auf den globalen Speicher zu minimieren. Weiterhin sollten die Zugriffe geordnet erfolgen, d.h. die Work-Items einer Work-Group lesen gleichzeitig die selbe Stelle oder greifen auf hintereinanderliegende Stellen zu, da die Zugriffe sonst je nach Hardware weniger parallelisiert erfolgen. Beides kann durch den Einsatz des lokalen oder privaten Speichers als manuell verwalteten Cache erreicht werden: Wird eine Speicherstelle mehrmals gelesen und/oder geschrieben, können die Zugriffe reduziert werde

Ein anderer Optimierungsansatz ist die Abstimmung der Work-Group-Größe eines Kernels auf die verwendete GPU. Die Work-Group-Größe spielt eine Rolle, weil ein SIMD-Core im Allgemeinen effizienter genutzt wird, wenn darauf mehr Work-Items zur Ausführung bereitliegen. Die Anzahl der Work-Items kann dabei von einer direkten maximalen Anzahl, durch die maximale Work-Group-Anzahl oder durch den zur Verfügung stehenden Speicher begrenzt sein. Dementsprechend kann aus dem Speicherverbrauch eines Kernels und der GPU-Architektur eine günstige Work-Group-Größe und ggf. eine günstige Variante des Kernels ermittelt werden. Hier wird hingegen der einfachere Weg gewählt, die für das verwendete System passende Work-Group-Größe experimentell zu ermitteln.

2.2 CUDA

In diesem Abschnitt wird ein kurzer Überblick über CUDA gegeben, eine weitere Plattform für GPGPU, die von SkePU verwendet wird. Dabei geht es vor allem um Gemeinsamkeiten und Unterschiede im Vergleich zu OpenCL.

CUDA wurde 2006 von Nvidia veröffentlicht und verfolgt mit parallel ausgeführten, numerierten Instanzen einer Funktion und verschiedenen Speicherbereichen ein ähnliches Konzept wie OpenCL. Dabei werden als Hardware zu Ausführung paralleler Funktionen nur GPUs und Beschleunigerkarten von Nvidia unterstützt. Diese Beschleunigerkarten basieren auf GPUs, verwenden aber andere Teiber und können keine Bildschirme ansteuern. Der wesentliche

Unterschied in der Nutzung ist, dass ein Host-Programm, das CUDA benutzt, mit einem speziellen Compiler (nvcc) übersetzt wird. Dieser erkennt und verarbeitet anhand einer zusätzlichen Syntax Kernel und Kernel-Aufrufe, welche ansonsten normalen Funktionen und Funktionsaufrufen ähneln, und lässt den restlichen Code von einem normalen Compiler übersetzen.

2.3 OpenMP

Dieser Abschnitt beschreibt kurz OpenMP, ein weiterer Parallelisierungsstandard, der in SkePU verwendet wird.

OpenMP ist ein Standard zur Erstellung paralleler CPU-Programme, bei dem es möglichst einfach sein soll, bisher rein sequentielle Programme zu parallelisieren. Dabei wird wird im Unterschied zu OpenCL und CUDA nicht nur Datenparallelität, sondern auch Taskparallelität unterstützt, d.h. es können verschiedene Funktionen gleichzeitig ausgeführt werden. Die Nutzung erfolgt über `#pragma`-Compilerdirektiven, die Codeblöcke als parallelisierbar markieren. Wird OpenMP vom Compiler nicht unterstützt, haben die Direktiven keine Auswirkungen, d.h. das Programm ist nicht parallel, aber übersetzbar und funktionsfähig.

Im Detail gibt es verschiedene Arten der Parallelisierung. Zunächst ist es möglich, einen Programmabschnitt mit der `parallel`-Direktive von mehreren Threads ausführen zu lassen. Dabei können die Threads wie in OpenCL ihre ID abfragen. Außerdem ist es möglich, innerhalb eines parallelen Abschnitts For-Schleifen zu parallelisieren, sodass die Iterationen unter mehreren Threads aufgeteilt werden. Die Voraussetzung dafür ist, dass die Anzahl der Iterationen zu Beginn feststeht und dass die Iterationen unabhängig voneinander ausgeführt werden können. Weiterhin können innerhalb eines parallelen Abschnitts mehrere Unterabschnitte parallel von jeweils einem Thread ausgeführt werden. Zusätzlich können jeweils Synchronisationsmechanismen verwendet werden und Variablen können als privat oder zwischen den Threads geteilt definiert werden.

2.4 Funktionsskelette

In diesem Abschnitt werden Funktionsskelette behandelt, eines der grundlegenden Konzepte von SkePU und SkelCL. Dabei werden nach einer allgemeinen Definition die Skelette *Map*, *Zip*, *Reduce* und *Scan* beschrieben, die in beiden Bibliotheken verfügbar sind.

Funktionsskelette sind Funktionen höherer Ordnung, also Funktionen, die andere Funktionen als Parameter erwarten. Die Funktionsskelette stellen dabei einen Algorithmus dar, bei dem eine oder mehrere vom Benutzer des Skeletts angegebene Argumentfunktionen in einem festgelegten Muster auf Daten angewendet werden. Dabei übergibt die Implementierung des Skeletts den Argumentfunktionen die Eingaben und sorgt ggf. für eine parallele Ausführung. Im folgenden werden vier gängige Funktionsskelette beschrieben, hierbei wird für die Anwendung eines Skelette mit einer Argumentfunktion auf Eingaben die Syntax $Skelett(Argumentfunktion, Eingaben...)$ genutzt.

2.4.1 Häufig verwendete Skelette

Map Das einfachste Funktionsskelett ist *Map*. Dabei wird auf jedes Element einer Eingabeliste a eine Argumentfunktion f angewendet und das Ergebnis an die selbe Stelle in der Ausgabeliste gesetzt (siehe auch Abbildung 2):

$$Map(f, a) = (f(a_0), f(a_1), ..., f(a_{n-1})) \tag{1}$$

Dementsprechend ist f eine beliebige Funktion mit einem Argument des Typs T_a (Typ der Elemente von a) und einem beliebigen Ergebnistyp. Da alle n Ausgabeelemente unabhängig voneinander berechenbar sind, lässt sich das Map-Skelett leicht parallelisieren.

Ein Beispiel für eine Map-Anwendung ist die Invertierung eines Vektors: Jede Komponente des Vektors wird mit -1 multipliziert und in die Ausgabe geschrieben.

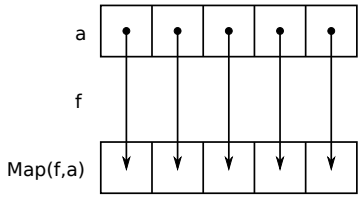


Abbildung 2: Prinzip des Map-Skeletts

Zip Beim *Zip*-Skelett werden zwei gleich lange Eingabelisten a und b zu einer Ausgabeliste verbunden, indem eine Argumentfunktion g auf jedes Paar korrespondierender Elemente angewendet wird:

$$Zip(g, a, b) = (g(a_0, b_0), g(a_1, b_1), ..., g(a_{n-1}, b_{n-1})) \tag{2}$$

Folglich hat g zwei Argumente, eines vom Typ T_a und eines vom Typ T_b, wobei die Argumenttypen nicht gleich sein müssen und der Ergebnistyp wiederum anders sein darf. Dieses Skelett wird in Abbildung 3 visualisiert. Auch hier sind die Ausgabeelemente unabhängig voneinander und die Parallelelisierung ist damit leicht.

Als Beispiel kann man die Vektoraddition betrachten: Die Komponenten der Vektoren bilden die Eingabeliste, die Argumentfunktion addiert beide Komponenten und die Ergebnisliste enthält die Komponenten des Summenvektors.

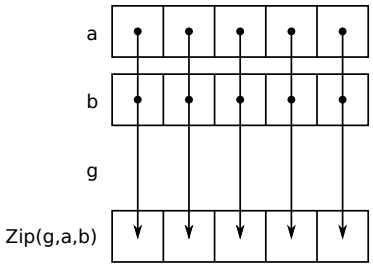


Abbildung 3: Prinzip des Zip-Skeletts

Reduce Da *Reduce*-Skelett reduziert eine Elementliste a auf ein einziges Element. Dazu werden wie in Abbildung 4 gezeigt alle Eingabeelemente durch mehrfache Anwendung eines assoziativen binären Operators $\oplus$ verknüpft:

$$Reduce(\oplus, a) = a_0 \oplus a_1 \oplus ... \oplus a_{n-1} \tag{3}$$

Die Funktion, die den Operator definiert, muss also zwei Argumente mit dem selben Typ wie die Ausgabe haben. Die Parallelisierung funktioniert bei einem Reduce-Skelett etwas anders: Es gibt nur ein Ergebnis, aber voneinander unabhängige Zwischenergebnisse, sodass ein möglichst balancierter Berechnungsbaum erstellt werden kann, bei dem die Operatoren innerhalb einer Ebene parallel ausgeführt werden.

Eine einfache Beipielanwendung ist die Berechnung der Summe mehrerer Zahlen, in diesem Fall ist der Operator die Addition.

Scan Das *Scan*-Skelett basiert auf dem Reduce-Skelett. Statt eines einzelnen Gesamtergebnisses bezüglich eines Operators $\otimes$ wird in der Ergebnisliste an der Stelle i die Verknüpfung aller Elemente vom ersten bis zu i-ten angegeben. Dabei gibt es zwei Varianten: Beim inklusiven Scan enthält das i-te Ergebnis den i-ten Eingabewert, beim exklusiven nicht. Da beim exklusiven Scan das erste Ergebniselement auf keinem Eingabelement basiert, wird dafür das neutrale Element bezüglich *otimes* gewählt. Außerdem ist klarzustellen, dass es kein $n+1$-tes Ausgabeelement gibt, das die Verknüpfung aller Elemente enthalten würde. Formal ist der exklusive Scan also folgendermaßen definiert:

$$Scan_{ex}(\otimes, a) = (0_{\otimes}, a_0, a_0 \otimes a_1, a_0 \otimes a_1 \otimes a_2, ..., a_0 \otimes ... \otimes a_{n-2}) \quad (4)$$

Der inklusive Scan ist entsprechend so definiert:

$$Scan_{in}(\otimes, a) = (a_0, a_0 \otimes a_1, a_0 \otimes a_1 \otimes a_2, ..., a_0 \otimes ... \otimes a_{n-1}) \quad (5)$$

Der exklusive Scan wird in Abbildung 5 gezeigt, der inklusive Scan ist in Abbildung 6 zu sehen.

Die Parallelisierung ist hier etwas schwieriger, da bei einer naiven Implementierung nach der obigen Definition der Argumentoperator $\otimes$ $\mathcal{O}(n^2)$-mal ausgeführt würde, im Vergleich zu $\mathcal{O}(n)$ bei einer sequentiellen Implementierung. Stattdessen wird der Stride-Algorithmus verwendet, der den Operator nur $\mathcal{O}(n)$ Mal anwendet und mit $\mathcal{O}(log(n))$ sequentiellen Berechnungsebenen auskommt.

Auch der Scan kann mit einem Additionsoperator verwendet werden, z.B. bei der Auswertung statistischer Daten.

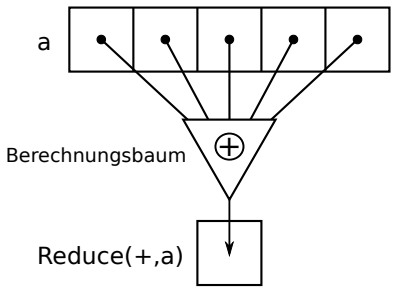


Abbildung 4: Prinzip des Reduce-Skeletts

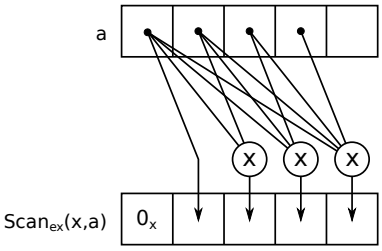


Abbildung 5: Prinzip des exklusiven Scan-Skeletts

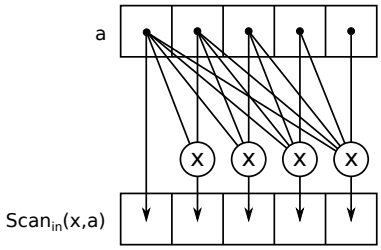


Abbildung 6: Prinzip des inklusiven Scan-Skeletts

2.5 Radixsort

Dieser Abschnitt behandelt den *Radixsort*, einen Sortieralgorithmus für natürliche Zahlen, dessen wichtigste Eigenschaft seine Laufzeitkomplexität ist. Die Grundidee des Radixsort ist, die zu sortierenden Zahlen in mehreren Iterationen nach jeweils einer Stelle zu einer Basis R zu sortieren. Dabei werden die Stellen der Reihe nach von der niedrigsten bis zur höchsten durchlaufen.

In jeder Iteration gibt es R *Buckets* genannte Listen, die den Ziffern zur Basis R zugeordnet sind. Die zu sortierenden Zahlen werden der Reihe nach anhand der Ziffern an der aktuell betrachteten Stelle in die Buckets eingefügt, wobei innerhalb eines Buckets die Eingabereihenfolge bewahrt wird. Am Ende einer Iteration werden die Buckets konkateniert, sodass die Zahlen nach der betrachteten Stelle sortiert sind und bei gleicher Stelle die vorherige Reihenfolge bestehen bleibt.

Dadurch sind die Zahlen nach der letzten Iteration sortiert: Aufgrund der letzten Iteration nach der höchstwertigen Stelle, bei gleicher höchstwertiger Stelle aufgrund der vorletzten Iteration nach der nächsten Stelle usw. bis zur niedrigstwertigen Stelle.

Die Laufzeitklasse des Radixsort ist $\mathcal{O}(n \cdot m)$ für n zu sortierende Zahlen und m zu betrachtende Stellen. Für Integer-Datentypen mit fester Länge kann der Radixsort folglich beliebige mit dem Datentyp darstellbare Zahlen in $\mathcal{O}(n)$ sortieren.

3 Gegebene Software

Dieses Kapitel stellt die Software vor, die die Grundlage dieser Arbeit bildet. Dazu werden zunächst die beiden zu nutzenden Bibliotheken, SkelCL und SkePU, beschrieben, wobei ihre grundlegenden Konzepte und ihre Nutzung im Vordergrund stehen. Danach wird die OpenCL-Implementierung des Radixsort von AMD erläutert, der Algorithmus, der mit Hilfe der beiden Bibliotheken neu implementiert werden soll.

3.1 SkelCL

Dieser Abschnitt behandelt die Konzepte hinter und die Benutzung von SkelCL. SkelCL ist eine in C++ implementierte Skelettbibliothek, d.h. es stellt Funktionsskelette für datenparallele Aufgaben zur Verfügung. Dabei wird intern OpenCL für die Parallelisierung verwendet, nach außen hin wird aber von OpenCL größtenteils abstrahiert. Die beiden wesentlichen Konzepte sind die Nutzung von Skeletten und der abstrahierte gemeinsame Speicher.

Der Rest des Abschnitts behandelt zunächst die zur Speicherabstraktion genutzten Klassen `Vector` und `Matrix`, danach werden die zu Verfügung stehenden Skelette und ihre Nutzung beschrieben, zuletzt geht es um die Struktur eines SkelCL nutzenden Programms.

Speicherabstraktion Eines der beiden Hauptkonzepte in SkelCL ist die Speicherabstraktion, die den in OpenCL zwischen Host und Device getrennten Speicher durch einen gemeinsamen ersetzt. Dafür wird die Template-Klasse `Vector<T>` verwendet, die die Schnittstelle und das Verhalten von `std::vector` größtenteils nachbildet und Elemente des Typs `T` speichert. Dabei werden die Daten intern auf dem Host und auf dem Device gespeichert, nach Änderungen an einer Version wird die andere erst bei einem Zugriff aktualisiert. Eine Änderung am Hostspeicher wird dabei normalerweise durch die Implementierung der Zugriffsoperatoren und -funktionen erkannt, eine Änderung am Device-Speicher wird meistens durch die verantwortlichen Skelette signalisiert. Wenn Änderungen nicht erkannt werden, kann der betreffende Speicher mit `dataOnDeviceModified()` bzw. `dataOnHostModified()` auch manuell als geändert markiert werden. Außerdem können Datentransfers in beide Richtungen synchron und asynchron erzwungen werden.

Zusätzlich steht auch die Klasse `Matrix<T>` zur Verfügung, die eine zweidimensionale Matrix darstellt und ihre Speicherbereiche wie `Vector` in den meisten Fällen automatisch verwaltet.

Für die Vektoren und Matrizen kann zusätzlich angegeben werden, wie sie bei der Nutzung mehrerer Devices auf diese verteilt werden. Dabei ist es

möglich, die Daten gleichmäßig auf die Devices zu verteilen, auf alle Devices zu kopieren oder nur auf einem Device zur Verfügung zu stellen.

Skelette Die Skelette sind in SkelCL als Template-Klassen implementiert, wobei die Template-Parameter die Ein- und Ausgabetypen der zu benutzenden Argumentfunktion sind. Ein Skelett wird ausgeführt, indem das Skelettobjekt mit dem ()-Operator wie eine Funktion benutzt wird. Bei der Instanziierung eines Skeletts wird der Quellcode der Argumentfunktion als String angegeben, wobei beachtet werden muss, dass die Argumente der Funktion keine Vektoren oder Matrizen, sondern die gerade betrachteten Elemente sind. Dementsprechend können die Argumentfunktionen als Stringliteral in den Quellcode eingefügt oder zur Laufzeit aus einer Datei geladen werden. Die Skelette können ihren Argumentfunktionen auch zusätzliche Argumente übergeben, die für alle Instanzen der Funktion gleich sind. Diese Zusatzargumente können auch Vektoren und Matrizen sein, die der Argumentfunktion als Arrays im globalen Speicher zur Verfügung stehen. Weiterhin kann beim Map- und Zip-Skelett die Work-Group-Größe festgelegt werden.

In SkelCL stehen die in Abschnitt 2.4 beschriebenen Skelette Map, Zip, Reduce und exklusiver Scan zur Verfügung. Außerdem gibt es das *Allpairs*-Skelett, das zwei Matrizen als Eingabe erwartet und wie bei Matrixmultiplikation jede Zeile der ersten Matrix mit jeder Spalte der zweiten Matrix kombiniert und die Ergebnisse in eine andere Matrix schreibt. Weiterhin existiert das *MapOverlap*-Skelett, das der Argumentfunktion einen quadratischen Abschnitt einer zweidimensonalen Eingabe zur Verfügung stellt, der um die Eingabekoordinaten bei einem normalen Map zentriert ist. Dabei kann die Größe des Bereichs über einen Parameter des Skeletts festgelegt werden. Der Quellcode dieses Skeletts ist aber noch nicht veröffentlicht worden.

Programmstruktur Bevor SkelCL in einem Programm benutzt werden kann, muss es erst mit `skelcl::init()` initialisiert werden, wobei festgelegt werden kann, wie viele Devices und welcher Device-Typ später verwendet werden sollen. Weiterhin müssen die Skelette, Vektoren und Matrizen instanziiert werden, danach können Skelette ausgeführt werden. Zusätzlich kann SkelCL mit `skelcl::terminate()` vor Programmende beendet werden.

Als Beispiel ist in Listing 1 die Implementierung des Skalarprodukts zweier Vektoren in SkelCL zu sehen. Darin wird SkelCL in Zeile 1 mit vorher ermittelten Device-Parametern initialisiert, in den Zeilen 3 und 4 werden die beiden Skelette zum elementweisen Multiplizieren und Aufsummieren erzeugt, wobei ihnen der Quellcode der Argumentfunktionen übergeben wird. Danach werden die zu multiplizierenden Vektoren in Zeile 6 und 7 erzeugt

```
1  skelcl::init(skelcl::nDevices(deviceCount).deviceType(deviceType));
2
3  Zip<int(int,int)> mult("int func(int x, int y){ return x*y; }");
4  Reduce<int(int)>   sum("int func(int x, int y){ return x+y; }", "0");
5
6  Vector<int> A(size);
7  Vector<int> B(size);
8
9  init(A.begin(), A.end());
10 init(B.begin(), B.end());
11
12 Vector<int> C = sum( mult(A, B) );
```

Listing 1: SkelCL-Beispiel: Skalarprodukt zweier Vektoren (Auszug)

und in Zeile 9 und 10 mit einer Funktion gefüllt. Die eigentliche Berechnung wird in Zeile 12 veranlasst, wo die Verwendung der Skelette als Funktionen zu sehen ist. Dabei ist es noch ein relevantes Detail, dass die Ausgabe des Reduce-Skeletts ein Vektor ist, der ein Element enthält.

3.2 SkePU

Dieser Abschnitt behandelt die Konzepte und Nutzung von SkePU. SkePU ist ebenfalls eine in C++ implementierte Skelettbibliothek, deren wesentliche Konzepte die Skelette und der abstrahierte gemeinsame Speicher sind. Dabei können intern verschiedene Backends zur Ausführung der Skelette verwendet werden: CUDA, OpenCL, OpenMP und eine nicht parallelisierte Implementierung. Allerdings muss das Backend schon zur Compilezeit festgelegt werden und ist für alle Skelette gleich.

Der Rest dieses Abschnitt behandelt die den Speicher abstrahierenden Klassen `Vector` und `Matrix`, die Skelette und Argumentfunktionen sowie den Aufbau eines SkePU-Programms.

Speicherabstraktion SkePU abstrahiert den je nach Backend geteilten Speicher mit Hilfe dreier Template-Klassen. Zwei davon sind `Vector<T>` und `Matrix<T>`, die sehr ähnlich funktionieren wie in SkelCL. Einen Unterschied gibt es beim Erkennen eines Schreibzugriffs auf den Host: SkelCL bietet überladene Zugriffsfunktionen, die in einer Version ein mit `const` schreibgeschütztes Element liefern und den Speicher nicht als geändert markieren, während die andere Version ein schreibbares Element liefert und von Speicheränderungen ausgeht. SkePU nutzt stattdessen die Containerklasse `ProxyElem`, die die meisten Zugriffe durch ihre Operatordefinitionen auf das eigentliche Vektorelement durchleitet, wobei Schreibzugriffe erkannt und markiert werden. Ein weiterer Unterschied ist, dass es keine Methoden gibt, die einen Speicher als

geändert markieren. Es gibt nur Transferanweisungen (`invalidateDevice()` für Transfers zum Device, `updateHost()` für Transfers zum Host), die nur bei der Nutzung von OpenCL oder CUDA einen Effekt haben.

Der dritte Datencontainer ist `SparseMatrix<T>`, dieser speichert dünn besetzte Matrizen in einem speziellen Format, das bei wenigen von 0 verschiedenen Elementen weniger Platz als ein normales Array verbraucht.

Skelette Die Argumentfunktionen werden in SkePU deutlich anders als in SkelCL implementiert. Bei den Argumentfunktionen können nur bestimmte Parametersignatur-Typen realisiert werden, dies sind unäre, binäre und ternäre Funktionen, Funktionen, die ein einzelnes Element und ein Array erwarten, und Funktionen, die keine Eingabe, sondern den Index ihres Ergebnisfeldes erhalten. Bei allen Funktionstypen sind die Eingabetypen und der Ausgabetyp gleich, zusätzlich kann ein konstantes Argument eines ggf. anderen Typs übergeben werden. Dieses konstante Argument darf allerdings kein Vektor und keine Matrix sein. Die Argumentfunktionen werden mit Makros definiert, die als Parameter den Funktionsnamen, den Parametertyp, die Parameternamen und den Funktionsrumpf erwarten. Folglich müssen die Argumentfunktionen im Quelltext stehen oder inkludiert werden. Inkludieren heißte dabei, den Quelltext in einer eigenen Datei zu speichern und mit `#include` vom Präprozessor einfügen zu lassen. Die Funktionsmakros werden vom Präprozessor zu einem Struct expandiert, dass verschiedene Versionen der Funktion für die verwendeten Backends enthält.

Die Skelette selbst sind als Template-Klassen realisiert, deren Template-Parameter die generierte Struktur der zu verwendenden Argumentfunktion ist. Bei der Instanziierung eines Skeletts wird eine neue Instanz der generierten Struktur der Argumentfunktion übergeben, die Ausführung funktioniert wie bei SkelCL mit dem ()-Operator.

Von den in Abschnitt 2.4 beschriebenen Skeletten stehen in SkePU alle zu Verfügung, anders als in SkelCL auch der inklusive Scan. Das Zip-Skelett wird dabei von der `Map`-Klasse übernommen, indem ein Map mit einer binären Funktion verwendet wird. Auf die selbe Weise lassen sich auch drei Datenmengen mit Hilfe einer ternären Funktion elementweise kombinieren.

Zusätzlich gib es noch vier weitere Skelette:

- *MapArray* erwartet als Argumente einen Vektor oder eine Matrix, auf die die Argumentfunktion gemappt wird, und einen anderen Vektor, der in der Argumentfunktion als Array verfügbar ist.

- *MapReduce* führt auf einer Datenmenge zunächst ein Map und dann ein Reduce aus, dementsprechend wird das Skelett mit zwei Argumentfunktionen initialisiert. Das Ergebnis ist das Selbe wie bei einer getrennten

Nutzung von Map und Reduce, die Ausführung kann aber effizienter sein.

- *MapOverlap* übergibt wie die SkelCL-Version der Argumentfunktion zusätzlich zum normal gemappten Element eine einstellbare Anzahl Nachbarn des Elements. Dabei werden ein- und zweidimensionale Eingaben unterstützt.

- *Generate* füllt einen Vektor oder eine Matrix ohne Eingabe, die Argumentfunktion erhält nur den Index des zu berechnenden Werts. Dieses Skelett kann genutzt werden, um parallel zufällige oder regelmäßige Daten zu generieren.

Programmstruktur Im Quelltext eines SkePU-Programms muss noch vor der Einbindung der SkePU-Header mit einer Definition von `SKEPU_OPENCL`, `SKEPU_CUDA` oder `SKEPU_OPENMP` das Backend festgelegt werden. Vor der Ausführung von Skeletten müssen diese und die Datencontainer instanziiert werden, eine Initialisierung von SkePU selbst ist aber anders als bei SkelCL nicht nötig. Als Beispielprogramm ist in Listing 2 ebenfalls das Skalarprodukt zu sehen. Darin werden zuerst in den Zeilen 1 bis 9 die Argumentfunktionen definiert. Danach folgt die `main`-Funktion, in der zunächst ein MapReduce-Skelett und die Eingabevektoren initialisiert werden. Nach zwei Testausgaben folgt die eigentliche Berechnung in Zeile 20. Hier ist zu sehen, dass das Ergebnis des Reduce-Schritts anders als bei SkelCL direkt als Datenelement übergeben wird.

3.3 OpenCL-Radixsort

Dieser Abschnitt erläutert die Radixsort-Implementierung in OpenCL, die zum Vergleich mit SkePU und SkelCL nachimplementiert werden soll. Dabei wird genauer auf die Funktionsweise eingegangen.

Die Radixsort-Implementierung von AMD ändert das Grundkonzept des Radixsort aus Abschnitt 2.5 in zwei Punkten:

Zum Einen werden die Zahlen während einer Iteration nicht sofort und nicht in Bucket-Listen einsortiert, sondern am Ende der Iteration direkt in ein einziges Ausgabearray eingetragen. Dazu muss bekannt sein, welchem Abschnitt des Ausgabearrays ein Bucket entspricht. Diese Information folgt aus den Bucketgrößen, die wiederum den Häufigkeiten der einzelnen Ziffern entsprechen. Deshalb wird zunächst ein Histogramm der Ziffernhäufigkeiten ermittelt, aus dem dann die Positionstabelle der Buckets berechnet wird. Zuletzt werden dann die Eingabezahlen anhand der Positionstabelle in ihre

```
1  // User-function used for mapping
2  BINARY_FUNC(mult_f, float, a, b,
3      return a*b;
4  )
5
6  // User-function used for reduction
7  BINARY_FUNC(plus_f, float, a, b,
8      return a+b;
9  )
10
11 int main(){
12     skepu::MapReduce<mult_f, plus_f> dotProduct(new mult_f, new plus_f);
13
14     skepu::Vector<float> v0(20, (float)2);
15     skepu::Vector<float> v1(20, (float)5);
16
17     std::cout<<"v0: " <<v0 <<"\n";
18     std::cout<<"v1: " <<v1 <<"\n";
19
20     float r = dotProduct(v0, v1);
21
22     std::cout<<"r: " <<r <<"\n";
23
24     return 0;
25 }
```

Listing 2: SkePU-Beispiel: Skalarprodukt zweier Vektoren

Ausgabebuckets verschoben, was der Algorithmus als Permutation bezeichnet.

Zum Anderen sollen diese drei Schritte - Histogrammermittlung, Positionstabellenberechnung und Permutation - parallelisiert werden. Dies wäre mit einem globalen Histogramm und einer globalen Positionstabelle ineffizient, weil das Erhöhen eines Zählers im Histogramm und das Schreiben in einen Bucket sychronisiert werden müssen, da sonst Histogrammdaten und Zahlen in der Ausgabe durch *race conditions* verloren gehen können. Als Gegenmittel wird die Eingabe in Blöcke aufgeteilt, für die getrennte Histogramme berechnet und eigene Buckets reserviert werden.

Im Detail arbeitet diese Implementierung mit $R = 256$ als Basis und sortiert n `uint`-Zahlen. Dabei muss n ein Vielfaches von $R \cdot G$ sein, wobei $G = 64$ die Work-Group-Größe für manche Kernel bestimmt. Diese Einschränkung von n lässt sich aber durch das Einfügen von Randwerten des Wertebereichs als Dummy-Elemente umgehen, da diese nach dem Sortieren leicht entfernt werden können. Weiterhin werden für n nur Werte bis 65536 zugelassen, diese Beschränkung wird mit der typischen Größe des lokalen Speichers auf GPUs begründet. Die Länge der Eingabeblöcke beträgt ebenfalls R.

Im Folgenden werden die drei Schritte und die dabei ausgeführten Kernel näher erklärt. Einen Überblick über den Ablauf und den Datenfluss gibt

Abbildung 7. Bei den Kernel-Auszügen werden bestimmte Variablendefinitionen ausgelassen, dabei sind nicht selbsterklärenden Bezeichnungen `gidx`/`gidy` für globale IDs, `tidx`/`tidy` für lokale IDs und `bidx`/`bidy` für Gruppen-IDs. Außerdem werden die Namen der Arrays an die Namen im Host-Quelltext angepasst.

3.3.1 Histogrammermittlung

In dieser ersten Phase wird für jeden Block der Länge R ein Histogramm berechnet, das aufgrund der Basis ebenfalls die Länge R hat. Das Konzept dieser Phase ist in Abbildung 8 zu sehen.

Dies geschieht im Kernel `Histogram`, von dem n Work-Items gestartet werden. Die wesentlichen Teile des Kernels finden sich in Listing 3. Jeweils R Work-Items bilden dabei eine Work-Group und berechnen zu einem Eingabeblock ein Histogramm. Dazu wird das Histogramm einer Gruppe zunächst im lokalen Speicher angelegt und mittels atomarer Operationen threadsicher ausgefüllt (Zeile 15-17). Nachdem dies für alle Work-Items der Gruppe abgeschlossen ist, wird das fertige Histogramm in den globalen Histogrammspeicher `histogramBins` kopiert (Zeile 22/23). `histogramBins` wird dazu als $\frac{n}{R} \times R$-Matrix interpretiert, in der jede Zeile ein Histogramm enthält.

```
1  __kernel
2  void histogram(__global const uint* unsortedData,
3                 __global uint* histogramBins,
4                 uint shiftCount,
5                 __local uint* sharedArray)
6  {
7      uint numGroups = get_global_size(0) / get_local_size(0);
8
9      /* Initialize shared array to zero */
10     sharedArray[localId] = 0;
11
12     barrier(CLK_LOCAL_MEM_FENCE);
13
14     /* Calculate thread-histograms */
15     uint value = unsortedData[globalId];
16     value = (value >> shiftCount) & 0xFFU;
17     atomic_inc(sharedArray+value);
18
19     barrier(CLK_LOCAL_MEM_FENCE);
20
21     /* Copy calculated histogram bin to global memory */
22     uint bucketPos = groupId  * groupSize + localId ;
23     histogramBins[bucketPos] = sharedArray[localId];
24 }
```

Listing 3: Histogram-Kernel (Auszug)

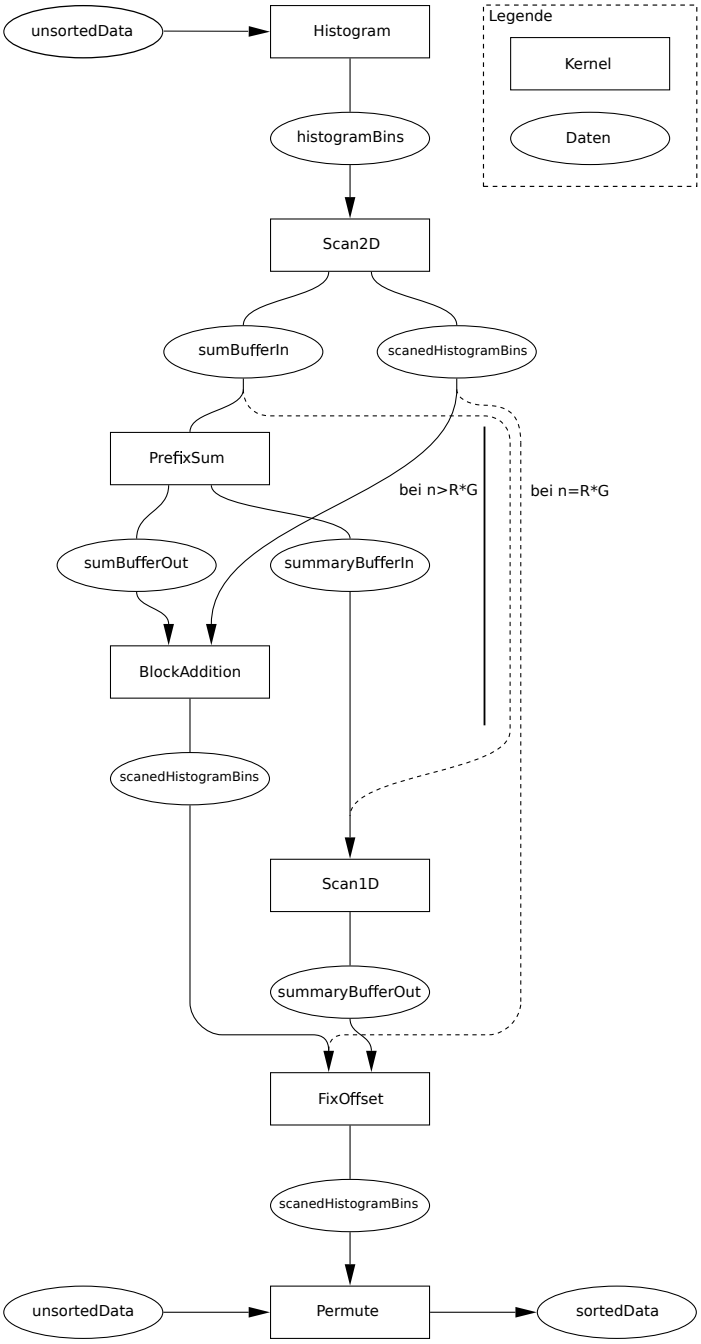


Abbildung 7: Struktur des Algorithmus

3.3.2 Berechnung der Positionstabellen

In der zweiten Phase werden aus den zuvor ermittelten Histogrammen die Positionstabellen für die Permutation im dritten Schritt berechnet.

Die Positionstabellen ergeben sich durch einen Scan mit der Addition als Operator aus den Histogrammen. Solche Anwendungen des Scan-Skeletts werden im folgenden als *Additionsscan* und *scannen* bezeichnet. Dieser Scan funktioniert, weil die Buckets primär nach ihrer Ziffer und sekundär nach ihrem Block geordnet sind. Dadurch liegen vor einem bestimmten Bucket genau die Buckets, die zu einer kleineren Ziffer gehören sowie die Buckets, die zur selben Ziffer gehören und für einen vorherigen Block gedacht sind. Folglich erhält man die Anfangsposition eines Buckets, indem man die Größen der vorherigen Buckets addiert.

Diese summierten Größen könnten direkt mit einem normalen exklusiven Additionsscan auf `histogramBins` ermittelt werden, wenn diese Matrix spaltenweise linearisiert wäre. Da die Matrix aber zeilenweise gespeichert ist, weil sonst das Schreiben am Ende des `Histogram`-Kernels zu Bankkonflikten führen würde, ist der Vorgang etwas komplizierter und wird auf mehrere Kernel verteilt. Die Idee dahinter ist, den Scan zwei mal aufzuteilen:

- In der ersten Aufteilung wird zunächst jede Spalte einzeln betrachtet und darauf ein exklusiver Additionsscan mit nicht hintereinanderliegenden Eingabeelementen ausgeführt, die Endsummen werden in einem zusätzlichen Zeilenvektor gespeichert. Dann wird der Vektor der Endsummen exklusiv gescannt, um zu jeder Spalte die Summe aller vorherigen Spalten zu erhalten. Dieser gescannte Vektor wird dann zu jeder Zeile der spaltenweise gescannten Matrix addiert, um die insgesamt gescannte Matrix zu erhalten. Nach dieser Aufteilung hat der

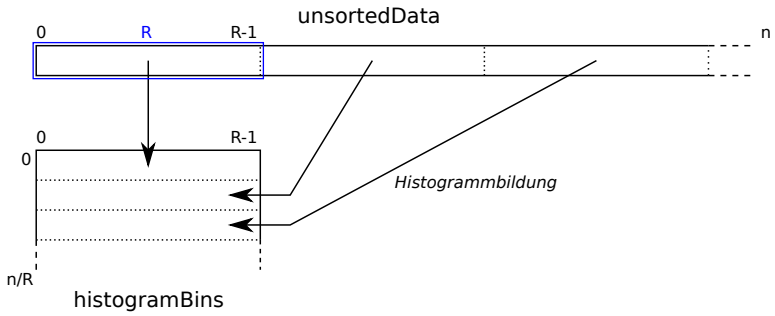


Abbildung 8: Funktionsprinzip des `Histogram`-Kernels. Der blau umrandete Eingabeblock wird von einer Work-Group verarbeitet.

Algorithmus die vereinfachte Form, die in Abbildung 7 mit den gestrichelten Abkürzungen markiert ist.

- Die zweite Aufteilung überträgt diese Strategie auf das Scannen der einzelnen Spalten im obigen ersten Schritt. Diese Spalten werden in Blöcke der Länge G unterteilt und blockweise exklusiv gescannt. Die Teilergebnisse werden wiederum gespeichert; diesmal wird dafür eine Matrix benötigt. Diese Matrix wird spaltenweise gescannt, um einerseits die Zwischensummen auf die blockweise gescannten Histogramme zu addieren und andererseits die Spaltensummen zu erhalten, die in der ersten Aufteilung benötigt werden. Nach der zweiten Aufteilung hat der Algotithmus die komplette in Abbildung 7 gezeigte Struktur

Falls n den minimalen zulässigen Wert $R{\cdot}G$ hat, findet diese zweite Aufteilung nicht statt.

Scan2D Der `Scan2D`-Kernel führt den ersten Teilscan aus, bei dem Abschnitte der Länge G aus den Spalten der Histogrammmatrix `histogramBins` gescannt werden. Abbildung 9 visualisiert diesen Vorgang, der Code ist in Listing 4 zu sehen.

Dazu wird für jedes Histogrammfeld ein Work-Item gestartet und innerhalb der Work-Group ein Stride-Algorithmus verwendet (Zeile 21-29), eine Work-Group umfasst dabei einen Scanabschnitt, also eine Spalte und G Zeilen. Die zu berabeitenden Elemente einer Gruppe werden im lokalen Speicher gecacht (Zeile 15), was die Leistung auf GPUs verbessert.

Die gruppenweise gescannten Histogramme werden in der analog zur Eingabe aufgebauten Matrix `scanedHistogramBins` ausgegeben (Zeile 35-40). Die jeweilige Summe aller Histogramme einer Gruppe wird in `sumBufferIn` gespeichert (Zeile 32), `sumBufferIn` ist dem entsprechend eine Matrix mit R Spalten für die Ziffern und $\frac{n}{RG}$ Zeilen für die Scan-Abschnitte einer Histogrammspalte. Gibt es nur eine Work-Group, werden die beiden folgenden Kernel die beiden folgenden Kernel, `PrefixSum` und `BlockAddition`, nicht benötigt und deshalb übersprungen. In diesem Fall wird `sumBufferIn` direkt als Eingabe für den darauffolgenden Kernel `Scan1D` verwendet.

PrefixSum Dieser Kernel führt den exklusiven Additions-Scan auf den Teilsummen in `sumBufferIn` aus. Die Zwischensummen in den Spalten werden in `sumBufferOut` gespeichert, die Spaltensummen in `summaryBufferIn`. Daher hat `sumBufferOut` die Größe $\frac{n}{RG} \times R$, `summaryBufferIn` die Länge R. Das Funktionsprinzip wird in Abbildung 10 gezeigt, der Code findet sich in Listing 5.

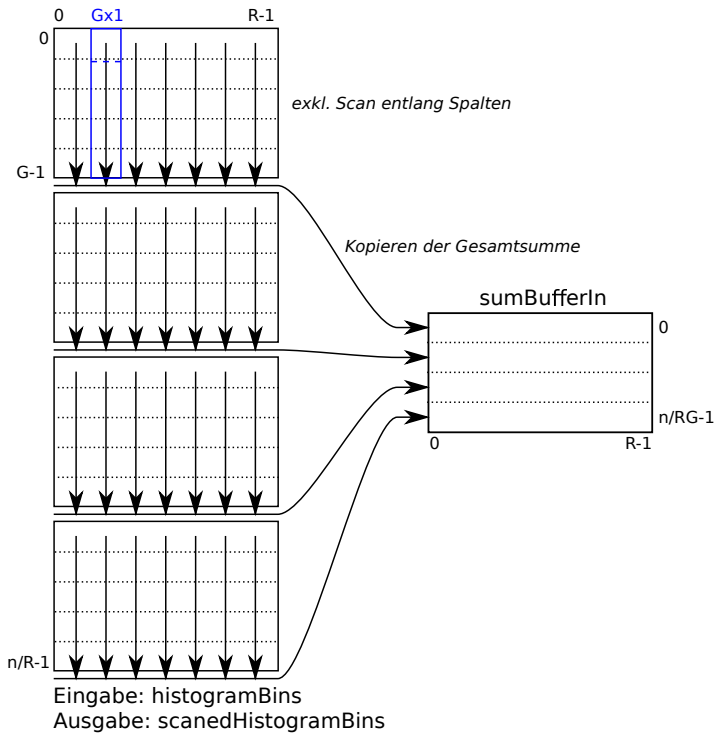


Abbildung 9: Funktionsprinzip des `Scan2D`-Kernels. Blau umrandet: Arbeitsbereich einer Work-Group

```
#define RADIX 8

__kernel void ScanArraysdim2(__global uint *scanedHistogramBins,
                             __global uint *histogramBins,
                             __local uint* block,
                             const uint block_size,
                             const uint stride,
                             __global uint* sumBufferIn)
{
    int lIndex = tidy * block_size + tidx;
    int gpos = (gidx << RADIX) + gidy;
    int groupIndex = bidy * (get_global_size(0)/block_size) + bidx;

    /* Cache the computational window in shared memory */
    block[tidx] = histogramBins[gpos];
    barrier(CLK_LOCAL_MEM_FENCE);

    uint cache = block[0];

    /* build the sum in place up the tree */
    for(int dis = 1; dis < block_size; dis *=2){
        if(tidx>=dis){
            cache = block[tidx-dis]+block[tidx];
        }
        barrier(CLK_LOCAL_MEM_FENCE);

        block[tidx] = cache;
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    /* store the value in sum buffer before making it to 0 */
    sumBufferIn[groupIndex] = block[block_size-1];

    /*write the results back to global memory */
    if(tidx == 0){
        scanedHistogramBins[gpos] = 0;
    }
    else{
        scanedHistogramBins[gpos] = block[tidx-1];
    }
}
```

Listing 4: Scan2D-Kernel (Auszug)

Der Scan wird hier mit einem Work-Item pro Element in `SumBufferIn` implementiert, dabei berechnet jedes Work-Item die Summe der vorhergehenden Elemente komplett selbst (Zeile 7-10).

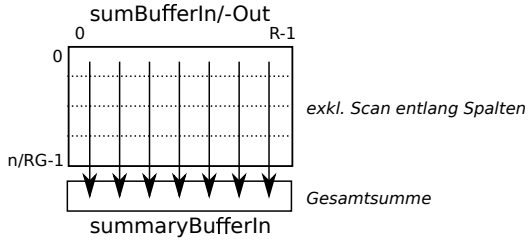


Abbildung 10: Funktionsprinzip des `PrefixSum`-Kernels

```
1  __kernel void prefixSum(__global uint* sumBufferOut,__global uint*
       sumBufferIn,__global uint* summaryBufferIn,int stride)
2  {
3      int Index = gidy * stride + gidx;
4
5      sumBufferOut[Index] = 0;
6
7      if(gidx > 0){
8          for(int i=0; i<gidx; i++)
9              sumBufferOut[Index] += sumBufferIn[gidy * stride + i];
10     }
11
12     if(gidx == stride -1)
13         summaryBufferIn[gidy] = sumBufferOut[Index] + sumBufferIn[gidy *
               stride + (stride-1)];
14 }
```

Listing 5: PrefixSum-Kernel (Auszug)

BlockAddition In diesem Kernel werden die Zwischensummen innerhalb der Spalten zu den blockweise gescannten Histogrammen addiert, um spaltenweise gescannte Histogramme zu erhalten. Dazu teilt der Kernel die Histogramme in `scanedHistogramBins` wieder in Gruppen der Größe G ein und addiert zu jedem teilweise gescannten Histogramm in einer Gruppe i die Spalten-Zwischensummen aus der i-ten Zeile von `sumBufferOut` (siehe Abbildung 11). Das Ergebnis wird in `scanedHistogramBins` zurückgeschrieben.

Es wird ein Work-Item pro Element in `scanedHistogramBins` gestartet, die Work-Group-Größe liegt bei $(G \times 1)$. Dadurch lesen alle Work-Items einer Work-Group das selbe Element in `sumBufferOut`, was den Zugriff bei GPUs häufig beschleunigt. Dies in Listing 6 in Zeile 9 zu sehen.

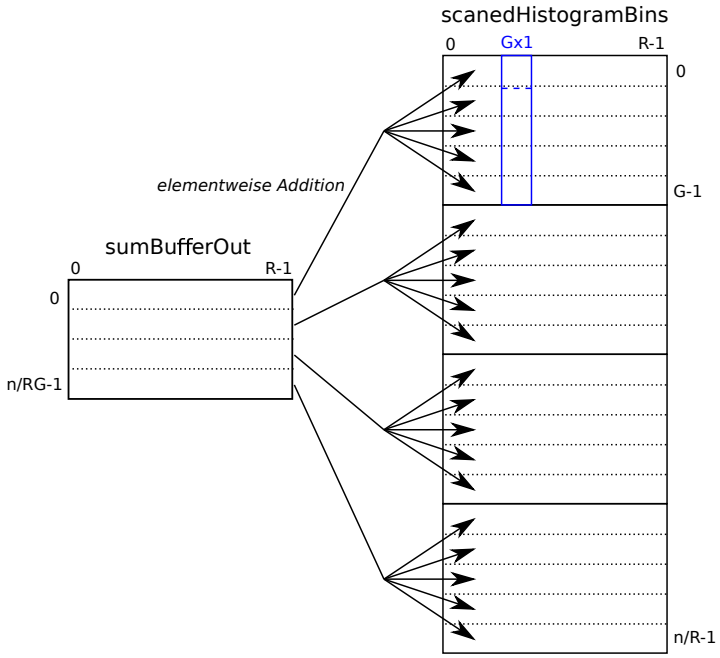


Abbildung 11: Funktionsprinzip des `BlockAddition`-Kernels. Blau umrandet: Arbeitsbereich einer Work-Group

```
1  #define RADIX 8
2
3  __kernel void blockAddition  (__global uint* sumBufferOut,__global uint*
       scanedHistogramBins,uint stride){
4      int gpos = gidy + (gidx << RADIX);
5
6      int groupIndex = bidy * stride + bidx;
7
8      uint temp;
9      temp = sumBufferOut[groupIndex];
10
11     scanedHistogramBins[gpos] += temp;
12 }
```

Listing 6: BlockAddition-Kernel (Auszug)

Scan1D Dieser Kernel führt einen Additions-Scan auf den Spaltensummen aus, um die Summen aller jeweils davor liegenden Spalten zu erhalten. Das Ergebnis dieses Scans auf `summaryBufferIn` wird im Array (bzw. Zeilenvektor) `summaryBufferOut` gespeichert, das ebenfalls die Länge R hat. Dies wird in Abbildung 12 dargestellt.

Auch dieser Scan ist durch ein Work-Item pro Eingabezahl und einen Stride-Algorithmus implementiert (Listing 7, Zeile 12-20), zusätzlich wird die komplette Eingabe der Länge R im lokalen Speicher gecacht (Zeile 9).

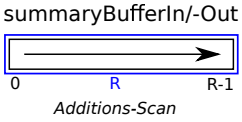


Abbildung 12: Funktionsprinzip des `Scan1D`-Kernels. Blaue Umrandung: Die einzige Work-Group verarbeitet alle Daten.

```
1  __kernel void ScanArraysdim1(__global uint *summaryBufferOut,
2                               __global uint *summaryBufferIn,
3                               __local uint* block,
4                               const uint block_size)
5  {
6      /* Cache the computational window in shared memory */
7      block[tid] = summaryBufferIn[gid];
8
9      uint cache = block[0];
10
11     /* build the sum in place up the tree */
12     for(int stride = 1; stride < block_size; stride *=2){
13         if(tid>=stride){
14             cache = block[tid-stride]+block[tid];
15         }
16         barrier(CLK_LOCAL_MEM_FENCE);
17
18         block[tid] = cache;
19         barrier(CLK_LOCAL_MEM_FENCE);
20     }
21
22     /*write the results back to global memory */
23     if(tid == 0){
24         summaryBufferOut[gid] = 0;
25     }
26     else{
27         summaryBufferOut[gid] = block[tid-1];
28     }
29 }
```

Listing 7: Scan1D-Kernel (Auszug)

FixOffset Hier wird der Zeilenvektor mit den gescannten Spaltensummen zu jeder Zeile der spaltenweise gescannten Matrix `scanedHistogramBins` addiert, sodass die fertige Positionstabelle entsteht. Die Addition wird mit einem Work-Item pro Element in `scanedHistogramBins` ausgeführt. Eine Illustration findet sich in Abbildung 13, der Quelltext in Listing 8.

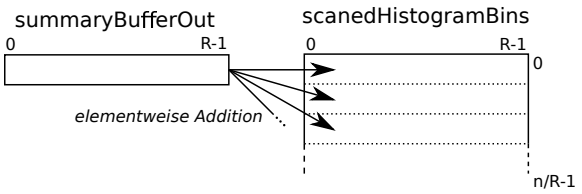


Abbildung 13: Funktionsprinzip des `FixOffset`-Kernels

```
1  #define RADIX 8
2
3  __kernel void FixOffset(__global uint* summaryBufferOut,__global uint*
       scanedHistogramBins){
4      int gpos = gidy + (gidx << RADIX);
5
6      scanedHistogramBins[gpos] += summaryBufferOut[gidy];
7  }
```

Listing 8: FixOffset-Kernel (Auszug)

3.3.3 Permutation

In dieser Phase findet die eigentliche Sortierung statt. Dazu werden im `Permute`-Kernel die Eingabezahlen eines Blocks der Reihenfolge nach betrachtet und anhand der Positionstabelle zu diesem Block im Ausgabearray `sortedData` abgelegt (Abbildung 14). Dabei muss die Bucketadresse in der Positionstabelle nach einer Nutzung inkrementiert werden, damit die nächste Zahl mit der selben Ziffer an der nächsten Position abgelegt wird.

Dadurch werden die Eingabezahlen nach der aktuell betrachteten Stelle sortiert, bei gleicher Ziffer bleibt die vorherige Reihenfolge erhalten: Stammen Zahlen mit gleicher Ziffer aus verschiedenen Blöcken, erhalten sie die richtige Reihenfolge, weil die entsprechenden Buckets nach Blöcken geordnet sind. Stammen sie aus dem selben Block, behalten sie ihre Reihenfolge, weil sie nach ihrer Eingabeposition abgearbeitet werden.

Weil die Zahlen eines Blocks der Reihenfolge nach abgearbeitet werden müssen, kann nur ein Work-Item pro Block gestartet werden. Ein Work-Item

speichert dabei seine sich verändernde Positionstabelle in einem nur von ihm genutzten Abschnitt des lokalen Speichers (Listing 9, Zeile 14). Hier wird der lokale Speicher verwendet, weil ein Histogramm 256 Zahlen enthält und deshalb eventuell nicht in den privaten Speicher passt. Die Work-Items werden in Work-Groups der Größe G gestartet, was aber nur für die Gechwindigkeit relevant ist, da innerhalb einer Work-Group keine Daten ausgetauscht werden.

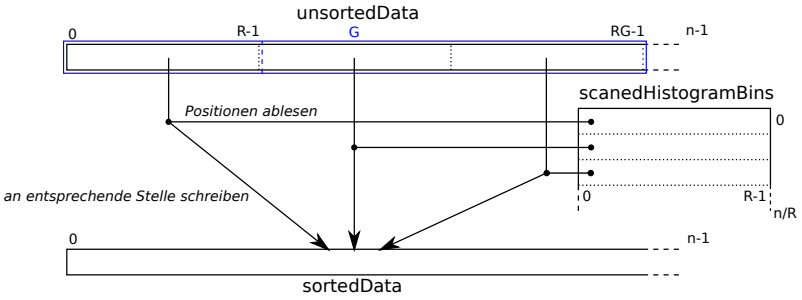


Abbildung 14: Funktionsprinzip des `Permute`-Kernels. Blau umrandet: Arbeitsbereich einer Work-Group, dieser umfasst G Blöcke der Länge R

```
1  #define RADIX 8
2  #define RADICES (1 << RADIX)
3
4  __kernel
5  void permute(__global const uint* unsortedData,
6               __global const uint* scanedHistogramBins,
7               uint shiftCount,
8               __local ushort* sharedBuckets,
9               __global uint* sortedData)
10 {
11     /* Copy prescaned thread histograms to corresponding thread shared block
           */
12     for(int i = 0; i < RADICES; ++i){
13         uint bucketPos = groupId * RADICES * groupSize + localId * RADICES
               + i;
14         sharedBuckets[localId * RADICES + i] =
               scanedHistogramBins[bucketPos];
15     }
16
17     barrier(CLK_LOCAL_MEM_FENCE);
18
19     /* Premute elements to appropriate location */
20     for(int i = 0; i < RADICES; ++i){
21         uint value = unsortedData[globalId * RADICES + i];
22         value = (value >> shiftCount) & 0xFFU;
23         uint index = sharedBuckets[localId * RADICES + value];
24         sortedData[index] = unsortedData[globalId * RADICES + i];
25         sharedBuckets[localId * RADICES + value] = index + 1;
26         barrier(CLK_LOCAL_MEM_FENCE);
27     }
28 }
```

Listing 9: Permute-Kernel (Auszug)

4 Implementierungen

Dieses Kapitel beschäftigt sich mit den Implementierungen des Radixsort in SkelCL und SkePU. Bei beiden Implementierungen soll der Ablauf der Radixsort-Implementierung mit der Unterteilung in die drei Phasen Histogrammerzeugung, Positionstabellenberechnung und Permutation beibehalten werden. Dabei wird großteils auch die Aufteilung in die einzelnen Kernel (siehe Abbildung 7) nachgebildet. Diese auf den Kerneln basierenden Schritte sollen in beiden Versionen zur Vereinfachung des Entwurfs möglichst ähnlich ablaufen. Deshalb sollen so weit wie möglich die Skelette eingesetzt werden, die den beiden Bibliotheken gemeinsam sind. Aufgrund dieser ähnlichen Struktur enthält dieses Kapitel zunächst einen Abschnitt über die gemeinsamen Konzepte beider Implementierungen, danach folgen die beiden Abschnitte mit individuellen Details der Programme.

4.1 Gemeinsame Konzepte

Zunächst soll es um allgemeine Überlegungen zur Umsetzung des Radixsort mit Skeletten gehen, die für beide Implementierungen gelten. Dazu wird zunächst ein Wechsel des grundlegenden Datentyps erklärt, bevor die zu verwendenen Skelette so weit wie möglich festgelegt werden. Außerdem werden gemeinsame Konstanten definiert.

Blöcke Der wichtigste Ansatz zur Übersetzung der gegebenen Kernel in Map, Zip, Scan und Reduce ist die logische Einteilung der Daten in Blöcke der Größe R. Diese Einteilung gilt nicht nur für die Histogramme, Positionstabellen und Zwischenstufen, sondern auch für die Ein- und Ausgabearrays der sortierten Zahlen, da diese in der Histogrammphase einem zu erstellenden Histogramm und in der Permutationsphase einer Positionstabelle zugeordnet sind. Die Idee ist nun, diese Blöcke als Dateneinheiten der Skelette zu verwenden: Aus einem Eingabeblock wird ein Histogramm berechnet, in der Scan-Phase werden (außer bei `Scan1D`) Histogramme addiert und bei der Permutation werden ein Eingabeblock und eine Positionstabelle kombiniert (dabei ist das Ergebnis allerdings kein Block). Dazu werden die Blöcke nicht mehr als Abschnitte eines Zahlenarrays gespeichert, sondern als Datenstrukturen, die jeweils ein Array mit R Zahlen enthalten.

Vereinfachter Scan Die Scan-Phase kann kann mit dem zur Verfügung stehenden Scan-Skelett deutlich vereinfacht werden, da die ggf. nötige Aufteilung der Histogramme in getrennt zu scannende Gruppen von der Implementierung des Skeletts übernommen wird. Damit kann der Ablauf der

Scan-Phase wie bei der AMD-Implementierung im Fall $n = R \cdot G$ vereinfacht werden, wie es in Abschnitt 3.3.2 beschrieben wird und in Abbildung 7 mit den gestrichelten Abkürzungen markiert ist. Folglich werden in dieser Phase nur die Schritte `Scan2D`, `Scan1D` und `fixOffset` benötigt. Bei der Verwendung des exklusiven Scan-Skeletts ist noch zu beachten, dass die Summe aller Histogramme in `summaryBufferIn` dabei nicht ausgegeben wird, aber leicht als Summe des letzten ungescannten und des letzten gescannten Histogramms berechnet werden kann.

Nicht parallelisierbare Anteile Der Scan auf dem globalen Histogramm `summaryBufferIn` in `Scan1D` lässt sich auf Block-Ebene nicht parallelisieren, dasselbe gilt für die Ermittlung von `summaryBufferIn` als Summe von zwei Blöcken. Es hätte daher sinnvoll sein können, diese Berechnungen auf dem Host durchzuführen, aber es stellte sich heraus, dass es effektiver ist, ein Map auf eine Eingabe mit einem Element zu starten, da so Datentransfers zwischen Host und Device vermieden werden.

Skelette Mit den drei oben beschriebenen Überlegungen lässt sich die gegebene Radixsort-Implementierung folgendermaßen in Skelette übersetzen:

- Die Histogrammerzeugung stellt ein Map auf die Eingabeblöcke mit den Histogrammen als Ausgabe dar; zusätzlich muss die Argumentfunktion die zu betrachtende Stelle kennen.
- Das Scannen der Histogramme ist ein exklusiver Scan auf dem Histogrammarray. Der Verknüpfungsoperator ist die elementweise Addition zweier Blöcke, das neutrale Element ein Block, dessen Felder alle 0 sind.
- Die Summe aller Histogramme, `summaryBufferIn`, ist die Summe des letzen gescannten und des letzten ungescannten Histogramms. Diese Summe wird mit einem Map auf ein einzelnes Element berechnet.
- Das Scannen von `SummaryBufferIn` findet wie oben schon beschrieben ebenfalls mit einem Map auf eine einelementige Eingabe statt.
- Der `FixOffset`-Kernel stellt ein Map auf die gescannten Histogramme dar, zusätzlich müssen alle Work-Items das gescannte Gesamthistogramm `SummaryBufferOut` kennen.
- Die Permutation lässt sich nicht mit den Standardskeletten darstellen. Es werden wie beim Zip jeweils ein Eingabeblock und ein Positionstabellenblock kombiniert, aber das Ergebnis ist kein Block, sondern R

Werte in der Ausgabe. Für diese Funktion sind daher bibliothekspezifische Skelette nötig. Außerdem muss auch hier die betrachtete Stelle bekannt sein.

Konstanten In den Programmen und in den folgenden beiden Abschnitten werden die folgenden Konstanten verwendet:

- `RADIX`: Basis, zu der sortiert wird, und Blockgröße
- `RADIX_BITS`: Anzahl Bits, die einer Stelle zur Basis `RADIX` entsprechen, also log_2(`RADIX`)
- `BITMASK`: Bitmaske, die zur Extraktion der letzten Stelle aus einer Zahl verwendet wird. Der Wert ist folglich `RADIX-1`.
- `ITERATIONS`: Anzahl der benötigten Sortieriterationen, folgt aus der Länge von `unsigned int`-Zahlen und `RADIX_BITS`
- `NUM_ELEM`: Anzahl der zu sortierenden Zahlen
- `NUM_GROUPS`: Anzahl der zu bearbeitenden Blöcke, der Wert beträgt aufgrund der Blockgröße `NUM_ELEM/RADIX`

4.2 Implementierung mit SkelCL

Dieser Abschnitt behandelt die Implementierungsdetails der SkelCL-Version des Radixsort. Dabei geht es um die Datenverwaltung, Details der Skelette und um Probleme und Einschränkungen, die bei der Implementierung auftraten. Wichtige Teile des Quellcodes sind in Listing 10 zu sehen.

Datenverwaltung Die Zahlenblöcke sind durch ein `struct` realisiert, das ein Array von `RADIX unsigned int`-Zahlen enthält.

Für die Speicherung der Blöcke werden sieben SkelCL-Vektoren verwendet. Davon enthalten fünf `NUM_GROUPS` Elemente und zwei nur ein einziges. `numbersA` und `numbersB` enthalten abwechselnd die Eingabe und die Ausgabe. Dazu werden sie über die Zeiger `unsortedPtr` und `sortedPtr` angesprochen, die nach jeder Iteration vertauscht werden (Zeile 45-47). `rawHistogram`, `scanedHistogram` und `finalHistogram` speichern die Histogramme und Positionstabellen. `sumBufferVec` und `scanedSumBufferVec` enthalten nur ein Element, nämlich `summaryBufferIn` und `summaryBufferOut`. Diese beiden Blöcke müssen in einen Container verpackt werden, da nur elementare Datentypen ohne Speichermanagement auf das Device übertragen werden können.

Skelette Die Übergabe des zusätzlichen Parameters für die zu betrachtende Stelle in den `Histogram`- und `Permute`-Schritten ist in SkelCL direkt über einen zusätzlichen Parameter bei der Skelettausführung möglich (Zeile 37). Wie in der vorgegebenen Implementierung gibt der Parameter an, um wie viele Bits eine Eingabezahl nach rechts verschoben werden muss, damit die danach niedrigsten Bits die gesuchte Stelle angeben.

Ein Problem bei der Nutzung des Scans im Scan2D-Schritt ist, dass dem Skelett bei der Instanziierung ein String übergeben werden muss, der das neutrale Element des verwendeten Operators darstellt. Das neutrale Element ist hier ein Block, der nur aus Nullen besteht. Da es sich um ein `struct` handelt, kann dieser Block nicht als Literal formuliert werden. Stattdessen wird dem Skelett ein Funktionsaufruf übergeben, dessen Ergebnis der gewünschte Nullblock ist (Zeile 7, `get_zero_block()`). Dies funktioniert, weil der Stringparameter mit dem neutralen Element mit einem Makro in den Kernel eingefügt wird. Der Quellcode der Nullblockfunktion wird im Benutzer-Quellcode des Scan-Kernels der eigentlichen Argumentfunktion vorangestellt, damit die Nullblockfunktion im Skelettkernel verfügbar ist.

Die Berechnung des Gesamthistogramms aus den letzten gescannten und ungescannnten Histogrammen wird durch ein Map realisiert (Zeile 8). Dabei wird auf den `unsigned int`-Vektor `sumBufferIndex` gemappt, dessen einziges Element die Position des jeweils letzten Blocks in `rawHistogram` und `scanedHistogram` angibt, die immer `NUM_GROUPS-1` ist. Die eigentlichen Datenquellen, die Histogrammvektoren, werden als Zusatzargumente übergeben (Zeile 40). Die Argumentfunktion liest dann die beiden Blöcke an der in `sumBufferIndex` angebenen Stelle und gibt deren Summe zurück.

Der nicht an Blöcke gebundene Zugriff auf die Ausgabe bei der Permutation wird möglich, indem der Ausgabevektor als Zusatzargument an ein Zip-Skelett übergeben wird, während die Hauptargumente die unsortierten Zahlen und die Positionstabellen sind (Zeile 43). Dadurch hat die Argumentfunktion (auch schreibenden) Zugriff auf das ganze Array von Blöcken und das eigentliche Funktionsergebnis ist ein Seiteneffekt. Dementsprechend ist der eigentliche Rückgabetyp der Argumentfunktion und des Skeletts `void`. Da der Ausgabevektor eigentlich keine Ausgabe ist, muss er manuell als geändert markiert werden (Zeile 44, `dataOnDeviceModified()`).

Probleme und Einschränkungen Ein wesentliches Problem ist die Basis und Blockgröße `RADIX`, da in den Kerneln Blöcke im privaten Speicher abgelegt werden müssen. Der in der vorgegebenen Implementierung genutzte Wert $R = 256$ bedeutet hier einen Speicherbedarf von 1024 Bytes pro Block, was für GPUs zu viel ist. Dies zeigte sich daran, dass bei Tests ein-

zelne Kernel übersprungen wurden oder das System einfror. Daher muss ein kleinerer Wert gewählt werden, hier wird 8 verwendet, da mit diesem Wert zusammen mit einer Work-Group-Größe von 64 bei Tests auf der GPU die höchste Leistung erzielt wurde, ohne das technische Probleme auftraten. Mit der Blockgröße ändert sich auch die Basis. Eine Basis von 8 bedeutet, dass in jeder Iteration nur 3 Bits der zu sortierenden Zahlen betrachtet werden, daher sind für `unsigned int`-Zahlen mit 32 Bit Länge 11 statt 4 Iterationen nötig.

Ein weitere, kleineres Problem war, dass das Reduce-Skelett im Kernel einen Fehler enthielt, bei dem eine Variable für Zwischenergebnisse nicht vom Typ der zu verarbeitenden Daten, sondern vom Typ `int` war. Dieser Fehler war aber schnell zu finden und trivial zu beheben.

Eine kleinere Einschränkung ist, dass SkelCLs Möglichkeiten, mehrere Devices zur Ausführung bestimmter Skelette zu verwenden, nicht genutzt werden (Zeile 3). Dies liegt daran, dass das Scan-Skelett derzeit nur dann verwendet werden kann, wenn SkelCL nur ein Device nutzt.

4.3 Implementierung mit SkePU

In diesem Abschnitt werden die Details der SkePU-Version des Radixsort behandelt. Dabei geht es wieder zunächst um die verwendeten Datenstrukturen, dann um die Implementierungsdetails und schließlich um Probleme und Einschränkungen des Programms. Ein Auszug des Quelltexts findet sich in Listing 11.

Datenverwaltung Die Datenverwaltung funktioniert sehr ähnlich zur Implementierung in SkelCL. Die Blöcke sind wieder als `struct block` realisiert, das ein `unsigned int`-Array der Größe `RADIX` enthält. Für die sortierten und unsortierten Zahlen werden ebenfalls abwechselnd die beiden SkePU-Vektoren `numbersA` und `numbersB` verwendet (Zeile 49-51), für die Histogramme werden die drei Vektoren `rawHistogram`, `scanedHistogram` und `finalHistogram` genutzt.

Bei den einelementigen Vektoren wegen einer Änderung an `Scan1D` nur `scanedSumBufferVec` benötigt.

Wegen des anderen Ablaufs der Permute-Phase (s.u.) werden zwei weitere Vektoren der Länge `NUM_BLOCKS` (`positionsTarget` und `positionsSource`) sowie eine Matrix `positionsTargetMatrix` mit einer Spalte und `NUM_BLOCKS` Zeilen verwendet.

```
1  int main()
2  {
3      skelcl::init(skelcl::nDevices(1).deviceType(device_type::GPU));
4
5      //Skels erzeugen
6      Map<block(block)> createHistogram(std::ifstream("histo.cl"),
           "histogram");
7      Scan<block(block)> scanHistogram(std::ifstream("add.cl"),
           "get_zero_block()", "add");
8      Map<block(unsigned int)> createSumBuffer(std::ifstream("sumbuffer.cl"),
           "lastSum");
9      Map<block(block)> scanBlock(std::ifstream("scan.cl"), "scanBlock");
10     Map<block(block)> fixOffset(std::ifstream("offset.cl"), "fixOffset");
11     Map<void(block)> permute(std::ifstream("permute.cl"), "permute");
12
13     //Buffer fuellen
14     fillVectorRandom(numbersA, NUM_BLOCKS);
15
16     //Sortieren
17     runSort(createHistogram, scanHistogram, createSumBuffer, scanBlock,
           fixOffset, permute);
18
19     //Probe
20     if(!checkSorted(*unsortedPtr)){
21         cleanup();
22         return 1;
23     }
24
25     cleanup();
26     return 0;
27 }
28
29 void runSort(skelcl::Map<block(block)> &createHistogram,
30              skelcl::Scan<block(block)> &scanHistogram,
31              skelcl::Map<block(unsigned int)> &createSumBuffer,
32              skelcl::Map<block(block)> &scanBlock,
33              skelcl::Map<block(block)> &fixOffset,
34              skelcl::Map<void(block)> &permute)
35 {
36     for(unsigned int iteration=0; iteration<ITERATIONS; iteration++){
37         createHistogram(rawHistogramOut, *unsortedPtr,
               iteration*RADIX_BITS);
38         rawHistogram.dataOnDeviceModified();
39         scanHistogram(scanedHistogramOut, rawHistogram);
40         createSumBuffer(sumBufferVecOut, sumBufferIndex, scanedHistogram,
               rawHistogram);
41         scanBlock(scanedSumBufferVecOut, sumBufferVec);
42         fixOffset(finalHistogramOut, scanedHistogram, scanedSumBufferVec);
43         permute(*unsortedPtr, finalHistogram, *sortedPtr,
               iteration*RADIX_BITS);
44         (*sortedPtr).dataOnDeviceModified();
45         Vector<block> *vecSwap = sortedPtr;
46         sortedPtr = unsortedPtr;
47         unsortedPtr = vecSwap;
48         Out<Vector<block>> *outSwap = sortedOutPtr;
49         sortedOutPtr = unsortedOutPtr;
50         unsortedOutPtr = outSwap;
51     }
52 }
```

Listing 10: Auszug aus der Implementierung des Radixsort mit SkeCL

Skelette Das wichtigste Detail ist die Realisierung der Permute-Phase. Dazu werden drei Skelettfunktionen benötigt, da es in SkePU nur wenige Möglichkeiten gibt, in einer Argumentfunktion frei auf einen Vektor zuzugreifen. Bei allen entsprechenden Kombinationen aus Argumentfunktionstyp und Skelett kann neben dem frei zugreifbaren Vektor nur ein weiterer Vektor oder eine Matrix übergeben werden. Für den normalen Permute-Kernel werden dagegen zusätzlich zu einem als Ausgabe verwendeten frei schreibbaren Vektor zwei Vektoren mit den Positionstabellen und den unsortierten Zahlen benötigt.

Die Lösung für dieses Problem ist, die Zuordnung der Work-Items zu den Zahlenblöcken umzukehren: Ein Work-Item verschiebt nicht die Zahlen eines unsortierten Blocks an beliebige Stellen der Ausgabe, stattdessen füllt es einen Ausgabeblock mit Zahlen aus beliebigen Eingabeblöcken (Zeile 21-30). Dadurch kann in der Funktion `permuteC` der normale Ausgabemechanismus von MapArray genutzt werden (Zeile 35). Die unsortierten Zahlen sind dabei die frei zugreifbare Eingabe, die Positionsinformationen für das Verschieben sind die per Map zugeordnete Eingabe (Zeile 48).

Diese Positionsinformationen geben zu jeder Ausgabeposition die Eingabeposition an, von der das Ergebnis gelesen werden muss, müssen aber erst aus den Bucket-Positionstabellen berechnet werden. Dazu sind zwei weitere Skelettfunktionen nötig, `permuteA` und `permuteB`.

Die erste Funktion `permuteA` berechnet mit einem Zip-Skelett aus den unsortierten Zahlen und den Bucket-Positionstabellen eine Zielpositionsliste, die jeder Eingabeposition die passende Position in der Ausgabe zuordnet (Zeile 45). Dazu werden analog zum normalen `permute`-Kernel zu den Zahlen eines Eingabeblocks in der Bucketpositionstabelle die Zielpositionen abgelesen und inkrementiert (Zeile 1-10).

Die zweite Funktion wandelt die Zielpositionen in Quellpositionen um, wobei jedes Work-Item einen Block liest und an die darin angegebenen Zielpositionen die jeweiligen Quellpositionen schreibt (Zeile 11-20). Dafür wird ein MapArray-Skelett verwendet, dessen gemapptes Argument die Eingabe ist, während das freie Argument als Ausgabe verwendet wird (Zeile 47). Ein Problem ist, dass ein Work-Item die Position des zu verarbeitenden Blocks in der Eingabe kennen muss, um die Quellpositionen angeben zu können. Diese Information erhält der Kernel aber nur, wenn die Eingabe eine Matrix ist, und eine Konvertierung zwischen Vektor und Matrix ist in SkePU nicht vorgesehen. Daher muss der Vektor mit den Zielpositionen zunächst auf dem Host in eine einspaltige Matrix kopiert werden (Zeile 46).

Damit findet die Permute-Phase in vier Schritten statt:

1. In `permuteA` werden aus den unsortierten Zahlen und den Bucket-Posi-

tionstabellen die Zielpositionen `positionsTarget` berechnet. Dafür wird ein Map mit einer `BINARY_FUNC_CONST`-Argumentfunktion verwendet.

2. Die Hostfunktion `vectorToMatrix()` kopiert die Zielpositionen in die Matrix `positionsTargetMatrix`.

3. Das MapArray-Funktion `permuteB` schreibt die Quellpositionen in den Vektor `positionsSource`. Dabei wird eine `ARRAY_FUNC_MATR`-Argumentfunktion verwendet.

4. Das MapArray-Funktion `permuteB` füllt die Ausgabe anhand der Quellpositionen mit den Eingabezahlen. Dafür wird eine `ARRAY_FUNC`-Argumentfunktion verwendet.

Ein weiteres Implementierungsdetail liegt in der Scan1D-Phase: In dieser Phase wird auch die Berechnung von `summaryBufferIn` übernommen. Dazu werden dem Map-Skelett die letzten Elemente von `rawHistogram` und `scanedHistogram` übergeben, wobei SkePUs Möglichkeit genutzt wird, einem Skelett statt einem Container ein Iteratorpaar zu übergeben (Zeile 42).

Probleme und Einschränkungen Diese Implementierung hat eine wesentliche Einschränkung: Die OpenCL- und CUDA-Backends können nicht genutzt werden. Bei OpenCL liegt das Problem in der `block`-Definition. Diese Definition müsste in den Kerneln wiederholt werden, damit der Datentyp verwendet werden kann. SkePU bietet aber anders als SkelCL keine Möglichkeit, solche Definitionen in die Kernel zu übernehmen, eine Wiederholung der Definition in der Argumentfunktion führt zu einem Compilerfehler. Dieser entsteht dadurch, dass die zusätzliche Definition auch in die CPU-Version der Argumentfunktion eingefügt wird, wo sie in einem anderen Namensraum als das Original liegt, sodass die Funktionsargumente den falschen Typ haben. Bei CUDA gab es zunächst einen Fehler im Scan-Skelett: Es wurde mehrmals 0 als neutrales Element verwendet, obwohl ein passendes Init-Element übergeben wurde. Dieser Fehler betrifft übrigens nicht nur die Verwendung von Structs, sondern auch die Nutzung mit einem Operator, dessen neutrales Element nicht 0 ist, z.B. mit der Multiplikation. Aber auch nach der Korrektur dieser Fehler und mit einer kleinen Blockgröße treten Laufzeitfehler bei der CUDA-Benutzung auf.

```
1  BINARY_FUNC_CONSTANT(permutePre, block, unsigned int, unsorted, histo,
       shiftCount,\
2      int i;\
3      block positions;\
4      for(i=0; i<RADIX; i++){\
5          int bucket = (unsorted.data[i] >> shiftCount) & BITMASK;\
6          positions.data[i] = histo.data[bucket]; \
7          histo.data[bucket] = histo.data[bucket]+1;\
8      }\
9      return positions;
10 )
11 ARRAY_FUNC_MATR(addressSwap, block, positionOut, positionIn, xidx, yidx,\
12     int i;\
13     for(i=0; i<RADIX; i++){\
14         unsigned int bidx = positionIn.data[i]/RADIX;\
15         unsigned int eidx = positionIn.data[i]%RADIX;\
16         unsigned int srcAddr = (yidx << RADIX_BITS) + i;\
17         positionOut[bidx].data[eidx] = srcAddr;\
18     }\
19     return positionIn;
20 )
21 ARRAY_FUNC(permuteFinal, block, unsorted, position,\
22     int i;\
23     block sorted;\
24     for(i=0; i<RADIX; i++){\
25         unsigned int bIdx = position.data[i]/RADIX;\
26         unsigned int eIdx = position.data[i]%RADIX;\
27         sorted.data[i] = unsorted[bIdx].data[eIdx];\
28     }\
29     return sorted;
30 )
31
32 int main(){
33     Map<permutePre> permuteA(new permutePre);
34     MapArray<addressSwap> permuteB(new addressSwap);
35     MapArray<permuteFinal> permuteC(new permuteFinal);
36
37     //sortieren
38     for(unsigned int iteration=0; iteration < ITERATIONS; iteration++){
39         createHistogram.setConstant(iteration*RADIX_BITS);
40         createHistogram(*unsortedPtr, rawHistogram);
41         scanHistogram(rawHistogram, scanedHistogram, EXCLUSIVE);
42         scanSumBuffer(scanedHistogram.end()-1, scanedHistogram.end(),
               rawHistogram.end()-1, rawHistogram.end(),
               scanedSumBufferVec.begin());
43         fixOffset(scanedSumBufferVec, scanedHistogram, finalHistogram);
44         permuteA.setConstant(iteration*RADIX_BITS);
45         permuteA(*unsortedPtr, finalHistogram, positionsTarget);
46         vectorToMatrix(positionsTargetMatrix, positionsTarget);
47         permuteB(positionsSource, positionsTargetMatrix,
               positionsTargetMatrix);
48         permuteC(*unsortedPtr, positionsSource, *sortedPtr);
49         Vector<block> *swap = unsortedPtr;
50         unsortedPtr = sortedPtr;
51         sortedPtr = swap;
52     }
53     return 0;
54 }
```

Listing 11: Auszug aus der Implementierung des Radixsort mit SkePU

5 Auswertung

In diesem Kapitel werden SkelCL und SkePU anhand der Ergebnisse bei der Implementierung des Radixsort evaluiert und verglichen. Dabei wird zunächst die Performance der Implementierungen mit SkelCL, SkePU und OpenCL ermittelt und verglichen. Danach wird die Ausdrucksfähigkeit und Nutzbarkeit der beiden Bibliotheken verglichen.

5.1 Geschwindigkeit

Dieser Abschnitt beschäftigt sich mit der Geschwindigkeit der drei vorliegenden Implementierungen des Radixsort.

Es werden 8 verschiedene Eingabegrößen getestet, für jede Eingabegröße werden 10 Messungen durchgeführt und ein Mittelwert gebildet. Dabei wird nur die für die Ausführung der Skelette und für ggf. anfallende Speichertransfers benötigte Zeit gemessen. Die Genauigkeit der Zeitmessung beträgt maximal 1ms. Die zu sortierenden Zahlen werden vom jeweiligen Programm zufällig erzeugt.

Die 8 Eingabegrößen sind in zwei Gruppen der Form $\{k, 2k, 3k, 4k\}$ aufgeteilt:

1. 2^{14} (16K), 2^{15} (32K), $3 \cdot 2^{14}$ (48K) und 2^{16} (64K) wurden ausgewählt, da die Implementierung von AMD maximal 2^{16} Elemente unterstützt.

2. 2^{22} (4M), 2^{23} (8M), $3 \cdot 2^{22}$ (12M) und 2^{24} (16M) wurden ausgewählt, weil die SkelCL-Implementierung auf dem für die Tests verwendeten System 2^{25} Elemente mit der GPU wegen eines Ressourcenmangels nicht verarbeiten kann.

Der Sinn der Struktur $\{k, 2k, 3k, 4k\}$ ist, das Skalierungsverhalten der Implementierungen innerhalb eines Größenbereichs zu messen, und mögliche Effekte beim Erreichen der maximalen Problemgröße zu erkennen.

Das Testsystem verfügt über eine Intel i7-860 CPU (4 physische Cores mit zweifachem SMP, 2,8 GHz), 4GB Arbeitsspeicher (davon ca. 3GB frei) und eine AMD Radeon HD 5970 Grafikkarte (zwei GPUs mit jeweils 1GB Speicher). Bei den Geschwindigkeitsmessungen muss kein Auslagerungsspeicher verwendet werden, von den GPUs wird in SkelCL und mutmaßlich auch bei der AMD-Implementierung nur eine verwendet. Die verwendete OpenCL-Implementierung stammt von AMD.

Die Tuningparameter sind bei SkelCL wie schon in Abschnitt 4.2 beschrieben eine Blockgröße von 8 und eine Work-Group-Größe von 64. Bei SkePU ist

die Blockgröße 256; der Wert wurde von der AMD-Implementierung übernommen. Kleinere und größere Werte wären dabei ebenfalls möglich: Eine kleinerer Blockgröße verlangsamt das Programm, da mehr Iterationen nötig sind, erlaubt aber eine feinere Wahl der Eingabegröße, eine größere Blockgröße hat den gegenteiligen Effekt. Bei den vier größeren Eingaben wird die SkePU-Version zusätzlich mit einer Blockgröße von 8 getestet, um eine bessere Vergleichbarkeit mit der SkelCL-Version zu erreichen. Denn wenn das OpenCL-Backend in SkePU verwendet werden könnte, müsste vermutlich eine ähnliche Blockgröße gewählt werden.

Ergebnisse mit kleineren Eingaben Bei den kleineren Problemgrößen werden die AMD- und SkelCL-Implementierung jeweils auf der CPU und einer GPU getestet, bei der SkePU-Implementierung wird das unparallelisierte und das OpenMP-Backend verwendet. Die Ergebnisse sind in den Abbildungen 15 und 16 zu sehen.

Hierbei fallen die Ergebnisse mit zwei Programmen besonders auf: Einerseits ist die unparallelisierte Implementierung in SkePU mit einem deutlichen Vorsprung die insgesamt schnellste. Folglich können Parallelisierungsvorteile den jeweiligen Overhead bei diesen Problemgrößen noch nicht ausgleichen. Andererseits ist die Implementierung in SkelCL auf der GPU um ein Mehrfaches langsamer als alle anderen Versionen, wobei die Ausführungszeiten bei allen vier Problemgrößen ähnlich nahe beieinanderliegen wie bei der AMD-Version auf der GPU. Folglich handelt es sich vermutlich um einen eingabeunabhängigen Overhead.

Weiterhin sind die drei parallel auf der CPU ausgeführten Versionen ab einer Eingabegröße von 32K ähnlich schnell, während die AMD-Implementierung auf der GPU deutlich schneller ist, aber ebenso deutlich hinter der sequentiellen SkePU-Implementierung zurückbleibt.

Außerdem fallen kleinere Unregelmäßigkeiten bei den verschiedenen Zeiten einer Programmversion auf, nämlich die ungewöhlich hohe Zeit bei SkePU mit OpenMP und 16K Elementen sowie die teilweise mit steigender Eingabegröße leicht fallende Ausführungszeiten. Der Grund für die hohe Zeit bei SkePU mit OpenMP und 16K Elementen ist unbekannt; die fallenden Zeiten können zufällige Schwankungen sein, da die Messwerte innerhalb der Testreihen sehr unterschiedlich sind. Ein Beispiel ist die Messreihe für die AMD-Implementierung auf der CPU mit 64K Elementen, bei der die gemessenen Zeiten zwischen 9ms und 35ms liegen.

Ergebnisse mit größeren Eingaben Bei den größeren Testdatenmengen kann die Implementierung von AMD nicht verwendet werden. Die SkePU-

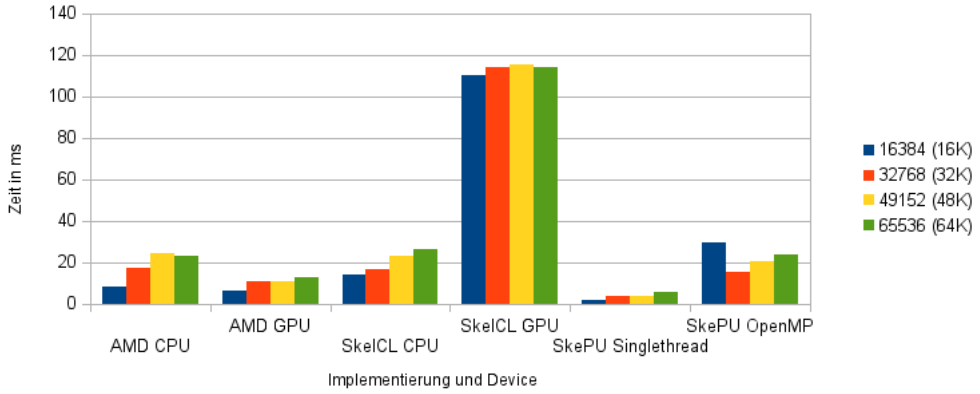


Abbildung 15: Leistungsmessung: Benötigte Zeit für kleinere Problemgrößen

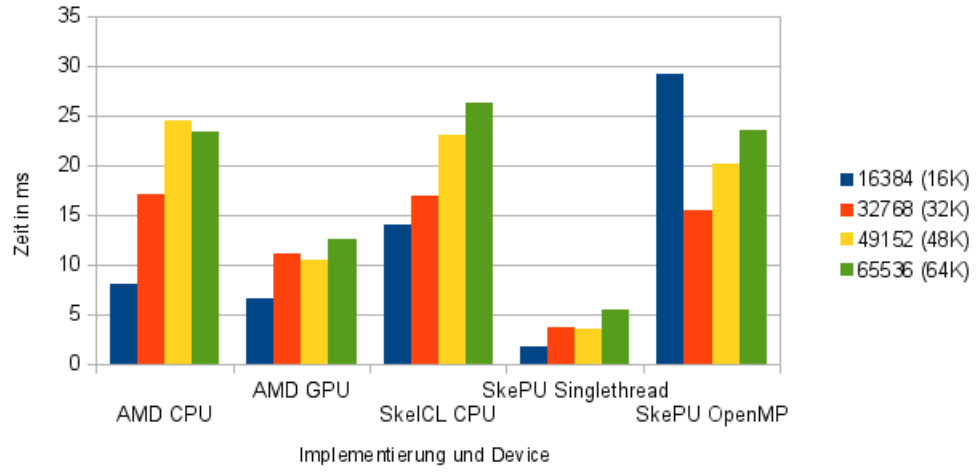


Abbildung 16: Leistungsmessung: Benötigte Zeit für kleinere Problemgrößen, für eine bessere Ablesbarkeit ohne die SkelCL-GPU-Ergebnisse

Implementierung wird wie zu Anfang dieses Abschnitts erklärt mit zwei verschiedenen Blockgrößen getestet. Die Ergebnisse der Tests sind in Abbildung 17 zu sehen.

Das deutlichste Ergebnis ist der Performanceunterschied zwischen den SkePU-Versionen mit kleinen und großen Blöcken: In der sequentiellen Variante bedeuten die größeren Blöcke eine etwa doppelt so hohe Geschwindigkeit, bei der OpenMP-Variante steigt zusätzlich der Beschleunigungsfaktor gegenüber der sequentiellen Version deutlich.

Bei den Eingabegrößen bis 12M bildet die Reihenfolge ungefähr die theoretischen Leistungsunterschiede der Hardware ab, allerdings nicht sehr deutlich: SkePU ist mit OpenMP (d.h. auf allen CPU-Cores) schneller als in der sequentiellen Version (d.h. auf einem CPU-Core), aber langsamer als SkelCL auf der GPU. SkelCL ist auf der CPU dagegen unerwarteterweise auch bei diesen großen Eingabemengen langsamer als die sequentielle SkePU-Version. Da der Unterschied zu SkePU mit OpenMP trotzdem nicht sehr groß ist, kann die Ursache im größeren Overhead bei der Nutzung von OpenCL liegen.

Ansonsten fällt auf, dass die Ausführungszeit bei diesen Eingabegrößen anders als bei den kleineren Eingaben annähernd proportional zur Eingabegröße ist. Dies ist damit zu erklären, dass der eingabeunabhängige Overhead der Parallelisierungstechniken und zufällige Schwankungen bei diesen Ausführungszeiten nicht mehr ins Gewicht fallen. Zusätzlich ist durch die große Anzahl Blöcke sichergestellt, dass die Hardware die meiste Zeit über ausgelastet ist, sodass der lineare Aufwand für den Algorithmus zu einer linearen Laufzeit führt. Die einzige Abweichung von den linearen Mustern tritt bei SkelCL auf der GPU mit 16M Zahlen auf. Der Grund für diese deutliche Verlangsamung ist vermutlich, dass die fast ausgereizten Ressourcen den Overhead in irgendeiner Weise erhöhen, z.B. weil Daten der Benutzeroberfläche in den Hauptspeicher ausgelagert werden müssen.

Fazit Die Leistungsmessungen führen zu verschiedenen Ergebnissen. Zum Einen hat die SkelCL-Implementierung einen ungewöhnlich großen Overhead bei der Verwendung der GPU, kann in anderen Fällen aber mit dem Vorbild von AMD mithalten und fällt bei größeren Eingaben unter Betrachtung der Blockgröße nicht allzu weit hinter die SkePU-Variante zurück. Zum anderen kann die Effizienz der SkePU-Implementierung im Vergleich zum Vorbild aufgrund der unterschiedlichen Parallelisierungstechniken kaum beurteilt werden, aber die Nutzung von OpenMP ist ab einer gewissen Problemgröße recht effektiv, auch wenn der Beschleunigungsfaktor deutlich unter dem mit vier Cores theoretisch möglichen Wert von 4 bleibt. Jedenfalls ist OpenMP die effektivste Parallelisierungstechnik für den Radixsort.

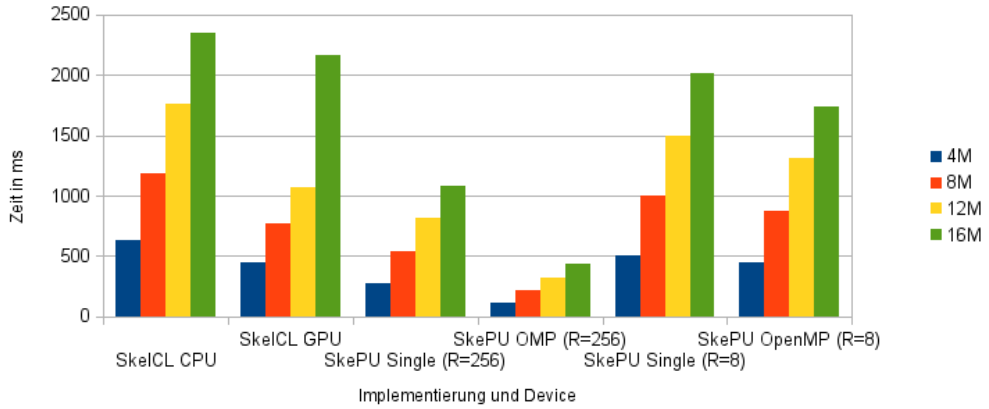


Abbildung 17: Leistungsmessung: Benötigte Zeit für größere Problemgrößen, R ist die Blockgröße

5.2 Ausdrucksfähigkeit und Nutzbarkeit

Dieser Abschnitt vergleicht, wie vielseitig und wie leicht sich SkelCL und SkePU verwenden lassen.

Über beide Bibliotheken lässt sich sagen, dass sie deutlich einfacher zu benutzen sind als OpenCL und viele datenparallele Algorithmen abbilden können. Dies ist auch an den deutlich kürzeren Quelltexten sichtbar. Der Vorteil gegenüber OpenCL liegt dabei vor allem in der Speicherabstraktion und in der automatischen Verwaltung des OpenCL-Kontexts. Die Skelette an sich sind auch nützlich, könnten aber auch ohne die Biliotheken als Kernelvorlagen für die manuelle OpenCL-Nutzung veröffentlicht werden. Außerdem ist häufig kein Wissen über die Parallelisierungstechnik im Backend notwendig, aber es kann bei Problemen leicht nötig werden, z.B. wenn manuelle Speichertransfers ausgelöst werden. Im Detail gib es aber Unterschiede:

SkePU SkePU ist mit den unterschiedlichen Backends bei der Hardwarenutzung flexibler, sofern alle Backends mit dem betrachteten Programm funktionieren. Zum Beispiel kann das OpenMP-Backend gewählt werden, wenn eine Anwendung auf der GPU ineffektiv ist oder wenn beim Anwender keine OpenCL- oder CUDA-Unterstützung vorrausgesetzt werden soll. Die Vorteile dieser Möglichkeit wurden bei den Performancevergleichen im vorherigen Abschnitt deutlich sichtbar. Dies wäre noch effektiver, wenn das Backend zur Laufzeit gewählt und gewechselt werden könnte. Weiterhin ist das MapOverlap-Skelett (siehe Abschnitt 3.2) für viele Bildbearbeitungsalgorithmen nützlich. Ein kleiner Zusatznutzen der Argumentfunktionsdefinition mit Makros ist, dass in IDEs und Texteditoren in den Argumentfunktionen Syntaxhervorhebung und ggf. Autovervollständigung zur Verfügung stehen (getestet mit QtCreator (IDE), Kate (GUI-Texteditor) und Nano (Konsolen-Texteditor)).

Der große Nachteil von SkePU ist die eingeschränkte Nutzung von frei zugreifbarem Speicher in den Argumentfunktionen, wie am Beispiel der Permute-Phase des Radixsort sichtbar wird (siehe Abschnitt 4.3). Die Probleme bei der Nutzung von Structs sind zwar eine noch stärkere Einschränkung, da davon z.B. auch komplexe Zahlen und geometrische Vektoren betroffen sind. Diese Probleme liessen sich aber in einer zukünftigen Version durch einen Definitionsübertragungsmechanismus leicht beheben. Eine weitere Einschränkung ist die Festlegung, dass die Argumente und das Ergebnis einer Argumentfunktion gleich sein müssen.

Durch diese beiden Einschränkungen kann es leicht nötig sein, rechnerische Umwege zu machen (z.B. Permute-Phase) oder Funktionen sequentiell auszuführen. Insbesondere zweiteres kann die Nutzung von GPUs ineffektiv

machen, da schon eine nicht durch ein Skelett realisierte Funktion in einem Algorithmus aufwendige Datentransfers zwischen Hauptspeicher und GPU auslösen kann.

SkelCL In SkelCL können die Skelette deutlich flexibler genutzt werden, was vor allem an der exklusiven Nutzung von OpenCL als Backend liegt. Sofern man entsprechende OpenCL-Kenntnisse besitzt, können dadurch vor allem im Map- und Zip-Skelett OpenCL-Features wie Work-Groups mit lokalem Speicher und Synchronisationsmechanismen sowie das Abfragen der Work-Item-ID genutzt werden. Zusammen mit der Möglichkeit, in einer Argumentfunktion freien Zugriff auf beliebig viele Arrays und Einzeldaten zu erhalten, lassen sich beliebige OpenCL-Kernel konstruieren. Dies ist möglich, indem man ein Map auf ein Array von Dummy-Elementen oder Work-Item-IDs durchführt und die eigentlichen Argumente und Ausgabepuffer als Zusatzargumente übergibt. Dazu muss nur ggf. ein gewisser Speicher-Overhead für die Dummy-Elemente in Kauf genommen werden.

Auch ohne solche extremen Zweckentfremdungen der Skelette lassen sich flexiblere Parallelisierungen erzielen, z.B. könnte die Permute-Phase des Radixsort auf Zahlenebene parallelisiert werden, indem ein Map auf die Eingabezalen durchgeführt wird, bei dem es Work-Groups der Größe `RADIX` gibt, in denen jeweils ein Histogramm im lokalen Speicher berechnet wird.

Zusätzlich kann ausgenutzt werden, dass die Argumentfunktionen sehr direkt in Kernel übersetzt werden. Dadurch können Probleme umgangen werden, wie an der Detaillösung für das neutrale Element des Scans in Abschnitt 4.2 zu sehen ist.

Weiterhin kann es Anwendungen geben, bei denen es praktisch ist, dass die Argumentfunktions-Quelltexte erst zur Laufzeit geladen werden. Dadurch werden z.B. benutzerdefinierte Filter für eine Bildbearbeitung möglich. Die Bedeutung dieser Möglichkeit wird allerdings dadurch relativiert, dass beide Bibliotheken auch unter der GPL-Lizenz verfügbar sind, sodass zusätzliche Argumentfunktionen auch in den Quelltext eingefügt werden können.

Der wesentliche Nachteil bei der Nutzung von SkelCL im Vergleich zu SkePU ist die exklusive Nutzung von OpenCL, weil OpenCL je nach Anwendung weniger effektiv ist als andere Parallelisierungstechnologien. Weiterhin ist das MapOverlap-Skelett auf zweidimensionale Daten beschränkt und noch nicht verfügbar. Auch SkePUs Generate-Skelett hat in SkelCL keine direkte Entsprechung. Beide Skelette könnten zwar mit einem Map und Zusatzargumenten nachgebildet werden, aber die Overlap-Zugriffe wären wie in [4] beschrieben relativ kompliziert. Das Generate-Skelett wäre einfacher nachzubilden, aber die Verwendung eines speziellen Skeletts macht ein Programm

verständlicher.

Dieser Teil der Bewertung lässt sich folgendermaßen zusammenfassen: Beide Bibliotheken vereinfachen die Nutzung der zugrundeliegenden Parallelisierungstechnologien deutlich und erfordern zumindest in den einfachen Anwendungsfällen keine Kenntnisse dieser Technologien. Dabei bringt SkePU zusätzlich Flexibilität bei der Wahl der Technologie, ist aber bei den realisierbaren Funktionen im wesentlichen auf die Grundformen der Skelette beschränkt. SkelCL ist bei den Funktionen sehr flexibel, ist aber auf die Nutzung von OpenCL festgelegt und erfordert für erweiterte Anwendungen wie Zusatzargumente OpenCL-Kenntnisse.

6 Fazit

Insgesamt gilt für SkelCL und SkePU, dass sie das parallele Programmieren deutlich vereinfachen, insbesondere sind in einfachen Fällen häufig keine Kenntnisse in speziellen Parallelisierungsmethoden notwendig.

SkelCL ist sehr vielseitig einsetzbar, sofern man über die nötigen OpenCL-Kenntnisse verfügt. Die Performance ist dabei, soweit das mit dem Beispiel des Radixsort zu beurteilen ist, unterschiedlich: In manchen Fällen ist die Geschwindigkeit mit normalem OpenCL vergleichbar, in manchen Fällen gibt es einen extremen Overhead.

SkePU kann unter Umständen eine höhere Leistung erreichen, indem das Backend gewechselt wird. Bei der Programmierung hingegen ist die Flexibilität geringer, sodass Funktionen unter Umständen nicht parallelisiert werden können, was die Leistung aufgrund von Datentransfers über die Funktion hinaus verringern kann.

Bei beiden Bibliotheken zeigte sich, dass sie bisher zumindest teilweise nicht mit zusammengesetzten Daten getestet wurden. Während es bei SkelCL nur den behebbaren Fehler in einem Kernel gab, ist SkePU mit Datenstrukturen nur eingeschränkt nutzbar. Immerhin ließe sich diese fehlende Funktionalität vermutlich leicht ergänzen, sodass man auf eine Implementierung in einer zukünftigen Version hoffen kann.

Literatur

[1] M. Steuwer, P. Kegel, und S. Gorlatch, *SkelCL - A Portable Skeleton Library for High-Level GPU Programming.* 2011 IEEE International Symposium on Parallel and Distributed Processing Workshop and Phd Forum (IPDPSW), Anchorage, USA, 2011.

[2] M. Steuwer, P. Kegel, und S. Gorlatch, *Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library.* 2012 IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW), Shanghai, China, 2012.

[3] Michel Steuwer, Malte Friese, Sebastian Albers und Sergei Gorlatch *Introducing and Implementing the Allpairs Skeleton for GPU Systems.* International Journal of Parallel Programming, (noch nicht erschienen)

[4] Michel Steuwer, Sergei Gorlatch, Matthias Buß, und Stefan Breuer, *Using the SkelCL Library for High-Level GPU Programming of 2D Applications.* Euro-Par 2012: Parallel Processing Workshops, S. 370-380 Springer 2012

[5] Johan Enmyren und Christoph W. Kessler, *SkePU: A multi-backend skeleton programming library for multi-GPU systems.* Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, Maryland, USA. ACM, 2010.

[6] Usman Dastgeer, *Skeleton Programming for Heterogeneous GPU-based Systems.* Licentiate thesis. Thesis No 1504. Department of Computer and Information Science, Linköping University, 2011.

[7] OpenCL-Beispiele, AMD Accelerated Parallel Processing (APP) SDK, Version 2.8.1, AMD, 2013. Webseite: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/

[8] *The OpenCL Specification*, Version 1.2 Khronos Group, 2012. Quelle: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[9] CUDA Toolkit, Nvidia, https://developer.nvidia.com/cuda-toolkit

[10] *OpenMP Application Program Interface*, Version 3.1, OpenMP Architecture Review Board, 2011. Quelle: http://www.openmp.org/mp-documents/OpenMP3.1.pdf