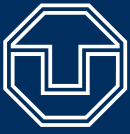


TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Datenbanken

OPTIMIERUNG VON DATENBANKALGORITHMEN FÜR HETEROGENE HARDWARE

Johannes Fett
Matr.-Nr.: 3482638

Betreut durch:
Prof. Dr.-Ing. Wolfgang Lehner

und:
Dipl. -Inf. Tomas Karnagel

Eingereicht am 7.4.2016

ERKLÄRUNG

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, 7.4.2016

ZUSAMMENFASSUNG

In der folgenden Arbeit werden die drei in OpenCL vorliegenden Datenbank-Operatoren Selection, GroupBy und Hashjoin für heterogene Hardware und OpenCL optimiert. Die drei Operatoren wurden aus dem Ocelot-Plugin für MonetDB entnommen. Es erfolgt eine Evaluation der Optimierungsstrategien der drei Operatoren. Bei der Optimierung der Operatoren konnte ein Speedup von bis zu 13,5 erzielt werden. Die Ausführungszeit aller untersuchten Operatoren ist durch die Optimierungen deutlich gesunken. Dabei wurden das Verhalten der Operatoren auf einem Prozessor, einer integrierter Grafikeinheit und einer dedizierten Grafikkarte untersucht. Aus den Optimierungen der Operatoren werden generelle Strategien zur Performanceoptimierung von OpenCL-Programmen entwickelt.

ABSTRACT

This work deals with optimizing three database operators, namely Selection, GroupBy and Hashjoin, all of which are implemented in OpenCL. The operators have been taken from the Ocelot plugin for MonetDB. Additionally, the performance of the three operators has been evaluated. The optimization of the operators yielded speedups of up to 13,5. The execution time of all examined operators could be significantly reduced by the optimizations. Within the scope of performance optimization, a central processing unit, an integrated graphics unit and a dedicated graphics card have been examined. General strategies for performance optimization of OpenCL programms are being developed through optimizing the three operators.

INHALTSVERZEICHNIS

1 Einleitung **9**

2 Grundlagen **11**

2.1 OpenCl 11

2.1.1 Speichermodell 11

2.1.2 Kernel-code 12

2.1.3 Host-code 13

2.1.4 Testumgebung 15

2.2 Untersuchte Datenbank-operatoren 16

2.2.1 Selection 18

2.2.2 Join 19

2.2.3 GroupBy 21

3 Optimierungen **25**

3.1 Selection 25

3.1.1 Overflow-Kernel 25

3.1.2 Local work size und global work size 27

3.1.3 Bewertung 28

3.2 Join 28

3.2.1 PreprocessHTKernel . . . 28
3.2.2 Local Memory Caching . . . 29
3.2.3 Local work size und global work size . . . 29
3.2.4 Bewertung . . . 30
3.3 GroupBy . . . 31
3.3.1 Local work size und global work size . . . 31
3.3.2 Overflow Hostcode . . . 31
3.3.3 Overflow Kernel . . . 32
3.3.4 Bewertung . . . 33

4 Ergebnisse **35**
4.1 OpenCL best practices . . . 35
4.1.1 Threads Auffüllen . . . 35
4.1.2 Vektorisierung . . . 36
4.1.3 Speicherzugriff . . . 36
4.1.4 Speicherarten . . . 37
4.1.5 Global Memory Caching . . . 37
4.1.6 Local work size . . . 37
4.1.7 Kerneldesign . . . 38
4.2 Probleme mit OpenCL und Tools . . . 38

5 Verwandte Arbeiten **41**

6 Ausblick **43**

A Anhang **47**
A.1 Inhalt des Datenträgers . . . 47
A.2 Datenbank-operator Beispielgraph . . . 47

1 EINLEITUNG

Datenbank-Operatoren aktueller Datenbank-Systeme müssen für die Parallelisierung der Datenbank-Operatoren die jeweilige Ziel-Architektur berücksichtigen. Um eine Entwicklung von parallelisierten Datenbank-Operatoren zu ermöglichen, die unabhängiger von den dabei verwendeten Hardware-Architekturen ist, wurde Ocelot entwickelt [Heimel et al., 2013].

Ocelot ist ein Plugin für die Opensource-Datenbank MonetDB. Dabei besteht Ocelot aus einer Menge von Datenbank-Operatoren, die mit mithilfe von OpenCL berechnet werden und den hierfür notwendigen Verwaltungsoperationen. Mithilfe von OpenCL ist es möglich Datenbank-Operatoren mit identischem Quellcode auf verschiedenen Devices auszuführen. Die Operatoren von Ocelot sind dabei nicht für konkrete Hardware entworfen und optimiert, da OpenCL zur Laufzeit des Plugins die Operatoren für die gewünschte Ausführungseinheit, auch Device genannt, kompiliert.

In der Folgenden Arbeit werden die drei Datenbank-Operatoren Hashjoin, GroupBy und Selection betrachtet. Die Operatoren werden hinsichtlich möglicher Optimierungen analysiert und diese anschließend umgesetzt und evaluiert. Aus den Optimierungen der einzelnen Operatoren werden allgemeine Strategien zur Optimierung von OpenCL-Programmen abgeleitet Die drei Operatoren wurden ausgewählt, da sie zu den am häufigsten verwendeten gehören und einen hohen Anteil an der Ausführungszeit von Datenbank-Anfragen haben.

Zunächst wird ein Überblick über die Funktionsweise der in OpenCL vorliegenden Operatoren und OpenCL gegeben. Dabei wird auf die Funktion des Operators innerhalb eines Datenbanksystems eingegangen, sowie auf die konkrete Funktionsweise des Operators in OpenCL.

Aus der Analyse der drei Operatoren ergeben sich Ansätze für Optimierungsstrategien. Diese werden entwickelt und an den gegeben Operatoren evaluiert. Die Optimierungen erfolgen lösgelöst vom MonetDB-System mit Zufallszahlen als Eingabedaten.

Ein Problem dabei ist die hohe Komplexität der Funktionen eines Operators, die gemeinsame Datenabhängigkeiten haben. Es wird versucht Optimierungen zu finden, für die möglichst geringe Modifikationen der Operatoren notwendig sind. Dadurch kann die Ausführungszeit komplexer Operatoren verbessert werden, ohne diese erheblich zu überarbeiten. Dabei wird darauf geach-

tet, ungültige Speicherzugriffe zu vermeiden und reproduzierbare, stabile Ergebnisse auf allen getesteten Devices zu erzielen. Ebenso wird das Verhalten von verschiedenen Devices bezüglich der vorgenommen Optimierungen getestet.

Aus diesen Vorarbeiten entehen optimierte Varianten der drei Datenbank-Operatoren. Diese werden bezüglich ihrer Ausführungszeit und Stabilität evaluiert. Die Evaluation erfolgt auf drei verschiedenen Devices. Es werden die einzelnen Optimierungen und der kummulative Effekt aller Optimierungen je Operator betrachtet. Als mächtigste Optimierungsstrategie erweist sich hierbei das Auffüllen der von OpenCL verwendeten Thread-Gruppen (local work size). Dabei müssen Änderungen am Operator vorgenommen werden, um ungültige Speicherzugriffe und verfälschte Berechnungsergebnisse zu vermeiden. Alle untersuchten Datenbank-Operatoren können so beschleunigt werden. Das höchste Speedup erzielt dabei der GroupBy-Operator mit einem Faktor von 13,5.

Zusätzlich erfolgt eine Einschätzung der verwendeten Softwaretools, sowie eine Betrachtung verwandter Arbeiten.

2 GRUNDLAGEN

2.1 OPENCL

OpenCL ist eine offen standardisierte Programmiersprache für Plattformunabhängiges paralleles Rechnen auf verschiedenen Devices, die von dem Industriekonsortium Khronos Group verwaltet und entwickelt wird. Ein Device (*cl_device*) ist dabei ein Hardwarebaustein, auf dem ein OpenCL Kernel ausgeführt werden kann.

OpenCL Code teilt sich in einen Kernelcode, der die gewünschte Berechnung auf einem Device ausführt, und einen Hostcode, der die Ausführung verwaltet. Zur Verwaltung werden ein oder mehrere Devices einer Platform zugeordnet. Auf einem Device befinden sich eine oder mehrere Berechnungseinheiten, auf denen der Kernel ausgeführt wird. Zum Beispiel beinhaltet eine Platform einer CPU der auch über eine integrierte Grafikeinheit verfügt, die integrierte Grafikeinheit und die CPU.

Die Menge aller ausgeführten Threads wird in OpenCL durch *work groups* in Teilmengen partitioniert. Dabei enthält eine *work group* eine Menge von Threads, die work item genannt werden. Eine *work group* wird je einer Berechungseinheit fest zugewiesen. Dabei kann eine Berechungseinheit mehrere work groups parallel zugewiesen bekommen. Es ist möglich, *work items* mit IDs in mehreren Dimensionen zu definieren (Zum Beispiel x,y,z).

Das im Kernel verwendete OpenCL C basiert auf ISO C99 und wurde für paralleles Rechnen modifiziert. Diese Arbeit bezieht sich auf OpenCL 1.2. Die Funktionen werden in den offiziellen API Dokumenten beschrieben. [Khronos, 2016a]

2.1.1 Speichermodell

OpenCL unterscheidet auf Device-Seite zwischen vier Speicherarten.

- Global Memory

- Constant Memory
- Local Memory
- Private Memory

Global Memory bezeichnet dabei Speicher auf den jedes *work item* unabhängig von seiner *work group* und zugewiesenen Berechnungseinheit lesend und schreibend zugreifen kann. Dieser Speicher ist am größten, hat jedoch die höchste Latenz. Hostseitig ist voller Zugriff erlaubt. Der Device-Treiber kann Teile des *Global Memory* Cachen um die Perfomance des OpenCL-Programms zu verbessern [Khronos, 2016b].

Constant Memory ist Speicher, auf den vom Device nur lesend zugegriffen werden kann. Dabei handelt es sich wie bei *Global Memory* um Speicher, den alle *work items* benutzen können.

Private Memory ist Speicher, der nur von jeweils einzelnen *work items* benutzt werden kann. Der Host hat keinen Zugriff darauf. Die Maximale Größe sieht man durch CLINFO-Befehl.

Local Memory ist geteilter Speicher für eine work group. Damit ist dieser Speicher an eine konkrete Berechnungseinheit gebunden, im Gegensatz zum *Global Memory* und *Constant Memory*. Der Zweck von *Local Memory* ist die Nutzung gemeinsamer Variablen innerhalb einer *work group* und um Kommunikation in einer *work group* zu ermöglichen.

Welche Speichertechnologien konkret für die vier Speichertypen von OpenCL verwendet werden, ist Device-abhängig.

Eine OpenCL-fähige Nvidia GPU verwendet für *Private Memory* und *Constant Memory* Registerspeicher, der sich auf dem Chip der Berechnungseinheit befindet. Der Zugriff auf Registerspeicher erfolgt im selben Taktzyklus je Instruktion [Nvidia, 2009, p. 33].

Die konkrete Menge Speicher, die pro Speicherart verfügbar ist, unterscheidet sich je nach Device.

Für die verwendeten Devices gelten folgende gemessene Speichergrößen[1] [2]:

Tabelle 2.1: Speichergröße

Speichermenge	CPU	GPU	IGPU
Local Memory [KiByte]	32	48	64
Global Memory [MiByte]	2047,87	2048	1297,6

2.1.2 Kernel-code

Im Kernel findet die Berechnung statt, der Kernelcode ist an C99 angelehnt. Eine .cl Kerneldatei kann mehrere Kernelfunktionen enthalten. Einer Kernelfunktion werden eine Menge von Parametern übergeben. Der Zugriff auf Speicherlemente wird mithilfe von globalId, localID geregelt.

[1] bestimmt mit clGetDeviceInfo(..,CL_DEVICE_LOCAL_MEM_SIZE,..)
[2] bestimmt mit clGetDeviceInfo(..,CL_DEVICE_GLOBAL_MEM_SIZE,..)

Listing 2.1: Kernel Beispiel

```
__kernel void example( __global int* buf, __global int* buf2 ){
        int x = get_global_id(0);

        buf2[x] = buf[x];

}
```

Der Kernelcode wird parallel auf dem ausgewählten Device ausgeführt. Als Parameter dienen im Beispiel 2.1 zwei Integer Arrays buf und buf2. Mithilfe der Funktion get_global_id(uint dimindx) erhält der Thread seine konkrete ID in der jeweiligen Dimension. Für dieses Beispiel wird angenommen, dass alle Threads ausschließlich über die X-Dimension aufgereiht sind. Daher liefert der Aufruf get_global_id(0) den globalen Index des Threads in X-Richtung.

Die globale ID wird dabei von 0 bis global_work_size -1 durchnummeriert. Im angegebenen Beispiel überschreibt ein Thread jeweils ein Element von buf2 durch ein Element von buf. Dabei ergibt sich das Offset zum Beschreiben der Arrays aus der fortlaufenden ThreadID. Der Thread mit einem Index i überschreibt buf2[i] mit buf[i].

Wobei gilt: $0 <= i <= global_work_size - 1$

In diesem Beispiel bestehen keine Abhängigkeiten zwischen den ausgeführten Threads, so dass diese ohne Beachtung von Synchronizität parallel ausgeführt werden können. Die parallele Berechnung der Kernelfunktion auf unabhängigen Daten entspricht somit der flynnschen Kategorie SIMD (Single Instruction, Multiple Data).

2.1.3 Host-code

Der Hostcode verwaltet die Ausführung einer Menge von Kerneln, die dafür notwendigen Speichertransfers und die Anbindung an ein OpenCL-fähiges Device. Im Folgenden werden alle notwendigen Schritte auf Hostseite aufgeführt um einen Kernel zu verwalten, auszuführen und die berechneten Ergebnisse zu benutzen. Für detailierte Informationen bezüglich Parameter und Rückgabewerte siehe [Khronos, 2016a].

Device auswählen und cl_context erstellen

Mit der Auswahl von *Platform* und *Device* wird ein *cl_context* erstellt, der zur Verwaltung der Ausführung von Kerneln auf dem gewählten Device dient.

Die Funktion *clGetPlatformIDs* gibt die konkreten IDs aller vorhandenen *Platforms* und deren OpenCL Versionen zurück. Dabei sind Mehrfachnennungen möglich, falls die *Platform* OpenCL 1.2, sowie OpenCL 2.0 untersützt.

Mit einer *cl_platform_id* und der Funktion *clGetDeviceIDs* wird eine Menge von *Devices* auf dieser *Platform* zurückgegeben. Eine Intel CPU Platform mit integrierter Grafik enthält ein CPU

und eine GPU. Dies kann unter anderem eingeschränkt werden auf CPU oder GPU *Devices* unter Angabe eines *cl_device_type*.

Der konkrete *cl_context* zur Verwaltung eines ausgewählten Devices wird mit der Funktion *clCreateContext* erstellt. Dabei werden Platform und Device angegeben, so dass der *cl_context* für ein konkretes Device auf exakt einer Platform gilt. Die verwendete OpenCL Version wird durch die Auswahl der *Platform* festgelegt.

Der *cl_context* dient als zentraler Verbindungspunkt zur Verwaltung des *Devices*. Es ist möglich, mehrere *cl_context* Entitäten zu benutzen, Speicherzugriffe sind jedoch nicht übergreifend möglich.

Auf dem vom *cl_context* bestimmten *Device* kann eine Menge von Kerneln ausgeführt werden. Zur Verwaltung der Ausführung der Kernel wird eine *cl_command_queue* benötigt, die einem *cl_context* zugewiesen werden muss.

Um eine .cl Datei mit einer Menge von Kerneln mit dem *cl_context* zu verknüpfen wird die Funktion *clCreateProgramWithSource* benutzt.

Zur Auswahl eines konkreten Kernels, der innerhalb dieses Quellcodes vorhanden ist, wird die Funktion *clCreateKernel* benutzt. Diese Funktion gibt unter Angabe des Funktionsnamens des Kernels diesen als *cl_kernel* Objekt zurück.

Speicherverwaltung

Um Daten zwischen Host und Device zu kopieren, wird eine *cl_mem* Entität benutzt, welche durch die Funktion *clCreateBuffer* erzeugt wird. Dabei muss neben *cl_context* auch die gewünschte Größe in Bytes angebenen werden. Ein *cl_mem* Objekt ist ein Puffer zum Zugriff auf Devicespeicher. Dabei wird für jedes Speicherobjekt im Kernel eine *cl_mem* Entität erzeugt. Mithilfe der Parameter der Funktion kann der Kernel auf lesenden oder schreibenden Zugriff auf die *cl_mem* Entität beschränkt werden. Statt Daten vom Host auf das Device zu kopieren, kann alternativ ein Speicherbereich auf dem Device alloziert werden.

Mögliche Fehler die dabei auftreten sind:

- CL_INVALID_BUFFER_SIZE: falls der Puffer größer ist, als das Device erlaubt
- CL_MEM_OBJECT_ALLOCATION_FAILURE: falls die Allokation auf dem Device fehlschlägt
- CL_OUT_OF_HOST_MEMORY: falls die notwendigen Ressourcen auf Hostseite nicht alloziert werden können

Kernel ausführen

Die erzeugte *cl_mem* Entität wird einem konkreten Kernel mittels der Funktion *clSetKernelArg* zugewiesen. Der dabei übergebene Index beim Funktionsaufruf entspricht der Argumentreihenfolge der Funktion im Kernel und beginnt bei 0.

Um den Kernel auszuführen wird die Funktion *clEnqueueNDRangeKernel* aufgerufen. Der Parameter *global_work_size* legt fest, wie viele *work_items* für diesen Kernel erstellt werden. *local_work_size* gibt an wie viele *work_items* eine *workgroup* bilden. Für *local_work_size* gelten folgende Einschränkungen:

- *global_work_size* muss ohne Rest teilbar sein durch *local_work_size*
- *local_work_size* darf maximal so groß sein wie *global_work_size*
- *local_work_size* darf die maximal von einer Berechnungseinheit berechenbare Anzahl *work_items* nicht überschreiten.

Alternativ kann mit Angabe von NULL dem Compiler die Entscheidung über die Größe der *work_groups* überlassen.

Ergebnisse zurückkopieren und Aufräumen

Mit dem Aufrufen der Funktion *clFinish* blockiert das Programm auf Hostseite, bis alle Kernel in der übergebenen *cl_command_queue* terminert sind.

Um die vom Kernel berechneten Ergebnisse vom Devicespeicher in den Hostspeicher zu kopieren wird *clEnqueueReadBuffer* verwendet.

Vor Beenden des Hostprogrammes müssen einige Aufräumoperationen durchgeführt werden:

- Speicher wird freigegeben mittels *clReleaseMemObject*
- Kernel wird freigegeben mittels *clReleaseKernel*
- Der Kernelcode wird freigegeben mittel *clReleaseProgram*
- Die command queue wird freigegeben mittels *clReleaseCommandQueue*
- Der cl_context wird freigegeben mittels *clReleaseContext*

Danach kann das Hostprogramm terminiert werden, ohne dass Ressourcen auf Host oder Device weiter blockiert werden.

2.1.4 Testumgebung

Ausführen von OpenCL Programmen ist auf einer Vielzahl verschiedener Devices möglich, die OpenCL treiberseitig unterstützen. Dazu gehören Intel und AMD CPUs, Nvidia und AMD GPUs, Xeon Phi, FPGAs (Field Programmable Gate Array), integrierte Grafikeinheiten einer CPU, sowie ARM Prozessoren. Eine Liste unterstützter Devices und deren maximale OpenCL Version ist durch Khronos veröffentlicht [Khronos, 2016c]. Für diese Arbeit wurden folgende Devices verwendet und abgekürzt:

- CPU: Intel Core i5-3450 3.1 GHz
- GPU: Nvidia GeForce GTX 660 Ti
- IGPU: Intel HD Graphics 2500

IGPU und CPU *cl_devices* sind dabei der selben Platform zugeordnet.

Zur Verwendung von OpenCL sind softwareseitig ein OpenCL-Treiber für das gewünschte Device und ein Compiler der OpenCL unterstützt notwendig. Entwicklung und Messungen erfolgen unter Windows 7, wobei darauf geachtet wurde, dass alle Kerneldateien und der Hostcode auch unter Linux und Mac funktionieren. Betriebsystemabhängige Codeabschnitte werden mit der *if defined* Klausel gekapselt.

Als Compiler wird der Intel C++ Compiler des INDE Plugin für Visual Studio verwendet. Als Entwicklungsumgebung und zum Debuggen wird Visual Studio 2013 benutzt. Debugging ist aufgrund von Limitierungen des INDE Plugins ausschließlich mit der CPU möglich. Die Ausführung des Codes ist auf allen vorhandenen Devices möglich.

Der im INDE mitgelieferte Intel Code Builder(ICB) als leichtgewichtige Entwicklungs- und Messumgebung wurde anfangs hinzugezogen, da dieser Performanceinformationen anzeigt und auch GPU debuggen kann. Bei mehrfachen Messungen der Ausführungszeit mit identischem Quellcode und Konfiguration, treten starke Abweichungen im zweisteilligen Prozentbereich auf. Mehr als die Hälfte der Ausführungen eines Kernels resultieren im Absturz des ICB, unabhängig von Kernel und Konfiguration. Daher wurde der ICB nicht für diese Arbeit verwendet.

2.2 UNTERSUCHTE DATENBANK-OPERATOREN

Listing 2.2: Datenbank-statement für Operatorgraph

```
SELECT abteilung, sum(gehalt)
FROM personal inner join gehalt
on gehalt.id = personal.id
WHERE abteilung = 'IT'
GROUP BY abteilung
```

Eine Anfrage an eine Datenbank wird von dieser intern in einen Baum von logischen Datenbank-Operatoren zerlegt (im Folgenden nur Operator). Ein Operator besitzt eine Menge von Parametern, eine Menge von Eingabedaten und errechnet eine Menge von Ergebnisdaten. Die Eingabedaten kommen aufgrund der Baumstruktur von einem vorherigen Operator, ebenso stellen Ergebnisdaten die Eingabedaten des nachfolgenden Operators dar. Davon ausgenommen ist der letzte Operator, welcher seine Ergebnisse an das Datenbanksystem zurückgibt. Der erste Operator der keinen Vorgänger hat, erhält seine Eingabe vom Datenbanksystem. Aufgrund ihrer Häufigkeit und des hohen Anteils an der Ausführungszeit wurden für diese Arbeit die drei Operatoren Selection, Hashjoin und GroupBy ausgewählt. Als Selection wird hierbei die Projektion, also die Beschränkung einer Menge von Attributen bezeichnet.

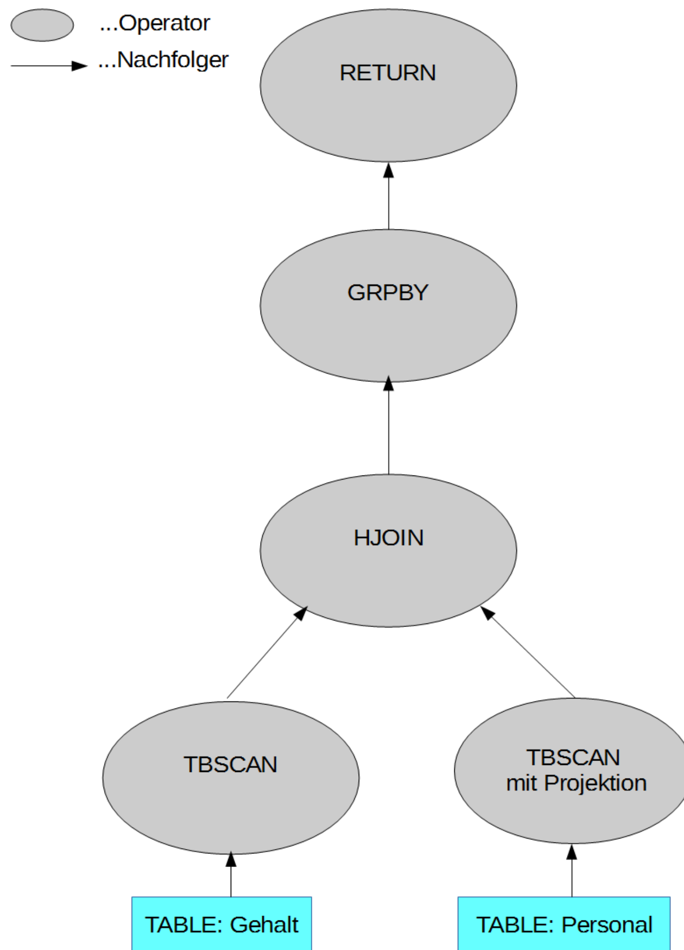


Abbildung 2.1: Der Operatorgraph der aus Listing 2.2 resultiert

Zum besseren Verständnis wurde ein minimales Beispiel eines möglichen Operatorbaumes mithilfe von DB2 erstellt. Bezüglich Tabellenschemata, Inhalt der Tabllen und Konfigurationen der Datenbank wird auf A.2 verweisen.

Diese Anfrage ergibt folgenden Graphen aus Datenbankoperatoren.

Alle drei Operatoren sind in diesem Graph enthalten. Der Operator TBSCAN (Tablescan) enthält die Projektion des Selectionoperators in der Klausel

$$WHERE\ abteilung = 'IT'$$

Als Eingabemenge dient jeweils eine Tabelle. Die Ergebnismengen der beiden TBSCAN-Operatoren sind gleichzeit die Eingabemengen des HJOIN-Operators (Hashjoin). Der HJOIN Operator führt einen Hashjoin auf den übergebenen Ergebnismengen der beiden TBSCAN-Operatoren aus. Dabei wird das Attribut ID in beiden Tabellen als Joinattribut verwendet. Die Ergebnismenge des HJOIN Operators ist eine neue Tabelle, die sowohl die Attribute der Tabelle Personal, als auch der Tabelle Gehalt enthält. ID kommt dabei als Joinattribut nur einmal vor.

Die Ergebnissmenge des HJOIN Operators wird als Eingabe für den GRPBY-Operator (Group By) verwendet. Auf diese Menge wird die Aggregatsfunktion angewendet:

$$sum(gehalt)$$

Nach dem GRPBY-Operator, übergibt der RETURN-Operator die Ergebnismenge an das Datenbank-System.

Für diese Arbeit werden Datenbankoperatoren lösgelöst von einem konkreten Datenbanksytem betrachtet und mit Zufallsdaten befüllt, um ihre Perfomance zu optimieren. Dabei sind die Kerneldateien .cl aus dem Ocelot Plugin für MonetDB entnommen. Das Ocelot Plugin ermöglicht Hardwareunabhängiges paralleles Berechnen von Datenbank-operatoren mittels OpenCL. Für einen Überblick über Ocelot wird auf [Heimel et al., 2013] verwiesen.

2.2.1 Selection

Funktionalität

Der Selection-Operator entscheidet, ob ein Attribut den Kriterien des Select-Statements entspricht, die dem Operator als Parameter übergeben werden. Die Kriterien werden in der WHERE-Klausel angegeben. Alle Attribute, die den Kriterien entsprechen, werden in die Ergebnismenge übernommen. Die Berechnung erfolgt Zeilenweise, für je ein konkretes Attribut (in den Formeln AT genannt). Der Operator entspricht damit der Projektion.

Übergeben werden dabei folgende Parameter:

- Untergrenze des Wertes (UG)
- Obergrenze des Wertes (OG)
- Flag für Existenz der Untergrenze (UG_E)
- Flag für Existenz der Obergrenze (OG_E)
- Flag für Inklusivität der Untergrenze (UG_I)
- Flag für Inklusivität der Obergrenze (OG_I)
- Flag für logische Negation des Ergebnisses (NEG)

Mithilfe dieser Parameter und des Attributes werden eine Reihe von Bedingungen überprüft, die alle wahr sein müssen, damit das Attribut in die Ergebnismenge des Selection-Operators übernommen wird. Je Attribut muss also ein Wahrheitswert berechnet werden.

Die Ergebnismenge des Operators ergibt sich aus der Menge aller Eingabetupel, für die alle Bedingungen erfüllt sind. Falls NEG = 1 ist, gilt:

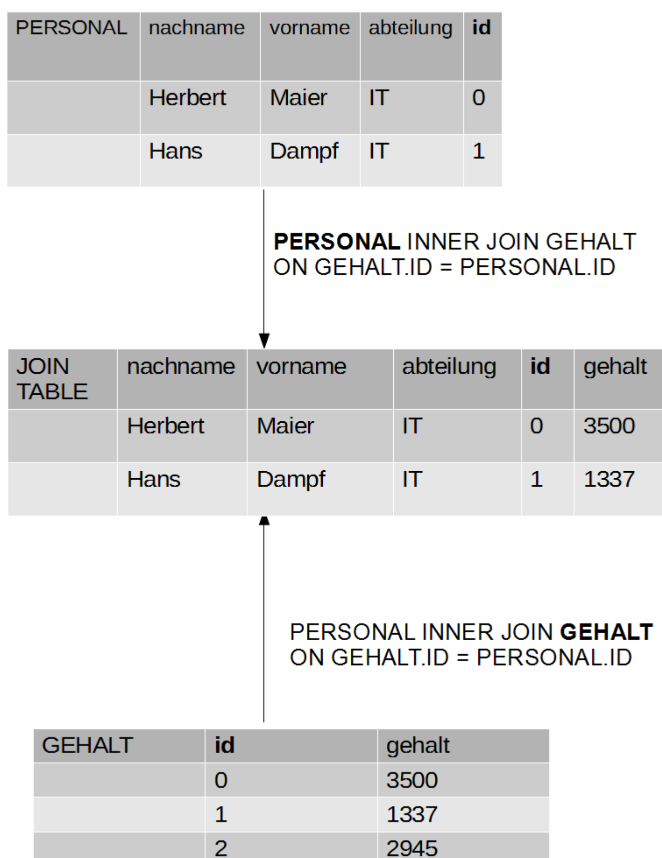
$$resultmenge_neg = (eingabemenge - resultmenge)$$

OpenCL-Implementierung

In der OpenCL-Implementierung werden je Thread acht Attribute mithilfe einer for-Schleife verarbeitet. Eine Abbruchbedingung überprüft bei jedem Schleifendurchlauf, ob die Gesamtanzahl der zu berechnenden Elemente überschritten wird. Alle Berechnungen erfolgen innerhalb einer Kernelfunktion.

Ein Präprozessormakro wird verwendet, um jeweils für einen konkreten Wert zu berechnen, ob er in die Ergebnismenge übernommen werden soll. Die for-Schleife ruft das Präprozessormakro acht mal auf, falls die Gesamtzahl der Elemente nicht überschritten wurde und gibt ein Bitmap zurück. Diese Bitmap wird als Ergebnis zurückgeschrieben. Die aktuelle Thread ID ($get_global_id(0)$) wird als Offset für Speicherzugriffe auf die Eingabedaten und die Ergebnisdaten verwendet.

2.2.2 Join



PERSONAL	nachname	vorname	abteilung	**id**
	Herbert	Maier	IT	0
	Hans	Dampf	IT	1

JOIN TABLE	nachname	vorname	abteilung	**id**	gehalt
	Herbert	Maier	IT	0	3500
	Hans	Dampf	IT	1	1337

GEHALT	**id**	gehalt
	0	3500
	1	1337
	2	2945

Abbildung 2.2: Ergebnis des Join-operators, der aus Listing 2.2 resultiert

Funktionalität

Bei einem Join werden jeweils zwei Tabellen anhand ihrer Joinpartner tupelweise auf Übereinstimmung getestet. Als Eingabedaten werden zwei Tabellen benutzt, wobei jeweils ein Attribut pro Tabelle als gemeinsames Joinattribut vorhanden sein muss. Die Ergebnissmenge besteht aus allen Eingabetupeln mit ihren Joinpartnern. Das Maximum eines Joins ist das kartesische Produkt der Eingabetabellen.

In Listing 2.2 wird das gemeinsame Joinattribut definiert durch folgende Bedingung:

$$on\ gehalt.id = personal.id$$

Es wird für jedes Tupel der linken Eingabemenge eine Menge von Partnern errechnet, für die die Bedingung erfüllt ist. Um die Berechnung zu beschleunigen, wird ein Hashwert für die Joinattribute errechnet.

Für ein Beispiel des Join Operators, siehe Abbildung 2.2.

OpenCL-Implementierung

In der OpenCL-Implementierung werden zur Berechnung der Ergebnistabelle mehrere Kerneln ausgeführt. Die Implementierung der Kernel ist aus Ocelot entnommen [Heimel et al., 2013]

Die Eingabedaten sind zwei Tabellen die aus Zufallszahlen erzeugt werden. Sie werden bezeichnet als Tabelle A und Tabelle B. Die Tabellen bestehen dabei aus genau einer Spalte, wobei diese Spalte jeweils das Joinattribut ist. Die Zufallszahlen ersetzen die Anbindung an das Ocelot Plugin.

Als erster Schritt werden mit der Kernelfunktion $init_zero_int$ die beiden cl_mem Objekte $table_buffer$ und $count_buffer$ initialisiert. Dabei wird jeder Integer an jeder Stelle des Arrays auf den Wert 0 gesetzt.

Für jeden Wert des Joinattributs von Tabelle A wird ein Hashwert errechnet. Diese werden von der Funktion $buildHashTablePessimistic$ in $table_buffer$ geschrieben. $table_buffer$ enthält damit die Hashtabelle für Tabelle A.

Die Kernelfunktion $countKeyCardinality$ zählt die Häufigkeit der Vorkommen eines Attributwertes des Joinattributs in Tabelle A. Dies wird mithilfe des errechneten Hashwertes bestimmt. Die Häufigkeit wird zurückgeschrieben in $offset_buffer$.

Mit den drei Funktionen prefixsum_seq, prefixsum_par, prefixsum_inc wird eine Präfixtabelle für Tabelle A erstellt. Präfixtabelle PA ist dabei um ein Element größer als Tabelle A. An der letzten Stelle von Präfixtabelle PA wird die Gesamtzahl der Vorkommen gespeichert.

Zu jedem Joinattribut in Tabelle A gibt es eine Anzahl von Vorkommen in Tabelle A. Dies ist die Kardinalität eines Attributs. In der Präfixtabelle für Tabelle A wird eine Variable für jedes Attribut um die Anzahl der zugehörigen Vorkommen erhöht. Das Erhöhen der Variable geschieht jedoch erst nach dem zurückschreiben des Eintrags in die Präfixtabelle.

Buildlookuptable erstellt eine Zuordnung von Zeilen IDs für Tabelle A. Sie bezieht sich auf Präfixtabelle PA.

hj_count berechnet für jeden Wert von Tabelle B die Anzahl zugehöriger Joinpartner in Tabelle A. Dabei berechnet jeder Thread mithilfe einer For-schleife die Anzahl für mehrere Tupel von Tabelle B. Dies wird über den Parameter tuples_per_thread im Hostcode festgelegt.

Dabei wird die Übereinstimmung der Joinattribute durch die Funktion $findInHashtable$ bestimmt. Falls ein Hashwert für das jeweilige Attribut in Tabelle B in der Hashtabelle von Tabelle A existiert, wurde ein Joinpartner gefunden.

Als Ergebnis wird in *cl_mem* offset2 die Anzahl von Joinpartnern von Tabelle B in Tabelle A zurückgeschrieben. Dabei entspricht der Index in offset2 dem Index in Tabelle B. Die Anzahl von Joinpartnern für ein Attribut in Tabelle B ist wie folgt definiert:

$$AnzahlJoinPartner[i] = offset2[i]$$

Die Kernelfunktion *hj_excec* berechnet zwei Spalten von Zeilen-IDs. Dabei wird jeder Zeilen ID von A eine Zeilen-ID von B zugewiesen. Diese Zeilen-IDs sind das Ergebnis des Hjoin-operators und stellen eine Zuordnung von Zeilen anhand der Joinattribute dar. Die Berechnung erfolgt attributweise, wobei jeder Thread mit einer for-Schleife eine Menge von Attributen berechnet, was über einen Parameter im Hostcode steuerbar ist. Für jedes Attribut von Tabelle B wird zuerst der Hashwert und die Anzahl der Joinpartner bestimmt. Unter Verwendung von Präfixtabelle PA, der lookuptable und offset2 werden die beiden Zusammenhängenden Spalten von Joinpartnern berechnet.

Unabhängig vom verwendeten Device verbraucht *hj_excec* den größten Teil der Ausführungszeit. Es fällt auf, dass *hj_excec* und *hj_count* einen identischen Zugriff auf die Hashtabelle anhand von Attributen von Tabelle B durchführen.

2.2.3 GroupBy

Funktionalität

Der GroupBy-operator gruppiert eine oder mehrere Spalten anhand einer Aggregatsfunktion. Für eine Übersicht der Aggregatsfunktionen wird auf [W3Schools, 2016] verwiesen. Als Eingabemenge fungiert eine Tabelle. In Abbildung 2.3 wird das Ergebnis des GroupBy-operators anhand Listing 2.2 gezeigt. In Listing 2.2 wird die Aggegratfunktion sum() wie folgt angewendet mit $sum(gehalt)$. Dieser Befehl legt fest, dass das Attribut $gehalt$ aufsummiert wird.

Um das Attribut zur Gruppierung der Ergebnismenge zu bestimmen, dient folgende Klausel:

$$GROUP\ BY\ abteilung$$

Anhand des Gruppierungsattributs abteilung wird die Tabelle gruppiert. Dafür wird für jedes Gruppierungs-attribut eine $group\ id$ berechnet, die für die Gruppierung verwendet wird. Als Er-

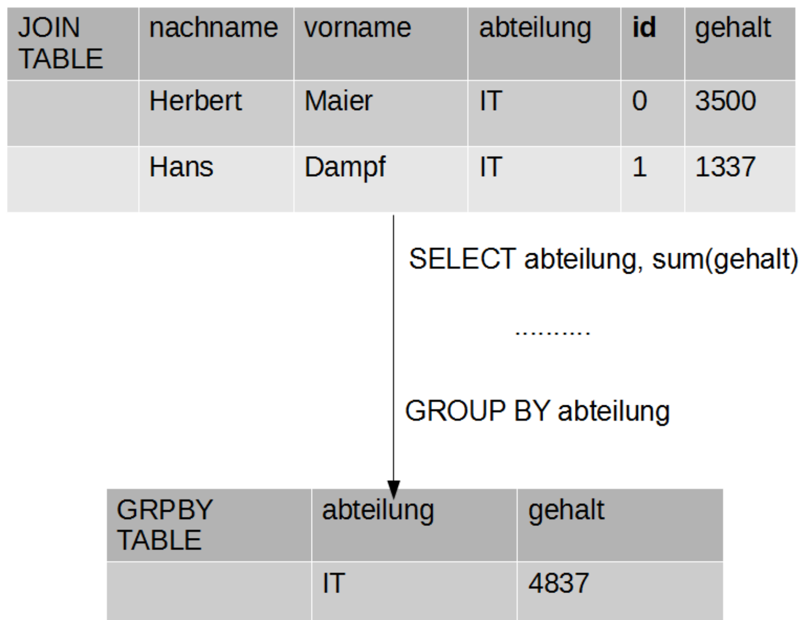


Abbildung 2.3: Ergebnis des GroupBy-operators, der aus Listing 2.2 resultiert

gebnis kommt jede Abteilung genau einmal vor. Bei Mehrfacheinträgen mit der selben Abteilung wird das Attribut *gehalt* aufsummiert und als Ergebnis angezeigt.

Das Ergebnis des Operators ist eine Zuweisung einer Gruppen-ID zu jeder Zeile einer Tabelle. Dabei kann die Bildung der Gruppen-ID von einem oder mehreren Attributen abhängen.

OpenCL-Implementierung

Die Implementierung des Kernels ist aus Ocelot entnommen [Heimel et al., 2013]. In der OpenCL-Implementierung werden zur Berechnung der Ergebnistabelle mehrere Kernelfunktionen ausgeführt.

Als Eingabe fungiert eine Zeile einer Tabelle, die aus Zufallszahlen erzeugt wurde. Sie wird im folgenden Tabelle A genannt.

Zur Vorbereitung weiterer Verarbeitungsschritte wird das Speicherobjekt *table_buffer* mit dem Integer-Wert 0 an jeder Stelle des Arrays initialisiert. Dies wird mit der Kernel-Funktion *init_zero_int* realisiert.

Anschließend wird eine Hashtabelle für Tabelle A erstellt. Es wird ein Hashwert für jeden Attributwert gebildet. Die Kernelfunktion *buildHashTableOptimistic* berechnet die Hashwerte. *buildHashTableOptimistic* überprüft während der Berechnung, ob Fehler aufgetreten sind und

gibt ihre Anzahl als int zurück. Um Fehler festzustellen, wird die Kernelfunktion *validateHashTable* aufgerufen. *buildHashTableOptimistic* verwendet die Kernelfunktion hash1.

Falls Fehler aufgetreten sind, wird die Hashtabelle neu berechnet durch die Funktion BuildHashTablePessimistic. Diese verwendet die Kernelfunktionen hash2 und hash3.

Nach dem Erstellen der Hashtabelle wird mithilfe der Kernelfunktion *prepareIDTable* die Tabelle für die Gruppen-IDs vorbereitet und im Speicherobjekt *idTable* gespeichert.

Es wird eine Präfixtabelle generiert nach dem selben Schema, das bereits in 2.2.2 erläutert wurde. Mithilfe der Präfixtabelle wird die Gruppen-ID je Hashwert berechnet und in *idTable* gespeichert.

Die Kernelfunktion *computeGroupingHt* weist jedem Attribut aus Tabelle A anhand seines Hashwertes eine Gruppen-ID zu und speichert diese im Speicherobjekt *computeGrouping*. Dabei entspricht der Index von *computeGrouping* dem Index von Tabelle A.

Die Kernelfunktion *generateExtentsTable* errechnet ein Beispielelement für jede Gruppen-ID. Dazu wird für jedes Attribut in Tabelle A die Gruppen-ID bestimmt. Die Gruppen-ID wird als Index für das Speicherobjekt *groups* verwendet und schreibt an dieser Stelle die Thread-ID. Die Thread-ID entspricht dem Index von Tabelle A, also dem Zeilenindex der Eingabetabelle. Da die Gruppen-ID als Index benutzt wird und der Kernel parallel ausgeführt wird, ist als Ergebnis nur sichergestellt, dass jeder Gruppen-ID genau einmal auftaucht. Mithilfe des Speicherobjekts *groups* lässt sich nachvollziehen, wie viele verschiedene Gruppen-IDs durch den Operator entstanden sind.

Die Kernelfunktion ComputeGroupingHt hat dabei den größten Anteil an der Ausführungszeit des Kernels, unabhängig vom verwendeten Device.

Bei den Kernelfunktionen *init_zero_int*, *generateExtentsTable*, *prepareIDTable* erfolgen Speicherzugriffe ausschließlich anhand der Thread-ID.

3 OPTIMIERUNGEN

Im Folgenden werden alle versuchten Optimierungen und am Ende jedes Abschnitts die maximalen Perfomancegewinne je Operator vorgestellt.

3.1 SELECTION

Die Kernelfunktion berechnet je Thread acht Wahrheitswerte mithilfer einer For-schleife. Für jeden Aufruf der For-schleife wird überprüft, ob der aktuelle Index die Gesamtzahl der Elemente überschreitet. Dies ist notwendig, da die Menge der Eingabedaten nicht garantiert durch 8 teilbar ist. Der aktuelle Index (pos) wird wie folgt berechnet:

$$pos = 8 * get_global_id(0)$$

pos wird innerhalb der Schleife nach jedem Durchlauf um eins inkrementiert und zum Zugriff auf die Eingabedaten verwendet. Nach der For-schleife werden die Ergebnisdaten zurückgeschrieben. Dies passiert auch, falls die Abbruchbedingung ausgelöst wurde. Sofern im Hostcode zuviele Threads erstellt wurden, löst die Kernelfunktion damit ungültige Speicherzugriffe aus. Abgesehen vom letzten Thread, werden für jeden Thread acht if-bedingungen erreicht.

3.1.1 Overflow-Kernel

Als Optimierungsidee wird die maximale Anzahl von Schleifendurchläufen vor der For-schleife berechnet, so dass auf eine Abbruchbedingung verzichtet werden kann, zugunsten einer variablen Anzahl an Schleifendurchläufen. Überlaufen durch zuviele Threads wird damit verhindert. Dabei wurde eine bool-Variable eingeführt um festzuhalten, ob überhaupt Daten zurückgeschrieben werden müssen. Da der bedingte ? Operator in OpenCL performanter als eine If-Bedingung ist, wurde dieser eingesetzt. Die Optimierungsidee wird im Folgenden als Overflow-kernel bezeichnet. Abhängig von der Eingabe ergeben sich folgende drei Möglichkeiten bei der Ausführung des Kernels:

- Teilweise Berechnung von Elementen. Die For-schleife berechnet zwischen 1 und 7 Elemente
- Vollständige Berechnung von 8 Elementen.
- Alle Eingabedaten wurden bereits berechnet, Thread terminiert ohne Berechnung und Zugriff auf Eingabe- und Ergebnisdaten.

Durch die Konstante *elem* wird der Kernelfunktion übergeben, wie viele Elemente genau berechnet werden sollen. Ein Thread durchläuft in der geänderten Variante des Kernels nur drei Verzweigungen verglichen mit acht in der Ausgangsvariante. Dadurch wird der optimierte Kernel compilerseitig vektorisiert. Dafür müssen allerdings drei Variablen im local Memory zusätzlich erzeugt werden, was den Speicherbedarf des Kernels erhöht.

Vektorisierung bedeutet im OpenCl Kontext, dass Lese- und Schreibzugriffe eines Threads jeweils für eine Anzahl Elemente parallel erfolgen. Dies kann explizit im Kernelcode erzwungen werden oder implizit durch den Compiler erfolgen. Je komplexer der Kontrollfluss des Kernels ist, desto weniger lohnt sich Vektorisierung. Durch die Reduktion von acht Verzweigungen auf drei gelangt der Compiler zu der Einschätzung, dass der Kernel von Vektorisierung profitiert.

Evaluation

Für Messungen der Ausführungszeit werden overflow- und standardkernel verglichen. Es wurde folgendes Testszenario verwendet:

- gpu: 160 Millionen Elemente
- cpu: 160 Millionen Elemente
- igpu: 16 Millionen Elemente

Die IGPU erlaubt wegen geringerem global memory nicht die Berechnung von 160 Millionen Elementen. Für die Ausführungszeit der CPU ergibt sich folgendes Bild:

Tabelle 3.1: CPU Messung Selection

CPU	standard	overflow
time in[ms]	184,4	113,2
speedup in [%]	–	38,61%

Aufgrund von Architektureigenschaften profitiert die CPU in hohem Maße von dem overflow Kernel. Das Pipelining der CPU löst bei falscher Branch Prediction aus, dass mehrere Instruktionen wiederholt werden müssen. Daher macht sich der sinkende Verzweigungsgrad des Kernels positiv bemerkbar, indem die Anzahl der Wiederholungen von Instruktionen gesenkt wird.

Tabelle 3.2: GPU Messung Selection

GPU	standard	overflow
time in[ms]	26	22,9
speedup in [%]	–	11,92%

Die GPU profitiert in geringerem Maße vom Overflow-kernel. Da sie eine höhere Anzahl paralleler Threads verarbeiten kann, hat die Verringerung der Verzweigungen weniger Einfluss.

Tabelle 3.3: IGPU Messung Selection

IGPU	standard	overflow
time in[ms]	36,8	37,4
speedup in [%]	–	-1,63%

Die IGPU verzeichnet als einziges Device einen Performanceverlust.

Als Occupancy bezeichnet man das Verhältnis der maximalen parallel ausführbaren Anzahl an work groups und den parallel auf einer Berechnungseinheit ausgeführten work groups. Je nach Device, kann die Verwendung von mehr Local Memory zu einer verringerten Occupancy führen. Da die IGPU, wie in Tabelle 2.1 angegeben, die größte Menge Local Memory je Berechnungseinheit hat, lässt sich der Performanceverlust nicht mit einer verringerten Occupancy erklären. Mithilfe der Funktion $clGetKernelWorkGroupInfo$ kann die benötigte Menge Local Memory und Private Memory für einen angegebenen Kernel je Device errechnet werden. Der benutzte OpenCL Treiber von Intel für die IGPU und CPU unterstützt diese Funktion zwar, gibt jedoch einfach den Wert 0 zurück, unabhängig von der Eingabe. Die fehlerhafte Funktion erlaubt es nicht, einen korrekten Wert für den Speicherbedarf vom Treiber zu erhalten. Daher kann die Occupancy für IGPU und CPU nicht berechnet werden.

3.1.2 Local work size und global work size

Der Overflow-Kernel verhindert Speicherzugriffe, nachdem Anzahl $elem$ Werte berechnet wurden. Weitere Threads können keine Speicherzugriffe verursachen. Daher ist es möglich die Global Work Size immer auf eine Teilbarkeit von 128 aufzufüllen und die Local Size auf 128 zu setzen. Die beste Performance ergibt sich für eine Local Size von 128 auf jedem Device. Der Original-Kernel erlaubt kein Auffüllen, da sonst Zugriffsverletzungen beim Speicherzugriff auftreten. Im Folgenden wird untersucht, ob das Auffüllen der Global Work Size die Ausführungszeit reduziert. Als Speedup wird der Quotient der beiden Ausführungszeiten bezeichnet. Dabei wird die Ausführunszeit des Original-Kernels durch die Ausführungszeit des Overflow-Kernels geteilt. Für die Messwerte und Graphen wird auf Anhang A.1 verwiesen.

Für die Messungen gelten folgende Bedingungen:

- Fall A: 2.000.001 Threads, Local Size nicht angegeben, Original-Kernel
- Fall B: Aufgefüllt auf 2.000.128 Threads, Local Size = 128, Overflow-Kernel

Ausführungszeit Kernelfunktion in [ms]	CPU	IGPU	GPU
Fall A	28,3	75,7	19,1
Fall B	11,4	37,7	2,7
Speedup	2,48	2,01	7,07

3.1.3 Bewertung

Die Benutzung des Overflow-Kernels ist auf allen Devices sinnvoll. Da ungültige Speicherzugriffe im Kernel-Code ausgeschlossen werden, kann die Global Work Size im Hostcode aufgefüllt werden, ohne Änderungen an den Speicherobjekten vorzunehmen. Dadurch kann auch auf der IGPU die Ausführungszeit verringert werden, sofern die Anzahl der Eingabedaten nicht bereits durch 128 teilbar ist. Die Ausführungszeit wird durch Auffüllen und Overflow-Kernel auf jedem Device verbessert. Die GPU profitiert mit einem Speedup von 7,07 in höherem Maße von den Optimierungen. Der Overflow-Kernel ist sowohl schneller als auch robuster gegenüber Fehlern in der Speicherverwaltung des Hostcodes.

3.2 JOIN

Im Folgenden werden Optimierungen am Hashjoin Operator erläutert.

3.2.1 PreprocessHTKernel

In den beiden Kernelfunktionen *HjCount* und *Hjexec* findet mithilfe der Kernelfunktion *findInHashTable* ein identischer Zugriff auf die Hashtabelle statt. Um die doppelte Berechnung zu vermeiden wird eine weitere Kernelfunktion namens *PreprocessHTKernel* eingeführt. Diese Funktion speichert die benötigten Zugriffe auf die Hashtabelle in einem cl_mem Objekt. *HjCount* und *Hjexcec* werden so verändert, dass sie auf dieses *cl_mem* Objekt zugreifen, anstatt *findInHashTable* aufzurufen.

Die Idee der neuen Kernelfunktion ist es, eine Berechnung zu sparen mithilfe eines weiteren Speicherzugriffs. Die Ausführung des veränderten Operators führt auf IGPU und CPU zu einem nicht nachvollziehbarem Absturz. Die GPU berechnet trotz der Änderungen des Ergebnis des Operators korrekt.

Evaluation

Für die Evaluation des PreprocessHtKernels wird Tabelle A mit zwei Millionen Elementen gefüllt und Tabelle B mit einer Millionenen Elemente. Dabei wird untersucht, ob die gesamte Ausführungszeit reduziert werden kann, indem PreprocessHtKernel eine identische Berechnung, die von den beiden Kerneln Hjcount und Hjexcec vorgenommen wird, ersetzt.

Tabelle 3.4: PreprocessHTKernel auf GPU, Hashjoin

Ausführungszeit [ms]	mit PreppproccessHtKernel	ohne PreppproccessHtKernel
PreprocessHt	6	0
HjCount	6	7
Hjexcec	305	306

Obwohl die beiden Kernelfunktionen *HjCount* und *Hjexcec* jeweils um eine Millisekunde schneller werden, erhöht sich die gesamte Ausführungszeit um vier Millisekunden. Das Einführen des PreprocessHTKernels führt also zu keiner Beschleunigung der Ausführungszeit des Operators.

3.2.2 Local Memory Caching

Die Kernelfunktionen *HjCount* und *Hjexcec* speichern den Rückgabewert der Kernelfunktion *findInHashTable* ohne Angabe eines Speichertyps in einem Integer-Array, also im Private Memory. Durch die explizite Angabe von *__local* vor dem Array wird es im Local Memory gespeichert. Wenn der Kernel im Debugmodus von Visual Studio gestartet wird, erfolgt eine korrekte Ausführung. Zum Messen der Ausführungszeit ist es notwendig, den Kernel ohne Debug auszuführen. Dies führt bei IGPU und CPU zu einem Absturz mit dem Hinweis auf eine lesende Zugriffsverletzung. Die GPU führt den veränderten Operator korrekt und ohne Fehlermeldung aus.

Evaluation

Für die Evaluation wird Tabelle A mit zehn Millionen Elementen gefüllt und Tabelle B mit 100.000 Elementen.

Tabelle 3.5: Local Mem Caching auf GPU, 10 Millionen mit 100.000 Elementen Hashjoin

Ausführungszeit in [ms]	mit __local	ohne __loca
alle Kernel ohne Kopieren der cl_mem	590	583
alle Kernel mit Kopieren der cl_mem	988	1001

Die Verschiebung in Local Memory hat einen Performanceverlust von 13 Milisekunden zur Folge. Dies entspricht einem Speedup von -1,3%.

3.2.3 Local work size und global work size

Zur Steuerung der *global_work_size* und *local_work_size* in *HjCount* und *Hjexcec* werden im Hostcode zwei Parameter verwendet.

- *local_size*: Größe einer work group, also *local_work_size*
- *processors*: Beinflusst Aufteilung der Zeilen auf Threads und *global_work_size*

Für die Berechnung der *global_work_size* wird folgende Formel verwendet:

$$global_work_size = processors * local_size$$

Die Anzahl der Tupel, die von einem Thread berechnet werden soll ergibt sich wie folgt:

$$Tupel_pro_thread = \frac{TabelleA_Tupels}{processors * local_size}$$

Eine Erhöhung des Parameters *processors* führt dazu, dass eine größere Anzahl von Threads erstellt wird und dass ein Thread jeweils weniger Tupel berechnet.

Evaluation Für die Messungen wird ein Hashjoin mit 100000 Elementen in Tabelle A und einer Millionen Elemente in Tabelle B berechnet. Um die kürzesten Ausführungszeiten von $HjCount$ und $Hjexcec$ zu bestimmen, werden alle Kombinationen der Parameter mit den folgenden Werten berechnet:

- *local_size* 32 , 64 , 128 , 256
- *processors* 16 , 32

Die Standardwerte der Parameter im Hostcode waren *local_size* = 256 und *processors* = 32. Im Folgenden wird die Kombination mit der kürzesten Ausführungszeit je Device mit der Ausführungszeit mit den Standardwerten verglichen.

Tabelle 3.6: Speedup je Device mit *processors* und *local_size*

Ausführungszeit in [ms]	IGPU	CPU	GPU
Standardwerte	258,8	43	66,4
beste Kombination	198,2	32,2	59
Speedup	1,31	1,02	1,13

Die besten Kombinationen sind dabei folgende:

Tabelle 3.7: beste Kombination *processors* und *local_size*

Device	*processors*	*local_size*
IGPU	16	32
CPU	16	32
GPU	16	128

Es fällt auf, dass alle Geräte am schnellsten sind mit *processors* = 16. Daraus folgt, dass eine Beabeitung von mehr Tupeln je Thread performanter ist als das Erstellen einer größeren Anzahl von Threads für die Berechnung der Tupel. Für die *local_size* gilt, dass CPU und IGPU am schnellsten sind, wenn *local_size* = 32 ist. Die GPU hingegen ist am schnellsten, wenn *local_size* = 128 ist. Dies liegt daran, dass die bis zu GPU 2048 Threads je Berechnungseinheit [Nvidia, 2016, p. 7] ausführen kann und die IGPU lediglich 6 [Intel, 2012, p. 18]. Die CPU profitiert mit 4 Cores auch nicht von einer höheren *local_size*. Für die Messwerte der einzelnen Durchläufe wird auf Anhang A.1 verwiesen.

3.2.4 Bewertung

Von den untersuchten Optimierungsstrategien brachte ausschließlich die Variation der Parameter *processors* und *local_size* einen Performancegewinn. Im Vergleich zum ursprünglichen Hostcode wurde für jedes Device ein Performancegewinn erreicht. Der Performancegewinn entspricht in diesem Fall den in Tabelle 3.6 genannten Zahlen.

Die Strategie, eine Berechung auf Kosten eines weiteren Zugriffs auf Global Memory zu sparen, erhöht die Ausführungszeit.

3.3 GROUPBY

Im Folgenden werden Optimierungsstrategien für den GroupBy Operator erläutert.

3.3.1 Local work size und global work size

Es wird untersucht, ob sich das Auffüllen der Work Groups positiv auf die Ausführungszeit der Kernelfunktionen auswirken kann. Hierfür wird zuerst die Ausführungszeit bei einer Eingabegröße der Tabelle A, die teilbar durch 128 ist verglichen mit einer Eingabegröße, die unteilbar durch 128 ist. Die Local work size von 128 ist auf der GPU performanter und führt zu keinem Performanceverlust auf CPU und IGPU, der größer als 1 ms ist.

- Fall A: 12 Millionen Elemente als Eingabedaten in Tabelle A
- Fall B: 12 Millionen - 1 Elemente als Eingabedaten in Tabelle A

Tabelle 3.8: Auffüllen von *global_size*

Ausführungszeit in [ms]	CPU	IGPU	GPU
Fall A	228	234	40
Fall B	265	263	85
Speedup	1,16	1,12	2,13

Falls die Eingabedaten teilbar durch 128 sind, verringert sich die Ausführunszeit deutlich. Die GPU profitiert davon in höherem Maße. Dies liegt in der GPU Architektur begründet, die jeweils 32 Instruktionen parallel ausführt. Da auch bei einer *WorkGroup*-Größe von eins jeweils 32 Instruktionen parallel ausgeführt werden, müssen viele verworfen werden. Deswegen ist der Performanceunterschied zwischen Fall A und Fall B bei der GPU am größten. Bei einer Teilbarkeit der Eingabedaten durch 128, zeigt sich bei allen Devices eine kürzere Ausführungszeit.

3.3.2 Overflow Hostcode

Da die Anzahl der Eingabedaten großen Einfluss auf die Ausführungszeit hat, wird untersucht ob es möglich ist die Eingabedaten auf eine durch teilbare 128 Anzahl aufzufüllen und dabei die Korrektheit des Ergebnisses zu gewährleisten.

Durch Auffüllen der Global Work Size entstehen Speicherzugriffe außerhalb der allozierten cl_mem Objekte. Dabei unterscheidet sich das Verhalten der Intel Hardware deutlich von der Nvidia GPU. IGPU und CPU lösen bei einem Speicherzugriff außerhalb des cl_mem Objekts einen Absturz des Programms aus und geben unkommentiert Speicheradressen auf der Konsole aus. Die GPU berechnet alle Kernel und terminiert ohne Absturz, trotz lesender und schreibender Speicherzugriffe außerhalb des allozierten Bereichs. Um Abstürze zu vermeiden, können die cl_mem Objekte ebenfalls aufgefüllt werden. Dabei wird das Endergebnis der Berechungen verfälscht.

Um eine korrekte Berechnung zu erhalten, könnten die Fehler im Nachhinein aus den Ergebnisdaten entfernt werden. Dies funktioniert leider nicht, da sich die Berechnungsfehler zwischen verschiedenen OpenCL-Compilern unterscheiden und nicht klar vorherseebar sind.

Daher dürfen nur Kernelfunktionen aufgefüllt werden, die dadurch keine ungültigen Speicherzugriffe und auch keine falschen Ergebnissdaten hervorrufen.

Folgende Kernelfunktionen konnten aufgefüllt werden unter Beibehaltung korrekter Ergebnisse und Vermeidung ungültiger Speicherzugriffe:

- *init_zero_int*
- *prepareIDTable*
- *generateExtentsTable*

Evaluation

Es wird untersucht, ob das Auffüllen der global size ohne Änderungen in den Kerneln einen Reduzierung der Ausführungszeit bewirkt. Dabei werden als Anzahl der Elemente in Tabelle A folgende Eingaben verglichen.

- Fall A: 11999989 Elemente (Primzahl) ohne Auffüllen im Hostcode
- Fall B: 11999989 Elemente (Primzahl) mit Auffüllen im Hostcode

Die Abkürzung S steht für Speedup.

Tabelle 3.9: Auffüllen mit Hostcode

Ausführungszeit in [ms]	CPU			IGPU			GPU		
Kernelfunktion:	A	B	S	A	B	S	A	B	S
init_zero_int	72	26	2,77	86	6	14,33	39	4	9,75
prepareIDTable	58	10	5,8	145	10	14,5	64	5	12,8
generateExtentsTable	232	2	116	210	14	15	58	1	58

Im Falle einer Primzahl als Kardinalität der Eingabedaten bewirkt das Auffüllen im Hostcode eine erhebliche Reduzierung der Ausführungszeit. Die Größte Verbesserung zeigt sich bei *generateExtentsTable* ausgeführt auf der CPU. Die Ausführungszeit reduziert sich um 230 ms. Auf allen drei Devices führen die Optimierungen zu einer Verbesserten Ausführungszeit. Falls die Kardinalität von A teilbar durch 128 ist, entspricht die Ausführungszeit des auffüllenden Hostcodes der des ursprünglichen Hostcodes.

3.3.3 Overflow Kernel

Die beiden Kernelfunktionen mit dem größten Anteil an Ausführungszeit sind *buildHashtable* und $compute_{g}roupingHt$. Da die Speicherzugriffe dieser beiden Kernel nicht fortlaufend anhand der Thread-ID sind, reicht das Auffüllen von global size und benutzter cl_mem Objekte nicht aus

um ein korrektes Ergebnis zu erhalten. Daher ist eine zusätzliche If-Bedingung in den Kerneln notwendig, um ungültige Speicherzugriffe bei einer aufgefüllten global work size zu vermeiden. Auf Seiten des Hostcodes werden die global work sizes aufgefüllt auf die nächste durch 128 teilbare Zahl.

Evaluation

Es wird untersucht, ob der veränderte Kernelcode in Kombination mit dem Auffüllen im Hostcode zu einer veränderten Ausführungszeit führt. Dazu wird folgender Testfall benutzt.

- Fall A: 11999989 Elemente (Primzahl) ohne Auffüllen im Hostcode
- Fall B: 11999989 Elemente (Primzahl) mit Auffüllen im Hostcode

Tabelle 3.10: Auffüllen mit Hostcode und Kernelcode

Ausführungszeit in [ms]	CPU			IGPU			GPU		
Kernelfunktion:	A	B	S	A	B	S	A	B	S
buildHashtable	277	123	2,45	444	91	4,88	199	3	66,33
compute_groupingHt	108	24	4,5	256	22	11,64	123	6	20,5

3.3.4 Bewertung

Im Folgenden wird der kombinierte Effekt der Optimierungen auf die Ausführungszeit je Device betrachtet. Dabei wird 11999989 als Anzahl Elemente in Tabelle A definiert. Es wird der ursprüngliche Host- und Kernelcode ohne Auffüllen mit dem veränderten Code der sowohl im Kernelcode, als auch im Hostcode auffüllt. Dabei wird die Ausführungszeit in ms angegeben und Speedup als dimensionsloser Faktor.

Tabelle 3.11: GroupBy Ausführungszeit Optimierungen

Device	Ohne Auffüllen	Mit Auffüllen	Speedup
CPU	804	238	3,38
IGPU	1228	231	5,32
GPU	502	37	13,57

Die Primzahl als Kardinalität der Eingabedaten stellt dabei den Fall mit dem höchsten Performancegewinn dar. Für den Fall einer lediglich ungeraden Zahl, die teilbar im Intervall 1 bis 128 ist, reduziert sich der kombinierte Speedup auf 1,15 für IGPU und CPU, sowie 2,34 für GPU. Die Optimierungen ergeben in keinem Fall eine Verschlechterung und in der überwiegenden Mehrzahl der Fälle eine Verbesserung der Ausführungszeit. Es ist somit sinnvoll, die veränderten Kernel auf jedem der drei Devices zu verwenden.

4 ERGEBNISSE

Im nachfolgenden Kapitel werden Strategien zur Optimierung von OpenCL Code erläutert, die bei der Optimierung der drei Datenbankoperatoren Selection, Join und GroupBy gewonnen wurden.

4.1 OPENCL BEST PRACTICES

Die gewonnen Optimierungsstrategien erstecken sich sowohl auf den Hostcode, als auch auf den Kernelcode.

4.1.1 Threads Auffüllen

Als Auffüllen wird hierbei das Auffüllen der global size auf eine Teilbarkeit von 128 bezeichnet. Durch die Teilbarkeit von 128, kann die local size auf 128 gesetzt werden. Dadurch werden je work group 128 Threads ausgeführt. Abhängig von der Architektur des Devices und dem Aufbau des Kernels, fällt der Performancegewinn unterschiedlich stark aus. Über alle Kernel profitierte die GPU am stärksten vom Auffüllen der local size. Dies liegt darin begründet, dass die GPU jede Instruktion genau 32 mal ausführt. Eine local Size von 1 ergibt dabei 31 Instruktionen, die verworfen werden müssen.

Das Auffüllen der global size alleine führt zu ungültigen Speicherzugriffen, falls diese von der Thread ID abhängen. Um ungültige Speicherzugriffe zu vermeiden gibt es zwei Möglichkeiten.

Im Hostcode können die verwendeten *cl_mem* Objekte ebenfalls aufgefüllt werden. Dabei muss vermieden werden, die aufgefüllten Werte für weitere Berechnungen zu verwenden, da sonst ein falsches Ergebnis entsteht.

Alternativ können die *cl_mem* Objekte nicht aufgefüllt und im Kernelcode eine If-Abfrage eingebaut werden. Die Abfrage lässt jeden Thread ohne Berechnung terminieren, dessen Thread-ID

durch das Auffüllen entstanden ist. Es hängt vom Kernelcode, der gesamten Anzahl Berechnungen und dem verwendeten Device ab, welche Strategie besser ist.

4.1.2 Vektorisierung

OpenCL verfügt über die Möglichkeit den Kernelcode zu Vektorisieren. Vektorisierung bedeutet eine Umwandlung des sequenziellen Abarbeitens einer Schleife in eine SIMD Operation, indem mehrere Schleifendurchläufe parallel verarbeitet werden. Dies kann entweder explizit durch Angabe im Kernelcode geschehen oder implizit durch eine automatische Vektorisierung während des Kompilierens. Der Compiler schätzt ab, unter welchen Bedingungen sich eine Vektorisierung die Ausführungszeit verringert. Falls der Compiler keine automatische Vektorisierung vornimmt, ergibt das explizite Vektorisieren im Regelfall einen Performanceverlust.

Vorraussetzung für die Vektorisierung ist die Verwendung von vektorisierbaren Statements innerhalb einer Schleife. Die Größe der Vektorisierung hängt vom Device ab und beträgt bei den drei getesteten Devices stets 4.

Um Compiler-seitige Vektorisierung zu erreichen, muss der Verzweigungsgrad möglichst niedrig sein. Ebenso sollte die Schleife sich außerhalb der Verzweigungen befinden, die durch If-Bedingungen entstehen.

Für den Selection-Operator wurde der Kernelcode so verändert, dass er automatisch vektorisiert wurde. Dadurch lässt sich ein Performancegewinn erzielen, ohne Änderungen im Hostcode vorzunehmen.

4.1.3 Speicherzugriff

Durch Auffüllen ohne Anpassungen im Hostcode oder fehlerhafte Verwaltung von Indices im Kernelcode kann es zu einem ungültigen Speicherzugriff kommen.

In einer solchen Situation verhalten sich die Devices von Intel und das Nvidia Device grundverschieden. Bei einem ungültigen lesenden Speicherzugriff führt die GPU den Kernel weiter aus und schreibt auch das korrekte Ergebnis zurück. Die Intel Devices terminieren beim ungültigen Speicherzugriff und geben einen Stacktrace auf der Konsole aus. Ebenso lässt sich weder Kernel- noch Hostcode mit Visualstudio debuggen, falls ein ungültiger Speicherzugriff erfolgt.

Bei einem ungültigen schreibenden Speicherzugriff zeigt die Intel Hardware das selbe Verhalten. Die GPU berechnet trotzdem das korrekte Ergebnis, sofern der Speicherzugriff nur wenige Indices zuviel benutzt. Für eine höhere Anzahl ungültiger Schreibvorgänge stürzt auch der Kernel auf der GPU ab. Es ist in keinem Falle eine sinnvolle Strategie, ungültige Speicherzugriffe zuzulassen, da nicht absehbar ist, wann es zu Abstürzen führt.

4.1.4 Speicherarten

Das explizite Angeben des *__local* Prefixes führt zu Performanceverlusten oder einer gleichbleibenden Performance. Falls kein Prefix im Kernelcode angegeben wird, wird Private Memory verwendet. Diese ist nur für den jeweiligen Thread benutzbar. Es hängt vom Device ab welche Speicherart dafür verwendet wird. Sofern Speicherzugriffe übergreifen innerhalb der *workgroup* auf das Speicherobjekt erfolgen sollen, kann Private Memory nicht verwendet werden.

Local Memory sollte verwendet werden, wenn Daten zwischen Threads einer *workgroup* ausgetauscht werden müssen. Local Memory ist im Regelfall schneller als Global Memory.

Falls Local Memory statt Global Memory verwendet werden soll reduziert sich je nach Device die Größenordnung des verfügbaren Speichers von Gigabyte auf Kilobyte. Das muss bei der Abschätzung der Größe von Eingabedaten bedacht werden.

Mithilfe der Funktion *clGetKernelWorkGroupInfo* können zusätzliche Informationen in Abhängigkeit des verwendeten Kernels gewonnen werden. Dabei kann die Menge verwendeter Local Memory und Private Memory angezeigt werden. Ebenso kann die geschätzte sinnvollste work group size ausgegeben werden. Das Intel Plugin gibt für alle drei genannten Funktionen einfach immer die Zahl 0 zurück. Das ist sachlich falsch und damit entfällt leider die Hilfe durch diese Funktion zur Performance-Optimierung.

4.1.5 Global Memory Caching

Ein Zugriff auf Global Memory hat bis zu 200 Taktzyklen Verzögerung zur Folge. Daher kann es sinnvoll sein eine Berechnung zwei mal in 2 verschiedenen Kerneln auszuführen, anstatt sie einmal auszuführen und in Global Memory zu speichern. Der zusätzliche Zugriff auf Global Memory kostet dabei mehr Ausführungszeit, als durch die fehlende Berechnung gespart wird.

4.1.6 Local work size

Die Angabe der Local work size erfolgt entweder explizit oder wird dem Compiler überlassen werden. Falls die einzig mögliche angegebene Local Size 1 beträgt bringt es einen Performancegewinn, keine Local Size anzugeben und dem Compiler die Abschätzung der Local Size zu überlassen.

Je nach Device unterscheidet sich, was die performanteste Größe der Local Size ist. Generell empfehlenswert für jedes der getesteten Devices ist eine Größe der Local Size von mindestens 64. 128 ist auf der GPU performanter und führt zu keinem Performanceverlust auf CPU und IGPU, der größer als 1 ms ist. Der Messfehler betrug dabei ebenso 1 ms. Die Angabe einer konkreten Local Size kann sich stark positiv auf die Performance auswirken.

4.1.7 Kerneldesign

Um Kernelfunktionen übersichtlich entwickeln zu können, ist es sinnvoll die Funktionalität möglichst klein zu halten. Dadurch ist die Korrektheit der Berechnung der Kernelfunktion mittels Debugger besser nachvollziehbar. Ein erhöhter Verzweigungsgrad der Kontrollstrukturen hat besonders auf Devices mit wenigen parallelen Berechungseinheiten Performancenachteile. Zusätzlich erlaubt das Aufteilen in einzelne Kernelfunktionen compilerseitige Autovektorisierung einzelner Kernelfunktionen.

4.2 PROBLEME MIT OPENCL UND TOOLS

Die Produktpalette der Intel OpenCL Tools ist unübersichtlich. Der benötigte Intel C++ Compiler für OpenCL ist sowohl im Intel OpenCL SDK, als auch im Intel Parallel Studio vorhanden. Das Intel integrated Native Developer Experience plugin für Visual Studio enthält wiederrum das OpenCL SDK mit dem notwendigen Compiler, sowie Debug Funktionalität. Mit dem Intel Plugin und Visual Studio ist es möglich die Kernelfunktionen zu Debuggen. Falls sowohl das Nvidia Visual Studio als auch das Intel INDE Plugin installiert wurden, reicht eine Neuinstallation von Visual Studio nicht aus um Debugfunktionalität zu erhalten. Die einfachste Variante ist in diesem Falle eine Neuinstallation des Betriebssystems. Das Nvidia Plugin ermöglicht kein Profiling und kein Debug für OpenCL.

Die Debugfunktion ist dabei teilweise unzuverlässig. Auch wenn der Kernelcode ohne ungültige Speicherzugriffe korrekt berechnet wird ist er nicht immer debugbar. Windows Updates können zu permanentem Verlust der Debugfunktionalität führen. Updates des Intel Plugins können ebenso zum Verlust der Debugfähigkeit führen. Problematisch ist es, dass es oft keinen Anhaltspunkt gibt, warum Debug nicht möglich ist. Der Hostcode ist auch nicht stets debugbar mit Visual Studio. Dabei werden Haltepunkte einfach übersprungen.

Updates des INDE Plugins ändern das Debugverhalten von Visual Studio. Die Reihenfolge der Platform-IDs wird vertauscht im Debug Modus. Daher ändert sich bei identischem Host-Code die ausgewählte Platform zwischen Intel und Nvidia.

Die Installation des INDE Plugins beeinträchtigt Visual Studio für andere Einsatzzwecke. Werbung von Intel bei jedem Start von Visual Studio, sowie Popup-Fenster, die zu Umfragen auffordern, stören die Arbeit.

Als leichtgewichtiges Gegenstück zum Entwickeln von OpenCL wird der Intel Code Builder im INDE Plugin mitgeliefert. Dabei entfällt der Hostcode und es wird nur Kernelcode betrachtet. Aufgrund der angesprochenen Instabilität und der großen Varianz der Messwerte, ist er nicht geeignet zum produktiven Einsatz. Falls diese Probleme in Zukunft behoben werden, kann er eine Alternative zu Visual Studio darstellen, aufgrund der deutlich geringeren Einrichtungszeit eines OpenCL Projektes. Der Code Builder kann CPU und IGPU debuggen im Gegensatz zu Visual Studio, das nur CPU Debuggen kann.

Falls das OpenCL Programm ohne Debugfunktionalität nur ausgeführt werden soll ist dies problemlos und zuverlässig mit Visual Studio 2015 und dem INDE Plugin möglich. Die hohe Anfäl-

ligkeit des INDE Plugins, seine Debugfähigkeit zu verlieren, gestaltet das Einrichten eines größeren OpenCL Projektes aufwändig. Da Updates des Plugins, sowie Betriesbssytem-Updates zu einem Verlust der OpenCL-Funktionalität von Visual Studio führen kann, ist es sinnvoll auf diese zu verzichten. Falls für andere Aufgaben Visual Studio oder aktuelle Updates notwendig sind, muss dies auf einem anderen Computer geschehen.

5 VERWANDTE ARBEITEN

Die Quelle der untersuchten Operatoren ist das Ocelot-Plugin für die Datenbank MonetDB. In [Heimel et al., 2013] wird das Design des Plugins beschrieben. Ocelots Anbindung an MonetDB erfolgt über den Ocelot Memory Manager, der mit dem MonetDB Storage Layer verbunden ist. Für die Verwaltung der OpenCL Anbindung sorgt das Modul OpenCL Context Management. Das Plugin erweitert dabei MonetDB und benutzt Kernkomponenten von MonetDB. Die in von Ocelot unterstützen Opertoren sind limitiert auf Selection, Join, Projektion, Aggreation und GroupBy auf Integer- und Fließkomme-Datentypen. Ocelot ist nicht in der Lage mehrere Operatoren parallel auszuführen. Der Grund dafür ist, dass Ocelot es nicht gestattet mehrere cl_context gleichzeitig zu verwenden. Je Operator ist ein eigener cl_context notwendig bei paralleler Verarbeitung. Das parallele Ausführen von mehreren Operatoren wird als weitere Verbesserungsmöglichkeit angegeben.

[Weber et al., 2012] stellen die Specmaster vor, eine OpenCL basierte Datenbank für das Durchsuchen von Protein-Informationen. Dabei wird die Performance von Suchoperationen auf drei Devices vorgestellt: Nvidia Gtx 480, AMD 7970 und Intel Sandy Bridge E5-2680. Als sinnvolle Orientierungsgröße der Local Work size für den CPU wird dort eins angegeben. Für die Operatoren, die in dieser Arbeit untersucht wurden, verschlechtert eine Local Work size von eins die Performance der CPU.

[Karnagel et al., 2015] betrachten GroupBy und Aggregatsfunktionen, die mithilfe von CUDA auf eine Nvidia GPU ausgeführt werden. Dabei werden hash- und sortierungs- basierte GroupBy Implementierungen verglichen. Durch experimentellen Nachweis wird unter Anderem gezeigt, dass die hashbasierte GroupBy Variante im Regelfall performanter ist. Es wird besonders auf Anomalien der Ausführungszeiten beim Skalieren der Eingabedaten des Operators eingangen.

[Rosenfeld et al., 2015] untersuchen verschiedene Varianten für Aggregation und Selection in Datenbanken auf heterogener Hardware. Parameter der Variationen waren dabei: Speicherzugriff, Schleifenauflösung, Instrukionslevel-Parallelität, Vektorbreite, Elemente pro Thread und Work group size. Durch Variation der Parameter entstehen mehrere Tausend Codevarianten. Die Auswahl der besten Variante hängt stark vom Device ab. Es wird ein wahrscheinlichkeitsbasierter multi-armed bandit-Algorithmus als Ausgangspunkt für die Auswahl der optimalen Variante vorgeschlagen. Aus einer Codebasis soll durch Variation von Parametern der je performantes-

te Kernel je Device ausgewählt werden. Damit entfällt manuelle Optimierung von Kerneln für einzelne Devices.

[Coplin and Burtscher, 2015] untersuchen den Einfluss von Quellcode-Optimierungen auf den Energieverbrauch einer GPU. Gegenstand der Optimierungen sind zwei n-body Algorithmen, die sich in ihrem Speicherzugriffs-Verhalten unterscheiden. Dabei wird ein Nvidia Tesla K20c Device verwendet. Die Algorithmen sind in CUDA implementiert. Es wird gezeigt, dass einige Optimierungen die Ausführungszeit auf Kosten des Energieverbrauchs erhöhen und andersum. Damit ist es möglich die betrachteten Algorithmen je nach Bedarf auf Ausführungszeit oder Energieverbrauch zu optimieren.

6 AUSBLICK

In dieser Arbeit wurden die Datenbank-Operatoren Selection, Join und GroupBy losgelöst vom Ocelot Plugin untersucht und hinsichtlich ihrer Ausführungszeit optimiert. Dabei wurden drei Devices verwendet die in Kapitel 2.1.4 spezifiziert sind. Daraus wurden generelle Strategien zur Performance-Optimierung von OpenCL Programmen entwickelt.

Eine Weiterführung dieser Arbeit wäre die Integration der veränderten Kernel in das Ocelot Plugin. Dafür ist eine Anbindung an den Ocelot Speichermanager notwendig, der in dieser Arbeit durch Zufallszahlen ersetzt wurde. Dadurch wird ein realitätsnaher Performancevergleich mit anderen Datenbanksystem ermöglicht. Für das Auffüllen der global work Size sind teilweise zusätzliche Speicheroperationen im Host-Code notwendig. Dabei sollte untersucht werden, inwiefern die Performance des Datenbanksystems davon beeinträchtigt werden kann.

Um eine breitere Einschätzung der Performance der Operatoren zu erstellen, sind weitere Testreihen auf anderen Devices möglich. Dabei kann getestet werden, ob unterschiedliche Teilbarkeiten der Local work size zu verschiedenen Resultaten auf unterschiedlichen Devices führen. Mögliche Devices wären unter anderem FPGAs oder Xeon Phi Parallelrechner.

Das Portieren der vorgestellten Operatoren von OpenCL 1.2 auf OpenCL 2.1 und die Verwendung der neuen Sprachfeatures stellt ebenfalls eine mögliche Fortführung dieser Arbeit dar. Dabei kann untersucht werden, inwiefern sich neue Sprachfeatures wie Direct Memory Mapping auf die Performance und Stabilität der Operatoren auswirken.

Ebenso kann untersucht werden, infwiefern sich verschiedene OpenCL-Compiler der selben Version auf die Ausführungszeit und Stabilität der Operatoren auswirken.

LITERATURVERZEICHNIS

[Coplin and Burtscher, 2015] Coplin, J. and Burtscher, M. (2015). Effects of source-code optimizations on gpu performance and energy consumption. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 48–58, New York, NY, USA. ACM.

[Heimel et al., 2013] Heimel, M., Saecker, M., Pirk, H., Manegold, S., and Markl, V. (2013). Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720.

[Intel, 2012] Intel (2012). Intel opensource hd graphics programmer's reference manuel. `http://files.renderingpipeline.com/gpudocs/intel/hd4000/IHD_OS_Vol1_Part1.pdf`. accessed 13.03.2016.

[Karnagel et al., 2015] Karnagel, T., Müller, R., and Lohman, G. M. (2015). Optimizing gpu-accelerated group-by and aggregation. In *VLDB*.

[Khronos, 2016a] Khronos (2016a). Opencl api 1.2. `https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/`. accessed 20.01.2016.

[Khronos, 2016b] Khronos (2016b). Opencl quick reference card. `https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf`. accessed 02.02.2016.

[Khronos, 2016c] Khronos (2016c). Supported devices opencl. `https://www.khronos.org/conformance/adopters/conformant-products`. accessed 20.01.2016.

[Nvidia, 2009] Nvidia (2009). Nvidia opencl best practices guide. `https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf`. accessed 05.02.2016.

[Nvidia, 2016] Nvidia (2016). Nvidia kepler gk110 architecture whitepaper. `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`. accessed 13.03.2016.

[Rosenfeld et al., 2015] Rosenfeld, V., Heimel, M., Viebig, C., and Markl, V. (2015). The operator variant selection problem on heterogeneous hardware. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015.*, pages 1–12.

[W3Schools, 2016] W3Schools (2016). Sql aggregate functions. `http://www.w3schools.com/sql/sql_functions.asp`. accessed 08.02.2016.

[Weber et al., 2012] Weber, R., Jenkins, D. D., Lineback, N., and Peterson, G. D. (2012). Accelerating tandem ms protein database searches using opencl. In *Proceedings of the 3rd International Workshop on Emerging Computational Methods for the Life Sciences*, ECMLS '12, pages 59–63, New York, NY, USA. ACM.

A ANHANG

A.1 INHALT DES DATENTRÄGERS

Mit dieser Arbeit wird ein Datenträger mit folgenden Inhalten abgegeben:

Die Messwerte werden nach Operatoren aufgeteilt.

- messungen/join.ods enthält alle genannten Messungen und Graphen des Join-Operators
- messungen/select.ods enthält alle genannten Messungen und Graphen des Selection-Operators
- messungen/groupby.ods enthält alle genannten Messungen und Graphen des GroupBy-Operators

Die verschiedenen Versionen des Quellcodes werden ebenfalls nach Operatoren aufgeteilt:

- source/select enthält alle Versionen von Host- und Kernelcode die für den Selection-Operator verwendet wurden.
- source/join enthält alle Versionen von Host- und Kernelcode die für den Join-Operator verwendet wurden.
- source/grp enthält alle Versionen von Host- und Kernelcode die für den GroupBy-Operator verwendet wurden.

A.2 DATENBANK-OPERATOR BEISPIELGRAPH

Mithilfe von DB2 9.5 wurde ein Beispiel konstruiert um das Zusammenspiel von Datenbankoperatoren zu verdeutlichen. Dabei wurden Zwei Tabellen mit fiktiven Daten gefüllt (je drei Tupel). Für die genauen Befehle wird auf folgende Dateien verwiesen:

- sql/inittables.sql Anlegen des Schemas
- sql/fillgehalt.sql Füllen der Tabelle Gehalt
- sql/fillpersonal.sql Füllen der Tabelle Personal
- sql/explain_commands.txt Die verwendeten Befehle um die Metadaten zu erzeugen, die unter anderem den Operatorgraphen enthalten.
- sql/output.txt Die erzeugten Metadaten
- sql/STMT.sql Das benutzte Datenbankstatement

Aus dem Graphen in output.txt wurde der Gerätename des verwendeten Computers als Tabellenprefix entfernt.