Karlsruher Institut für Technologie

Automatische OpenCL-Code-Analyse zur Bestimmung von Speicherzugriffsmustern

Bachelorarbeit

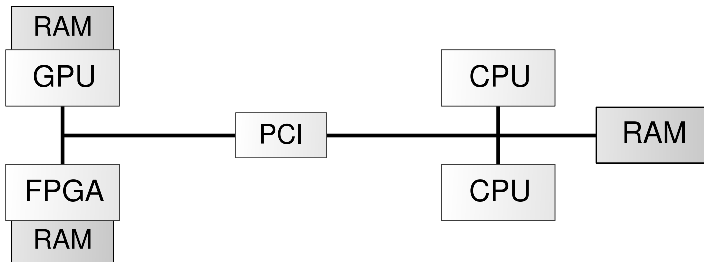
Moritz Lüdecke | 8. Juli 2014

INSTITUT FÜR TECHNISCHE INFORMATIK - LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELVERARBEITUNG

```
if (Setting->work_dim == 1) {
    id = get_global_id(0) - get_global_offset(0);
} else if (Setting->work_dim == 2) {
    id = (get_global_id(1) - get_global_offset(1)) * get_global_size(0)
        + (get_global_id(0) - get_global_offset(0));
} else {
    id = ((get_global_id(2) - get_global_offset(2)) * get_global_size(1)
        * get_global_size(0))
        + ((get_global_id(1) - get_global_offset(1))
            * get_global_size(0))
        + (get_global_id(0) - get_global_offset(0));
}
return id;
```

Motivation

Karlsruher Institut für Technologie



Programmierer muss angeben,

- ... auf welche Hardwarekomponenten der Code ausgeführt werden soll.
- ... welche Daten auf der jeweiligen Hardwarekomponente verarbeitet werden soll.

Motivation

KIT Karlsruher Institut für Technologie

Großes Ziel

- Alle Recheneinheiten ausnutzen
  - Wenn genug Tasks vorhanden sind: Task parallel ausführen
  - Ansonsten aufteilen

Vermeiden

- Doppelte Berechnungen
- Unnötige Kopiervorgänge

Ziel der Bachelorarbeit

- Abhängigkeit der Eingabedaten auflösen
- Daraus ein Speicherzugriffsmuster bilden

Motivation

Karlsruher Institut für Technologie

Großes Ziel

- Alle Recheneinheiten ausnutzen
  - Wenn genug Tasks vorhanden sind: Task parallel ausführen
  - Ansonsten aufteilen

Vermeiden

- Doppelte Berechnungen
- Unnötige Kopiervorgänge

Ziel der Bachelorarbeit

- Abhängigkeit der Eingabedaten auflösen
- Daraus ein Speicherzugriffsmuster bilden

Motivation

Karlsruher Institut für Technologie

Großes Ziel

- Alle Recheneinheiten ausnutzen
  - Wenn genug Tasks vorhanden sind: Task parallel ausführen
  - Ansonsten aufteilen

Vermeiden

- Doppelte Berechnungen
- Unnötige Kopiervorgänge

Ziel der Bachelorarbeit

- Abhängigkeit der Eingabedaten auflösen
- Daraus ein Speicherzugriffsmuster bilden

Beispiel: Matrixmultiplikation

Karlsruher Institut für Technologie

```
# C = A * B
# mit C[Zeile][Spalte]
for i = 0 to A.length − 1:
    for j = 0 to B[0].length − 1:
        C[i][j] = 0
        for k = 0 to A[0].length − 1:
            C[i][j] += A[i][k] * B[k][j]
```

$$\begin{pmatrix} 3 & 0 & -2 & 4 \\ 8 & 1 & 5 & 3 \\ 11 & 5 & 1 & 0 \\ 8 & 1 & 2 & 1 \end{pmatrix} * \begin{pmatrix} 1 & -2 & 3 & 4 \\ 5 & 0 & 1 & 0 \\ 2 & 0 & 0 & 5 \\ 3 & 2 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 11 & 2 & 9 & -2 \\ 32 & -10 & 25 & 54 \\ 38 & -22 & 38 & 49 \\ 20 & -14 & 25 & 41 \end{pmatrix}$$

Speicher A, Speicher B, Speicher A & B

OpenCL

- **Open C**omputing **L**anguage
- Einheitliches Architekturkonzept
- Aufteilung in Host- und Kernelcode

Host

- Code wird auf der CPU ausgeführt
- Verwaltet einzelne OpenCL-Geräte (Devices)
- Initialisiert Ausführung (Anzahl der Work-Items und Work-Groups)

Kernel

- Kann auf verschiedenen Architekturen ausgeführt werden
- Wird auf OpenCL-Geräten einmal ausgeführt
- Wird für jedes Work-Item ausgeführt
- OpenCL C (basiert auf ISO C99)

OpenCL

Karlsruher Institut für Technologie

- **Open C**omputing **L**anguage
- Einheitliches Architekturkonzept
- Aufteilung in Host- und Kernelcode

Host

- Code wird auf der CPU ausgeführt
- Verwaltet einzelne OpenCL-Geräte (Devices)
- Initialisiert Ausführung (Anzahl der Work-Items und Work-Groups)

Kernel

- Kann auf verschiedenen Architekturen ausgeführt werden
- Wird auf OpenCL-Geräten einmal ausgeführt
- Wird für jedes Work-Item ausgeführt
- OpenCL C (basiert auf ISO C99)

OpenCL: Beispielcode

Karlsruher Institut für Technologie

Quadratfunktion im OpenCL-Kernelcode

```
__kernel void square(__global float* input,
                     __global float* output,
                     const unsigned int count) {
    int i = get_global_id(0);
    if (i < count) {
        output[i] = input[i] * input[i];
    }
}
```

OpenCL: Work-Item-Funktionen

Karlsruher Institut für Technologie

- `uint get_work_dim ()`
- `size_t get_global_size (uint` *dimindx*`)`
- `size_t get_global_id (uint` *dimindx*`)`
- `size_t get_local_size (uint` *dimindx*`)`
- `size_t get_enqueued_local_size (uint` *dimindx*`)`
- `size_t get_local_id (uint` *dimindx*`)`
- `size_t get_num_groups (uint` *dimindx*`)`
- `size_t get_group_id (uint` *dimindx*`)`
- `size_t get_global_offset (uint` *dimindx*`)`
- `size_t get_global_linear_id ()`
- `size_t get_local_linear_id ()`

OpenCL: Work-Item-Funktionen

Karlsruher Institut für Technologie

- Alle Work-Item-Funktionen bauen auf fünf Faktoren auf
- Vier werden im Hostcode festgelegt

Feste Werte im Hostcode

- `cl_uint        work_dim`
- `const size_t *global_work_offset`
- `const size_t *global_work_size`
- `const size_t *local_work_size`

Iterierbarer Wert

- Aktuelle `global_work_id`

LLVM / Clang

Karlsruher Institut für Technologie

LLVM

- Compiler-Infrastruktur
- Fast alle OpenCL-Implementationen bauen auf LLVM auf
- „Bytecode“: LLVM IR (**I**ntermediate **R**epresentation)

Clang

- LLVM-Frontend
- C, C++, Objective-C und Objective-C++
- Clang AST
- APIs:
  - LibClang
  - Clang Plugins
  - LibTooling

LLVM / Clang

Karlsruher Institut für Technologie

LLVM

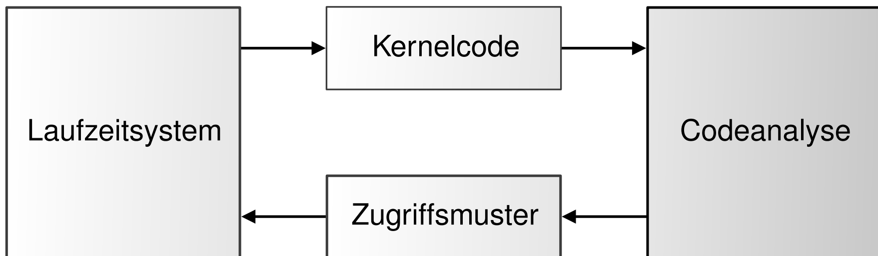
- Compiler-Infrastruktur
- Fast alle OpenCL-Implementationen bauen auf LLVM auf
- „Bytecode“: LLVM IR (**I**ntermediate **R**epresentation)

Clang

- LLVM-Frontend
- C, C++, Objective-C und Objective-C++
- Clang AST
- APIs:
  - LibClang
  - Clang Plugins
  - LibTooling

Compilierervorgang in LLVM

Bestimmung des Zugriffsmusters



Zielsetzung

- Eingabe: Kernelcode
- Ausgabe: Speicherzugriffsmuster

Bestimmung des Zugriffsmusters

Karlsruher Institut für Technologie

Möglichkeiten

1. Instrumentierung
2. LLVM Passes
3. Statische Codeanalyse mit Clang

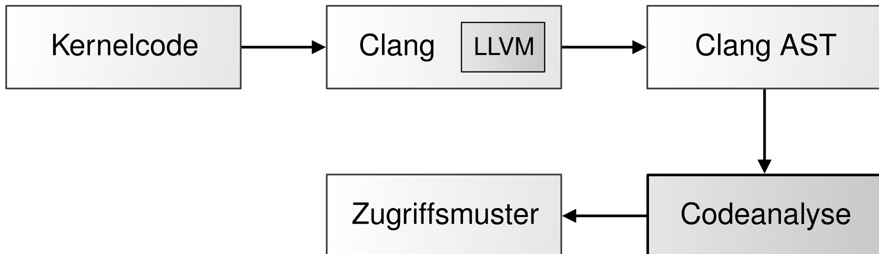
Bestimmung des Zugriffsmusters

Karlsruher Institut für Technologie

Möglichkeiten

1. Instrumentierung
2. LLVM Passes
3. Statische Codeanalyse mit Clang

Bestimmung des Speicherzugriffsmusters durch **Statische Codeanalyse mit Clang**:



Bestimmung des Zugriffsmusters

Karlsruher Institut für Technologie

Zur Erinnerung: **Quadratfunktion im OpenCL-Kernelcode**

```
__kernel void square(__global float* input,
                     __global float* output,
                     const unsigned int count) {
    int i = get_global_id(0);
    if (i < count) {
        output[i] = input[i] * input[i];
    }
}
```

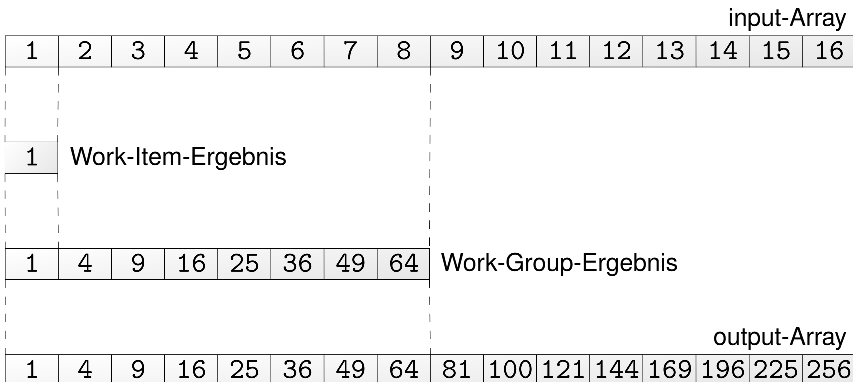
Rahmenbedingungen der Quadratfunktion

- Zwei Work-Groups mit je acht Work-Item
- Annahme: `count` > 15

Bestimmung des Zugriffsmusters

Rahmenbedingungen der Quadratfunktion

- Zwei Work-Groups mit je acht Work-Item
- Annahme: `count` > 15



Problematik unbekannter Größen

Karlsruher Institut für Technologie

```
int i = get_global_id(0) + 1;
if (i < count) {
    output[i] = input[i] * input[i];
}
```

Problem

- Generell: Rückgabewert der Work-Item-Funktion für jedes Work-Item unterschiedlich
  - ⇒ Jedes Work-Item liefert ein anderes Zugriffsmuster zurück
- Während der Codeanalyse ist der Rückgabewert einer Work-Item-Funktionen unbekannt

Problematik unbekannter Größen

Karlsruher Institut für Technologie

```
int i = get_global_id(0) + 1;
if (i < count) {
    output[i] = input[i] * input[i];
}
```

Lösung

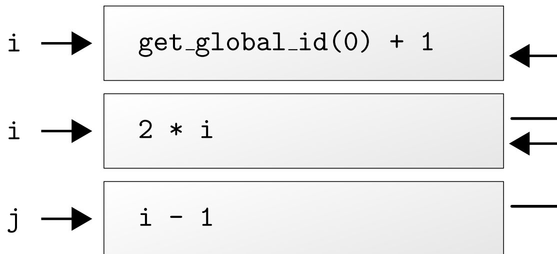
- Berechnungen müssen festgehalten werden, um sie später durchführen zu können
- Rückgabewerte von Work-Item-Funktionen und Variablenwert müssen Referenziert werden

$\Rightarrow$ Zugriffsmuster kann nur mit Kenntnis der Rahmenbedingungen im Hostcode berechnet werden

Erzeugung des Codebaums

```
int i = get_global_id(0) + 1;
i = 2 * i;
int j = i - 1;
```

Referenzliste:

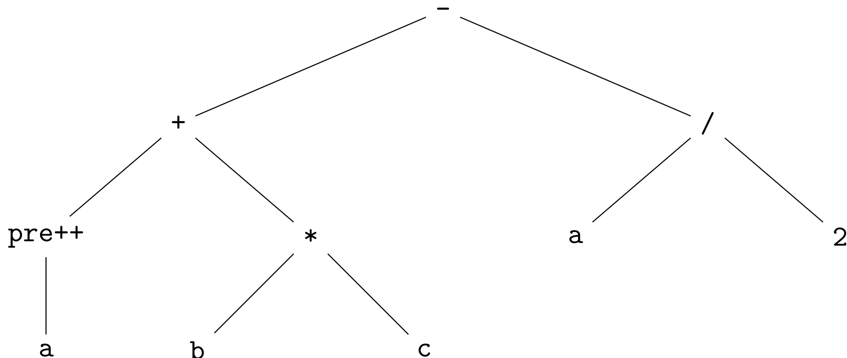


Erzeugung des Codebaums

```
(++a + b * c) − a / 2
```

Funktionalen Darstellung:

```
−(+(pre++(a), *(b, c)), /(a, 2))
```



Evaluation

Karlsruher Institut für Technologie

Rahmenbedingungen:

Parameter	Wert
global_work_size	32
local_work_size	8
global_work_offset	0
work_dim	1
clEnqueueTask	false

- Gleiche Rahmenbedingung für alle OpenCL-Kernels
- Kleine Werte zur Wahrung der Übersichtlichkeit

Evaluation

Karlsruher Institut für Technologie

Systemkomponenten zur Bestimmung der Laufzeit:

Prozessor	AMD FX 6100 6-Core
Taktfrequenz	3,3 GHz
Arbeitsspeichergröße	8 GB DDR3
Betriebssystem	Arch Linux x86_64
Linuxkernel	3.14.5-1-ARCH

Vectoraddition

Karlsruher Institut für Technologie

```
__kernel void vectorAdd(__global const float *a,
                        __global const float *b,
                        __global float *c) {
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Parameter	Typ	Zugriffsmuster
a	Array	R: 11111111.11111111.11111111.11111111
b	Array	R: 11111111.11111111.11111111.11111111
c	Array	W: 11111111.11111111.11111111.11111111

- Laufzeit der Codeanalyse inkl. Zugriffsmustergenerierung: 20 ms

Rodinia: backprop

Karlsruher Institut für Technologie

Kernelfunktion: `bpnn_adjust_weights_ocl`

Parameter	Typ	Zugriffsmuster
`delta`	Array	R: 01111111.1
`hid`	Variable	R
`ly`	Array	R: 01
`in`	None	
`w`	Array	R: 01111111.10001111.1111 W: 01111111.10001111.1111
Korrektur		R: ?1111111.1?... W: ?1111111.1?...
`oldw`	Array	R: 01111111.10001111.1111 W: 01111111.10001111.1111
Korrektur		R: ?1111111.1?... W: ?1111111.1?...

Rodinia: Übersicht

KIT – Karlsruher Institut für Technologie

Benchmark	Schleifen	If -Bedingung	Struct -Aufrufe	Bedingte Operator	Funktions-aufrufe	Laufzeit [ms]
backprop	X	X	-	-	-	28
bfs	X	X	X	-	-	25
cfd	X	X	X	-	X	55
gaussian	-	X	-	-	-	30
heartwall	X	X	-	-	-	35
hotspot	X	X	-	X	-	31
lavaMD	X	X	X	-	-	27
lud	X	X	-	-	-	52
myocyte	-	X	-	-	X	66
nn	-	X	X	-	-	18
nw	X	X	-	-	X	302
pathfinder	X	X	-	X	-	34
srad	X	X	-	-	-	35
streamcluster	X	X	X	-	-	39

Weitere Arbeiten

- Schleifen
- If-Bedingungen
- Structs
- Bedingte Operatoren
  - `(a < b) ? a : b`
- Verschachtelte Arrayaufrufe
  - `array[array[array[0]]]`
- Mehrdimentionale Arrayaufrufe
  - `array[x][y]`
- Funktionsaufrufe
- Verschachtelte Funktionsaufrufe
  - `function1(function2(function3(0)))`
- De- und Inkrementierung eines Arrayelements und des Indexwertes
  - `++array[++i]`
- Kernelfunktionsparameter berücksichtigen
- Sub-Groups-Funktionen
- Weitere Datenformate: Booleans, Gleitkommazahlen...

Zusammenfassung

Karlsruher Institut für Technologie

- Statische Codeanalyse Mithilfe LLVM und Clang AST
- Variabenlwerte werden intern durch einen Codebaum festgehalten, damit diese zu einem späteren Zeitpunkt berechnet werden können.
- Speicherzugriffsmuster wird in zwei Schritten ermittelt:
  1. Ermittlung Speicherzugriffsmuster eines Work-Items durch Codeanalyse
  2. Generierung Speicherzugriffsmuster eines ganzen OpenCL-Devices
- Erster Schritt muss dabei nur einmal durchgeführt werden
- Das Zugriffsmuster gibt an, welche Kernelfunktionsparameter und deren Arrayelemente während der Laufzeit beschrieben und gelesen werden.
- Es kann nicht immer das richtige Zugriffsmuster bestimmt werden