Parallele Algorithmen mit OpenCL

Universität Osnabrück, Henning Wenke, 2013-05-08

Aufräumen

- Ressourcen in umgekehrter Abhängigkeitsreihenfolge freigeben
- Objekte haben Reference-Count (RC), initial 1
- `clRelease<...>()` verringert RC um 1
- Falls Objekt nicht verwendet & RC 0, wird es gelöscht

Platform	-
Device	`clReleaseDevice()`
Context	`clReleaseContext()`
CommandQueue	`clReleaseCommandQueue()`
Program	`clReleaseProgram()`
Kernel	`clReleaseKernel()`
Buffer	`clReleaseBuffer()`

OpenCL API Fehlerbehandlung

- Fehler eines API-Calls wird durch Konstante mitgeteilt
  - `CL_SUCCESS` - kein Fehler, oder:
  - `CL_DEVICE_NOT_FOUND`
  - `CL_INVALID_VALUE`
  - `CL_INVALID_KERNEL_ARGS`
  - `CL_INVALID_OPERATION`
  - …
- Funktionsrückgabe oder über Parameter `errcode_ret`

```
int clFinish(                  // Rückgabe: Fehlerkonstante
    CLCommandQueue command_queue
)
```

```
CLProgram clCreateProgramWithSource(
    CLContext context, String string,
    IntBuffer errcode_ret // Rückgabe einer Fehlerkonstante
)
```

Beispiel

```
// Bisheriges Beispiel: Setze Parameter 2 des Kernels 'kernel'
int err = clSetKernelArg(kernel, 2, c);
System.out.println(err == CL_SUCCESS);            // Ausgabe: TRUE

// Setze Parameter 90 des Kernels 'kernel'
err     = clSetKernelArg(kernel, 90, c);
System.out.println(err == CL_SUCCESS);            // Ausgabe: FALSE
System.out.println(err == CL_INVALID_ARG_INDEX); // Ausgabe: TRUE
```

```
kernel void vec_add(
 global int* a,
 global int* b,
 global int* c){
    int i = get_global_id(0);
    c[i] = a[i] + b[i];
}
```

Erläuterungen zu Fehlern

- In OpenCL Spezifikation für jede Funktion mögliche Fehlerkonstanten aufgelistet:
  - http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/
- Dort Erläuterung der Konstanten für diese Funktion
- Beispiel: `clCreateBuffer` liefert `CL_INVALID_HOST_PTR`
- Nachschlagen liefert Ursachen für dieses Szenario:
  - "If `host_ptr` is `NULL` and `CL_MEM_USE_HOST_PTR` or `CL_MEM_COPY_HOST_PTR` are set in `flags`"
  - "If `host_ptr` is not `NULL` but `CL_MEM_COPY_HOST_PTR` or `CL_MEM_USE_HOST_PTR` are not set in `flags`"
- ... manchmal hilft das

Übersetzungsfehler

- Fehlermeldungen des „OpenCL C Compilers“ werden nicht auf Kommandozeile ausgegeben
- `clBuildProgram()` liefert Fehlerkonstante, z.B.: `CL_BUILD_PROGRAM_FAILURE`
- In diesem Fall: Hole Ausgaben mit `clGetProgramBuildInfo()`
- Hausaufgabe…

Laufzeitfehler

- Alles wie bisher initialisiert (a, b & c jeweils 10 elementig)
- Kernel abgewandelt

```
kernel void vec_add(
 global int* a,
 global int* b,
 global int* c){
    int i = get_global_id(0);
    c[i] = a[i] + b[i + 90000];
}
```

```
// ... Beispiel bis hier bis auf Kernel unverändert
clCreateProgramWithSource(...);
clBuildProgram(...);
clCreateKernel(...);

...

// Berechnung in Auftrag geben
int err = clEnqueueNDRangeKernel(queue, kernel,...);
System.out.println(err == CL_SUCCESS); // Ausgabe: TRUE,
                                       //  da Ausführung asynchron
...

err = clFinish(queue);
System.out.println(err == CL_SUCCESS); // Ausgabe: TRUE oder FALSE
// Hinweis: Pointer kennen Länge nicht. Aber: Zugriffsfehler möglich
```

Kapitel

OpenCL C Einführung

Allgemeines

- An C (ISO C99) angelehnte Sprache
  - *Informell: Ähnlich zu Java, aber ohne Objektorientiertheit*
- Zusätzlich Erweiterungen für parallele Algorithmen
- Kein dynamisches Allokieren von Speicher
- Keine Arrays variabler Länge
- Keine Rekursion
- Startpunkt immer Funktion mit Keyword `kernel`

Built-in Functions

- Allgemeine Funktionen
  - Z.B.: `dot(), sin(), max(), abs(), …`
  - http://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf
- Für Synchronisation & asynchrones kopieren
- Work-Item (≈Kernel-Instanz) Functions, z.B.:

```
// Liefert Index in der jeweiligen Index-Dimension D, D ∈ (0, 1, 2)
//  Wert kann in jedem Work-Item anders sein
int get_global_id(uint D)

// Liefert Indexanzahl in jeweiliger Index-Dimension D.
//  Wert in jedem Work-Item identisch
int get_global_size(uint D)

// Liefert ersten Index in jeweiliger Index-Dimension.
//  I.d.R. 0
int get_global_offset(uint D)
```

Daten: Address Space Qualifiers

		`global`	`const`	`local` (später)	`private` (default)
Host	Allocation	Dynamic	Dynamic	Dynamic	-
	Read/Write access	R / W	R / W	-	-
Kernel	Allocation	-	Static	Static	Static
	Read/Write access	R / W	R	R / W	R / W
	Visibility	global (alle Work-Items)	global	Work-group	Work-Item

```
kernel void vec_add(
 global int* a, global int* b, global int* c){
  private int i = get_global_id(0);
  c[i] = a[i] + b[i];
}
```

Beispiel

Matrixmultiplikation

Matrix aus Zahlen

- $m \times n$ Matrix ist rechteckiges Zahlenschema mit $m \cdot n$ Elementen
- Geeignet, einige Berechnungen elegant darzustellen
- Hinweise:
  - Spaltenvektor ~ $m \times 1$ Matrix
  - Zeilenvektor ~ $1 \times n$ Matrix
- Dann besteht Matrix aus:
  - n Spaltenvektoren, bzw.:
  - m Zeilenvektoren

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Ein Element:

$a_{i,j} = a_{zeile,spalte}$

Letztes Element:

$a_{m,n} = a_{maxZeile,maxSpalte}$

Matrix in linearem Speicher

- Reserviere linearen Speicher, dem Produkt aus Zeilen- & Spaltenzahl entsprechend
- Lege 2D → 1D Indextransformation fest

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

- Row-Major-Order: $index1D \leftarrow i \cdot n + j$

$$\{ a_{11}, a_{12}, a_{13}, a_{14}, a_{21}, a_{22}, a_{23}, a_{24}, a_{31}, a_{32}, a_{33}, a_{34}, a_{41}, a_{42}, a_{43}, a_{44} \}$$

- Column-Major-Order: $index1D \leftarrow j \cdot m + i$

$$\{ a_{11}, a_{21}, a_{31}, a_{41}, a_{12}, a_{22}, a_{32}, a_{42}, a_{13}, a_{23}, a_{33}, a_{43}, a_{14}, a_{24}, a_{34}, a_{44} \}$$

- Wahl theoretisch egal…
- …kann aber großen Einfluss auf Speicherzugriffsmuster haben

Matrixmultiplikation

- Möglich, wenn Spaltenzahl der linken mit Zeilenzahl der rechten Matrix identisch
- Dann ist l $\times n$ Matrix $\boldsymbol{C}$ Produkt aus $l \times m$ Matrix $\boldsymbol{A}$ und $m \times n$ Matrix $\boldsymbol{B}$.
- Ihre Komponenten sind:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj}$$

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \left(\begin{array}{c|c} c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} & c_{12} = a_{11} \cdot b_{12} + a_{12} \cdot b_{22} \\ \hline c_{21} = a_{21} \cdot b_{11} + a_{22} \cdot b_{21} & c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22} \end{array} \right)$$

Beispiel: Matrix Vektor Multiplikation

- Gegeben:
  - $m \times n$ Matrix $\boldsymbol{A}$, als Java `int[] A` in Row-Major-Order
  - $n \times 1$ Matrix $\boldsymbol{b}$, als Java `int[] b`
  - Ergebnis: $m \times 1$ Matrix $\boldsymbol{c}$, mit $\boldsymbol{c} = \boldsymbol{A} \cdot \boldsymbol{b}$, als `int[] c`

- Komponenten c_i von $\boldsymbol{c}$ sind: $c_i = \sum_{k=0}^{n-1} a_{ik} \cdot b_k$

```
//  Berechnung der ci mit Java
for(int i = 0; i < m; i++){
   int sum = 0;
   for(int k = 0; k < n; k++)
      sum += A[getIndexRowMO(i, k, n)] * b[k];
   c[i] = sum;
}
```

```
// Liefert 1D Row-Major-Order-Index des Elements ij einer
//  Matrix mit n Spalten
int getIndexRowMO(int i, int j, int n){
   return i * n + j;
}
```

Parallele Formulierung

```
//  Java-Beispiel (letzte Folie), umformuliert
for(int i = 0; i < m; i++)
   calcMatVecProductComponent(A, b, c, n, i);
```

```
// Berechnet und schreibt Komponente i des m x 1 Vektors c, mit c = A * b.
//  Dabei ist A eine m x n Matrix und b ein n x 1 Vektor
void calcMatVecProductComponent(int[] A, int[] b, int[] c, int n, int i){
   int sum = 0;
   for(int k = 0; k < n; k++)
      sum += A[getIndexRowMO(i, k, n)] * b[k];
   c[i] = sum;
}
```

```
//  Alternative: Berechne Funktion parallel (Pseudocode)
For each (i | i ∈ {0,…,m − 1}) in parallel do
   call calcMatVecProductComponent(A, b, c, n, i)
End
```

- Aufgabe: Formuliere `calcMatVecProductComponent` als Kernel
- Unterschied zu Vektoraddition?

Überführung in Kernel

Java

```
int getIndexRowMO(int i, int j, int n){
   return i * n + j;
}

void calcMatVecProductComponent(
 int[] A,
 int[] b,
 int[] c,
 int n,
 int i)
{


 int sum = 0;
 for(int k = 0; k < n; k++){
    sum += A[getIndexRowMO(i,k,n)]*b[k];
 }
 c[i] = sum;

}
```

```
// Ausführung m-mal sequentiell
for(int i = 0; i < m; i++)
   calcMatVecProductComponent(..., i);
```

OpenCL C

```
int getIndexRowMO(int i, int j, int n){
   return i * n + j;
}

kernel void calcMatVecProductComponent(
 global int* A,
 global int* b,
 global int* c,
 const  int  n
      )
{

 int i = get_global_id(0);
 int sum = 0;
 for(int k = 0; k < n; k++){
    sum += A[getIndexRowMO(i,k,n)]*b[k];
 }
 c[i] = sum;

}
```

```
// Ausführung m-fach parallel beantr.
clEnqueueNDRangeKernel(..., myKernel,
  m_,     // Parameter global_work_size
...);
```