Parallele Algorithmen mit OpenCL

Universität Osnabrück, Henning Wenke, 2013-05-15

Beispiel

Matrixmultiplikation (Fortsetzung)

Wiederholung: Formel

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

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

- Andere Bezeichnungen:

$$c_{row,col} = \sum_{k=0}^{A.cols-1\ (=B.rows-1)} a_{row,k} \cdot b_{k,col}$$

- Hinweis: $c_{row,col}$ ist Skalarprodukt aus Zeilenvektor row von $\boldsymbol{A}$ und Spaltenvektor col von $\boldsymbol{B}$

Indextransformationen

- 2D Index
  - $row : \{0, \dots, rows - 1\}$
  - $col : \{0, \dots, cols - 1\}$

$$\boldsymbol{A} = \overbrace{\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}}^{cols} \Big\} rows$$

- 2D → 1D Indextransformation
  - Row-Major-Order: $index1D \leftarrow row \cdot cols + col$
  - Liefert $index1D$: $\{0, \dots, rows \cdot cols - 1\}$
  - Elemente, gemäß 1D-Index in linearem Speicher angeordnet:

$$\{ a_{00}, a_{01}, a_{02}, a_{03}, a_{10}, a_{11}, a_{12}, a_{13}, a_{20}, a_{21}, a_{22}, a_{23}, a_{30}, a_{31}, a_{32}, a_{33} \}$$

- 1D → 2D Indextransformation (RMO)
  - $row \leftarrow index1D \,/\, cols$
  - $col \leftarrow index1D \,\%\, cols$

Komponentenberechnung mit OpenCL C

```
// 2D -> 1D Indextransformation. Wie letzte Woche.
int getIndexRowMO(int row, int col, int colCnt){
   return row * colCnt + col;
}

// Berechnet Komponente C_rowA,colB der Matrix C, mit C = A * B und liefert sie zurück
int calc_c_rowCol(
 int rowA, int colB,              // Zeilen- / Spaltenindex
 global int* A, global int* B,    // Matrix A und B (Row-Major-Order)
 int COLS_A, int COLS_B           // Spaltenzahlen Matrix A, B
 ){
   int sum = 0;
   for(int k = 0; k < COLS_A; k++) {
      sum += A[getIndexRowMO(rowA, k, COLS_A)] * B[getIndexRowMO(k, colB, COLS_B)];
   }
   return sum;
}
```

$$c_{row,col} = \sum_{k=0}^{A.cols-1} a_{row,k} \cdot b_{k,col}$$

Matrixmultiplikation mit OpenCL C

```
kernel void matrixMul_Index1d(// Berechnet je 1 Element von C = A * B mit 1D-Index
 global int* A, global int* B, global int* C,  // Matrizen A, B, C
 const int COLS_A, const int COLS_B            // Hinweis: COLS_B = COLS_C
){
    int cIndex1D = get_global_id(0);// Index zu ber. Elem {0,...,c.cols*c.rows-1}
    int colC = cIndex1D % COLS_B; // Zeilenindex und ...
    int rowC = cIndex1D / COLS_B; // ... Spaltenindex des zu berechnenden Elements
    C[cIndex1D] = calc_c_rowCol(rowC, colC, A, B, COLS_A, COLS_B); // Letzte Folie
}
```

```
clEnqueueNDRangeKernel((...),work_dim ← 1, global_work_size ← {c.cols * c.rows});
```

--- Oder ---

```
kernel void matrixMul_Index2d(// Berechnet je 1 Element von C = A * B mit 2D Index
 global int* A, global int* B, global int* C,  // Matrizen A, B, C
 const int COLS_A              // COLS_B kann optional  auch übergeben werden...
){
    int rowC = get_global_id(0);       // Zeilenindex  {0, ..., c.rows - 1}
    int colC = get_global_id(1);       // Spaltenindex {0, ..., c.cols - 1}
    int COLS_B = get_global_size(1);  // Hinweis: COLS_B = COLS_C
    int cIndex1D = getIndexRowMO(rowC, colC, COLS_B);
    C[cIndex1D] = calc_c_rowCol(rowC, colC, A, B, COLS_A, COLS_B); // Wie oben!
}
```

```
clEnqueueNDRangeKernel((...),work_dim ← 2, global_work_size ← {c.rows, c.cols});
```

Vergleich: 1D / 2D Indexkombinationen

- Beispiel: Berechne 3 x 2 Matrix C aus C = A * B, analog zur letzten Folie, mit:
  - A: 3 x 2 Matrix,
  - B: 2 x 2 Matrix

1D-Kernel: `clEnqueueNDRangeKernel((...), global_work_size ← {3 * 2});`

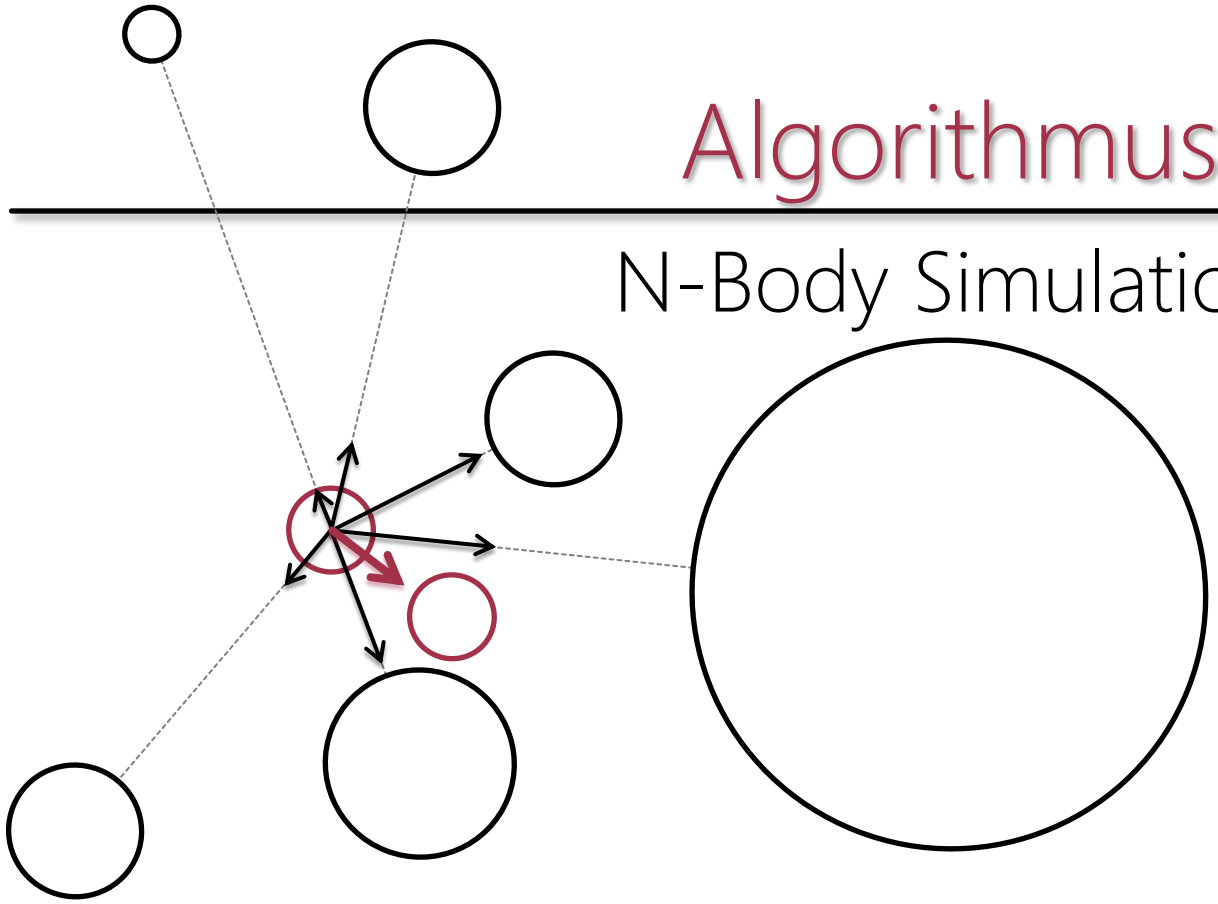
Instanz / Work Item	A	B	C	D	E	F
`cIndex1D ← get_global_id(0)`	0	1	2	3	4	5
`colC ← cIndex1D % 2`	0	1	0	1	0	1
`rowC ← cIndex1D / 2`	0	0	1	1	2	2

2D-Kernel: `clEnqueueNDRangeKernel((...), global_work_size ← {3, 2});`

Instanz / Work Item	A	B	C	D	E	F
`rowC ← get_global_id(0)`	0	1	2	0	1	2
`colC ← get_global_id(1)`	0	0	0	1	1	1
`cIndex1D ← rowC · 2 + colC`	0	2	4	1	3	5
`Entspricht Work Item des 1D-Kernels`	A	C	E	B	D	F

Algorithmus

N-Body Simulation



Quelle: GPU Gems 3

Überblick

- All-pairs approach: Jeder der n Körper interagiert mit jedem anderen → Laufzeit $O(N^2)$
- Wechselwirkung je zweier Teilchen unabhängig bestimmbar
- Wechselwirkungen der Teilchen z.B. abhängig von:
  - Distanz
  - Masse, z.B. für Gravitationskraft
  - Radius, etwa für Kollisionen
- Heute
  - Teilchen punktförmig
  - Nur Gravitationskraft
  - Massen aller Teilchen gleich

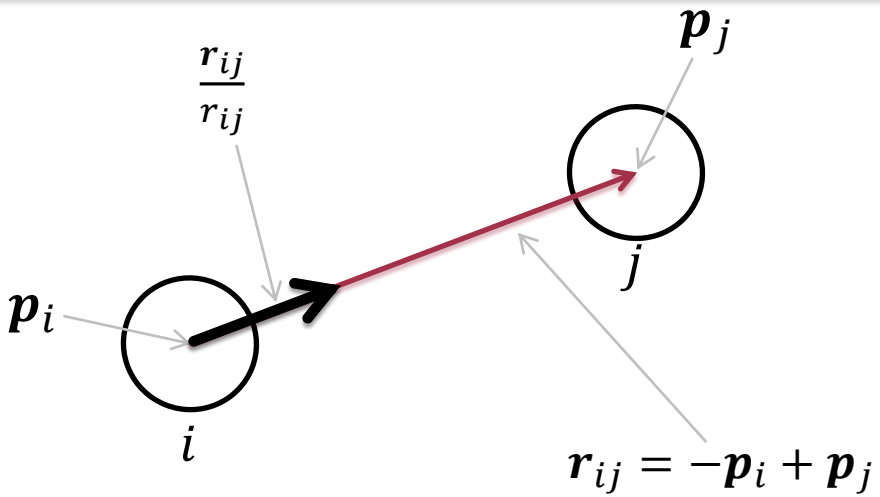
Simulation

- Anfänglich für alle Teilchen $i, i \in \{0, \dots, N-1\}$ gegeben:
  - Geschwindigkeit: $\boldsymbol{v}_i$
  - Position: $\boldsymbol{p}_i$
- Gesucht: Neue Position $\boldsymbol{p}_{i,neu}$
- Simulation findet in diskreten Zeitschritten Δt statt
- In jedem Simulationsschritt:
  - Berechne ∀ Teilchen i aktualisierte Geschwindigkeit und neue Position, ausgehend von Geschwindigkeit des Teilchens i und Positionen aller Teilchen j des vorherigen Iterationsschritts. Berechne dazu:
  1. Auf i wirkende Gesamtkraft $\boldsymbol{F}_i$
  2. Daraus Beschleunigung…
  3. Aktualisiere damit Geschwindigkeit …
  4. Neuen Ort $\boldsymbol{p}_{i,neu}$ wenn sich i Δt Zeiteinheiten lang mit aktualisierter Geschwindigkeit ausgehend von $\boldsymbol{p}_i$ bewegt
- Nächster Iterationsschritt: Vertausche $\boldsymbol{p}_{i,neu}$ und $\boldsymbol{p}_i$

Hinweis: Vektorielle Größen in Fettdruck

Paarweise wirkende Gravitationskraft

- Gegeben: Teilchen i und j, mit:
  - $\boldsymbol{p}_i$: Position von i
  - $\boldsymbol{p}_j$: Position von j
  - m_i: Masse von i
  - m_j: Masse von j



- Dann ist
  - $\boldsymbol{r}_{ij} = -\boldsymbol{p}_i + \boldsymbol{p}_j$ Vektor von $\boldsymbol{p}_i$ nach $\boldsymbol{p}_j$
  - Normiert: $\frac{\boldsymbol{r}_{ij}}{r_{ij}}$
- Für die auf i wirkenden Gravitationskraft von j gilt:
  - $\boldsymbol{f}_{ij} = G \cdot \frac{m_i \cdot m_j}{r_{ij}^2} \cdot \frac{\boldsymbol{r}_{ij}}{r_{ij}}$ ← Kraft wirkt von i ausgehend in Richtung j
  - $= -\boldsymbol{f}_{ji}$

- Kraft proportional zum Produkt der Massen
- Kraft nimmt quadratisch mit Abstand von i und j ab

Gravitationskonstante

Resultierende Kraft

- Berechne für jeden Körper i die resultierende Kraft aus Gesamtheit aller anderer Körper j
- Diese Kraft F_i ergibt sich aus Summe über alle paarweise zu berechnenden auf i wirkenden Kräfte $\boldsymbol{f}_{ij}$
  - $\boldsymbol{F}_i = \sum_{j=1}^{N} \boldsymbol{f}_{ij}\,, mit: i \neq j$

    $= G m_i \sum_{j=1}^{N} \left(\frac{m_j \boldsymbol{r}_{ij}}{r_{ij}^3} \right), mit: i \neq j$
- Problem: $\boldsymbol{F}_i \rightarrow \infty$, für $r_{ij} \rightarrow 0$
- Lösung: Größe $\varepsilon^2, mit\ \varepsilon > 0$ für Abschwächung hinzufügen
  - $\boldsymbol{F}_i \approx G m_i \sum_{j=1}^{N} \left(\frac{m_j \boldsymbol{r}_{ij}}{\left(r_{ij}^2 + \varepsilon^2\right)^{\frac{3}{2}}} \right)$
- Aufwand für ein $\boldsymbol{F}_i$: $O(N)$
- Gesamtaufwand: $O(N^2)$

Beschleunigung

- Es gilt: $\boldsymbol{F} = m \cdot \boldsymbol{a}$, mit $\boldsymbol{a}$: Beschleunigung

  $\Longrightarrow \boldsymbol{a} = \frac{\boldsymbol{F}}{m}$

- Damit ergibt sich für die resultierende Beschleunigung des Teilchens i:
  - $\boldsymbol{a}_i = \frac{\boldsymbol{F}_i}{m_i}$

Geschwindigkeit

- Es gilt: $\boldsymbol{a} = \frac{d\boldsymbol{v}}{dt}$, $\boldsymbol{v}$: Geschwindigkeit

  $\Longrightarrow d\boldsymbol{v} = \boldsymbol{a} \cdot dt$
- Simulation verwendet diskrete Zeitschritte Δt
  - Innerhalb eines Zeitschritts Δt sind alle Größen nicht zeitabhängig

    $\Longrightarrow \Delta\boldsymbol{v} = \boldsymbol{a} \cdot \Delta t$
  - Geschwindigkeitsänderung in einem Zeitschritt der Simulation
- Gegeben: Bisherige Geschwindigkeit $\boldsymbol{v}_i$ des Partikels i aus vorherigem Simulationsschritt
- Aktualisiere Geschwindigkeit für den aktuellen Simulationsschritt: $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i + \Delta\boldsymbol{v}_i$

Neue Position

- Bereits bekannt:
  - Position des Teilchens aus letztem Simulationsschritt: $\boldsymbol{p}_i$
  - Für diesen Simulationsschritt aktualisierte Geschwindigkeit: $\boldsymbol{v}_i$
  - Δt
- Es gilt:
  - $\boldsymbol{v} = \frac{d\boldsymbol{p}}{dt}$
  - $\Longrightarrow d\boldsymbol{p} = \boldsymbol{v} \cdot dt$
  - Diskret: $\Delta\boldsymbol{p} = \boldsymbol{v} \cdot \Delta t$
- Folglich gilt für die Ortsänderung des Teilchens i:
  - $\Delta\boldsymbol{p}_i = \boldsymbol{v}_i \cdot \Delta t$
- Setze neue Position $\boldsymbol{p}_{i,neu}$ des Teilchens i:
  - $\boldsymbol{p}_{i,neu} \leftarrow \boldsymbol{p}_i + \Delta\boldsymbol{p}_i$

Algorithmus

- Daten aller Teilchen liegen konsekutiv im Speicher:
  - `p[]`: $\{p_{0,x}, p_{0,y}, p_{0,z}, p_{1,x}, \dots, p_{N-1,y}, p_{N-1,z}\}$
  - `pNeu[]`: $\{pNeu_{0,x}, pNeu_{0,y}, pNeu_{0,z}, pNeu_{1,x}, \dots, pNeu_{N-1,y}, pNeu_{N-1,z}\}$
  - `v[]`: $\{v_{0,x}, v_{0,y}, v_{0,z}, v_{1,x}, \dots, v_{N-1,y}, v_{N-1,z}\}$

  Daten des Teilchens i = 0
- Kernel-Indizierung: 1, 2 oder 3D?
  - 1D: Naheliegend. Index über Partikel
  - 2D: Denkbar, wenn alle $\boldsymbol{f}_{ij}$ parallel berechnet werden sollen.
  - 3D: Für passende Räumliche Datenstruktur. Sonst ungeeignet, da Teilchen nicht diskret im Raum verteilt…
- Algorithmus (berechnet die $\boldsymbol{f}_{ij}$ für jedes i sequentiell):

```
While (Iterating)
  | For Each (Particle i | i ∈ {0, …, N-1}) in parallel do
  |   | call nBodyIter(p, pNeu, v, i, DELTA_T)
  | End
  | call visualize(pNeu)
  | toggle(p, pNeu)
End
```

- Implementieren von `nBodyIter` als OpenCL C Kernel: Hausaufgabe