Parallele Algorithmen mit OpenCL

Universität Osnabrück, Henning Wenke, 2013-06-26

Kapitel

Optimizations

Motivation: Parallele Algorithmen

- Lösbarkeit
  - Jedes durch PA lösbare Problem auch sequentiell lösbar
  - Man braucht parallele Algorithmen nicht.
- Speicher: PA benötigt mindestens gleiche Menge
- Komplexität: Bleibt oder wird schlechter
- Laufzeit:
  - Kann sich in Abhängigkeit der Parallelität stark verkürzen
  - Vektoraddition: $O(n) \rightarrow O(1)$ Parallel-Time
  - Radix Sort für Integer: $O(n) \rightarrow O(\log(n))$ Parallel-Time
  - … wenn in Praxis entsprechende Hardware genutzt wird
- Kurz: Wir wollen bekannte Probleme schneller lösen
  - Theoretisch: Durch Parallelität
  - Praktisch: Durch Optimierungen für bestimmte Hardware, z.B. GPUs

Optimierung: Allgemeines

- Optimierung (hier) := Anpassung an Hardware
  - Typisch um Laufzeit i.d. Praxis zu verbessern
  - Kann theoretische Eigenschaften verschlechtern
  - Kann auf anderer Hardware langsamer / unausführbar werden
  - Verwandt: Anpassung an Daten
- OpenCL Code…
  - … ist portierbar
  - Optimierungen sind es nicht
  - Lesbarkeit kann leiden…
- Wir optimieren:
  - Für GPUs…
  - … von Nvidia
  - Für AMD GPUs: Projektthema?
  - Für CPUs: Projektthema?

Abschnitt

Speicherhierarchie

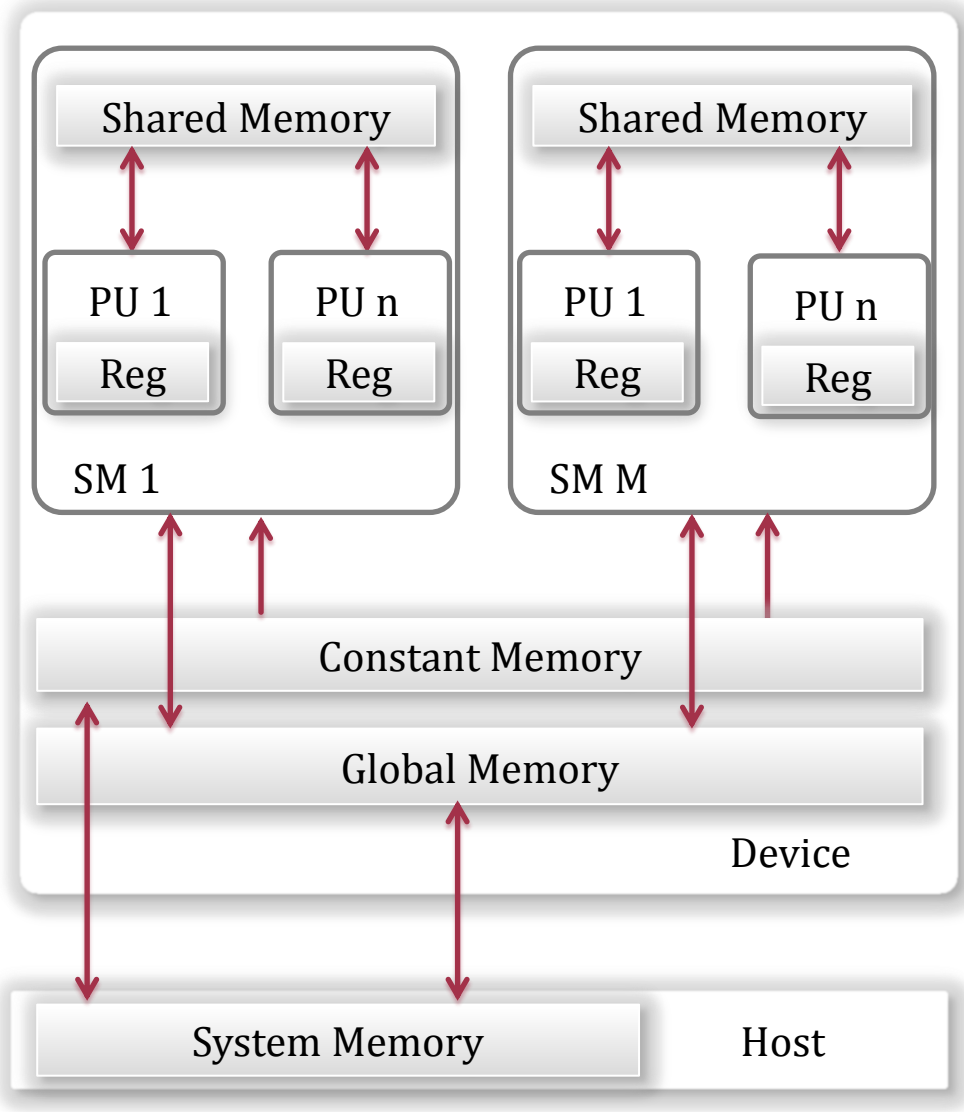
Limitierung durch Speicherbandbreite

- Definiere: Compute to Global Memory Access Ratio
- Matrixmultiplikation: 1.0
- Ausführung auf GTX 670:
  - Speicherbandbreite zum Global Memory: 192,3 GB/s
  - Bei 4 Byte je Wert mögliche Berechnungen: 48 GFLOPS/s
  - Aber: GTX 670 liefert bis 2460 GFLOPS/s
  - Durch Speicherbandbreite auf 2% der Maximalleistung limitiert
- Beobachtung: Jeder Wert wird redundant geladen
- Lösung: Speicherhierarchie
  - Off-Chip/On-Chip Memories, mit unterschiedlichen:
  - Bandbreiten, Latenzen, Größen
  - Zugriffsrechten & Zugriffsmustern

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

Architektur mit Nvidia Kepler (grob)

- Processing Unit (PU)
  - Führt Berechnungen aus
  - Arbeiten nicht unabhängig
  - Privater Speicher: Register (Reg)
- Streaming Multiprocessor (SM/SMX)
  - Arbeiten unabhängig
  - Shared Memory: Alle PUs eines SM haben Zugriff
- Global / Constant Memory
  - Global: Alle SM haben Zugriff
  - Constant: Alle SM haben Lesenden Zugriff
  - Host hat Zugriff

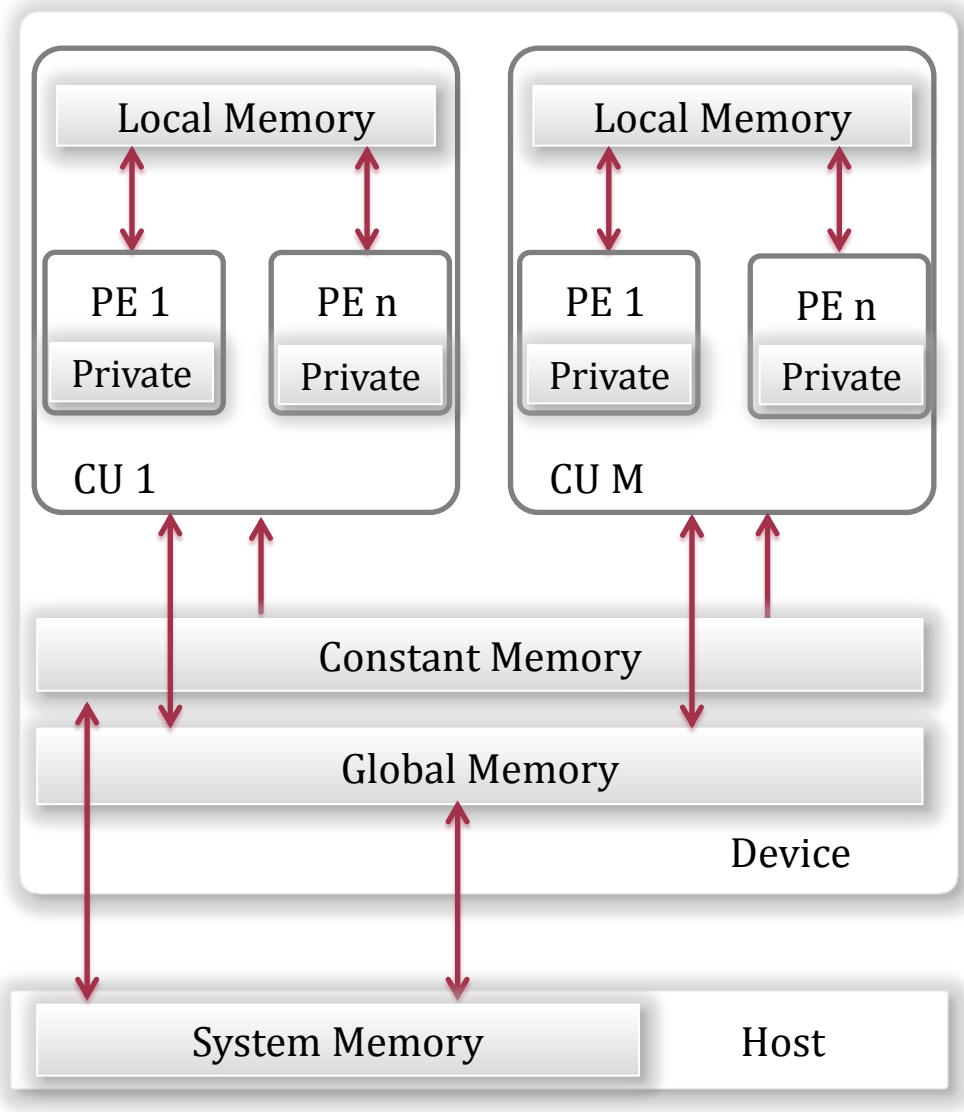


Ressourcen (Kepler / GTX 670)

- GPU (Nvidia GeForce GTX 670) enthält:
  - 7 Streaming Multiprocessors
  - 2 Gbyte Global memory
  - 64 Kbyte Constant Memory
- Streaming Multiprocessor (Nvidia Kepler GK 104)
  - Enthält 192 Processing Units
  - 48 Kbyte Shared Memory
  - Anzahl 32-Bit Register: 64000
  - Verwaltet bis 16 Work-Groups gleichzeitig $\Rightarrow$ $16 \cdot 1024 = 16384$ Threads
- Gesamt (GTX 670)
  - 1344 Processing Units
  - 114688 Threads

Mapping auf OpenCL

- Compute Unit (CU) ≅ SM
- Local Memory ≅ Shared Memory
- Private deklarierte Variablen in Registern hinterlegt
- Ausnahme: Arrays
- Processing Element (PE) ≅ PU
- Eine CU kann mehrere Work-Groups ausführen…
- …innerhalb derer via Local Memory Ergebnisse ausgetauscht werden können
- Work-Group wird niemals auf verschiedene CUs aufgeteilt



Local Memory

- „User Managed Cache“
- On-Chip Memory, trägt nicht zu Bandbreite zum Global Memory bei
- Größere Bandbreite & geringere Latenz als Global Memory
- Langsamer als Register
- Jede Work-Group hat unabhängiges Lokal Memory
- Ermöglicht schnellen Datenaustausch innerhalb einer Work-Group
- Lebenszeit: Kernellaufzeit
- Sehr begrenzt verfügbar

Beispiel: Vektoraddition

```
kernel void vecAddLocal(global int* a, global int* b, global int* c){
   local int aL[256]; // Deklariere lokale Daten mit Compile-
   local int bL[256]; //  Zeit Größe 256
   local int cL[256]; // Zugriff: Alle Work-Items der Work-Group

   int id  = get_global_id(0);
   int idL = get_local_id(0);

   aL[ idL ] = a[ id ];                  // Kopiere Daten...
   bL[ idL ] = b[ id ];                  // ... in Local Memory

   barrier(CLK_LOCAL_MEM_FENCE); // Stelle Laden der Daten sicher
   cL[idL] = aL[idL] + bL[idL];  // Führe Berechnung im Local Mem aus
   barrier(CLK_LOCAL_MEM_FENCE); // Stelle Rückschreiben des
                                 //  Ergebnisses in Local Mem sicher
   c[id] = cL[idL]; // Schreibe Ergebnis in Global Memory zurück
}
```

```
// Nötige Parameter für Ausführung (Auswahl)
// Hinweis: Jedes Work-Item lädt nur eigene Daten -> Local Mem unnötig
//          ... und verwendet nur eigene Daten    -> Barrier unnötig
clEnqueueNDRangeKernel((...),global_work_size ← {VEC_DIM},
                             local_work_size ← {256});
```

Beispiel: Alternative Deklaration

```
kernel void vecAddLocal(global int* a, global int* b, global int* c,
   local int* aL, // Deklariere lokale Daten. Lege Größe durch
   local int* bL, //  durch Host (pro Kernelaufruf) fest
   local int* cL // Zugriff: Alle Work-Items der Work-Group
){ // Restlicher kernel identisch mit letzter Folie
   int id  = get_global_id(0); int idL = get_local_id(0);
   aL[idL] = a[id]; bL[idL] = b[id];
   barrier(CLK_LOCAL_MEM_FENCE);
   cL[idL] = aL[idL] + bL[idL];
   barrier(CLK_LOCAL_MEM_FENCE);
   c[id] = cL[idL];
}
```

```
// Setze Größe in Bytes der Local Memory Kernelparameter eines Kernel
//  Objects "vecAddKernel" mit clSetKernelArg
clSetKernelArg(vecAddKernel, 3, 256 * 4);
clSetKernelArg(vecAddKernel, 5, 256 * 4);
clSetKernelArg(vecAddKernel, 6, 256 * 4);

// Kernelaufruf identisch mit letzer Folie
clEnqueueNDRangeKernel((...),global_work_size ← {VEC_DIM},
                             local_work_size ← {256});
```

Was stimmt hier (vermutlich) nicht?

```
kernel void vecAddLocal(
    global int* a, global int* b, global int* c,
    local int* aL, local int* bL, local int* cL
){
    int id  = get_global_id(0);
    aL[id] = a[id];
    bL[id] = b[id];
    barrier(CLK_LOCAL_MEM_FENCE);
    cL[id] = aL[id] + bL[id];
    barrier(CLK_LOCAL_MEM_FENCE);
    c[id] = cL[id];
}
```

- Zugriff auf Local Memory mit globalem Index
- Funktioniert nur…

  … wenn es genau eine Work-Group gibt, da dann

  … globaler & lokaler Index identisch

Was stimmt hier nicht?

```
kernel void vecAddLocal(
    global int* a, global int* b, global int* c,
    local int* aL, local int* bL, local int* cL
){
    int id  = get_global_id(0);
    int idL = get_local_id(0);
    barrier(CLK_LOCAL_MEM_FENCE);
    cL[idL] = aL[idL] + bL[idL];
    barrier(CLK_LOCAL_MEM_FENCE);
    c[id] = cL[idL];
}
```

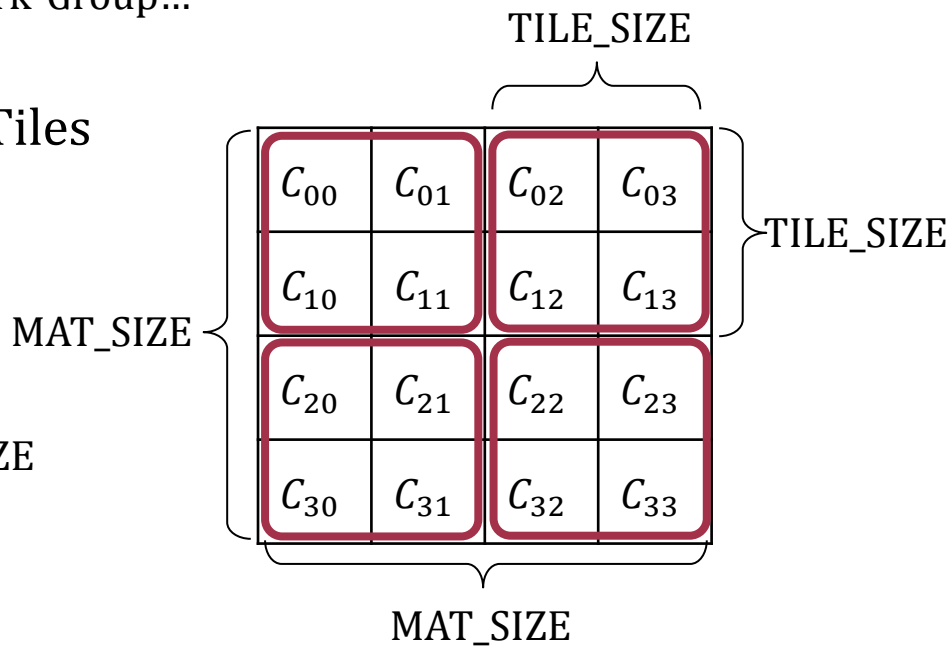
- Local Memory Variablen aL & bL werden gelesen, aber nie geschrieben
- Funktioniert niemals, da nur Kernel sie schreiben kann & ihre Lebensdauer der des Kernels entspricht

Beispiel

Tiled Matrixmultiplikation

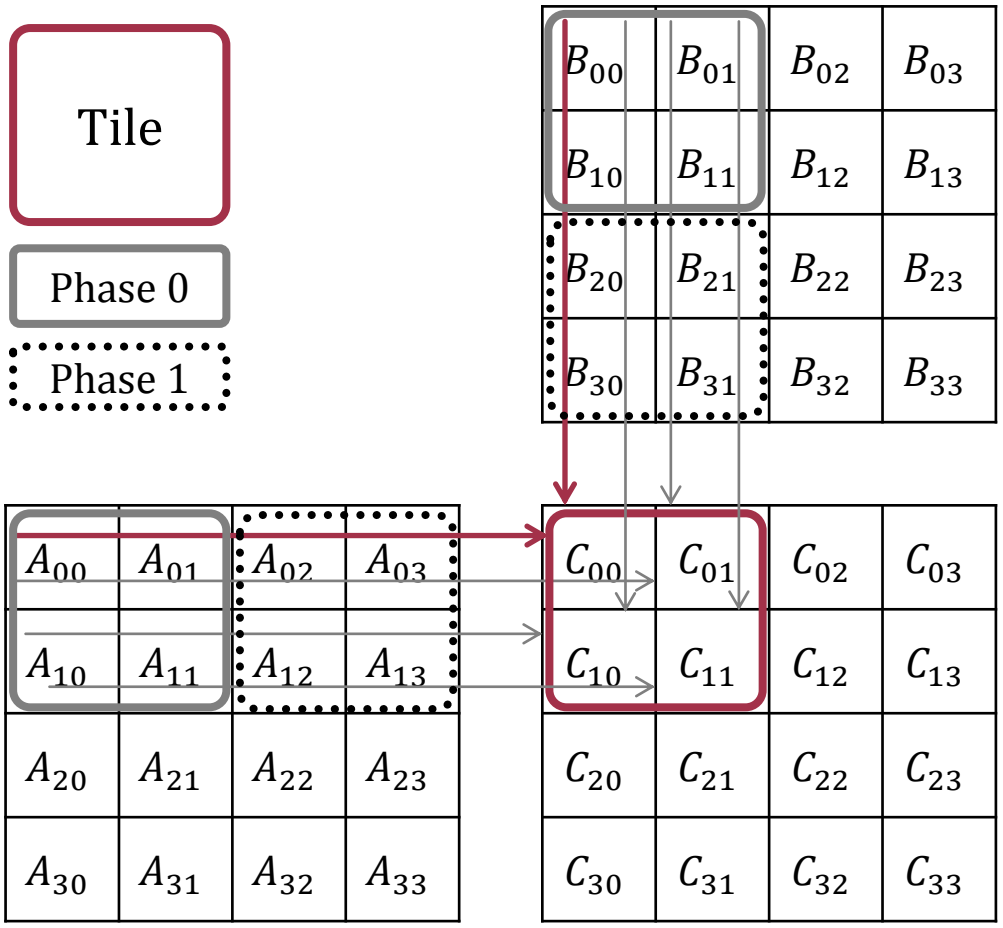
Tiles

- Idee: Lade Daten aus Global Memory in Lokal Memory & verwende mehrfach
- Als Ganzes?
  - Lokal Memory sehr begrenzt
  - Außerdem dann zwingend genau eine Work-Group…
  - … kann nur ein SM verwendet werden
- Aufteilen der Matrix in unabhängige Tiles
  - Verarbeitet durch je eine Work-Group
  - Jeweils eigenes Local Memory
- Matrizen A, B, C seien
  - Quadratisch
  - Größe: Zeilenzahl = Spaltenzahl = MAT_SIZE
  - MAT_SIZE sei Zweierpotenz
- Tile
  - Quadratisch
  - Größe: Zeilenzahl = Spaltenzahl = TILE_SIZE
  - TILE_SIZE sei Zweierpotenz
  - TILE_SIZE ≤ MAT_SIZE



Tile

Berechnung für ein Tile



- Für jedes Element existiert ein Work-Item
- Jedes Work-Item lädt in jeder Phase je einen Wert aus A & B in Local Memory
- Es gibt MAT_SIZE / TILE_SIZE Phases. (Hier: 2)
- Werden sequentiell abgearbeitet
- Jedes Work-Item berechnet in jeder Phase einen Summanden des Elements, welches es repräsentiert, aus Daten des LM
- Alle für ein Tile benötigten Daten werden genau einmal in LM geladen
- Hier: 16 statt 32 Werte je Tile aus GM laden

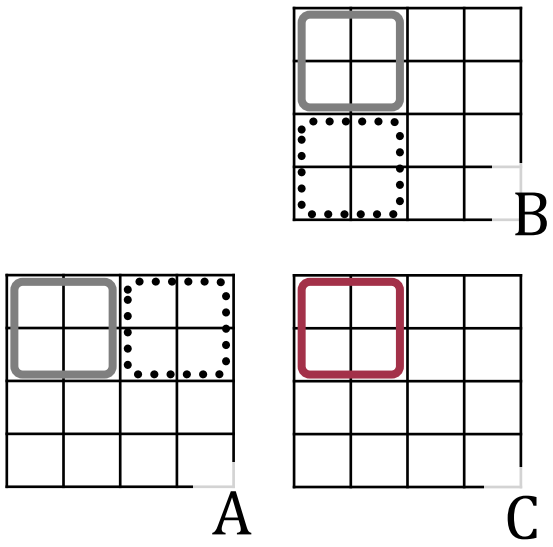
Work Item A berechnet: $C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30}$

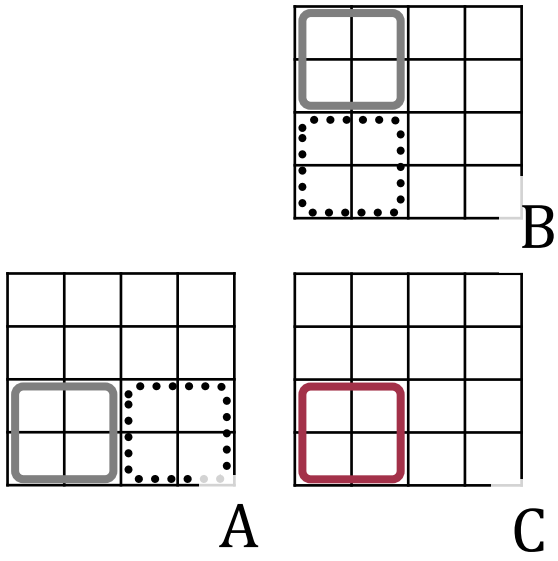
Work Item B berechnet: $C_{01} = A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21} + A_{03}B_{31}$

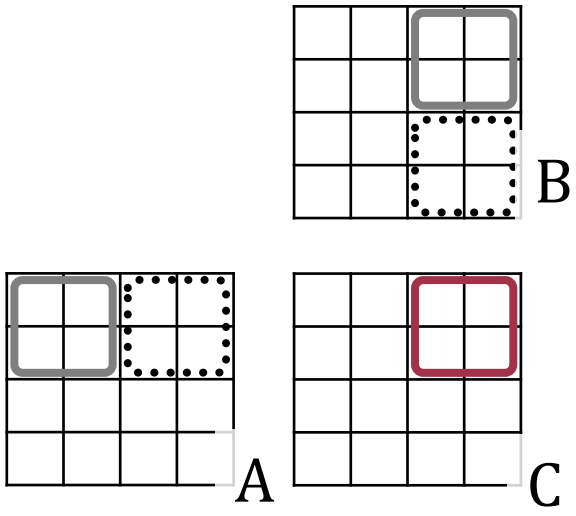
Work Item C berechnet: $C_{10} = A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20} + A_{13}B_{30}$

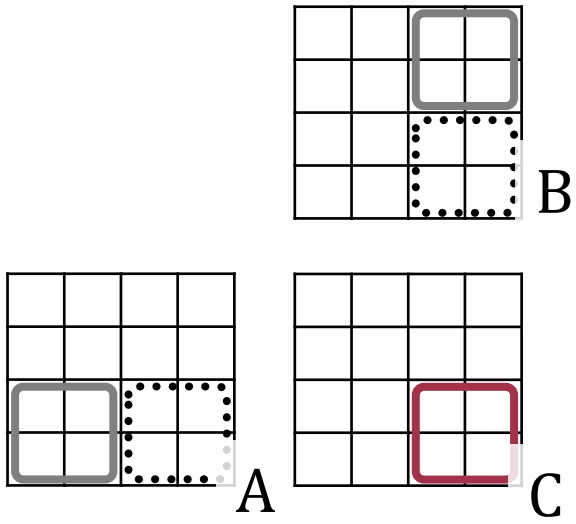
Work Item D berechnet: $C_{11} = A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$

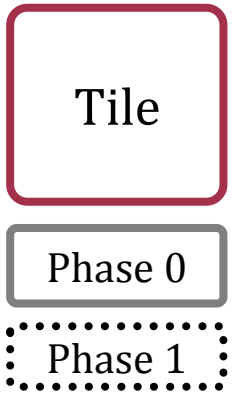
Alle Tiles











Synchronisation & Parallelität

Tile

Phase 0

Phase 1

B_{00}	B_{01}	B_{02}	B_{03}
B_{10}	B_{11}	B_{12}	B_{13}
B_{20}	B_{21}	B_{22}	B_{23}
B_{30}	B_{31}	B_{32}	B_{33}

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}

C_{00}	C_{01}	C_{02}	C_{03}
C_{10}	C_{11}	C_{12}	C_{13}
C_{20}	C_{21}	C_{22}	C_{23}
C_{30}	C_{31}	C_{32}	C_{33}

$$
\begin{aligned}
C_{00} &= A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} + A_{03}B_{30} \\
C_{01} &= A_{00}B_{01} + A_{01}B_{11} + A_{02}B_{21} + A_{03}B_{31} \\
C_{10} &= A_{10}B_{00} + A_{11}B_{10} + A_{12}B_{20} + A_{13}B_{30} \\
C_{11} &= A_{10}B_{01} + A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}
\end{aligned}
$$

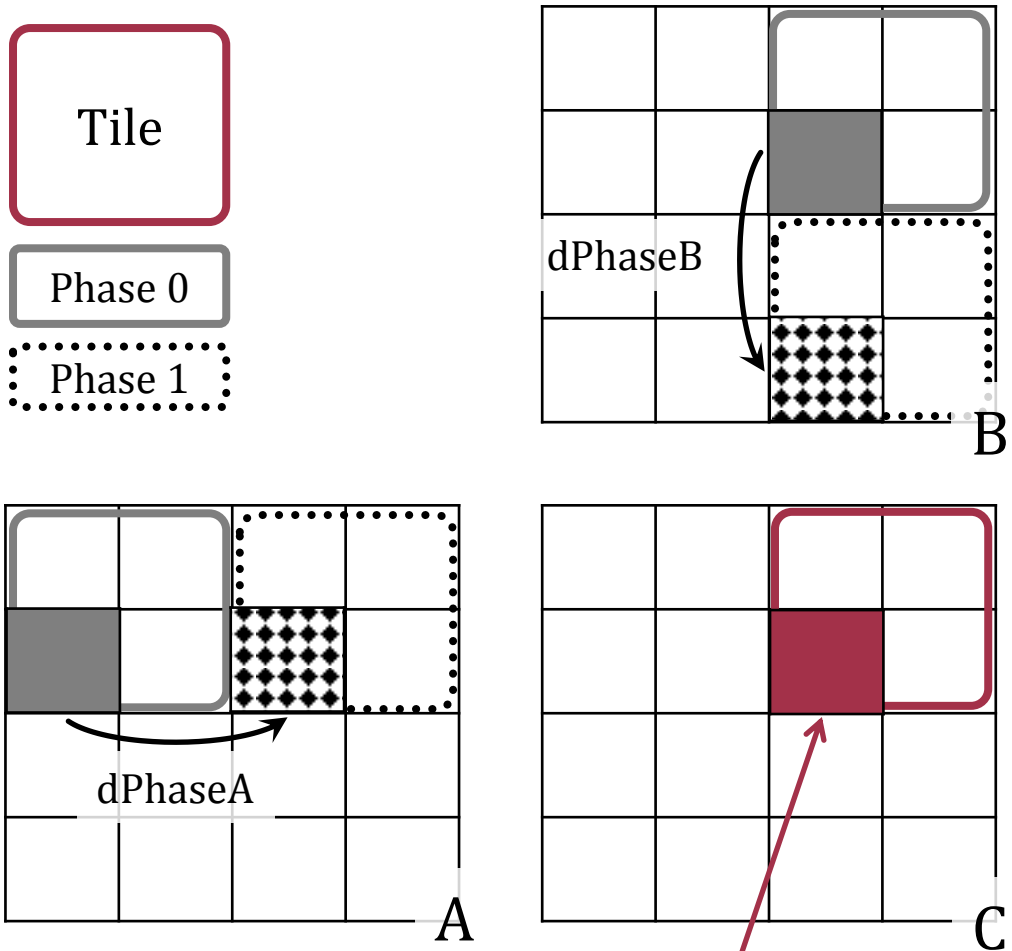
Innerhalb eines Tiles

- Daten jeder Phase aus A & B überschreiben Daten vorheriger Phase im Local Memory
- Synchronisation
  - In jeder Phase:
  - … vor Datenladen und …
  - … nach Datenladen
  - ⇒ Jedes Tile muss in eigener Work-Group ausgeführt werden
- Parallelität: Alle Work Items

Verschiedene Tiles

- Synchronisation: /
- Parallelität: Entspricht Tile-Anzahl

Datenladeschema



- x=2
- y=1
- lX=0
- lY=1

- Koordinaten eines Elements
  - x, y: Global in Matrix. Hier in (0, 3)
  - lX, lY: Lokal in Tile . Hier in (0, 1)
- Erstes zu ladendes Element
  - Hinweis: Linearer Speicher
  - Aus A: y · MAT_SIZE + lX
  - Aus B: lY · MAT_SIZE + x
- Erhöhe dann 1D-Zugriffs-Index in jeder Phase um:
  - dPhaseA = TILE_SIZE
  - dPhaseB = TILE_SIZE · MAT_SIZE
- Gesamt: Lade in Phase phase in Abhängigkeit von x, y, lX, lY:
  - A: y · MAT_SIZE + phase · TILE_SIZE + lX
  - B: (phase · TILE_SIZE + lY) · MAT_SIZE + x
- Aus Global Memory zu ladende Daten je Element:
  - 2 · PHASE_CNT = 2 · MAT_SIZE / TILE_SIZE
  - Statt: MAT_SIZE · 2
  - Ersparnis: Faktor TILE_SIZE

Algorithmus

```
Data: Matrix A, B, C // A,B initialisiert
// Konstanten, Nebenbedingungen: Siehe vorherige Folien
For each (Tile t) in parallel do
    For each (Elem e of t) in parallel do
        // e kapselt Indices x, y, lX, lY der vorherigen Folien
        cXY ← 0 // Initialisiere mit e korrespondierendes Matrixelement mit 0
        For each (Phase p, p in (0, ... ,  MAT_SIZE / TILE_SIZE - 1)) do
            aTileOfA[e] ← A[calcIndexA(e, p, TILE_SIZE, MAT_SIZE)] // Kopiere Daten
            aTileOfB[e] ← B[calcIndexB(e, p, TILE_SIZE, MAT_SIZE)] //  in Local Mem
            // Stelle Vorhandensein der Daten des Tiles t in LM sicher
            <Synchronise Tile t>
            cXY ← cXY + TILE_SIZE Summanden der Phase p des Skalarprodukts
            // Keine neuen Daten laden, bevor Berechnungen abgeschlossen
            <Synchronise Tile t>
        End
    End
End
```

- Für Ausführung durch OpenCL Kernel muss gelten:
  - Work-Items: MAT_SIZE * MAT_SIZE
  - Work-Groups: Muss Anzahl der Tiles entsprechen, also:

    MAT_SIZE * MAT_SIZE / (TILE_SIZE * TILE_SIZE)
- Wir verwenden 2D-Kernel-Indizierung

OpenCL Kernel (2d Index)

```
#define TILE_SIZE 16
kernel void matMul_Tiled(
 global float* A, global float* B, global float* C, const int MAT_SIZE){
    local float aTileOfA[TILE_SIZE][TILE_SIZE];
    local float aTileOfB[TILE_SIZE][TILE_SIZE];

    int x  = get_global_id(0); // Globaler Index des Elements cXY in
    int y  = get_global_id(1); //   der Matrix
    int lX = get_local_id(0);  // Lokaler Index des Elements im
    int lY = get_local_id(1);  //   zugeordneten Tile

    float cXY = 0;
    for(int phase = 0; phase < MAT_SIZE / TILE_SIZE; phase++){
       aTileOfA[lX][lY] = A[y * MAT_SIZE + phase * TILE_SIZE + lX];
       aTileOfB[lX][lY] = B[(phase * TILE_SIZE + lY) * MAT_SIZE + x];
       barrier(CLK_LOCAL_MEM_FENCE);
       for(int k=0; k < TILE_SIZE; k++){
          cXY += aTileOfA[lX][k] * aTileOfB[k][lY];
       }
       barrier(CLK_LOCAL_MEM_FENCE);
    }
    C[MAT_SIZE * y + x] = cXY;
}
```

Evaluation

- Test: Matrix 1024*1024, TILE_SIZE 16, 32Bit float
- Messung auf GPU (GTX 670)
  - Ohne Local Memory: 156 ms
  - Mit: 17 ms
  - Faktor: 9,2
  - Vergleich: Bandbreitenersparnis Faktor 16
- Messung auf CPU (Intel Core i7 2700k)
  - Ohne Local Memory: 3333 ms
  - Mit: 3850 ms
  - … schadet hier sogar!