Parallele Programmierung
GPU Parallelisierung 2

Vorlesung 11

Prof. Dr. Luc Bläser

Quiz – Letzte Woche

```
__global__
void VectorAddKernel(float *A, float *B, float *C, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  C[i] = A[i] + B[i];
}
```

```
N = 1000000;
VectorAddKernel<<<1, N>>>(A, B, C, N);
```

Wieso funktioniert dies nicht?

Probleme

- Blockgrösse max. 1024
  - Braucht $\lceil N/1024 \rceil$ Blöcke = 98 Blöcke
- Boundary Check in Kernel
  - `if (i < N)`

Korrektur

```
__global__
void VectorAddKernel(float *A, float *B, float *C, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) {
     C[i] = A[i] + B[i];
  }
}

N = 1000000;
int nofBlocks = (N + 1023) / 1024;
VectorAddKernel<<<nofBlocks, 1024>>>(A, B, C, N);
```

Inhalt Heute

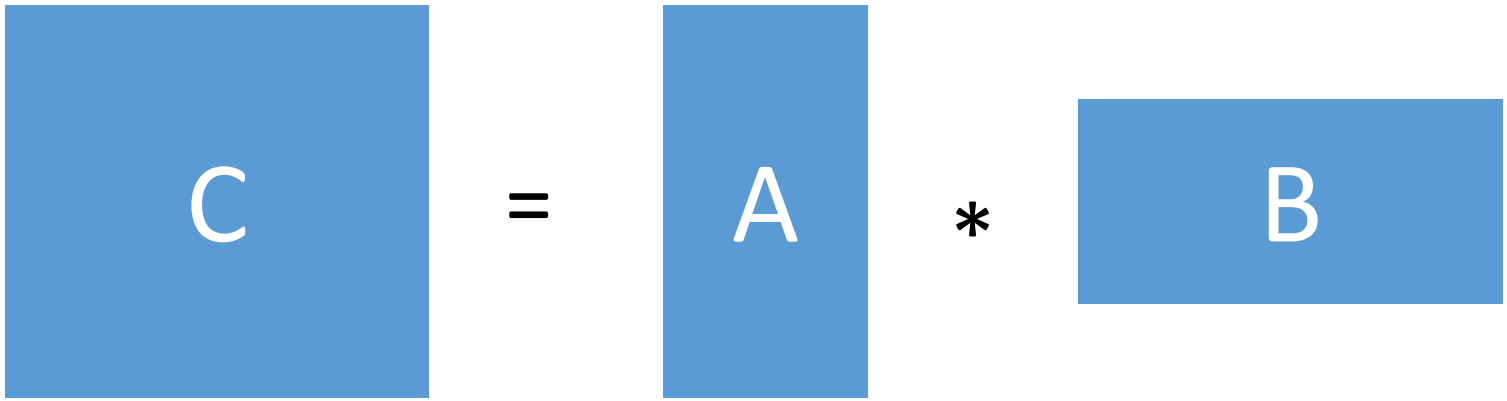
- CUDA-Optimierungstechniken
  - Speichermodell
  - Synchronisation
  - Warps
  - Divergenz
  - Coalescing
- OpenCL im Anhang

Lernziele

- CUDA (bzw. OpenCL) Wissen vertiefen und Optimierungstechniken kennenlernen
- Speichermodell und Synchronisation einsetzen können

Matrix Multiplikation

$$C = A * B$$



Sequentielle Matrix Multiplikation

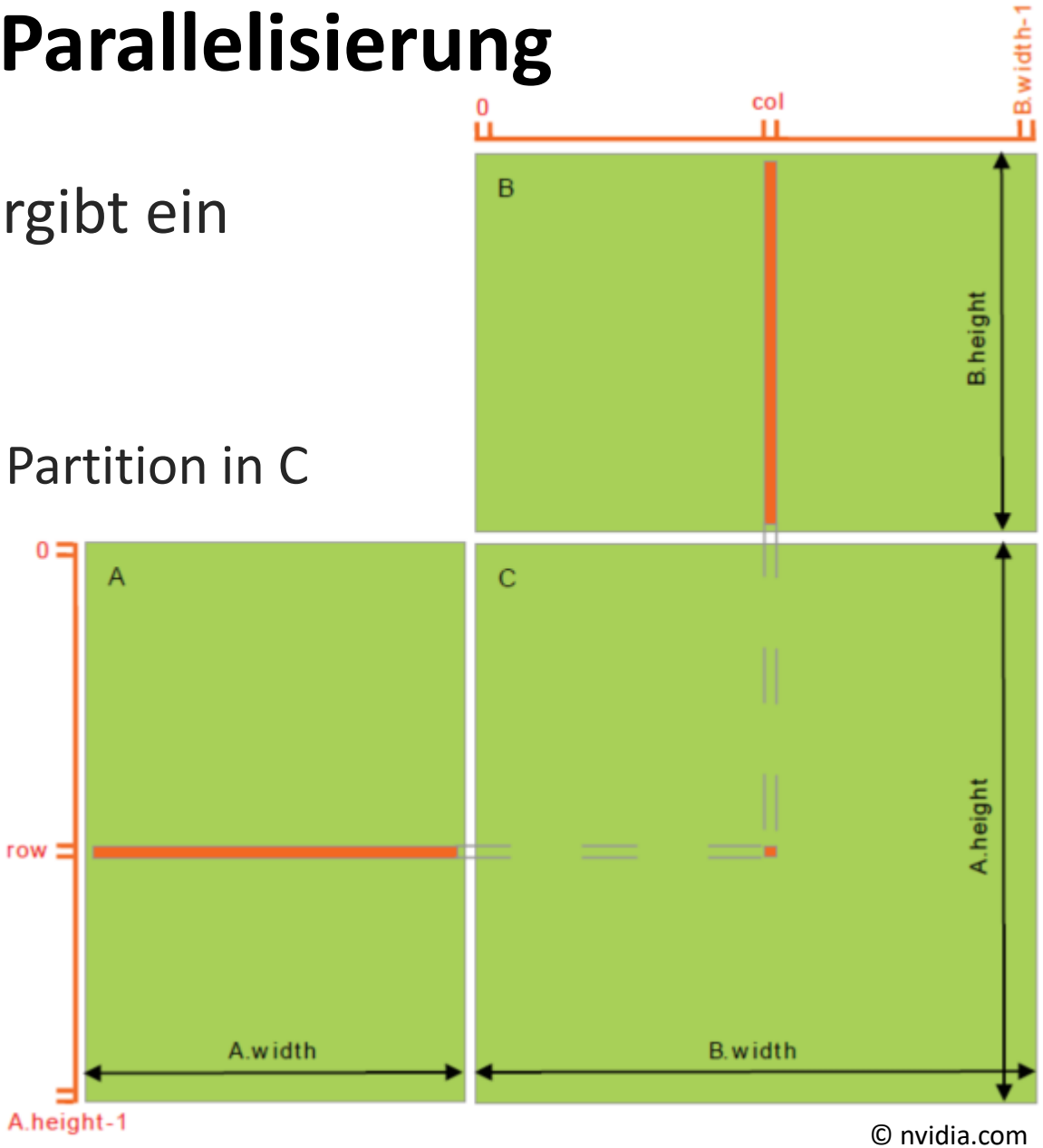
$$C_{i,j} = \sum_{0}^{K-1} A_{i,k} * B_{k,j}$$

C#
```
float sum = 0;
for (int k = 0; k < K; k++) {
  sum += A[i, k] * B[k, j];
}
C[i, j] = sum;
```

Keine multidimensionale Arrays in C

Parallelisierung

- Jeder Thread ergibt ein Element in C
  - N * M Threads
  - Ein Block: Eine Partition in C



© nvidia.com

Kernel: Erster Versuch

```
__global__
void multiply(float *A, float *B, float *C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  float sum = 0;
  for (int k = 0; k < K; k++) {
    sum += A[i * K + k] * B[k * M + j];
  }
  C[i * M + j] = sum;
}
```

Äquivalent zu A[i, k] * B[k, j]

Äquivalent zu C[i, j]

Was fehlt in diesem Design?

Mit Randbehandlung

```
__global__
void multiply(float *A, float *B, float *C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < M) {
    float sum = 0;
    for (int k = 0; k < K; k++) {
      sum += A[i * K + k] * B[k * M + j];
    }
    C[i * M + j] = sum;
  }
}
```

Ist das effizient?

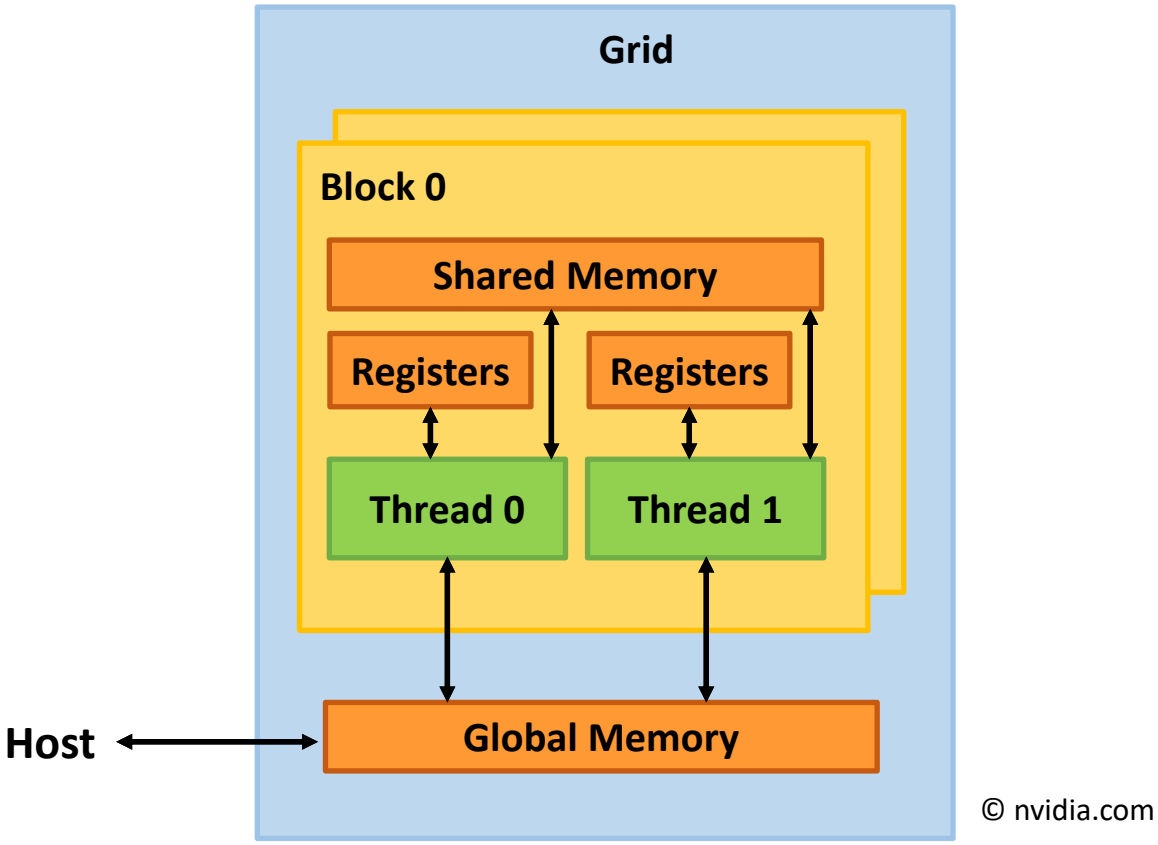
Hohe Speicherkosten

- Global Memory ist relativ teuer: ca. 600 Zyklen
- Threads lesen wiederholt selbe Elemente von A & B

Thread 0, 0:	A[0, 0] .. A[0, K-1]	B[0, 0] .. B[K-1, 0]
Thread 0, 1:	A[0, 0] .. A[0, K-1]	B[0, 1] .. B[K-1, 1]
Thread 1, 0:	A[1, 0] .. A[1, K-1]	B[0, 0] .. B[K-1, 0]
Thread 1, 1:	A[1, 0] .. A[1, K-1]	B[0, 1] .. B[K-1, 1]

=> Effizienzsteigerung mit Cache Speicher

CUDA Speicherstufen



© nvidia.com

Vereinfachtes Speichermodell

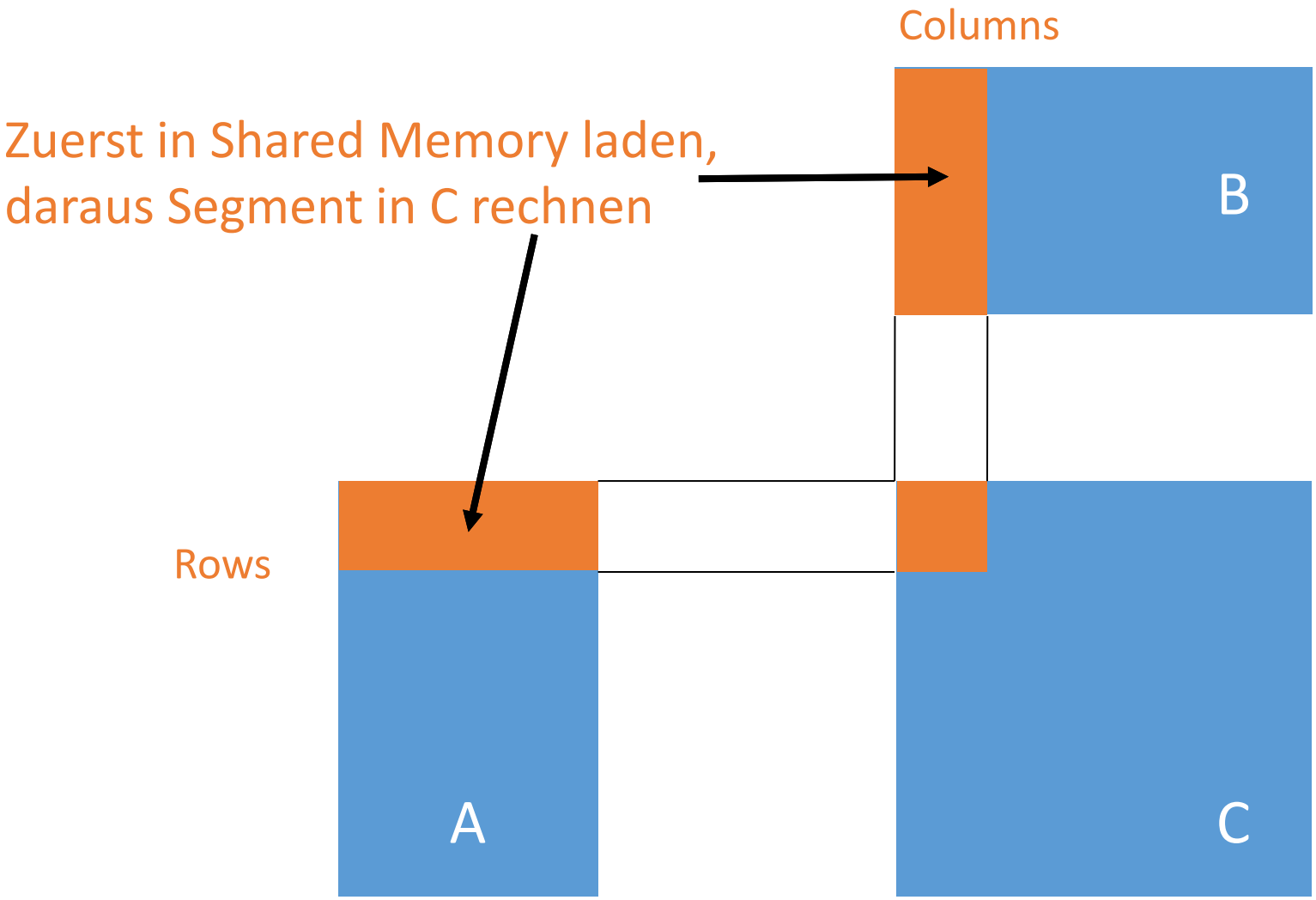
Speicherstufen

- Shared Memory
  - Per Streaming Multiprozessor
  - Schnell ca. 4 Zyklen
  - Nur zwischen Threads innerhalb Block sichtbar
  - Paar KB
  - `__shared__ float x;`
- Global Memory
  - «Main Memory» in GPU Device
  - Langsam ca. 400-600 Zyklen
  - Allen Threads sichtbar
  - Mehrere GB
  - `cudaMalloc()`

Schnelle Matrix Multiplikation

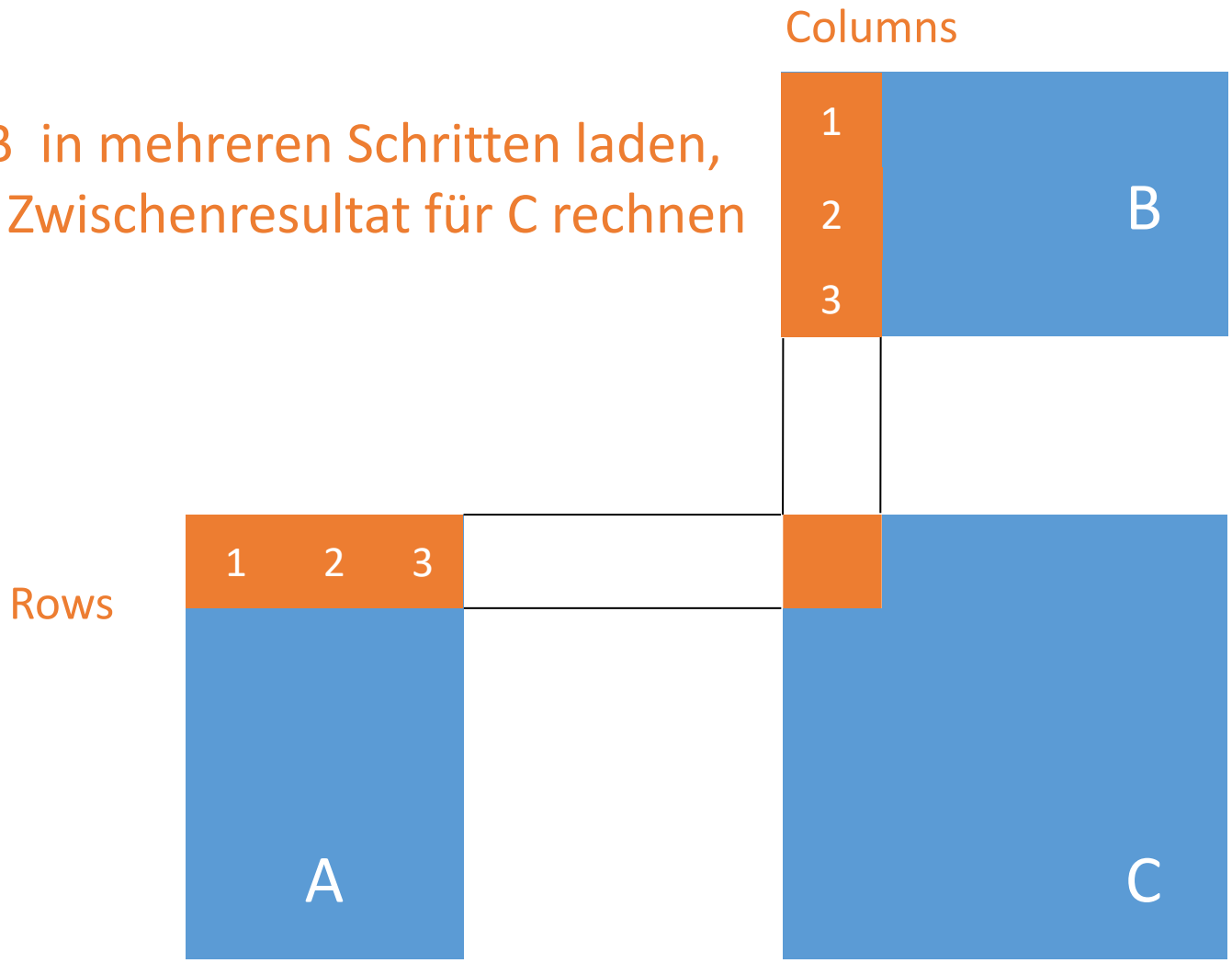
Wie kann man Shared Memory hier nutzen?

Tiled Matrix Multiplication



Begrenztes Shared Memory

A und B in mehreren Schritten laden,
Daraus Zwischenresultat für C rechnen



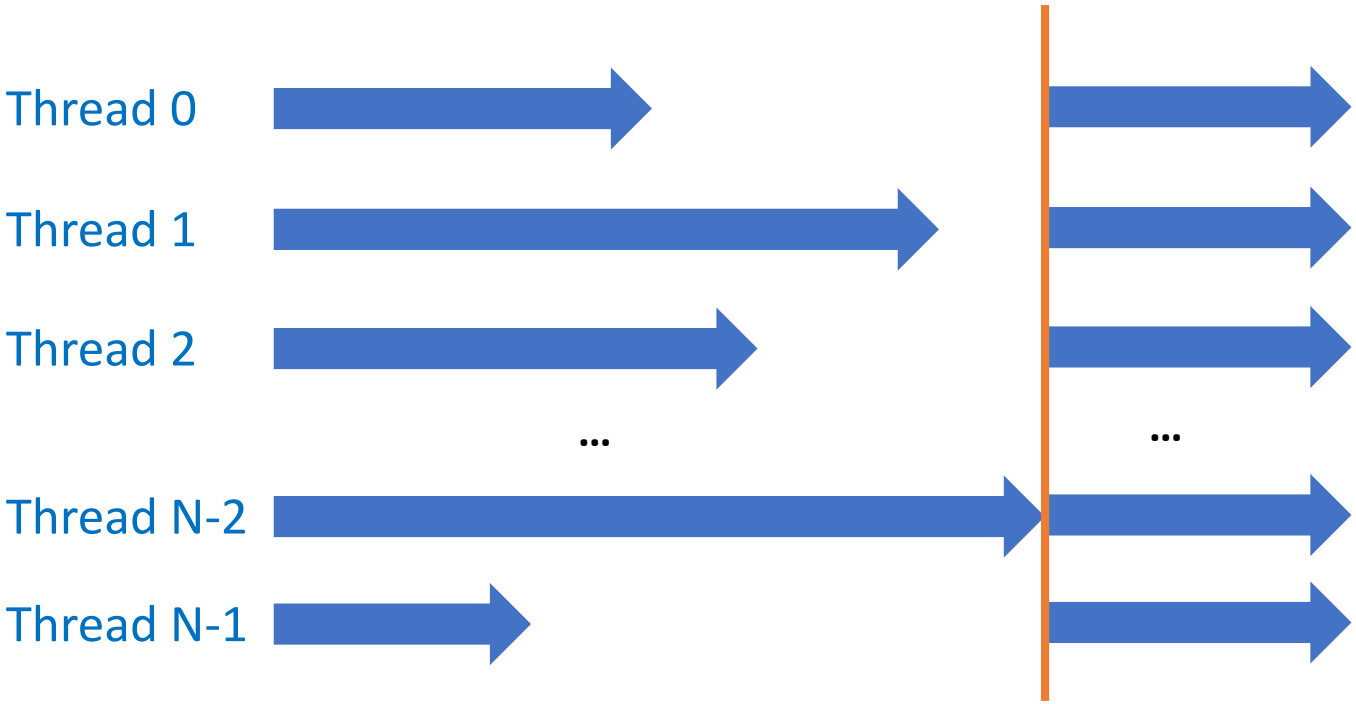
Gerüst des Algorithmus

```
float sum = 0.0;
for (int tile = 0; tile < nofTiles; tile++) {
  // Tile von A und B in Shared Memory lesen
  // Jeder Thread liest ein Element von jedem Tile
  __syncthreads();
  // Multipliziere Zeile von A-Tile mit
  // Spalte von B-Tile aus dem Shared Memory
  sum += partialProduct;
  __syncthreads();
}
C[row * M + col] = sum;
```

Was könnte der Zweck von __syncthreads() sein?

Block-Barriere

- `__syncthreads()`
  - Synchronisiert alle Threads innerhalb eines Blocks
  - Keine Synchronisation zwischen Blöcken



Block Barriere

- Jedes __syncthreads Statement ist andere Barriere
  - In if-else nur erlaubt, falls alle Threads eines Blocks entweder nur if- oder nur else-Branch pro Runde wählen
  - Sonst undefiniert (Blockade/Fehlverhalten)

```
if (threadIdx.x < blockDim.x/2) {
   __syncThreads();
   …
} else {
   __syncThreads();
}
```

Verschiedene Barrieren

Fehler

Erweiterung mit Shared Memory

```
__shared__ float Asub[TILE_SIZE][TILE_SIZE];
__shared__ float Bsub[TILE_SIZE][TILE_SIZE];

int tx = threadIdx.x, ty = threadIdx.y;
int col = blockIdx.x * TILE_SIZE + tx;
int row = blockIdx.y * TILE_SIZE + ty;

for (int tile = 0; tile < nofTiles; tile++) {
  Asub[ty][tx] = A[row * K + tile * TILE_SIZE + tx];
  Bsub[ty][tx] = B[(tile * TILE_SIZE + ty) * M + col];
  __syncthreads();
  // Multipliziere Zeile von A-Tile mit
  // Spalte von B-Tile aus dem Shared Memory
  __syncthreads();
}
```

Shared Memory: Deklaration

- Deklaration mit Keyword `__shared__`
- Statische Array-Grösse notwendig
  - Begrenzter Speicher (z.B. 48 KB)
- Mehrdimensionalität bei statischer Grösse erlaubt
  - Sonst muss es in C linearisiert werden (Bsp Matrix A, B, C)

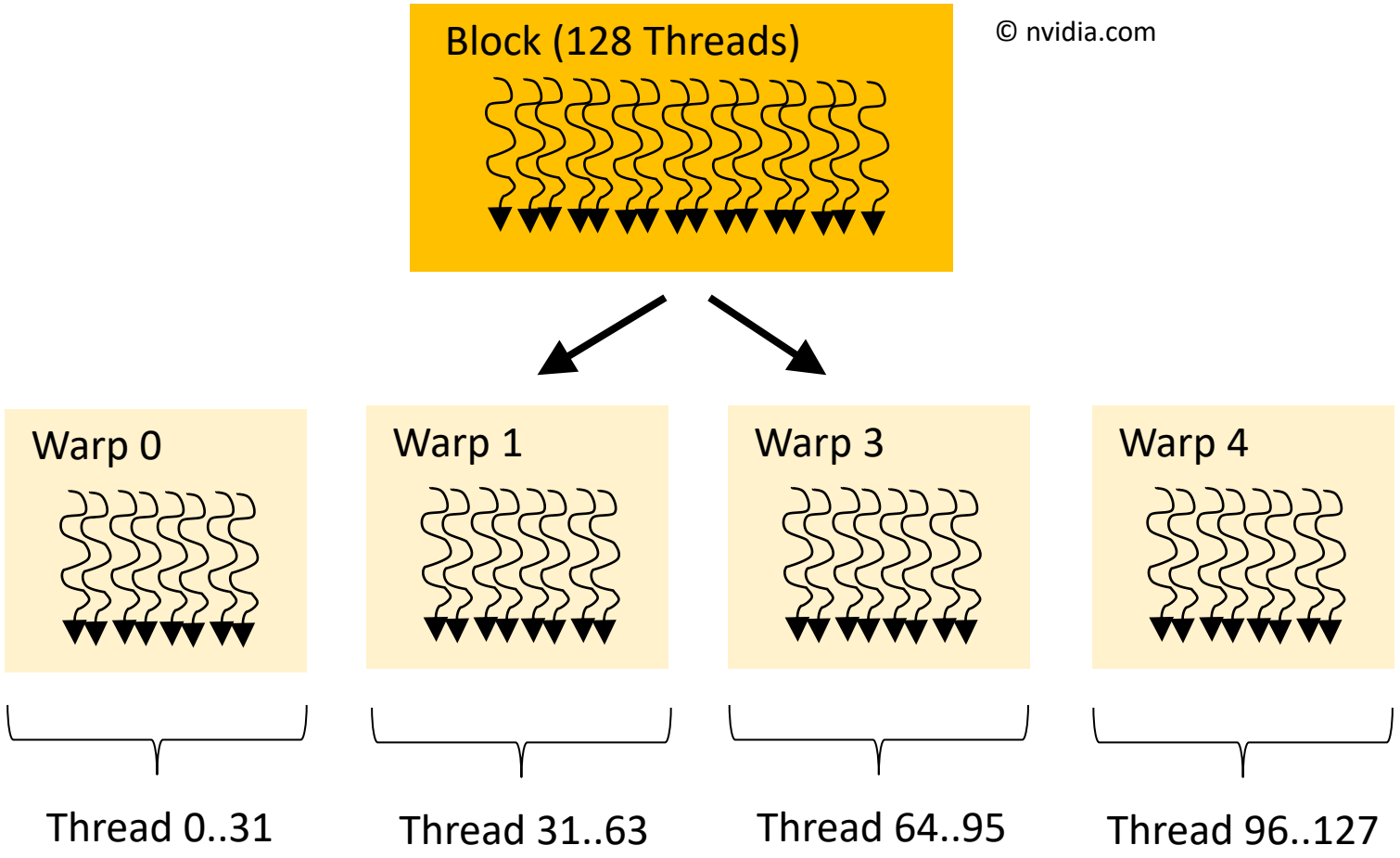
Matrix Multiplication Continued

```
float sum = 0.0;
for (int tile = 0; tile < nofTiles; tile++) {
  Asub[ty][tx] = A[row * K + tile * TILE_SIZE + tx];
  Bsub[ty][tx] = B[(tile * TILE_SIZE + ty) * M + col];
  __syncthreads();
  for (int ksub = 0; ksub < TILE_SIZE; ksub++) {
    sum += Asub[ty][ksub] * Bsub[ksub][tx];
  }
  __syncthreads();
}
C[row * M + col] = sum;
```

Was fehlt noch?

Warp

- Block wird intern in Warps zerlegt (zu je 32 Threads)



Warp Ausführung

- Alle Threads in Warp führen gleiche Instruktion aus
  - SIMD: Gleiche Instruktion auf verschiedenen Daten
  - Verzweigungen werden abwechselnd ausgeführt (später)
- Stream Multiprozessor kann alle Warps eines Blocks beherbergen
  - Aber nur wenige laufen gleichzeitig echt parallel (1 bis 24)

Divergenz

- Unterschiedliche Verzweigungen im selben Warp
  - if/switch/while/do/for
  - SM führt Instruktion der einen Verzweigung durch
    - Die anderen Threads müssen warten
  - Dann wieder der anderen Verzweigung
    - Die einen Threads müssen warten

=> Performance-Problem

Verzweigungen im Kernel

- Schlechter Fall
  - Divergenz innerhalb derselben Warp

```
if (threadIdx.x > 1) {
  …
} else {
  …
}
```

- Guter Fall
  - Gleiche Verzweigung innerhalb Warp

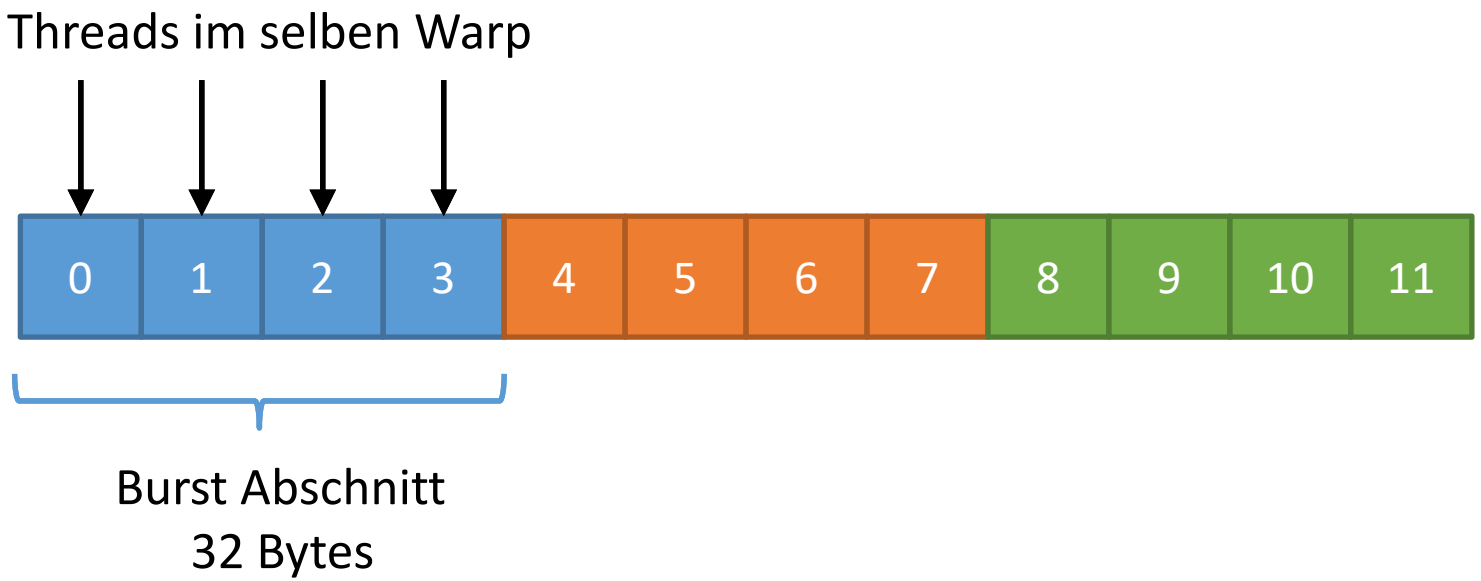
```
if (threadIdx.x / 32 > 1) {
  …
} else {
  …
}
```

Warps werden sequentiell nach threadIdx.x/y/z in 32 Schritten aufgeteilt

Memory Coalescing

- Zugriffsmuster der Threads sind entscheidend
  - Für Performance
- Falls Threads auf 32-Byte-Bereiche zugreifen
  - Kombinierter Zugriff pro Bereich (Burst)
- Sonst teure Einzel-Zugriffe
  - Falls Threads verschiedene Bursts zugreifen
  - Z.B. je 400 Zyklen pro Global Device Memory Access

Coalescing



Lese/Schreibe den Burst in einer Transaktion pro Warp

Auch Coalescing

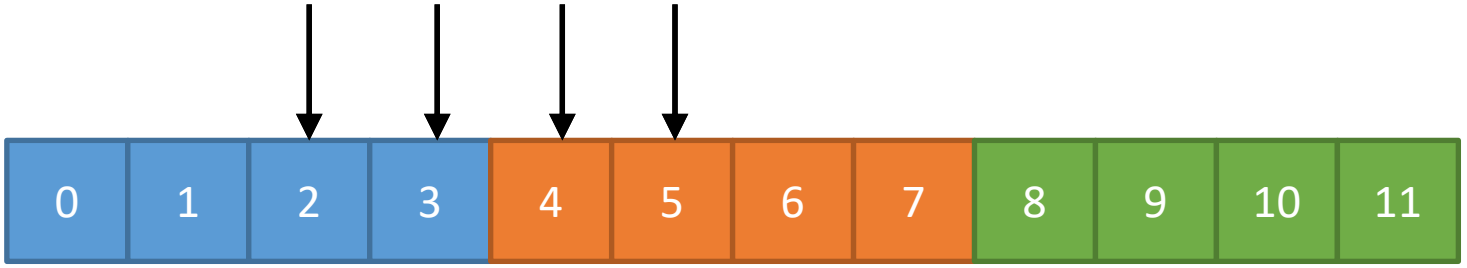
Vertauschtes Lesen/Schreiben im selben Burst

Auch Coalescing

Nur einzelne Elemente im Burst werden zugegriffen

Misaligned => Kein Coalescing

3 Transaktionen



2 Transaktionen

Coalescing Details

Je nach Compute Capability

- < 6.0: Burst Section
  - 128 Bytes für >= 4-Byte-Elemente
  - 64 Bytes für 2-Byte-Elemente
  - 32 Bytes für 1-Byte-Elemente
- < 1.2: Kein Coalescing bei Vertauschen oder Lücken

Coalescing in Anwendung

- Zugriffe möglichst wie folgt (um)designen

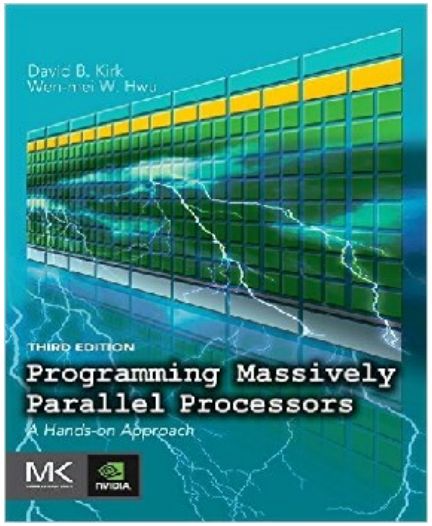
```
data[(Ausdruck ohne threadIdx.x) + threadIdx.x]
```

Rückblick: Lernziele

- CUDA (bzw. OpenCL) Wissen vertiefen und Optimierungstechniken kennenlernen
- Speichermodell und Synchronisation einsetzen können

Literatur zum Nachschlagen

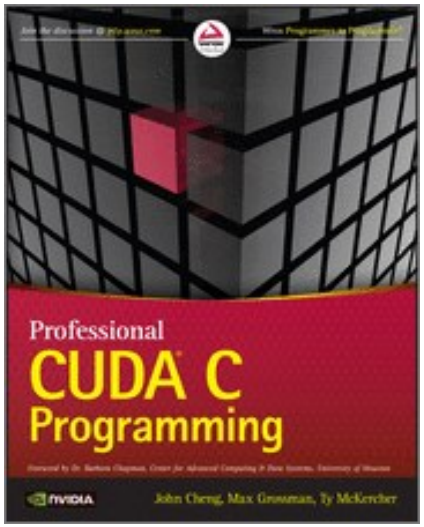
- D. B. Kirk and W.-m. W. Hwu. Programming Massively Parallel Processors, 3rd Edition, Morgan Kaufmann, 2016.

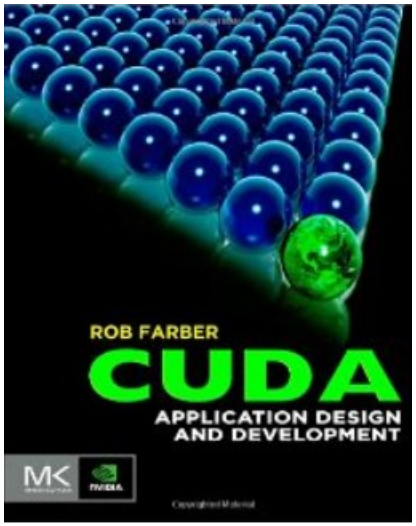


- Nvidia CUDA C Programming Guide
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide

Weitere Literatur (nach Bedarf)

- John Cheng, Max Grossman, Ty McKercher. Professional CUDA C Programming, Wrox, 2014.
- R. Farber. CUDA: Application Design and Development, Morgan Kaufmann, 2011.





Alternative GPU Parallelisierung

- Khronos Group. OpenCL Specification (Version 2.2)
  - http://www.khronos.org/opencl
- Microsoft. C++ AMP
  - http://msdn.microsoft.com/de-de/library/hh265137.aspx
- OpenACC
  - http://openacc.org

Anhang: OpenCL

Weitere Unterschiede OpenCL/CUDA

Aspekt	OpenCL	CUDA
Shared Memory	__local	__shared__
Block Barrier	barrier(CLK_LOCAL_MEM_FENCE)	__syncthreads()
Thread Id in Block	get_local_id(0) get_local_id(1)	threadIdx.x threadIdx.y
Block Id in Grid	get_group_id(0) get_group_id(1)	blockIdx.x blockIdx.y

OpenCL Kernel Body

```
__local float Asub[TILE_SIZE][TILE_SIZE];
__local float Bsub[TILE_SIZE][TILE_SIZE];

int tx = get_local_id(0), ty = get_local_id(1);
int col = get_group_id(0) * TILE_SIZE + tx;
int row = get_group_id(1) * TILE_SIZE + ty;
```

OpenCL Kernel Continued

```
float sum = 0.0;
for (int tile = 0; tile < nofTiles; tile++) {
  Asub[ty][tx] = A[row * K + tile * TILE_SIZE + tx];
  Bsub[ty][tx] = B[(tile * TILE_SIZE + ty) * M + col];
  barrier(CLK_LOCAL_MEM_FENCE);
  for (int ksub = 0; ksub < TILE_SIZE; ksub++) {
    sum += Asub[ty][ksub] * Bsub[ksub][tx];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
C[row * M + col] = sum;
```

Randprüfung noch einbauen