Mining Massive Datasets

Lecture 14

Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

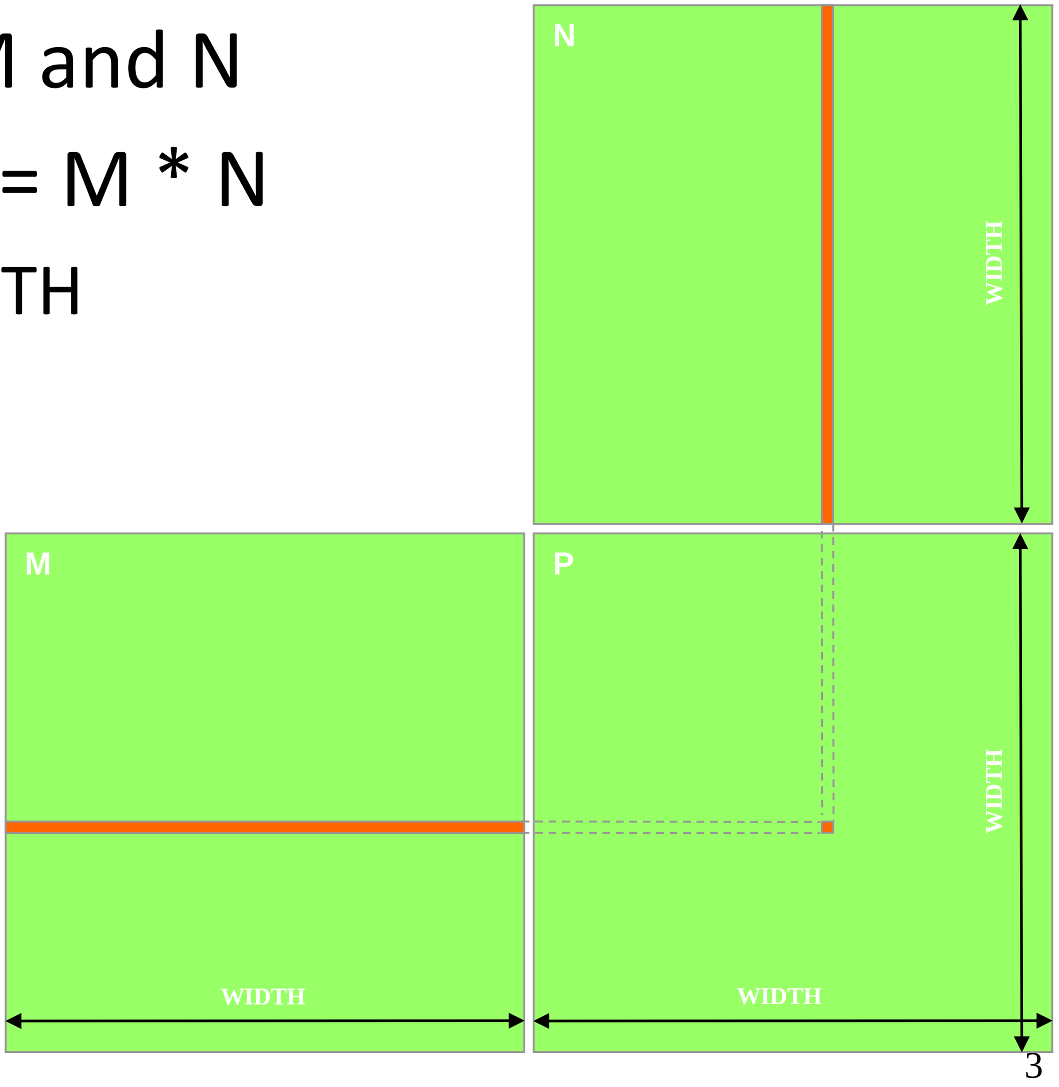
RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Introduction to GPU-Programming with CUDA: Motivation

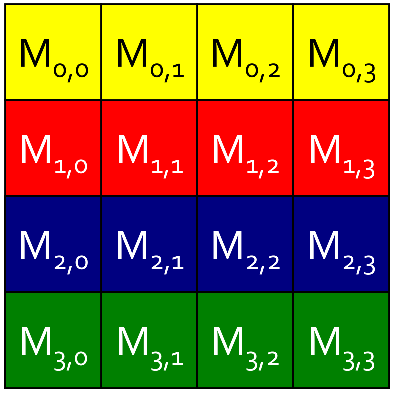
Slides (partially modified) by:
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

Square Matrix Multiplication Example

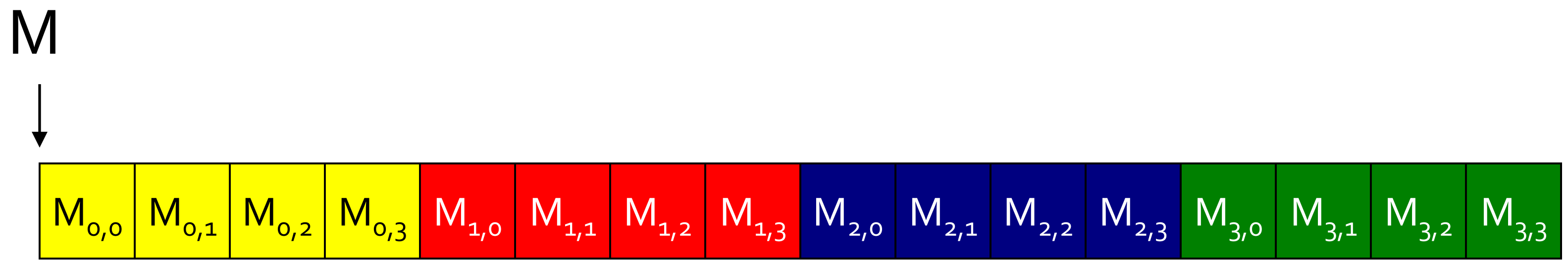
- Input: matrices M and N
- Output: matrix P = M * N
  - Size WIDTH x WIDTH



Each Matrix is Stored in a 1D-Array



$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

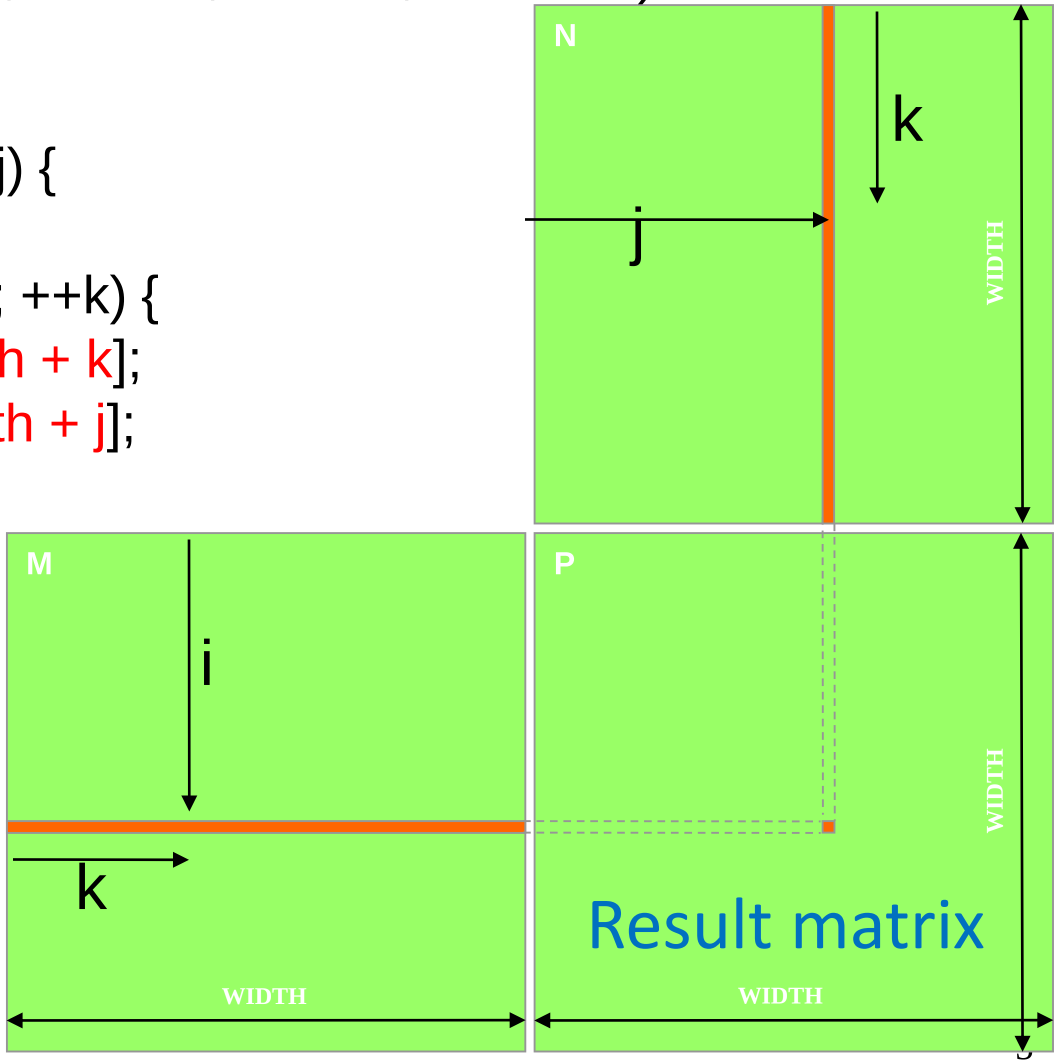


M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

=> Element M[i,j] is adressed via M[i*WIDTH + j]

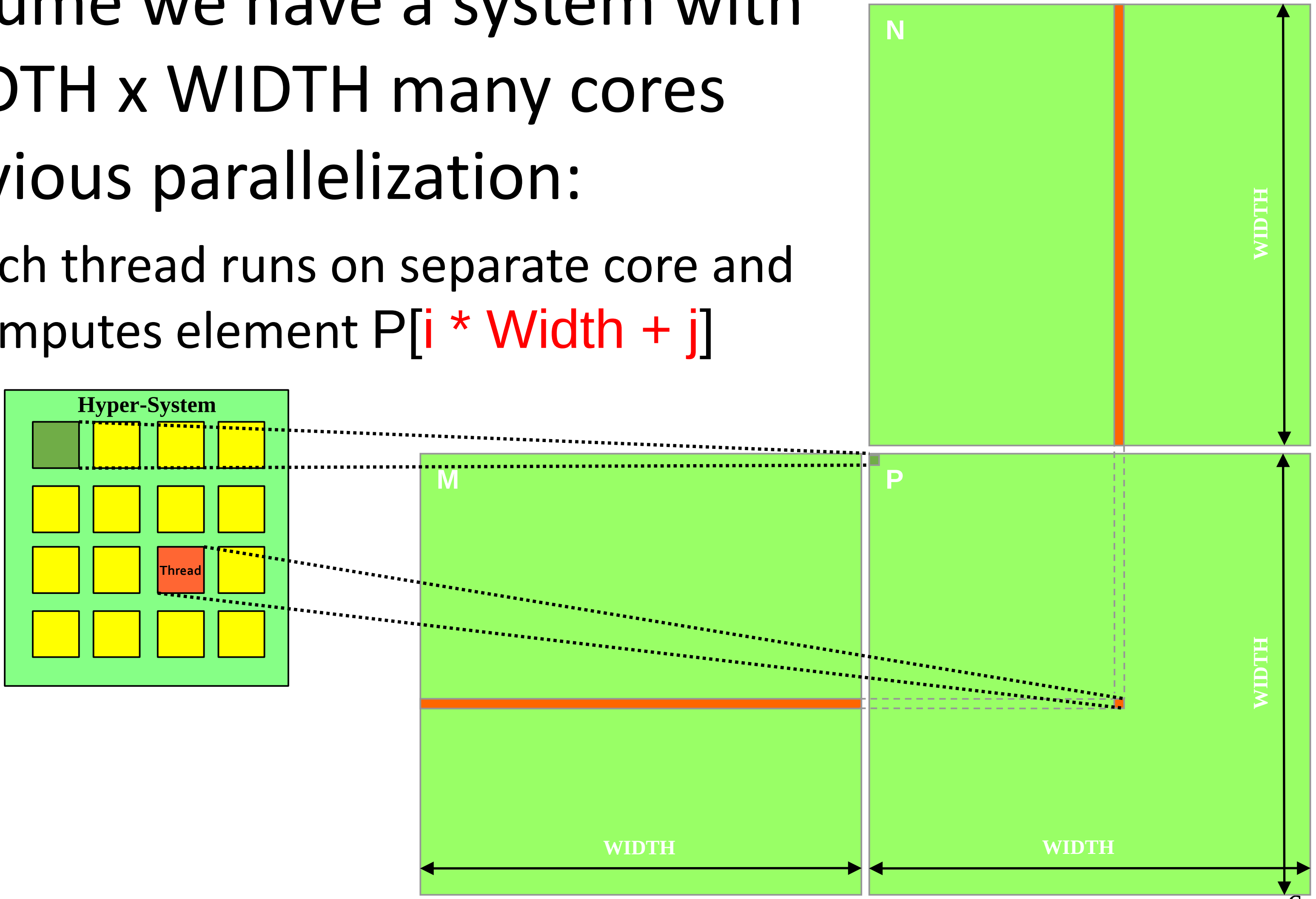
A Simple Host (CPU) Version in C

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Square Matrix Multiplication Example

- Assume we have a system with WIDTH x WIDTH many cores
- Obvious parallelization:
  - Each thread runs on separate core and computes element P[i * Width + j]

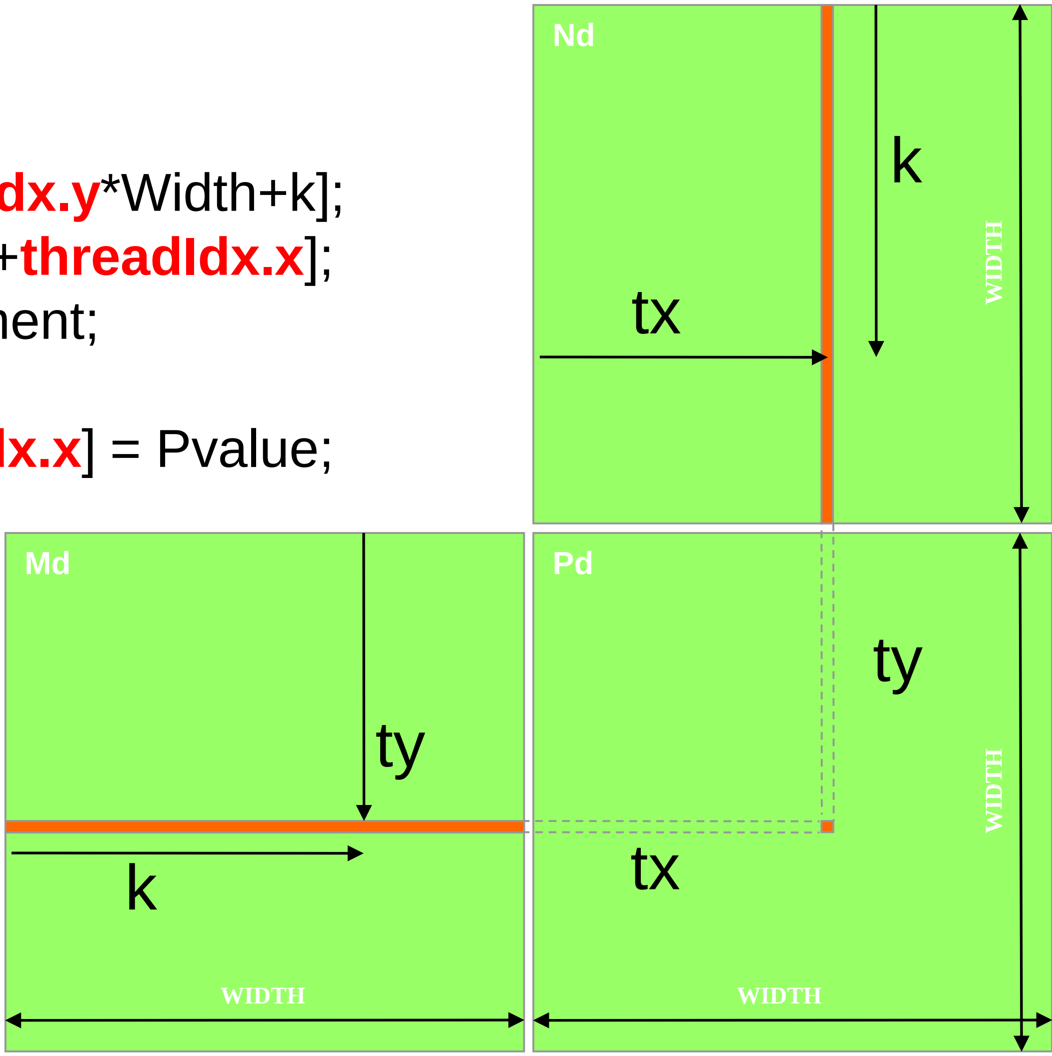


6

Each Thread Runs Following Code

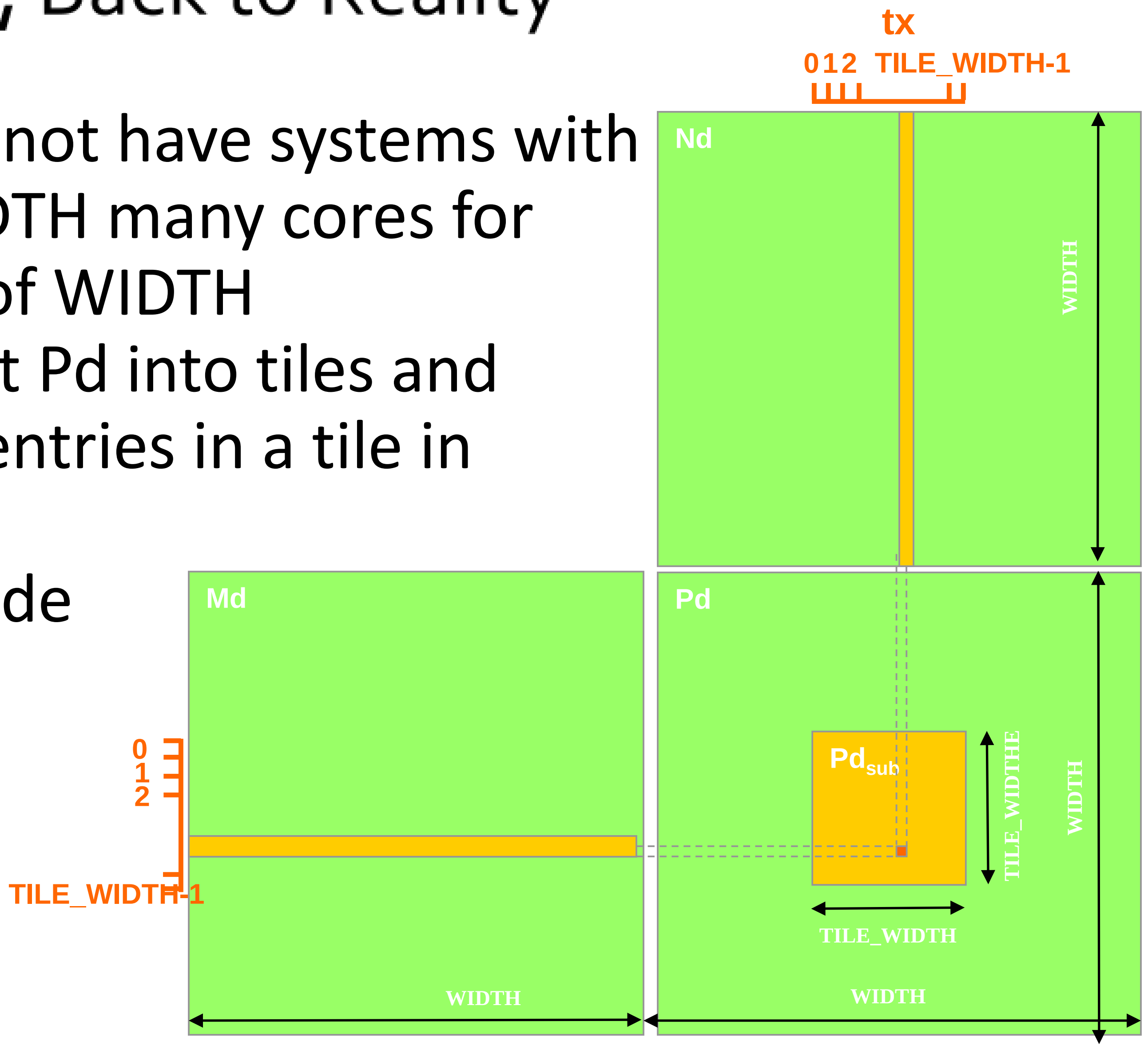
```
void MatrixMulKernel (float* Md, float* Nd, float* Pd, int Width)
{
float Pvalue = 0;
for (int k = 0; k < Width; ++k) {
        float Melement = Md[threadIdx.y*Width+k];
        float Nelement = Nd[k*Width+threadIdx.x];
        Pvalue += Melement * Nelement;
   }
  Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```

- Where do **threadIdx.x** and **threadIdx.y** come from?
  - Their values are "magically" assigned by hardware
  - Different pairs of values for each thread



Back to Life, Back to Reality

- Problem: we not have systems with WIDTH x WIDTH many cores for large values of WIDTH
- Solution: Split Pd into tiles and compute all entries in a tile in parallel
- => How to code this easily?



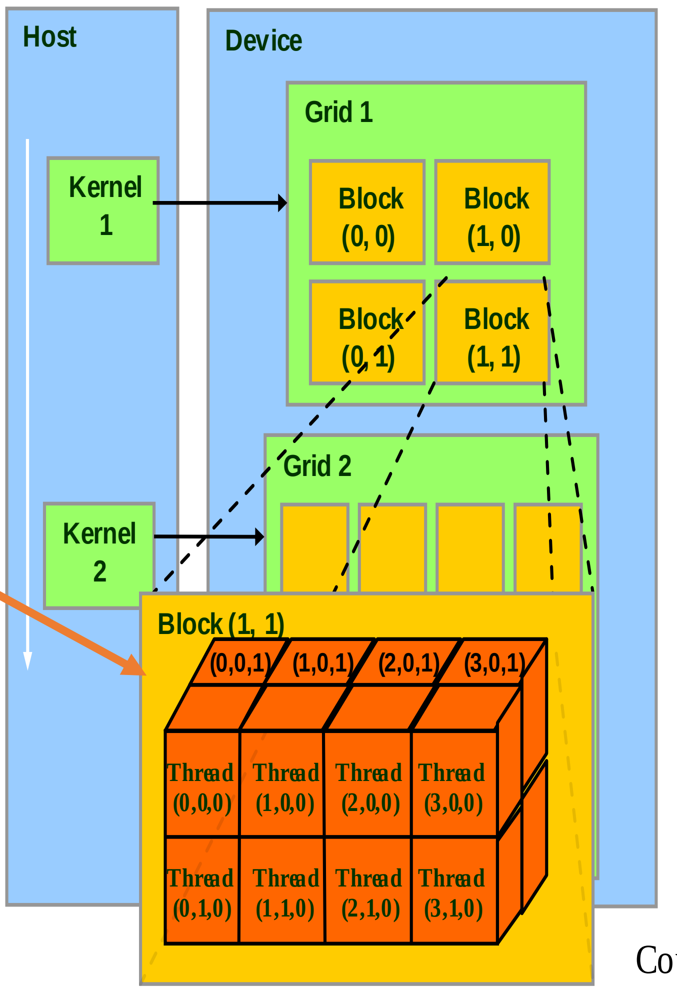
Introduction to GPU-Programming with CUDA: CUDA Architecture

Slides (partially modified) by:
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

Blocks of Threads in CUDA

- The **CUDA** programming model allows very large number of (virtual) threads
- These are grouped into **blocks**
  - Each block contains multiple threads, arranged as a 1D, 2D, or 3D array; here: 3D array

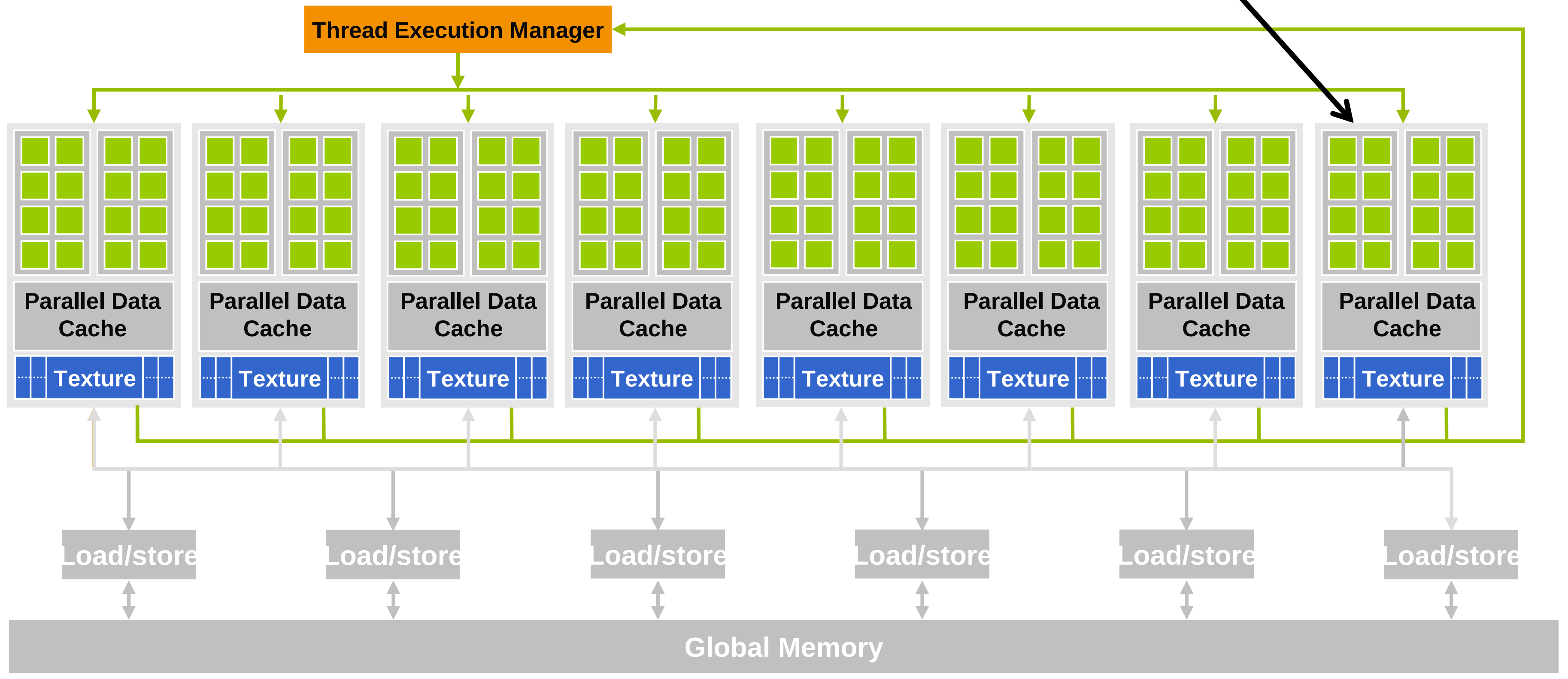
- Blocks correspond to units of hardware (**streaming multiprocesors, SM**)



Courtesy: NDVIA

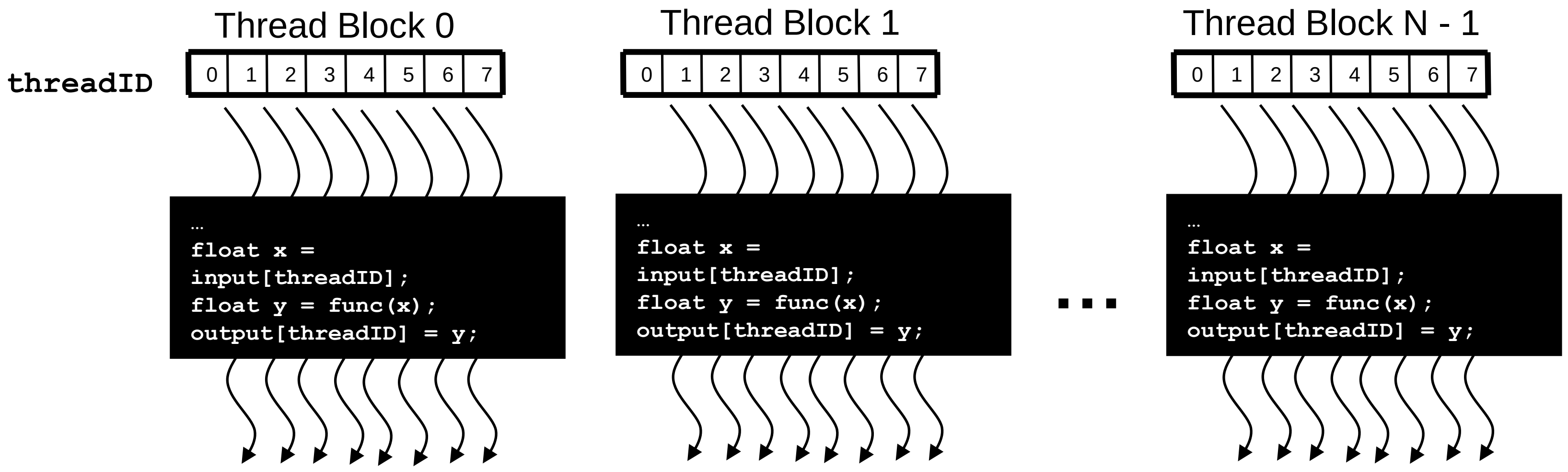
G8o CUDA mode – A **Device** Example

- Each block (or a part of it, a **warp**) runs on a separate streaming multiprocessor SM
- Example: the G80 CUDA GPU



Threads within Blocks

- Each SM has many **streaming processors** (**SP**s)
  - Each SP executes "own" thread, all SPs the same code
  - Threads within a block cooperate via **shared memory**, **atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate

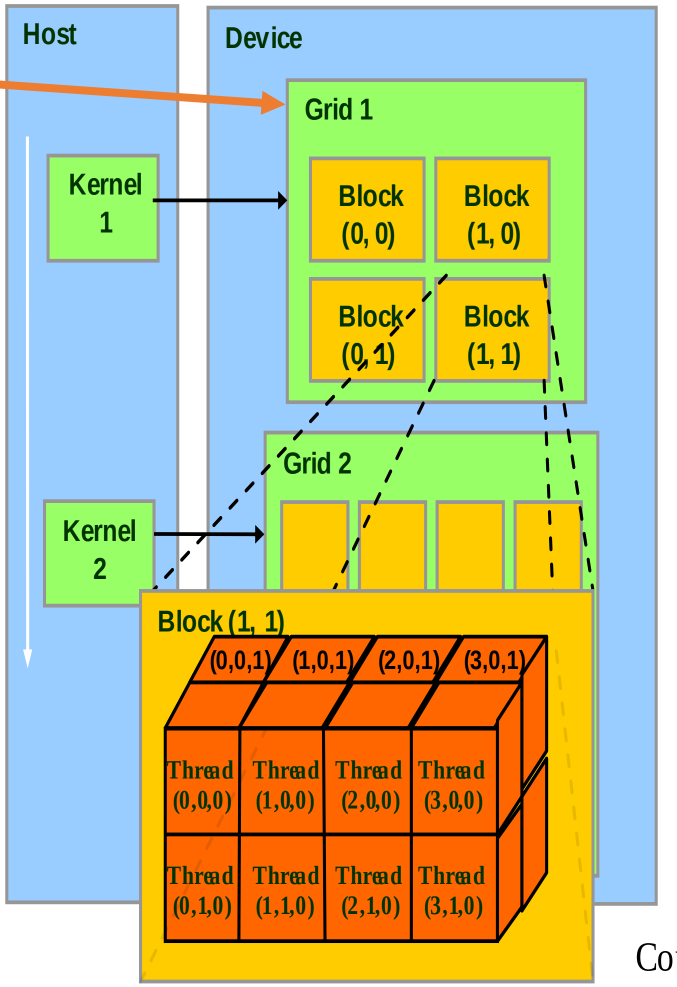


```
…
float x =
input[threadID];
float y = func(x);
output[threadID] = y;
```

Grid of Blocks

- Blocks are organized into a 1D or 2D **Grid**
- Grid is the unit of execution
  - => A program is always executed as a grid of blocks

- BUT: We cannot specify the order of execution of the blocks
  - => Each block is completely independent of other blocks

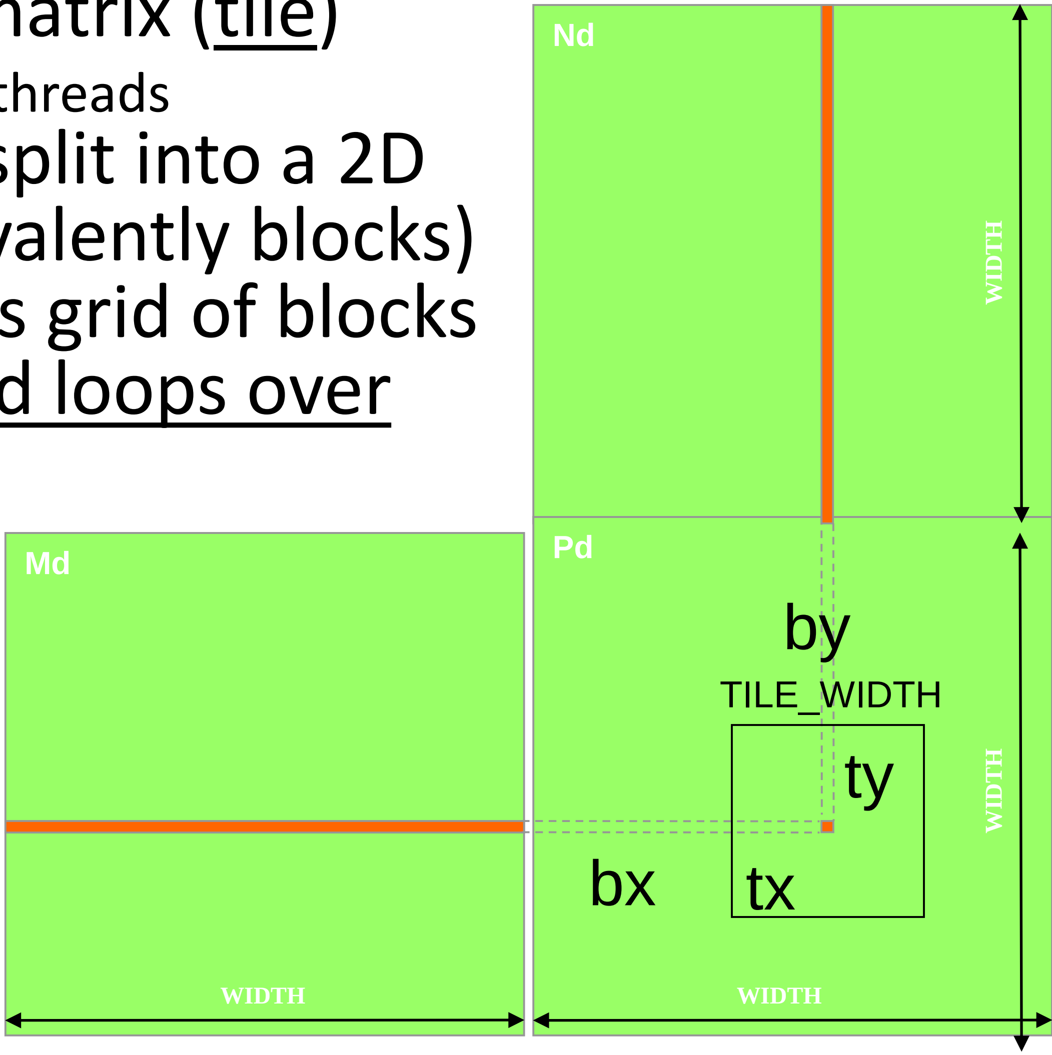


Courtesy: NDVIA

Back to the Matrix Example

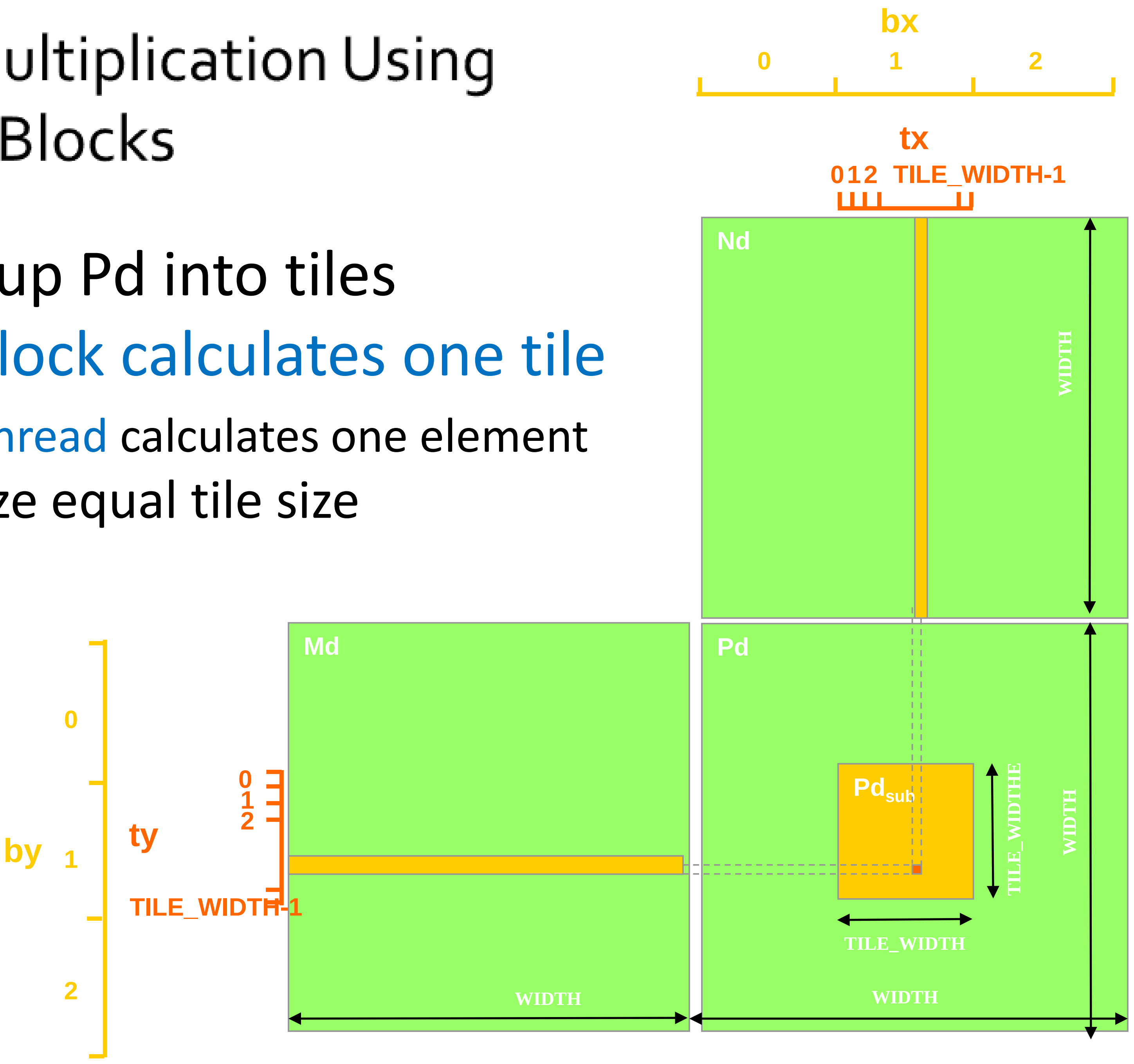
- Each 2D thread block computes a $(\text{TILE\_WIDTH})^2$ sub-matrix (tile)
  - Each has $(\text{TILE\_WIDTH})^2$ threads
- The whole matrix is split into a 2D grid of tiles (or, equivalently blocks)
- => Because of CUDA's grid of blocks we don't need nested loops over tiles!

You still need to put a loop around the kernel call for cases where WIDTH/ TILE_WIDTH is greater than max grid size (64K)!

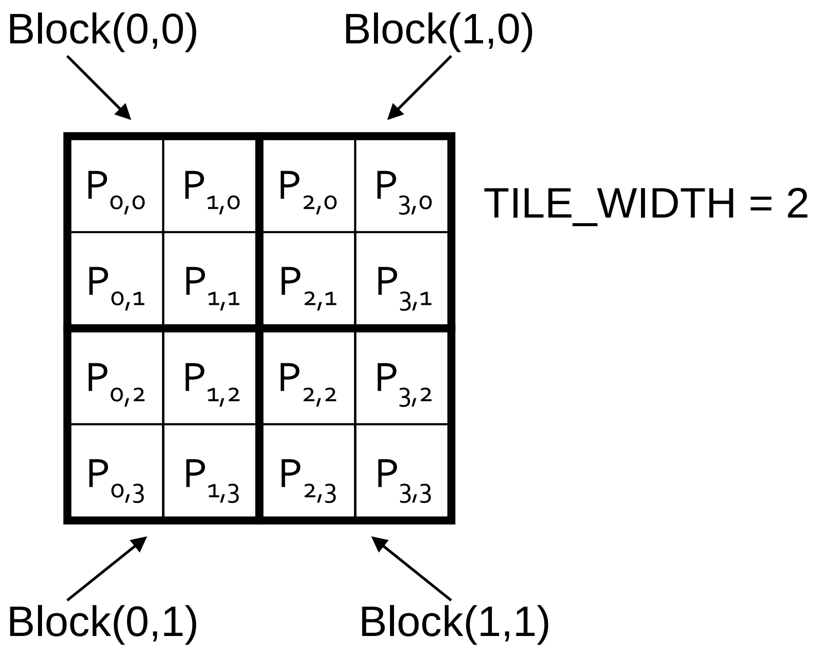


Matrix Multiplication Using Multiple Blocks

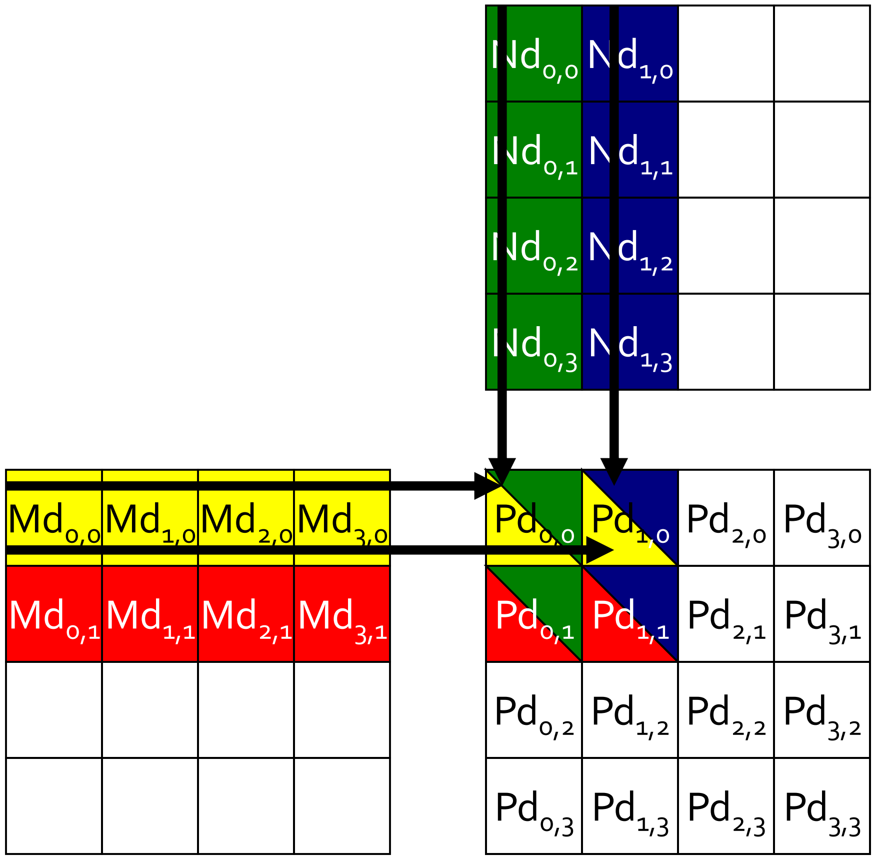
- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
- Block size equal tile size



A Small Example



A Small Example: Multiplication



Revised Matrix Multiplication Kernel using Multiple Blocks

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one                              matrix
for (int k = 0; k < Width; ++k)
   Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

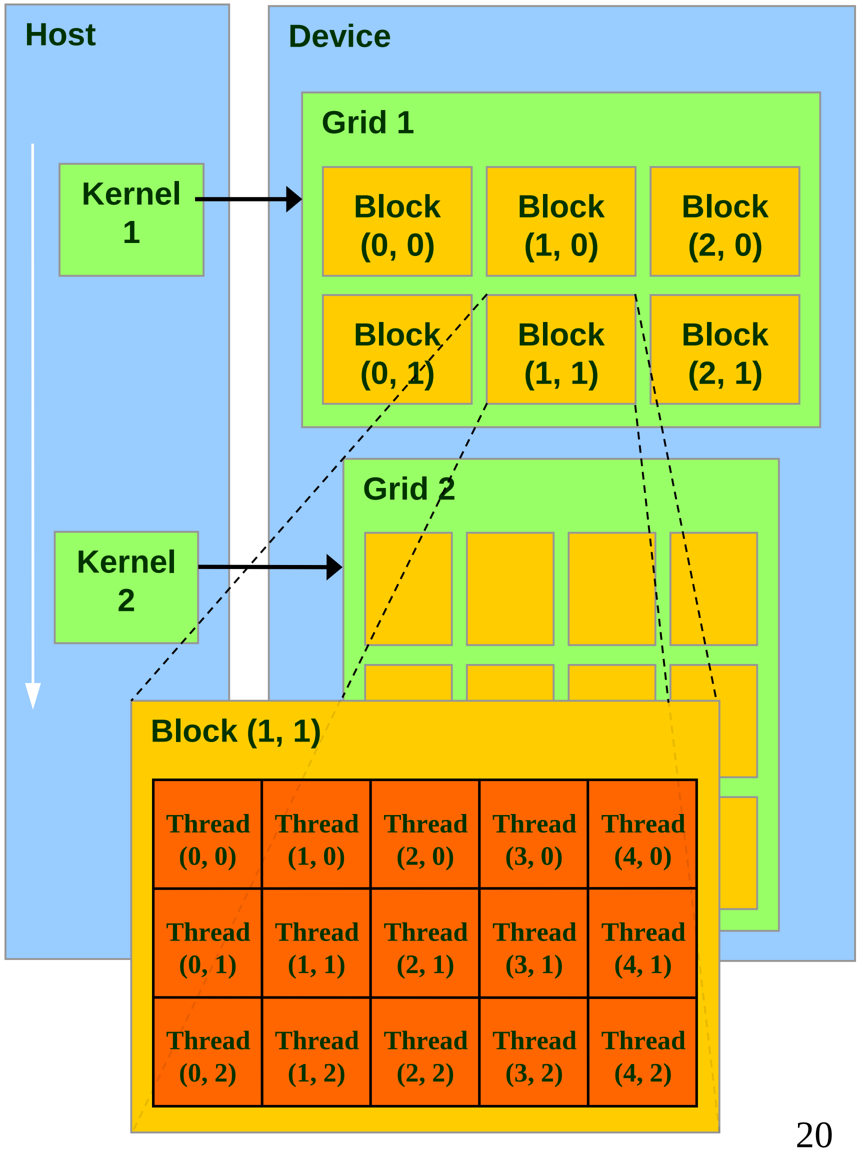
Values provided „magically“ by the CUDA-environment

Introduction to GPU-Programming with CUDA: Blocks, Threads and Warps

Slides (partially modified) by:
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

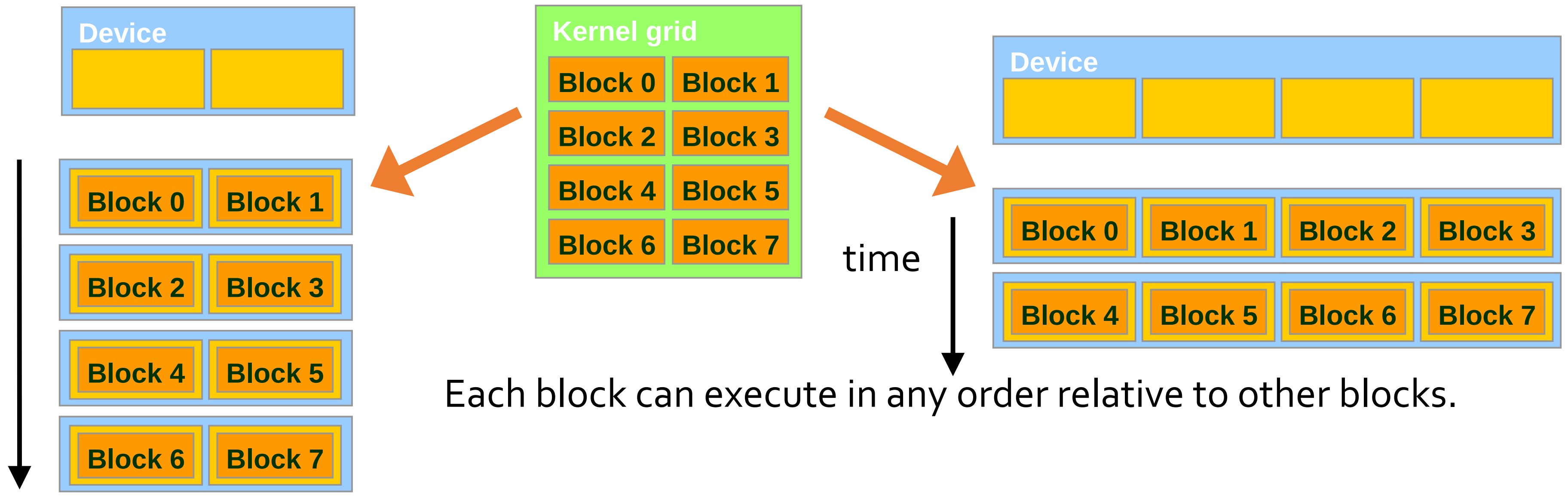
Grid of Blocks and Block of Threads

- Threads are organized in CUDA in two levels:
  - **Block of threads**: a container of threads with 1D, 2D or 3D indexing
  - **Grid of blocks**: a container of blocks with 1D or 2D indexing
- A single <u>identical</u> program (**kernel**) is launched on a grid of blocks
  - Each thread receives "magically" different block-in-grid and thread-in-block indices



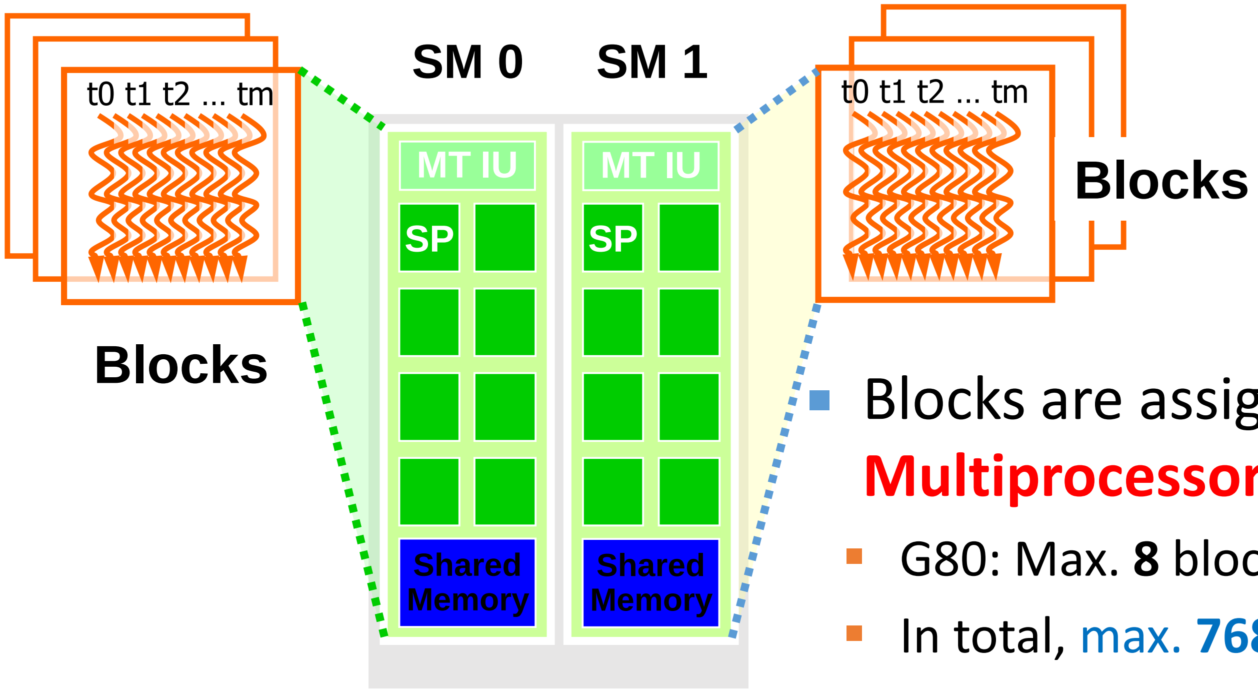
Why Execution over a Grid of Blocks?

- 1. Our code doesn't need one or two outer loops for iterating over blocks (= tiles for MM)
- 2. Scalability: If a (future) GPU has more processors, more blocks can be executed in parallel (=> Good to have many blocks!)



Each block can execute in any order relative to other blocks.

G80 Example: Executing Thread Blocks

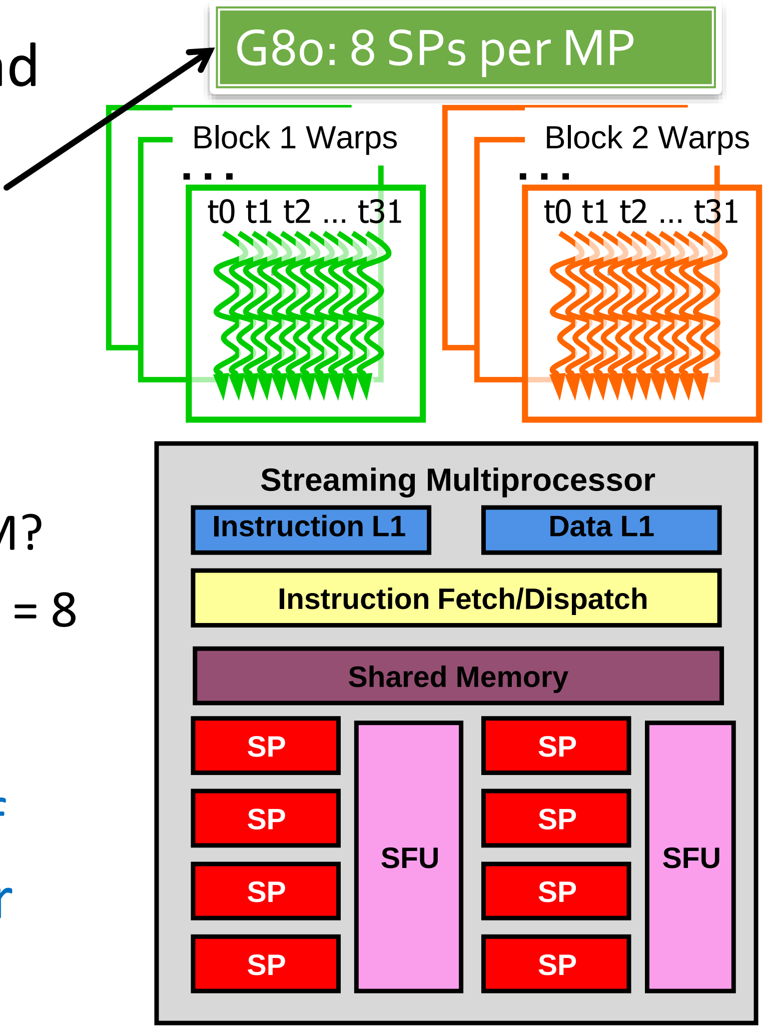


- Blocks are assigned to **Streaming Multiprocessors**
  - G80: Max. **8** blocks per SM
  - In total, max. **768** threads per SM
  - Use max #threads per SM (later more):
    - **Good**: 256 (threads/block) * 3 blocks => 768 threads
    - **Bad**: 512 (threads/block) * 1 block => 512 threads

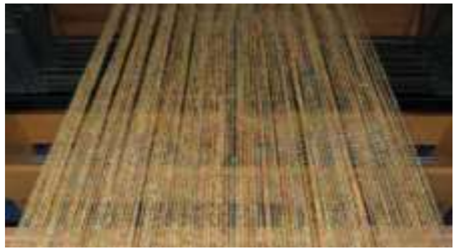
- HD keeps track of block id's, thread id's and their status => limitations

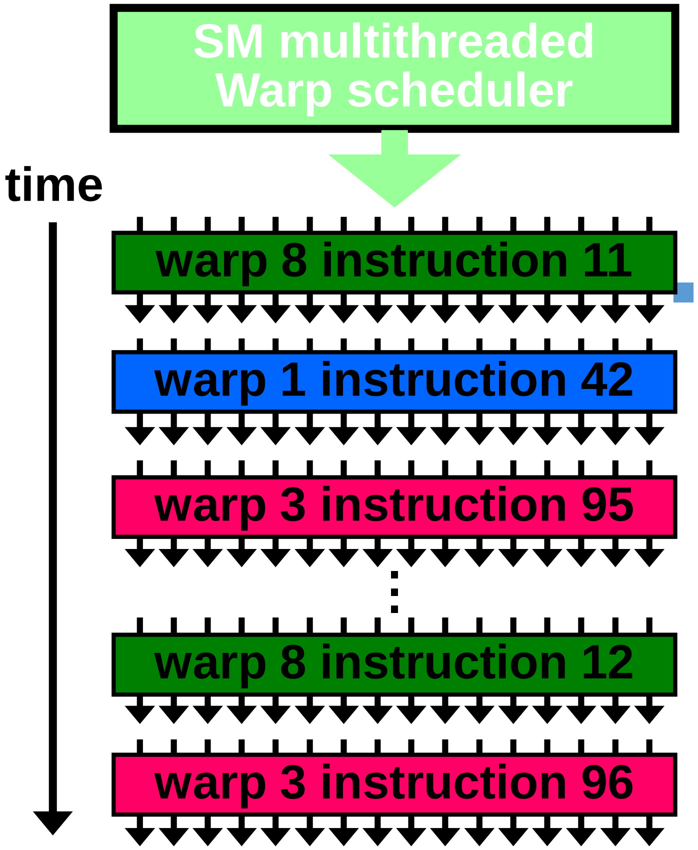
Thread Scheduling/Execution

- Each block is divided in 32-thread **warps**
  - 4 cycles per warp needed on G80
- How many warps per SM?
  - 3 blocks are assigned to a SM
  - Each block has 256 threads
  - How many warps are there in an SM?
    - Each block is divided into 256/32 = 8 Warps
    - There are 8 * 3 = 24 Warps
- At any point in time, <u>only one</u> of the 24 warps will be selected for instruction fetch and execution



SM Warp Scheduling





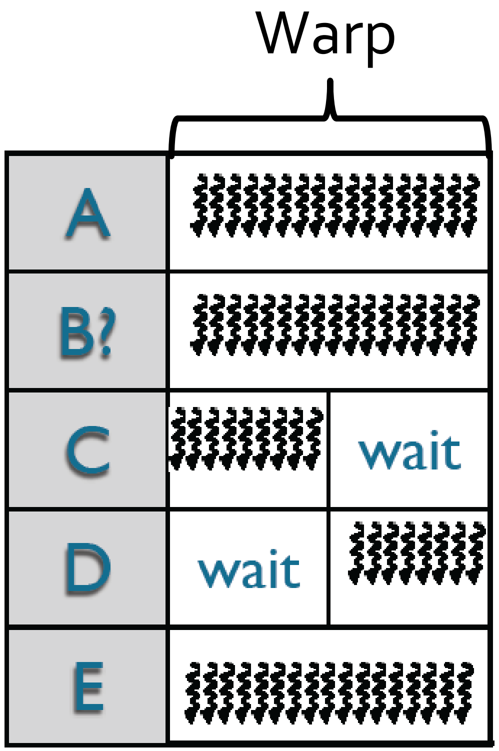
- Switching between warps allows for (memory) **latency hiding**
  - While a warp waits for data from memory, other warps (with ready data) are executed
  - No overhead: hardware implements zero-overhead Warp scheduling
- It is important to have many threads per SM (see "Use max #threads per SM") on slide 8
  - More threads => more warps per SM => higher probability to find a warp with ready data (operands) => better latency hiding

Thread Divergence at Branching

- Main performance concern with branching is **divergence**
  - = Threads within a single warp take different paths

```
A;
if B then
      C
else
      D;
E;
```

- G80: Different execution paths are serialized

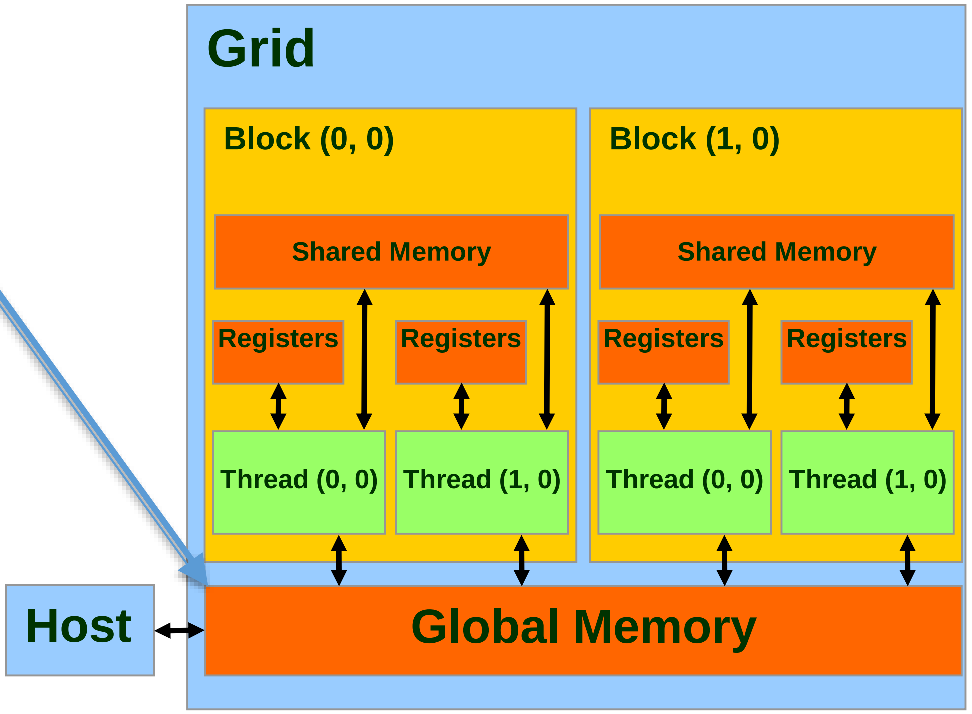


Introduction to GPU-Programming with CUDA: Memory Efficiency

Slides (partially modified) by:
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

CUDA Memory Model Overview

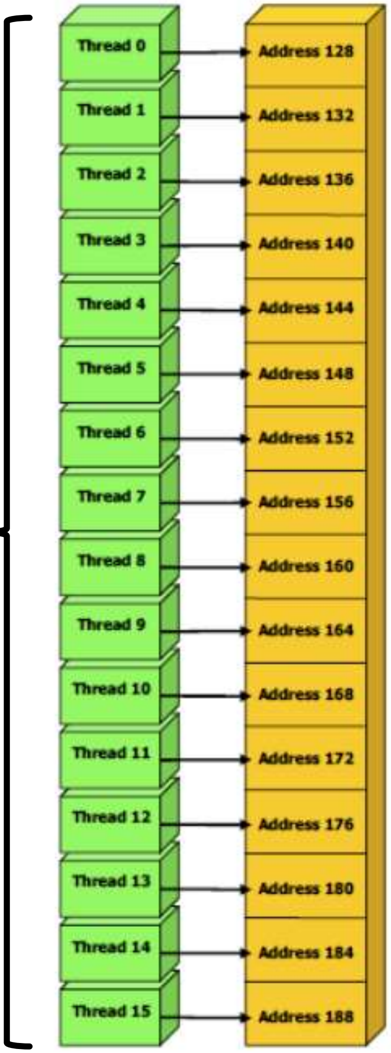
- Global memory
  - Main means of communicating R/W data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on optimizing access to global memory only



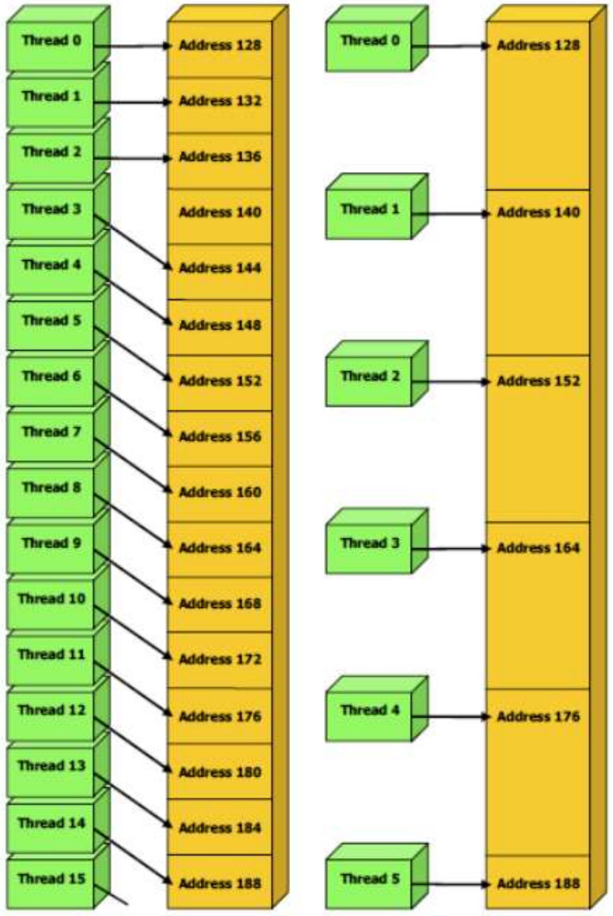
Optimizing Memory Access of a Single Warp

- When accessing global memory, peak performance utilization occurs when all threads in a half warp read or write continuous memory locations
- Such (good) access pattern is called **coalesced** access pattern (dt. vereinigt)

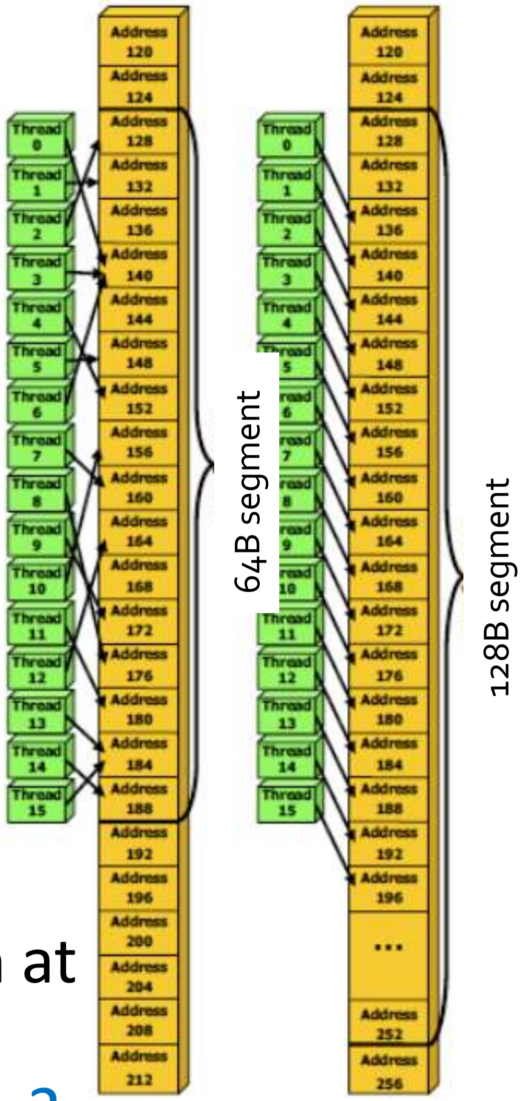
Half Warp = 16 threads



Coalesced Access and Compute Capability



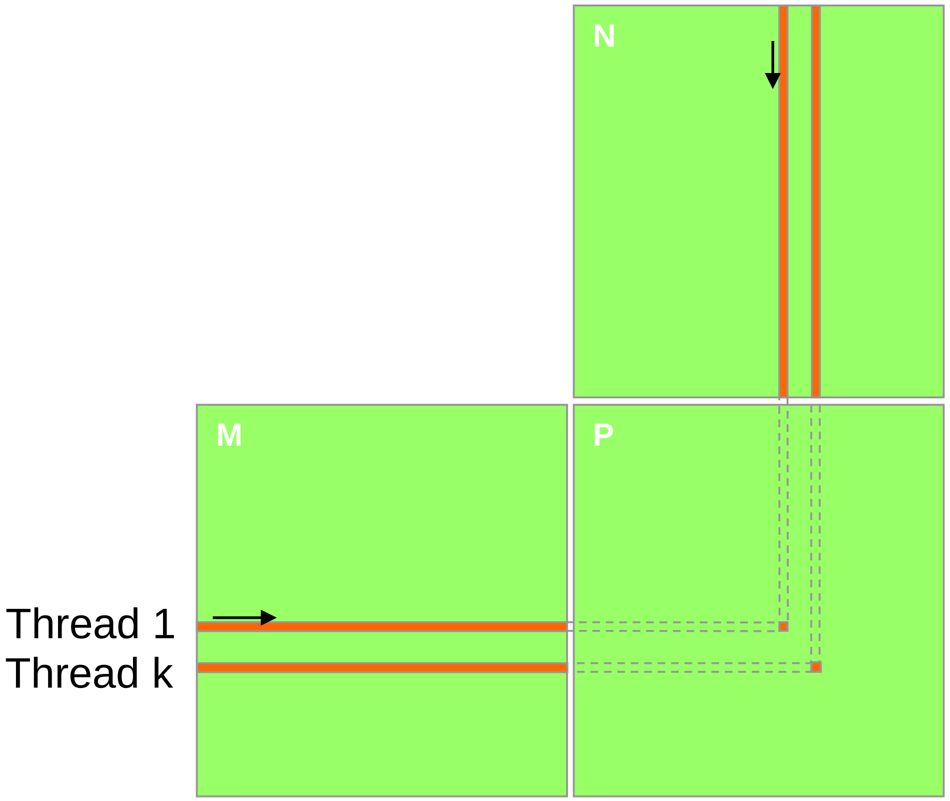
- 16 mem. transactions (each) at compute capability 1.0 or 1.1



- 1 memory transaction at compute capability 1.2

Optimizing Matrix Access /1

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Layout:

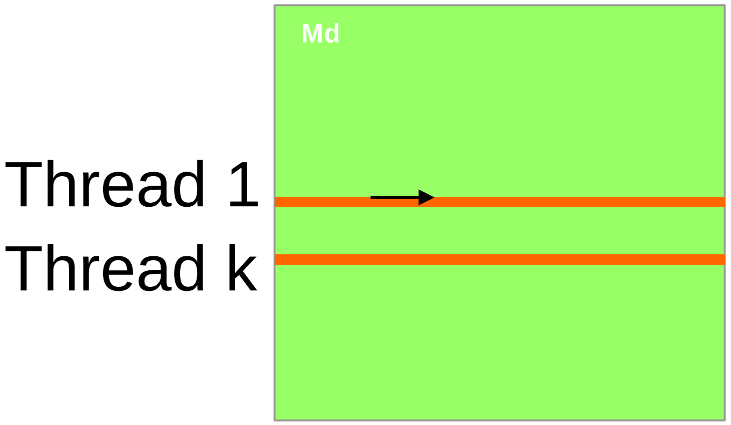
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Matrices are stored in the global memory

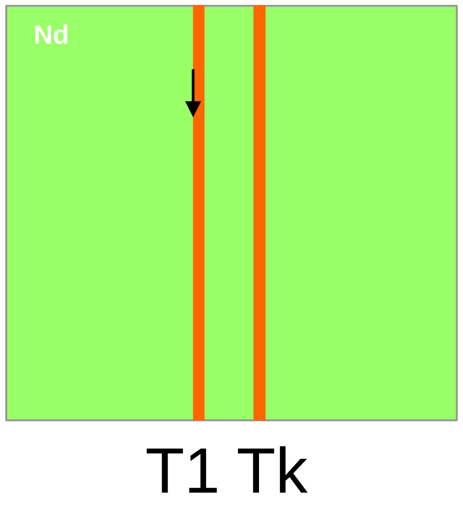
Optimizing Matrix Access /2

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Md: Good or bad?



Nd: Good or bad?

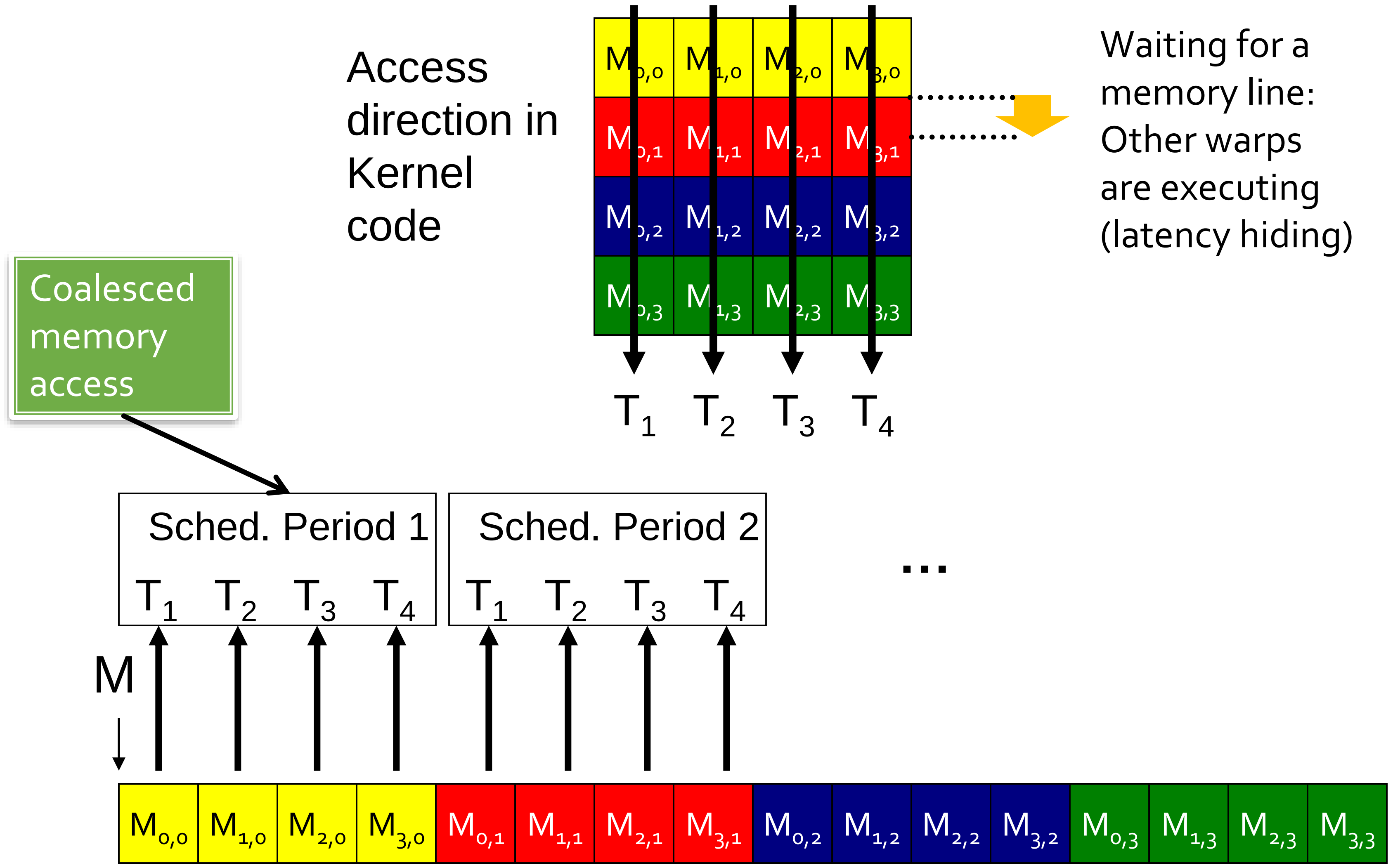


Layout:

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

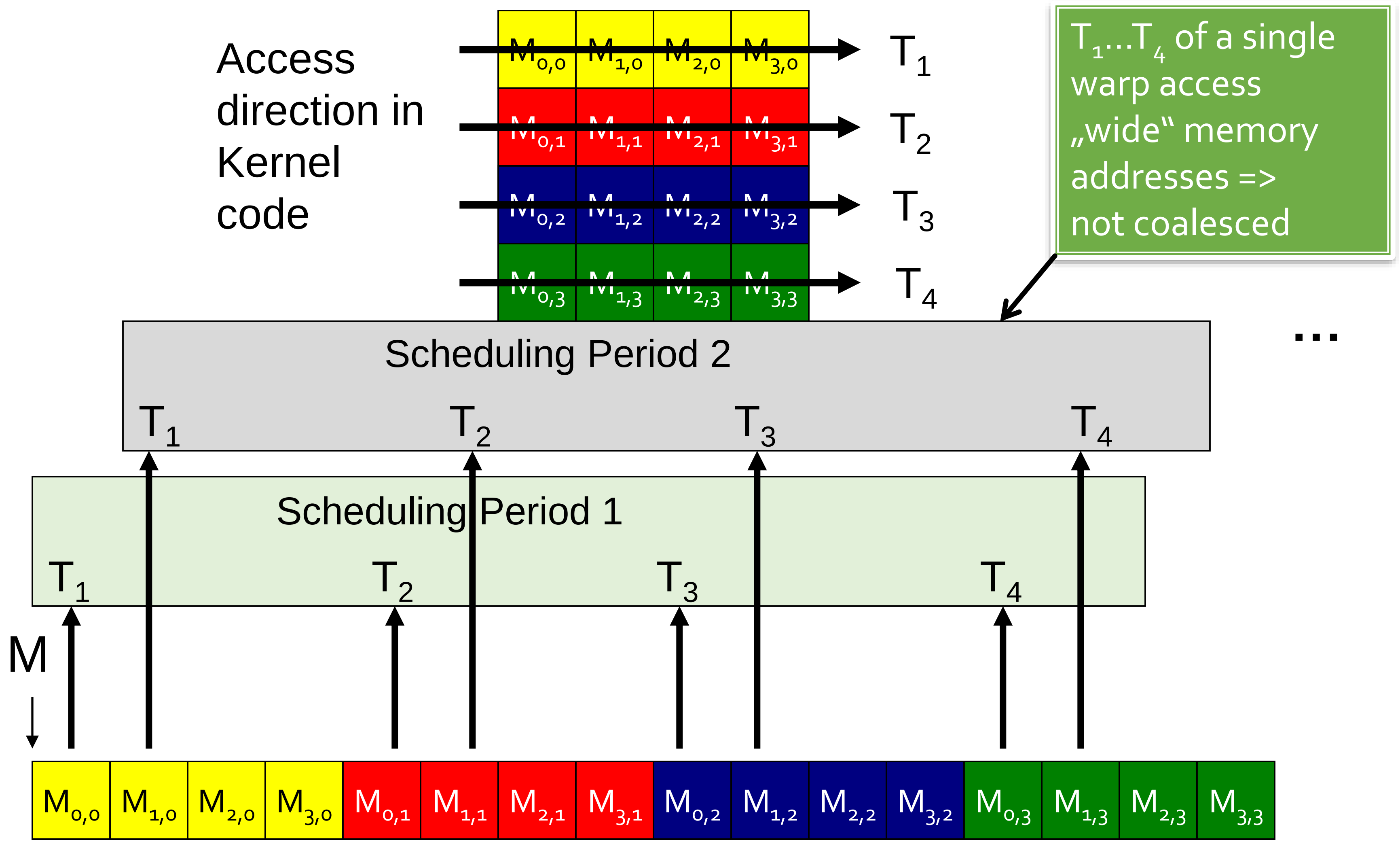
Matrices are stored in the global memory

Nd: Good or bad?



Access direction in Kernel code
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
T_1 T_2 T_3 T_4
Waiting for a memory line: Other warps are executing (latency hiding)
Coalesced memory access
Sched. Period 1
T_1 T_2 T_3 T_4
Sched. Period 2
T_1 T_2 T_3 T_4
...
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

Md: Good or bad?


Access direction in Kernel code
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ T_1
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ T_2
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ T_3
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$ T_4
Scheduling Period 2
T_1 T_2 T_3 T_4
Scheduling Period 1
T_1 T_2 T_3 T_4
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
…

T_1…T_4 of a single warp access „wide" memory addresses => not coalesced

Thank you.

Questions?

Additional Slides

Introduction to GPU-Programming with CUDA: Clustering on GPUs

Was sollen wir parallelisieren?

- Zur Vereinfachung werden i.A. nur Teile eines Programmes auf GPU programmiert
  - Aufwändiges wird auf GPU parallelisiert, Rest seriell auf CPU ausgeführt
- Was parallelisieren wir bei k-Means?
  - Berechnung der Abstände zwischen Punkten (Samples) und Centroiden und ihre Zuordnung zu den Centroiden

Serieller Code (CPU)

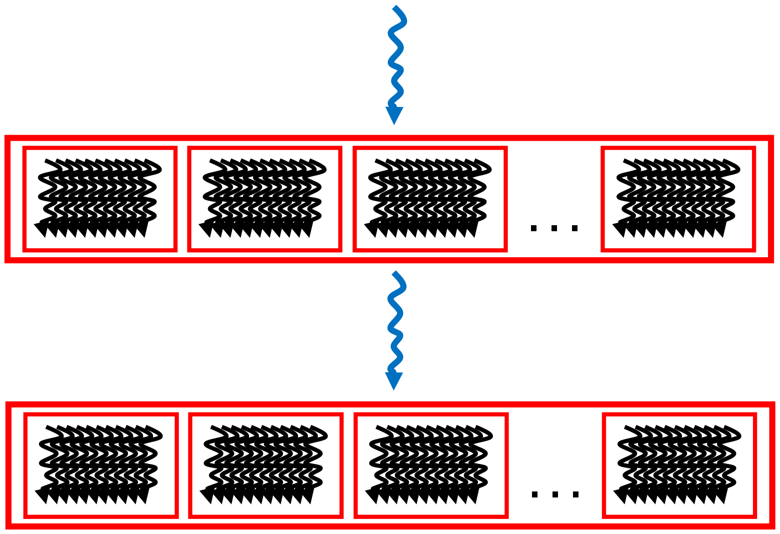
Paralleler Code (GPU)

KernelA<<< nBlk, nTid >>>(args);

Serieller Code (CPU)

Paralleler Code (GPU)

KernelA<<< nBlk, nTid >>>(args);



Kernel Routine - Übersicht

- Sei nts die Gesamtanzahl der GPU-Threads
- Der Thread t (t aus {0,..,nts-1}) bearbeitet Samples mit Indices t, t+nts, t+2*nts,..., t+p*nts (< n, #Samples)
- Für jeden dieser Samples s_i:
  - Für jeden Centroid C_j (der k aktuellen Centroide):
    - Berechne die quadrierte Entfernung $D^2(s_i, c_j)$
    - Falls $D^2(s_i, c_j)$ kleiner ist als bisherige min. Centroid-Entfernung:
      - Setze C_j als den nächsten (nearest) Centroid für s_i fest
- Warte, bis alle Threads fertig sind

Kernel – Parameter und Eingaben

- Wir haben die Anzahl der Blöcke (gridDim, ein 1D-Vektor) und die Anzahl der Threads pro Block (blockDim, auch ein 1D-Vektor) festgelegt
  - Abhängig von der GPU und Datengröße
- Jeder Thread bekommt „magisch“ die Werte:
  - gridDim, blockDim, blockIdx, threadIdx

- Wir haben als weitere Eingabe für den Kernel:
  - dataSize (= n), numDims (= d), numClusters (= k)
  - data[][] – Matrix: n Samples mit jeweils d Koordinaten
  - centroids[][] – Matrix: k aktuelle Centroide, jeweils d El.
  - membership[] – Vektor von n Integers: Zuordnung von Samples zu den Centroiden bzw. Clustern

Kernel - Pseudocode

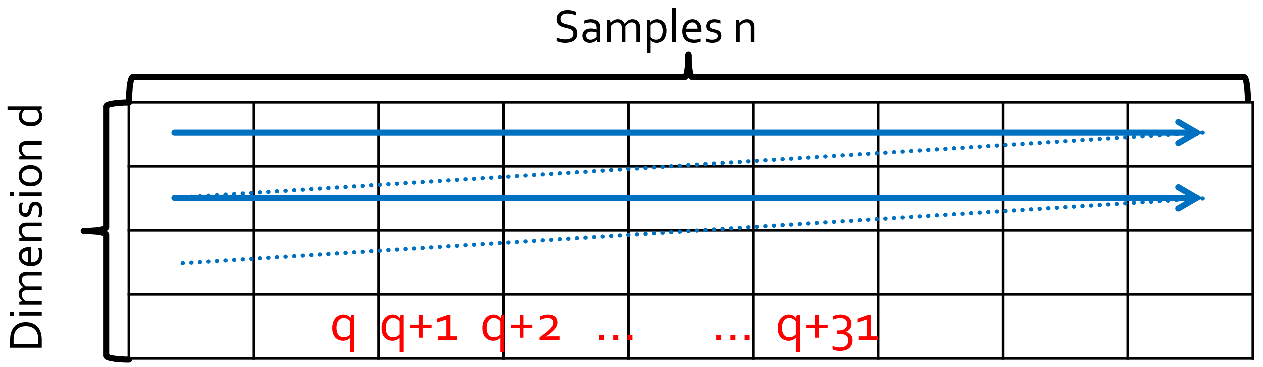
```
threadId = blockDim*blockIdx + threadIdx
nts = blockDim*gridDim               // total # of threads
for i = threadId to dataSize-1 step nts:    // get sample $s_i$
    minDistance = MAX_DOUBLE
    clusterIdx = 0
    for j = 0 to numClusters-1:               // for each centroid $C_j$ ...
        distance = 0
        for dim = 0 to numDims-1: // compute $D^2(s_i, c_j)$
            distance = distance + (data[dim][i] – centroids[j][dim])^2
        if distance < minDistance then:
            minDistance = distance
            clusterIdx = j
    membership[i] = clusterIdx
synchronize threads
```

Code für Sample S_i und Centroid C_j

```
for dim = 0 to numDims-1: // compute $D^2(s_i, c_j)$
    distance = distance + (data[dim][i] – centroids[j][dim])^2
if distance < minDistance then:
    minDistance = distance
    clusterIdx = j
```

- Roter Code: Berechnet distance = $D^2(s_i, c_j)$
- Violetter Code: Überprüft, ob distance ein neues Minumum ist, ggf. aktualisiert clusterIdx
- Der rote Code ist kritisch für die Leistung
  - Warum? Was ist da interessant?

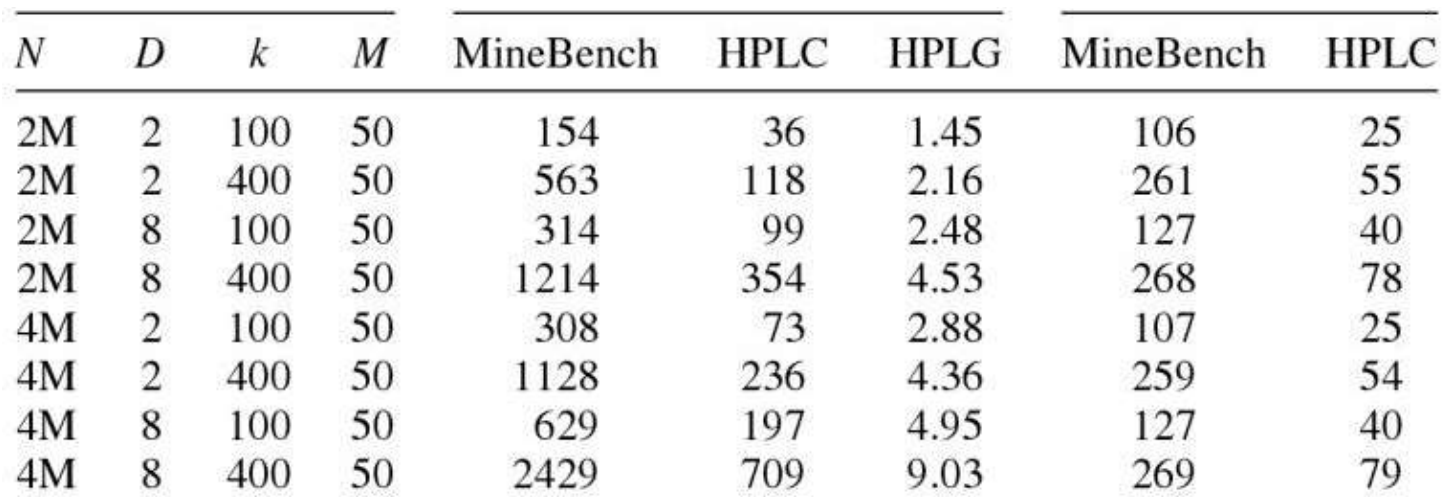
Zugriff auf Matrix data



- Matrix data ist im Speicher transponiert
  - Angrenzende Speicherzellen halten (i.A.) die gleiche Koordinate von zwei verschiedenen Samples – warum?
- distance = distance + (**data[dim][i]** – centroids[j][dim])^2
  - Diese Anweisung wird <u>gleichzeitig</u> für eine ganze Warp ausgeführt, mit i-Werten: q, q+1,q+2, ..., q+31
  - D.h. Wir bekommen einen „coalesced"-Zugriff auf data!

Referenz und Ergebnisse

- Kapitel 5 aus „Scaling Up Machine Learning“
  - Meichun Hsu, Ren Wu and Bin Zhang: „Uniformly Fine-Grained Data-Parallel Computing for Machine Learning Algorithms” (Achtung: viele grobe Fehler!)



Datensatz				Zeiten (s)			Speedups (CPU vs. GPU)	
N	*D*	*k*	*M*	MineBench	HPLC	HPLG	MineBench	HPLC
2M	2	100	50	154	36	1.45	106	25
2M	2	400	50	563	118	2.16	261	55
2M	8	100	50	314	99	2.48	127	40
2M	8	400	50	1214	354	4.53	268	78
4M	2	100	50	308	73	2.88	107	25
4M	2	400	50	1128	236	4.36	259	54
4M	8	100	50	629	197	4.95	127	40
4M	8	400	50	2429	709	9.03	269	79

- MineBench = open source kmeans; HPLC = optimiertes kmeans auf CPU; HPLG = kmeans auf GPU
- CPU: Xeon 5345@ 2.33 GHz; GPU = GeForce GTX 280 (1GB)