PDC Summer School 2016

Introduction to CUDA: Threads and Memories

2016-08-19

Michael Schliephake

Szilárd Páll

KTH – CSC – HPCViz

Overview

1. What is CUDA?
2. Heterogeneous computing
3. CUDA Threads
4. CUDA Memories
5. Matrix Multiplication

CUDA

Compute **U**nified **D**evice **A**rchitecture

- Released in 2007 / along the G80 arch
- Main goal:
  expose NVIDIA GPUs for computing
- Data-parallel computing

CUDA

Compute **U**nified **D**evice **A**rchitecture

- Parallel programming platform
  - CUDA language: C/C++ extension
  - programming model:
    SIMT hw multi-threading
  - API: ”high” & low level
- Software ecosystem: dev tools (compiler, debugger, etc.), driver, runtime, libraries, etc.

Heterogeneous computing

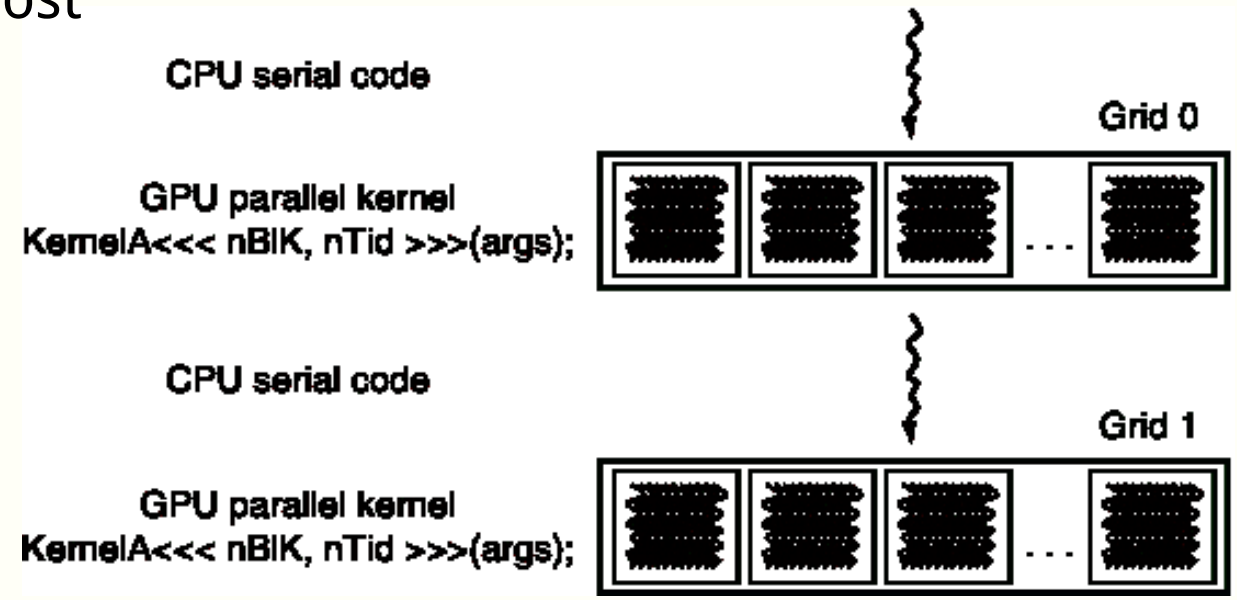
- Generally:
  use of multiple types of architectures
  (e.g. CPU+GPU, CPU+FPGA, CPU+DSP)

- Terminology
  - **Host:** CPU executing the program
  - **Device:** GPU accelerator

Heterogeneous computing: Execution Scheme

Host: CPU executing the program
Device: GPU accelerator

1. Initialization
2. Alloacate **host**- and **device**-memory
3. While „Something to do“
   - Transfer data host → device
   - Execute computational kernels on the device
   - Transfer data device → host
4. Deallocate memory
5. End.



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

CUDA GPU programming model

- Compute intensive code:

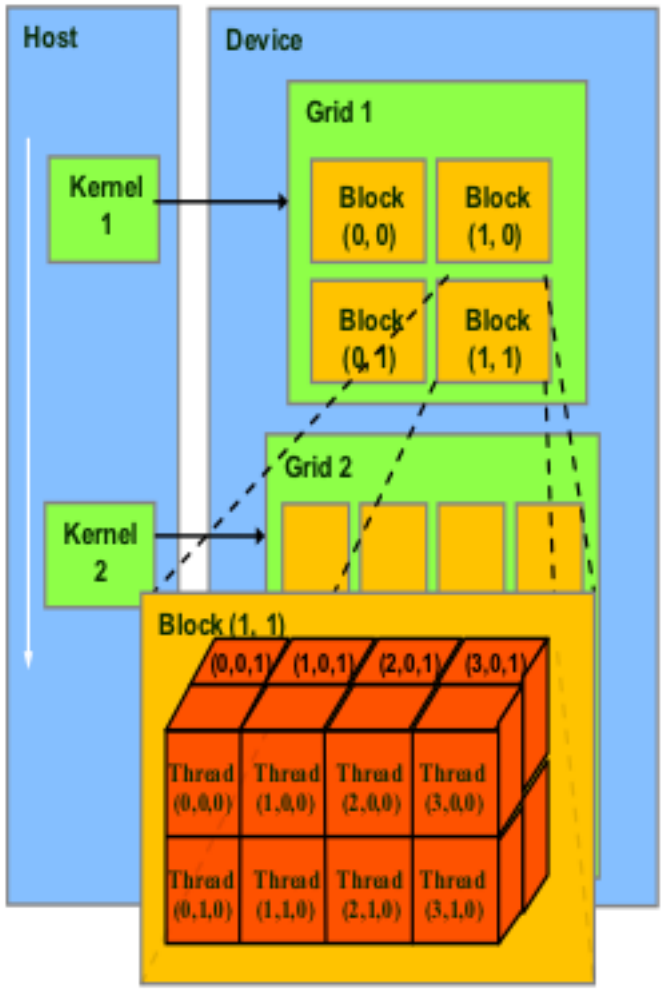
  Data-parallel part of the program executes on the device as a **kernel**

- Kernels are **launched** on the device

  → executed by many threads

- One kernel at a time
  - Slightly relaxed more recently

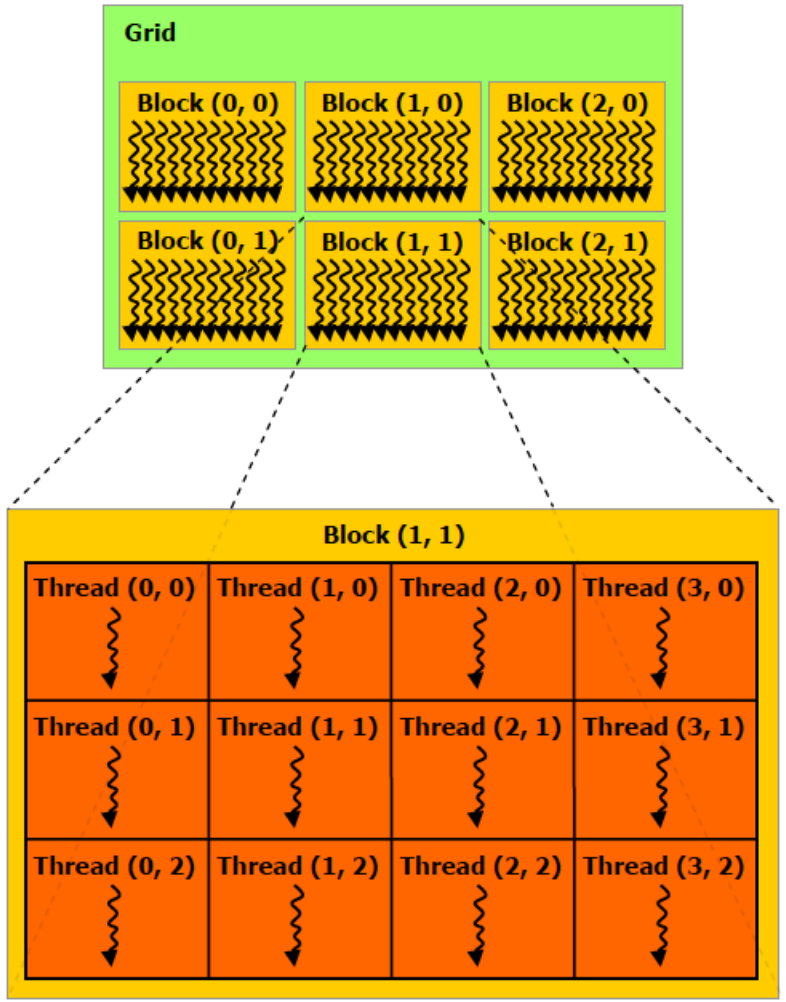
Grids, Blocks, Threads



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Threads and Blocks

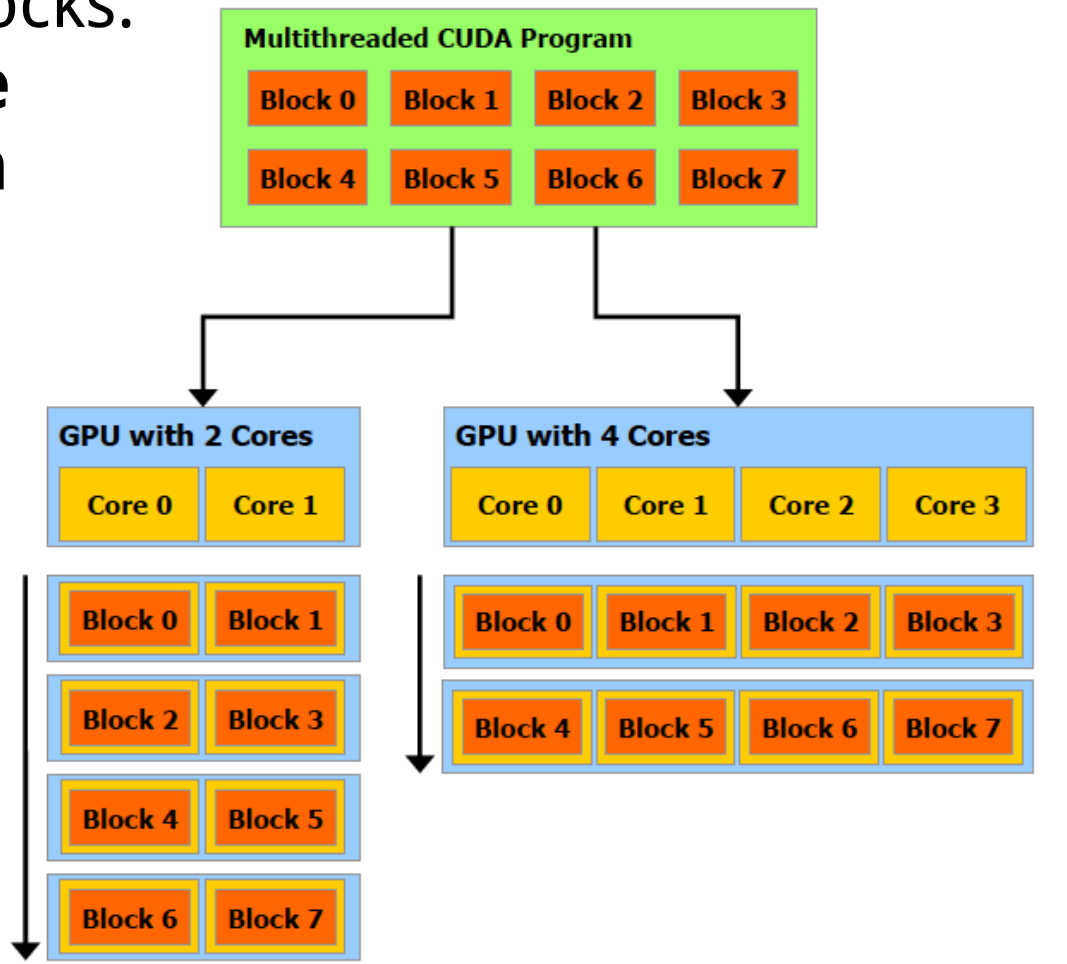
- Thread
  - Independent thread of execution
- Block
  - correspond to parallel tasks
  - must be independent
    - Execution order not determined, sequential or parallel
    - Coordination possible



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Scalable Execution

Independence of blocks: guarantees **scalable program execution**



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

CUDA threading

- Kernels execute (data) **parallel part** of the program on the device
- Program for single thread first

  optimize for execution later (idealized)

- CUDA threads != CPU threads
  - Data-parallel tasks (SIMD)
  - **Lockstep** execution
  - **Lightweight:**

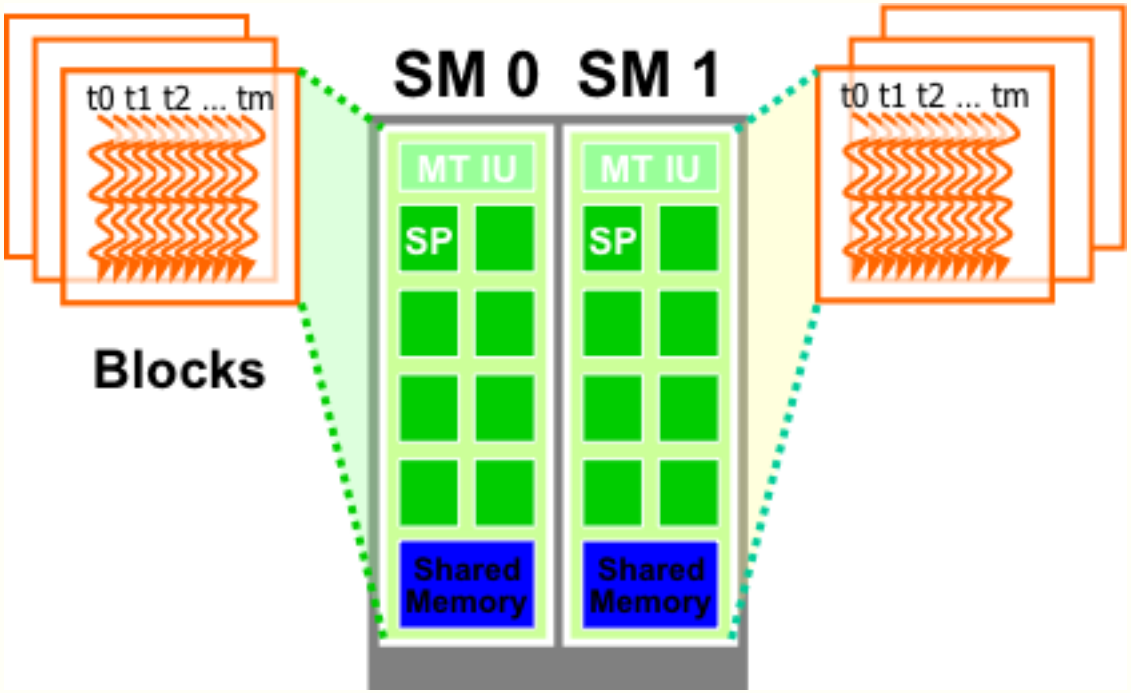
    very low creation and context switch overhead

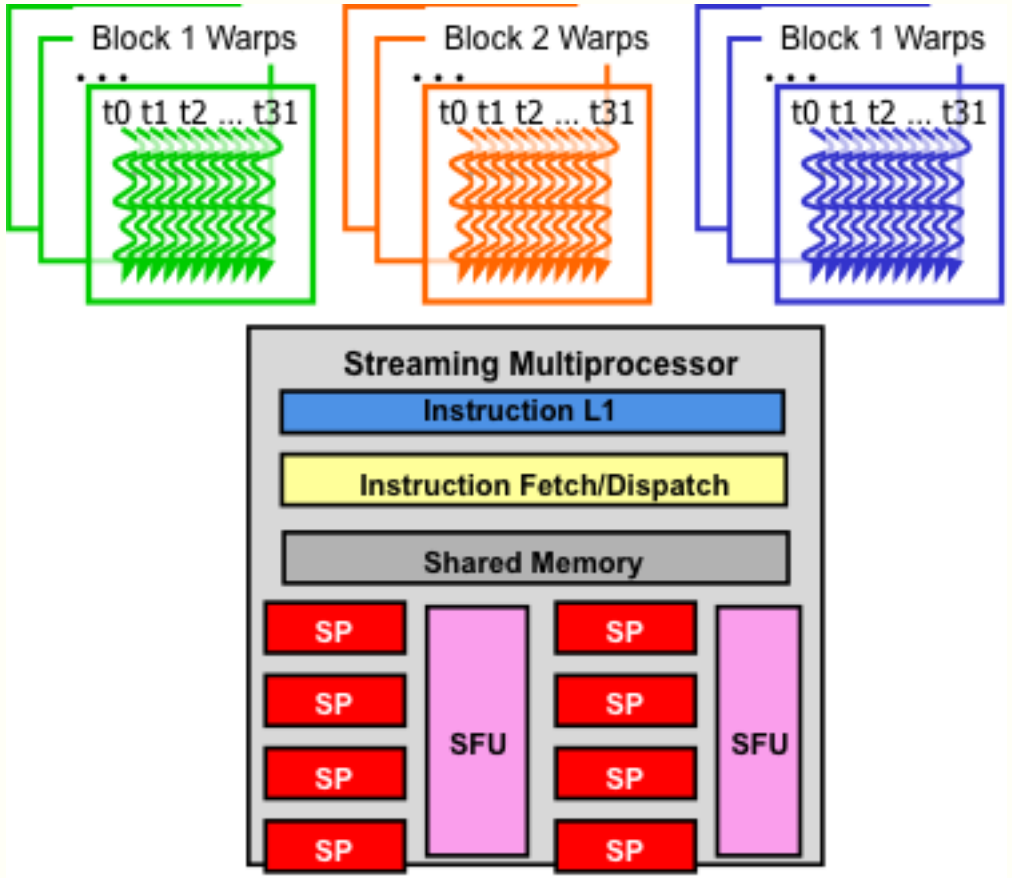
SIMT: Single Instruction Multiple Data

- NVIDIA hardware threading **programming model flexibility vs efficiency**

- **Warp:** unit of hardware execution, 32 (for now)
  - same instruction per cyle
  - Fast switch → latency hiding
- **High throughput**: large # of threads needed
  - Large enough problem
  - Over-decomposition

Warp Scheduling






```
void __syncthreads();
```

Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

CUDA Hello World

```
__global__ void kernel( void ) {
}

int main( int argc, char *argv[] ) {
   kernel<<<1,1>>>();
   printf(”Hello, world!\n”);
   return 0;
}
```

CUDA compilation

- C extensions: needs a compiler
  - C++ support (CUDA 7.0: C++11)
- Host & device code can be mixed
  - Multi-pass compilation
  - Runtime API
- Mixed / device code is compiled using nvcc
  - Source to source compiler for host
    - calls ”host” C++ compiler (e.g. gcc)
  - GPU device code generator
  - Links the runtime

CUDA compilation (cont.)

- Nvcc generates PTX or binary code
- Just-in-time compilation of PTX code
- Binary code (architecture specific)
  - f.ex. `-code=sm_13`
  - forward-compatible within major revision
- Certain PTX features require minimum revision
  - f.ex. `-arch=sm_20,sm_37`
  - forward-compatible
- Compilation to different files for future architectures

Function Modifier

	Execution	Callable from
__**device**__ float deviceFunc()	device	device
__**global**__ void kernelFunc()	device	host
__**host**__ float hostFunc()	host	host

Specification of Host and Device Code

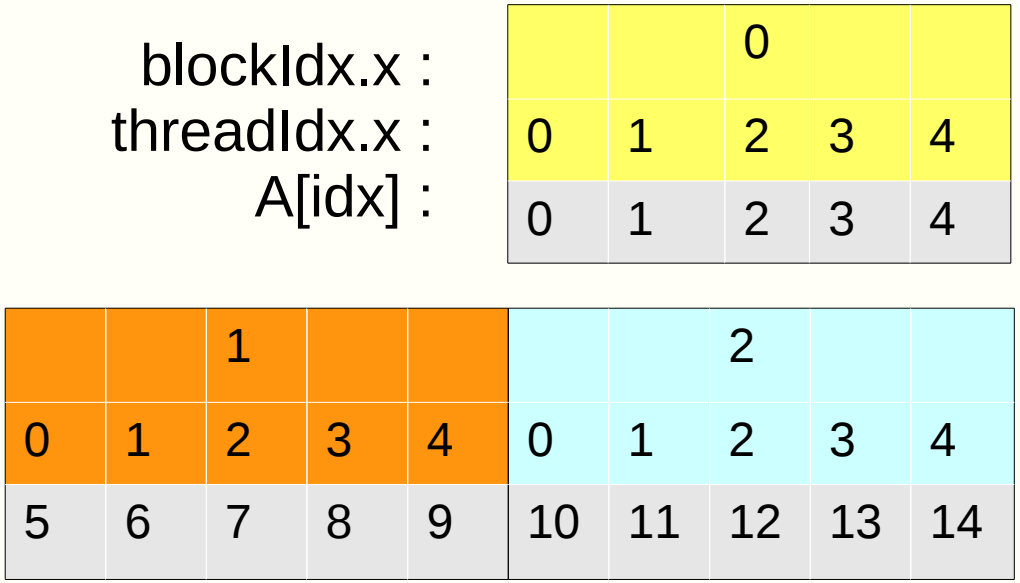
- Execution Kernel

```
function_name<<<nBlock, nThread>>>(args);
```

  - `nBlock` – number of blocks
  - `nThread` – number of threads
  - args – pointer to data etc.
- Thread learns its position via
  - `blockIdx, threadIdx` – global variables for current Position
  - `gridDim, blockDim` – global variables for extensions of the indices

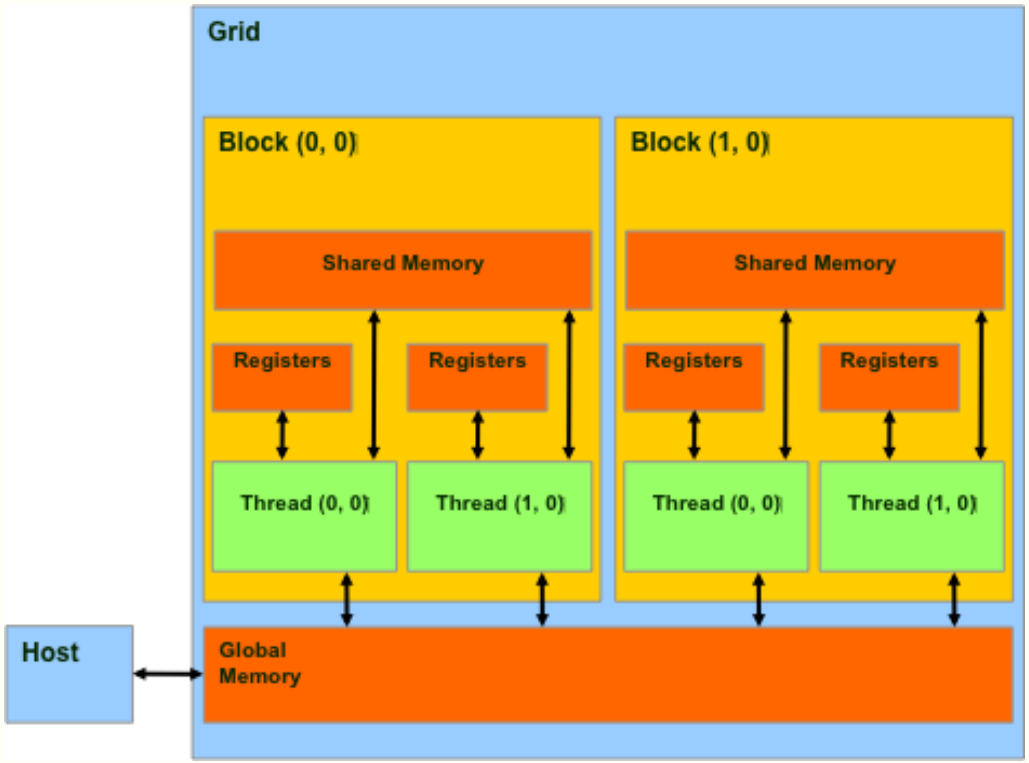
Work Decomposition Example

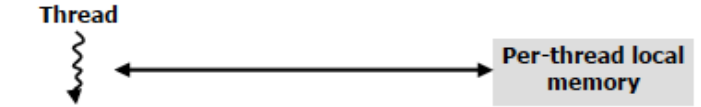
```
__global__ void kFunc(int *A) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    A[idx] = idx;
}

void main() {
// Allocate and

initialise data memories

dim3 grid, block;
grid.x  = 3;
block.x = 5;
kern<<<grid, block>>>(d_A);

// Copy results back
}
```

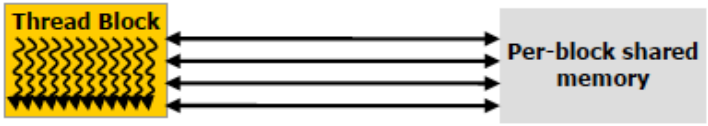


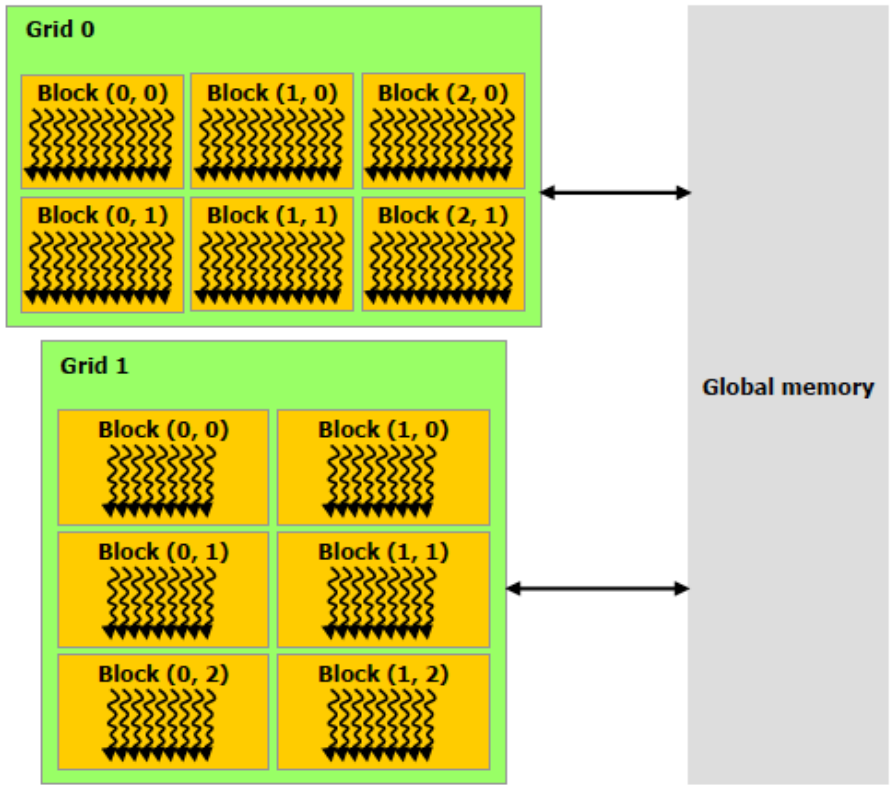
Memory access

Simplified View to the execution of a block





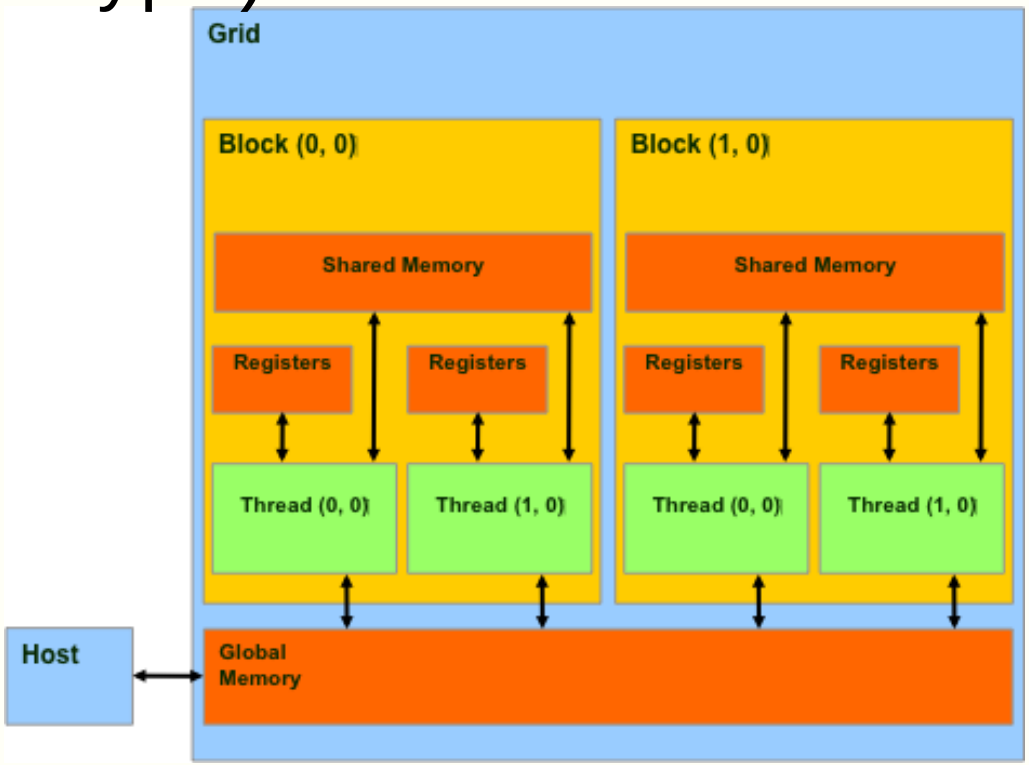




Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Memory Allocation and Transfer

- `cudaMalloc(pointer, size)`
- `cudaFree(pointer)`
- `cudaMemcpy(dest, src, type)`
  - Host -> Host
  - Host -> Device
  - Device -> Host
  - Device -> Device



ImageCourtesy of: David Kirk/NVIDIA and Wen-mei W. Hwu

Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Declaration of Variables

Declaration	Memory	Scope	Lifetime
__device__ __local__ int LocalVar	local	thread	thread
__device__ __shared__ int SharedVar	shared	block	block
__device__ int GlobalVar	global	grid	application
__device__ __constant__ int ConstantVar	constant	grid	application

- Automatic Variables
  - In register
  - Arrays: registers or local memory (if large or indexed dynamically)

GPU memory types & characteristics

- **Global** memory: relatively slow
  - Cached (L1 and/or L2)
  - Requires coalescing
- **Shared** memory: programmable on-chip memory (cache)
  - Banked, 1 word/thread/warp throughput
    → without **bank conflicts!**
- **Constant** memory:
  - Read-only, fully cached gmem segment
  - very efficient for read-only **uniform** use
- Texture memory

Unified memory

- Same memory on CPU & GPU
  - Automatic migration
  - Skip cudaMalloc/cudaMemCpy
  - Avoid complex data flow
- Limitations
  - Coherence only at kernel launch & sync
  - Concurrent modification not allowed
  - granularity
- Pascal improvements

CUDA Memories

First Shared Memory Example

```
__global__ void slow_access(int *data,...)
{
   // compute this thread's position
   unsigned int i = blockDim.x*blockIdx.x
                                + threadIdx.x;

   if (i > 0) {
      // Load needed data
      int ia = data[i];
      int ib = data[i-1];
      // Do some computation
      //...
   }
}
```

Every element is read twice from the global memory

CUDA Memories

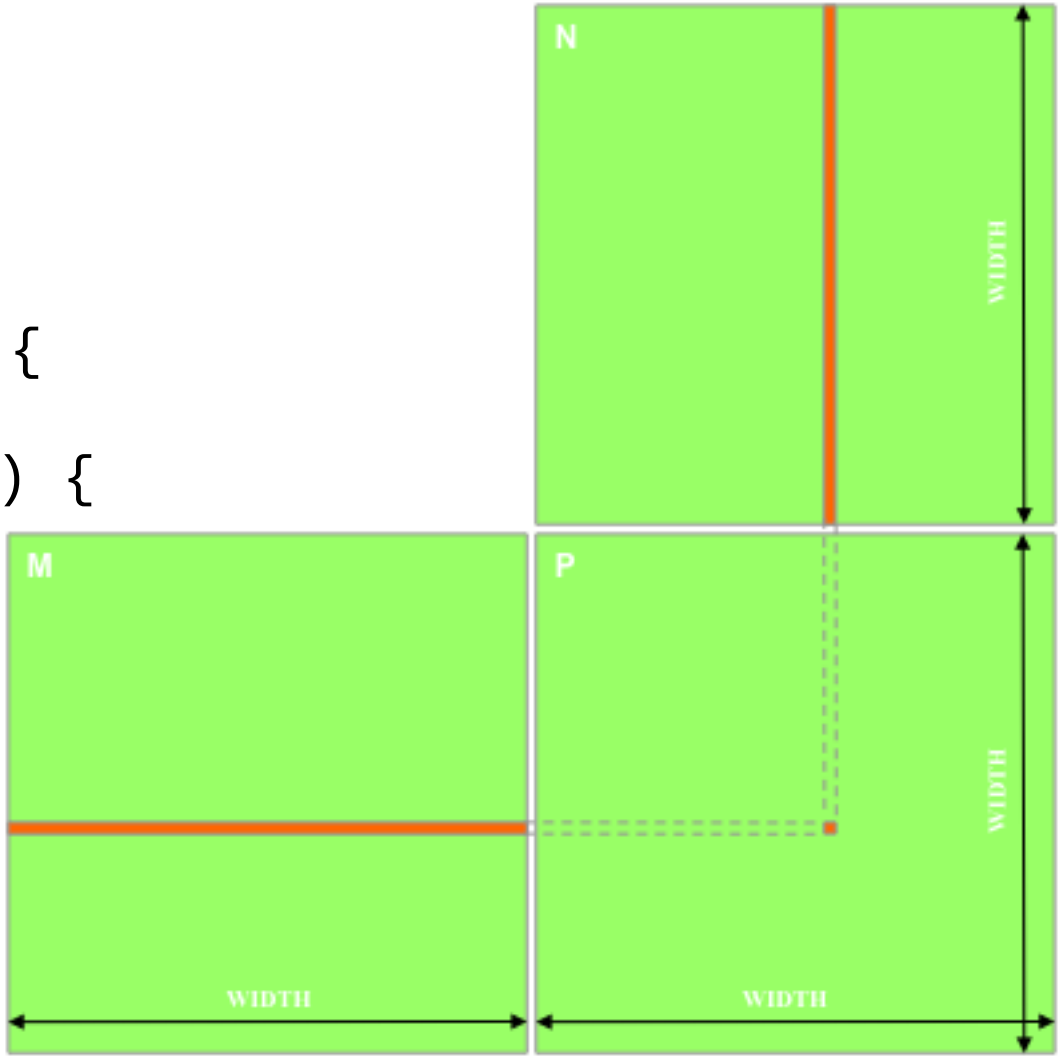
First Shared Memory Example

```
__global__ void faster_access(int *data,...)
{
    // Compute this thread's position
    int tx = threadIdx.x;
    __shared__ int data_sh[BLOCK_SIZE];
    int i = blockDim.x*blockIdx.x + tx;

    data_sh[tx] = data[i];
    _syncthreads();
    if (tx > 0) {
        // Compute with data_sh[tx] and data_sh[tx-1]
        // ...
    }
    else if(i > 0) {
        // Handle thread block boundary
        // Compute with data_sh[tx] and data[i-1]
        // ...
    }
}
```

Every element is read once by one thread of the block
→ Half the memory accesses compared to previous example

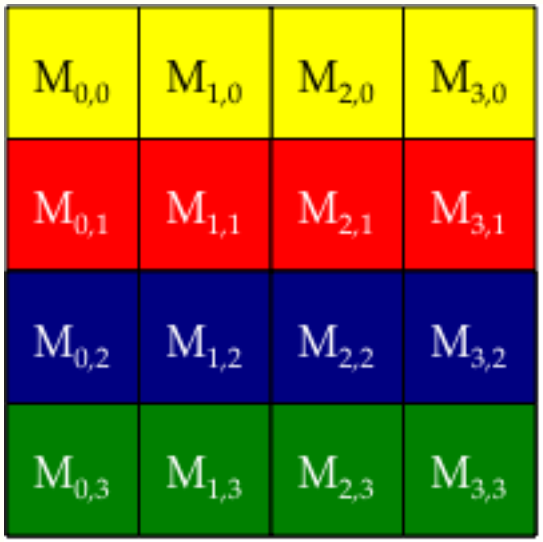
Matrix multiplication
Base algorithm

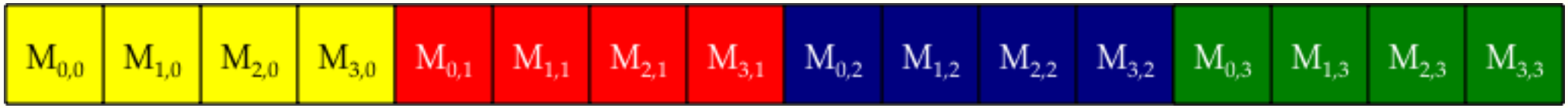
```
float M[WIDTH][WIDTH];
float N[WIDTH][WIDTH];
float P[WIDTH][WIDTH];

for (int i = 0; i < WIDTH; ++i)
  for (int j = 0; j < WIDTH; ++j) {
    float sum = 0.;
    for (int k = 0; k < WIDTH; ++k) {
      float a = M[i*WIDTH + k];
      float b = N[k*WIDTH + j];
      sum += a * b;
    }
    P[i * Width + j] = sum;
  }
}
```



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Memory Layout of Matrices





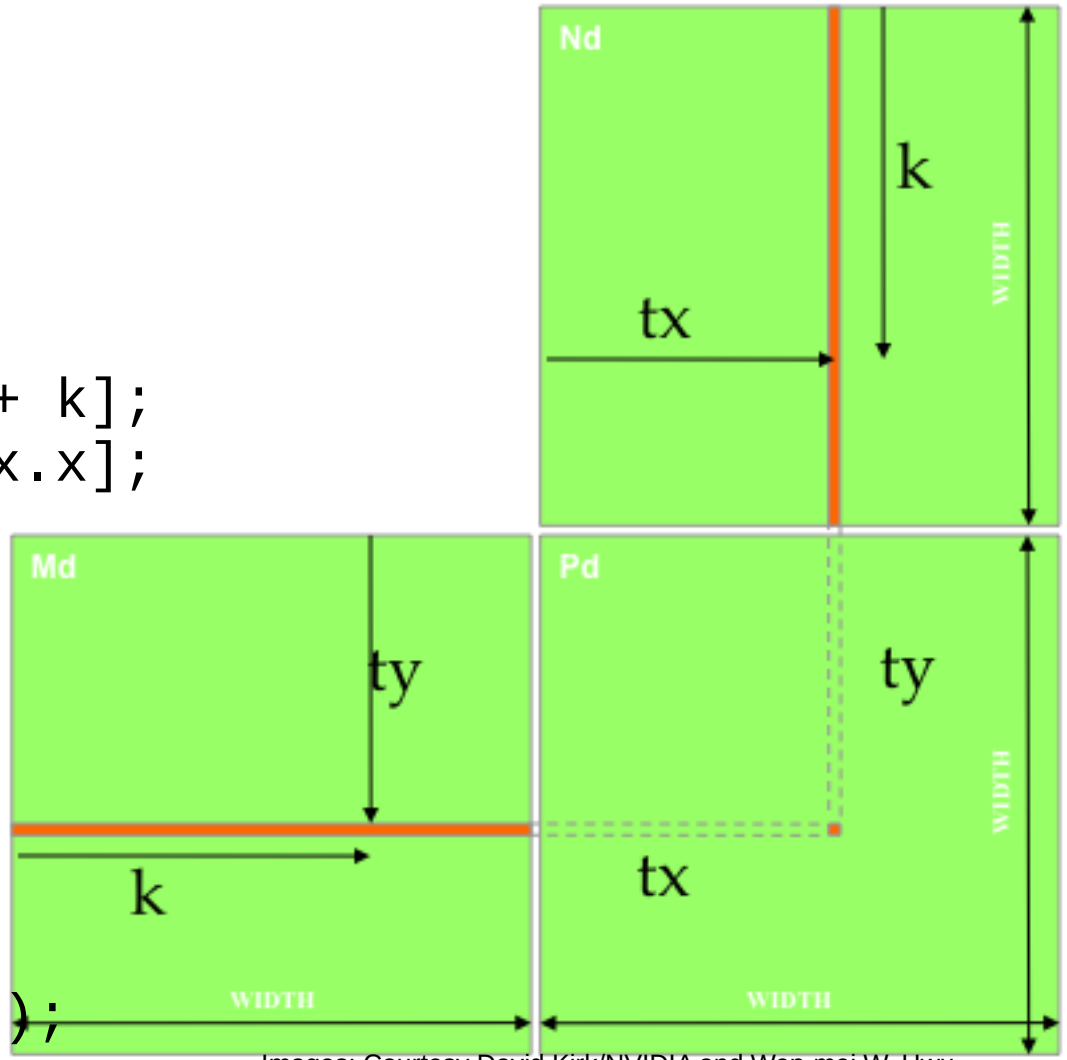
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Compute Kernel Fragment

```
float Md[WIDTH*WIDTH];
float Nd[WIDTH*WIDTH];
float Pd[WIDTH*WIDTH];

float sum = 0.;
for (int k = 0; k < WIDTH; ++k) {
  float a = Md[threadIdx.y*WIDTH + k];
  float b = Nd[k*WIDTH + threadIdx.x];
  sum += a * b;
}
Pd[threadIdx.y*WIDTH
      + threadIdx.x] = sum;
```

Kernel activation

```
dim3 dGrid(1,1),
      dBlock(WIDTH, WIDTH);
kernelfunc<<<dGrid, dBlock>>>(...);
```



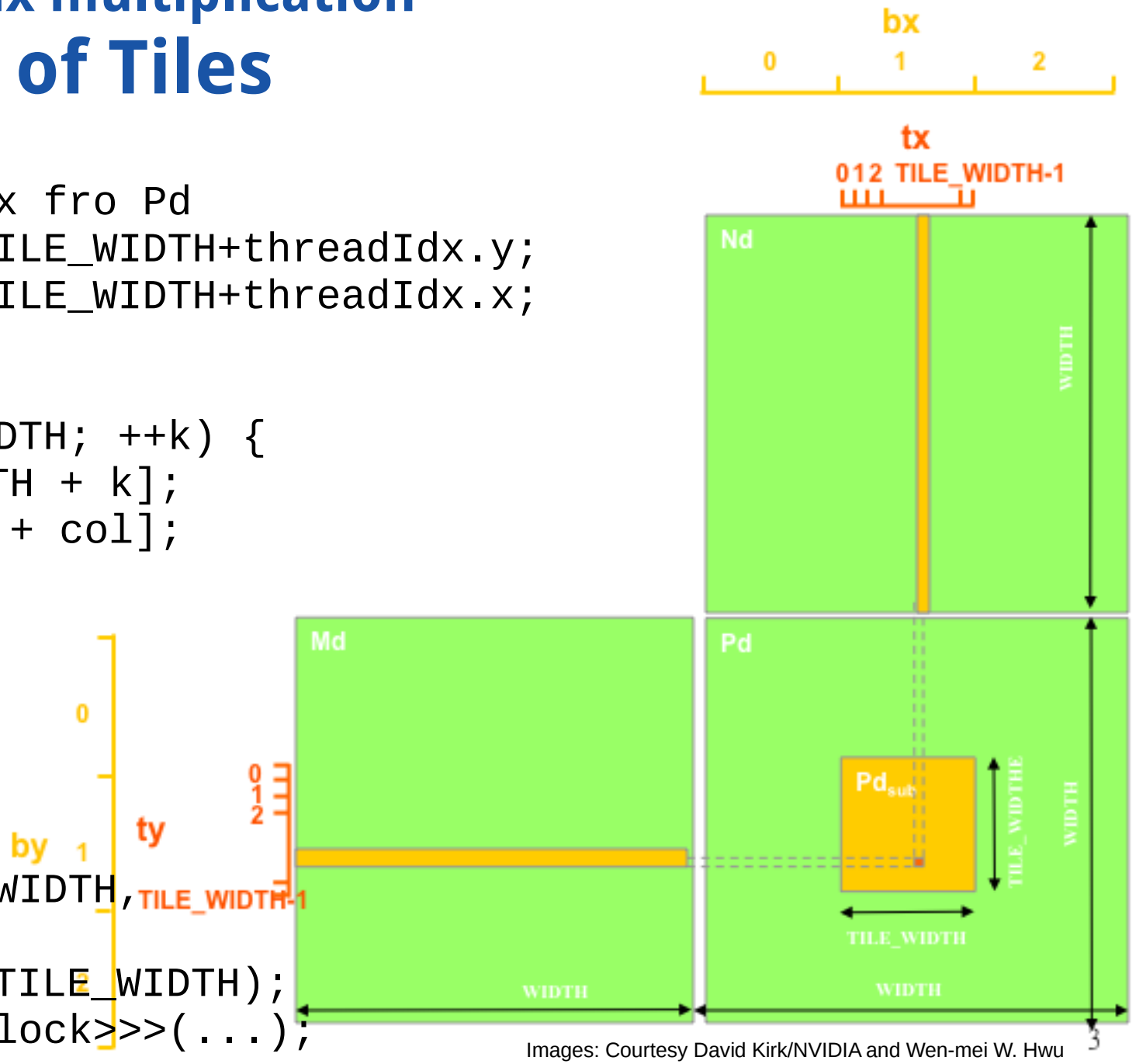
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Use of Tiles

```
// Row and column index fro Pd
int row = blockIdx.y*TILE_WIDTH+threadIdx.y;
int col = blockIdx.x*TILE_WIDTH+threadIdx.x;

float sum = 0.;
for (int k = 0; k < WIDTH; ++k) {
  float a = Md[row*WIDTH + k];
  float b = Nd[k*WIDTH + col];
  sum += a * b;
}
Pd[row*WIDTH
    + col] = sum;
```

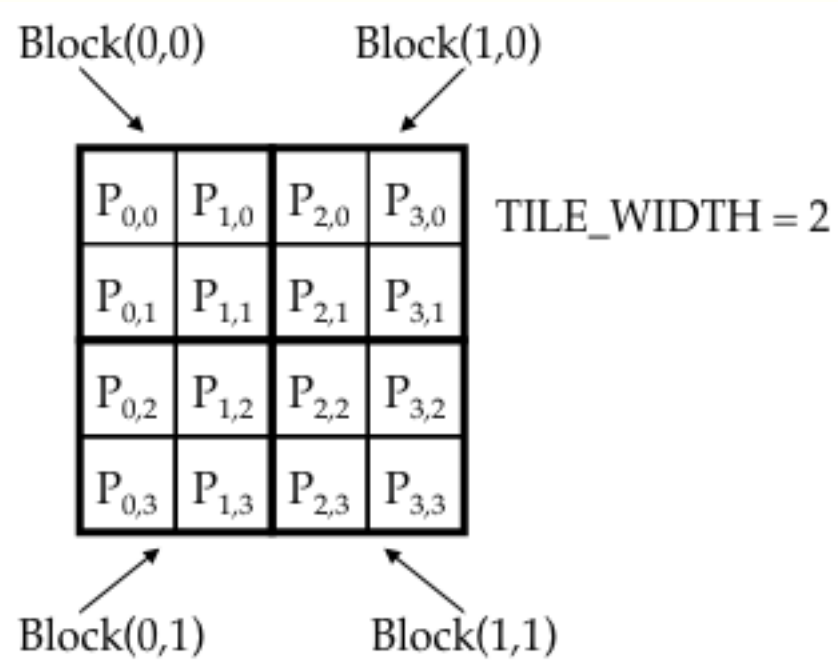
Kernel activation

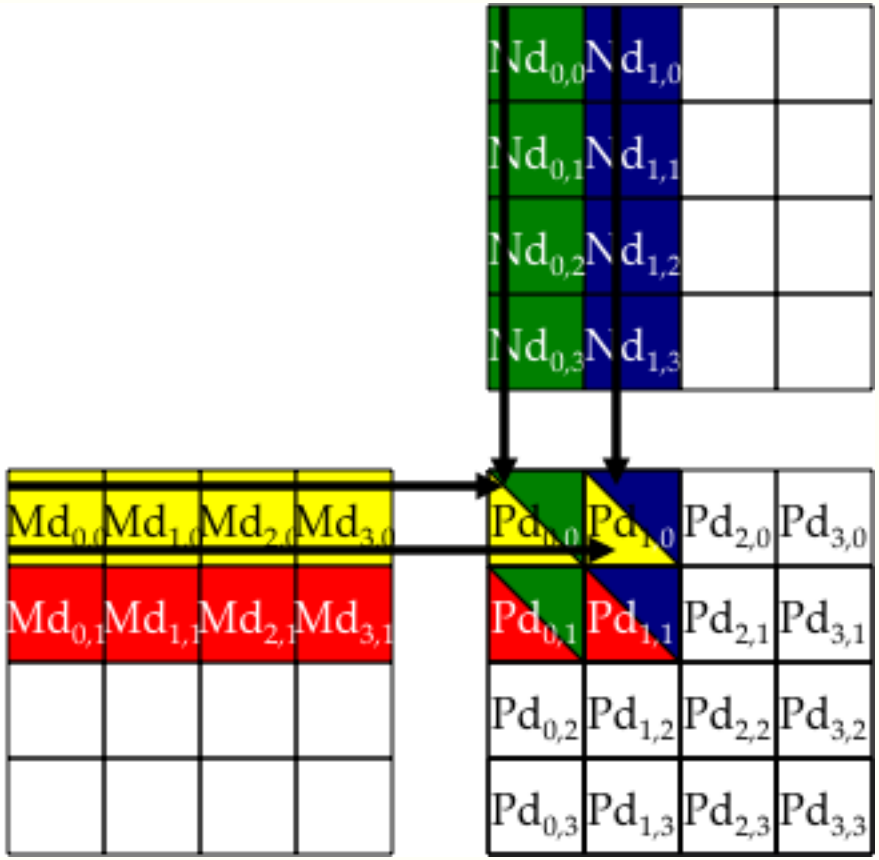
```
dim3 dGrid(WIDTH/TILE_WIDTH,
   WIDTH/TILE_WIDTH),
   dBlock(TILE_WIDTH, TILE_WIDTH);
kernelfunc<<<dGrid, dBlock>>>(...);
```



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Example for Tiles

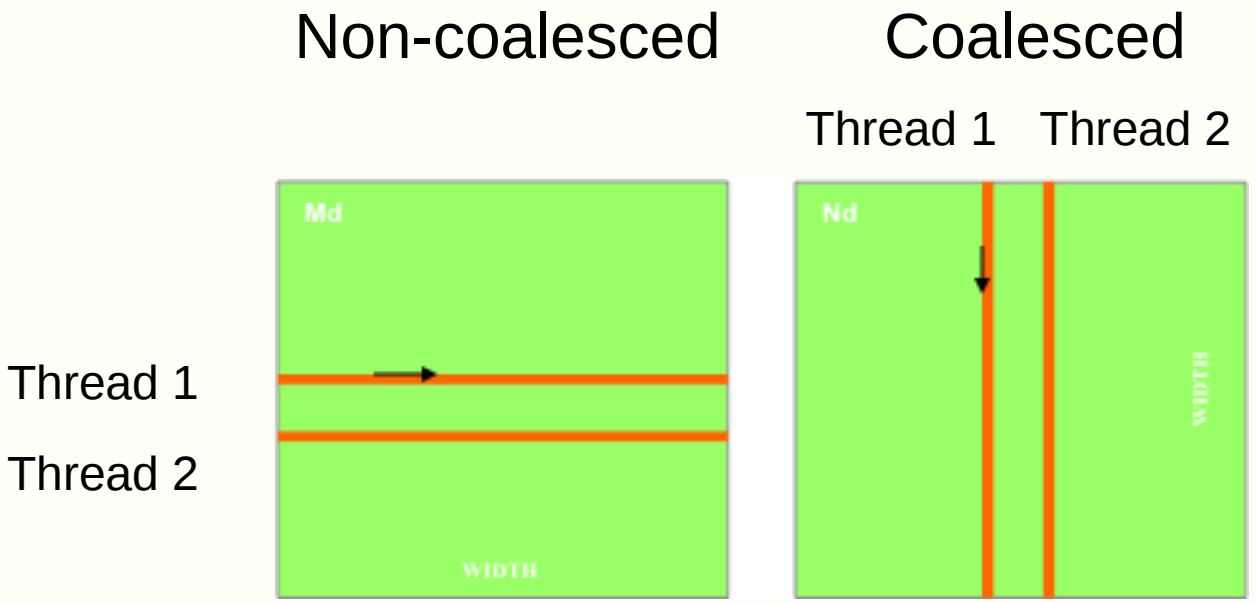




Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Memory Coalescing I

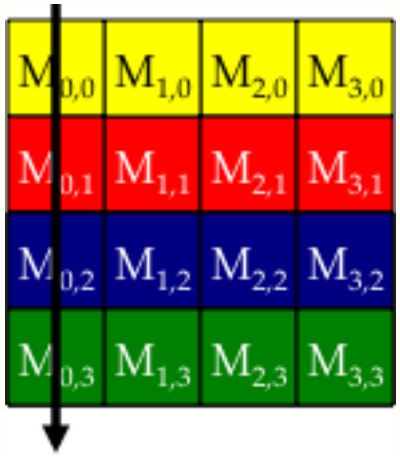
- Accessign consecutive memory locations in a warp increases performance



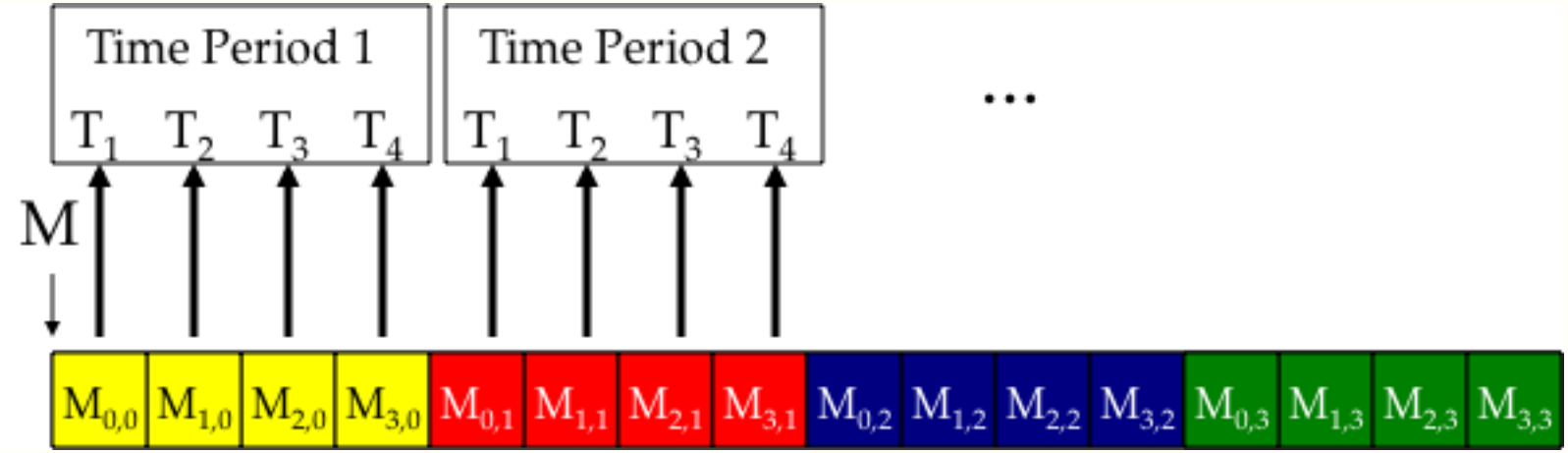
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Memory Coalescing II

Access order in the algorithm



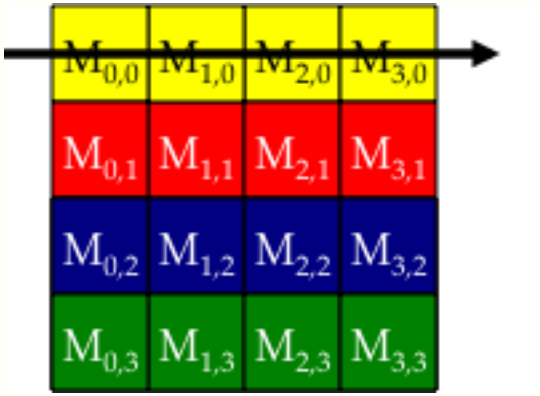
Device memory access



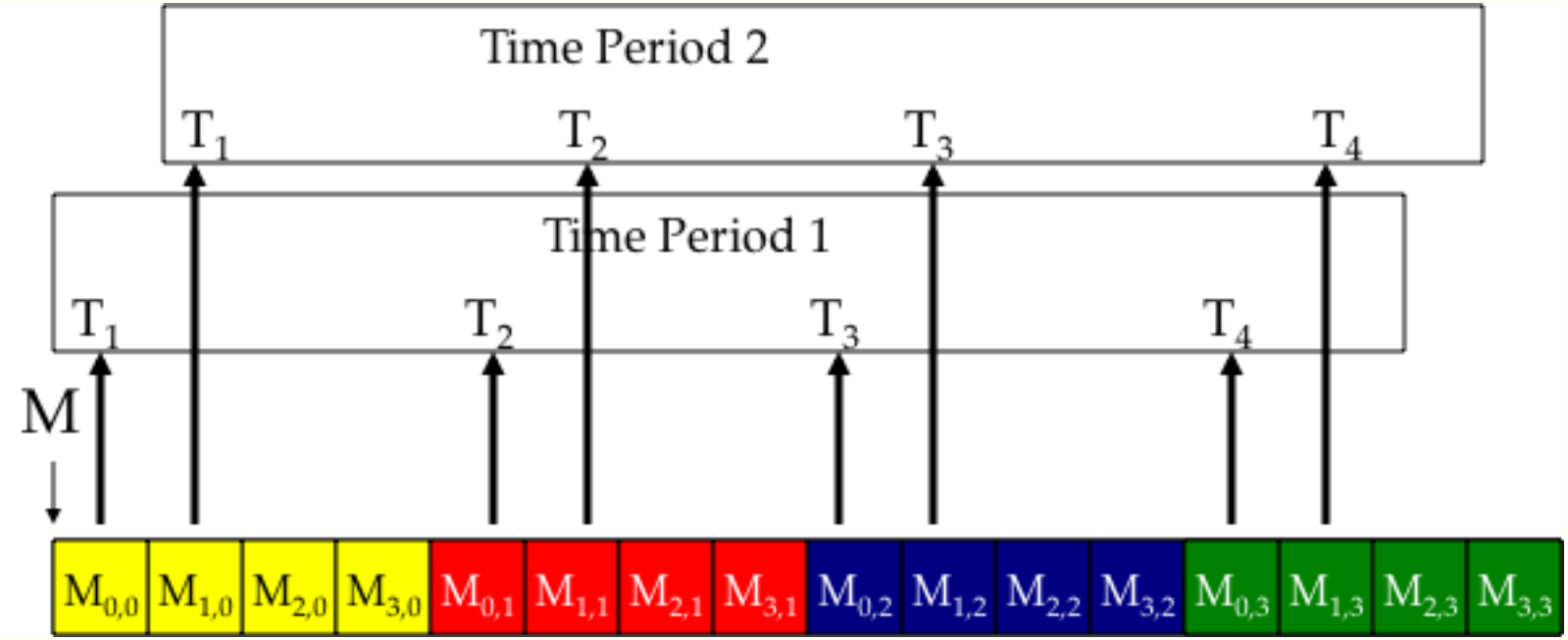
Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Memory Coalescing III

Access order in the algorithm

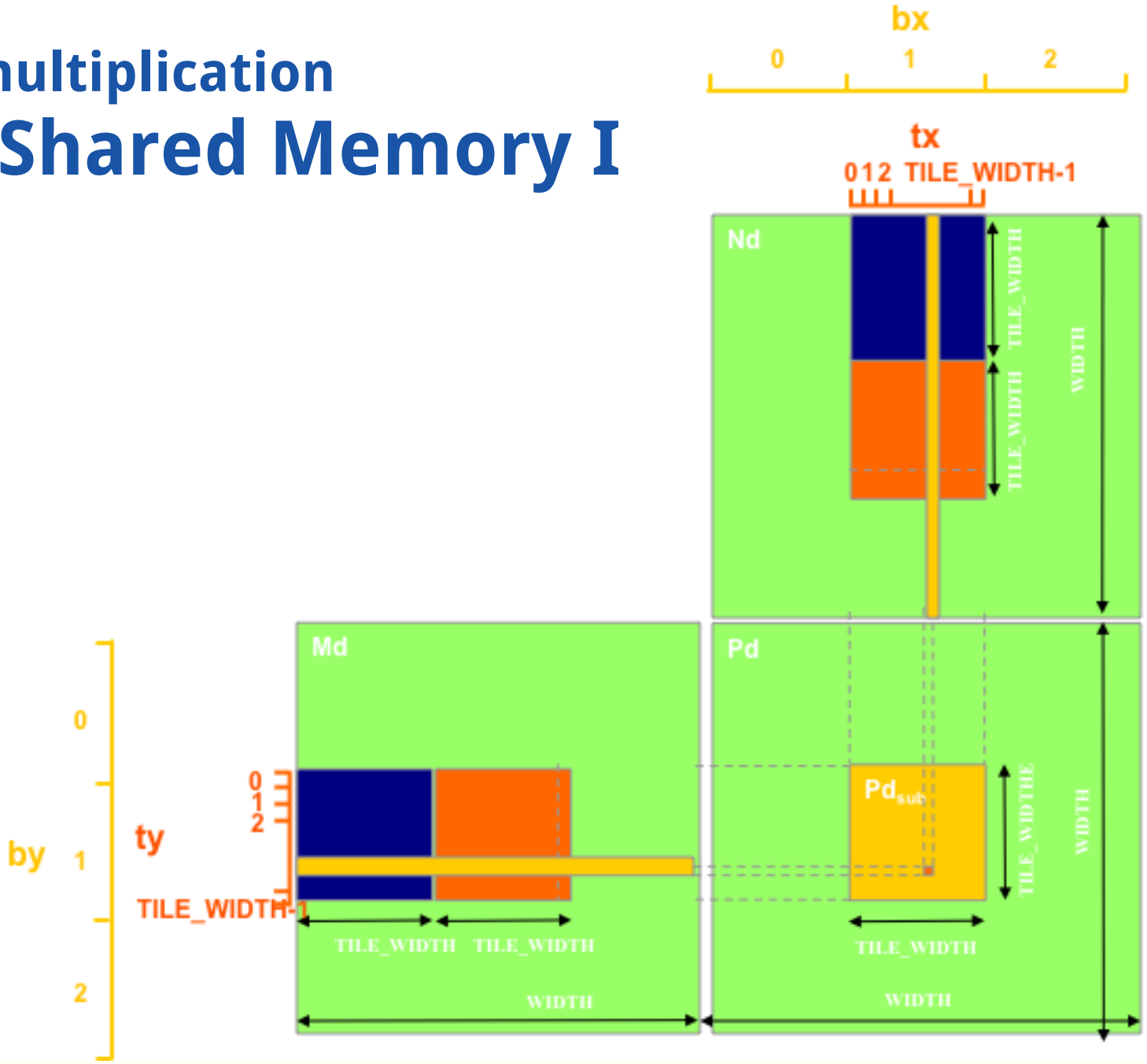


Device memory access



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication
Using Shared Memory I



Images: Courtesy David Kirk/NVIDIA and Wen-mei W. Hwu

Matrix multiplication

Using Shared Memory II

```
__shared __float Mds[TILE_WIDTH][TILE_WIDTH];
__shared __float Nds[TILE_WIDTH][TILE_WIDTH];
int bx = blockIdx.x;  int by = blockIdx.y;
int tx = threadIdx.x; int ty = threadIdx.y;
int Row = by*TILE_WIDTH + ty;
int Col = bx*TILE_WIDTH + tx;
float sum = 0;

for (int m = 0; m < Width/TILE_WIDTH; ++m) {
      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
      Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
      __syncthreads();
      float sum;
      for (int k = 0; k < TILE_WIDTH; ++k)
        sum += Mds[ty][k] * Nds[k][tx];
      __syncthreads();
}
Pd[Row*Width + Col] = sum;
```

SIMT vs SIMD/SMT

- **flexibility vs efficiency**
- Flexiblity: SIMD < SIMT < SMT
- Efficiency: SIMD > SIMT > SMT

  (as long it fits the workload)
- Compared to SIMD; SI common, but

  multiple: register sets, addresses, flow paths
- Compared to SMT:
  - enough data parallelism → high throughput
  - More registers → more efficient latency hiding

Ref: http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html