Praktikum: Paralleles Programmieren mit CUDA

07-17. April 2014

AG „Parallele und Verteilte Architekturen“

Course Information

- Room: 04-220
- Website:
  - http://hpc.informatik.uni-mainz.de/teaching/teaching-winter-term-2013-14/parallel-programming-with-cuda-lab
  - Username: cuda2014
  - Password: cuda2014
- People
  - Prof Dr Bertil Schmidt
    - bertil.schmidt@uni-mainz.de
    - Office: 03-136
  - Dr Tuan Tu Tran
    - trant@uni-mainz.de
    - Office: 03-131

Schedule

Day		Topic	Lab Material
1	07-Apr	Introduction and "Hello World" program	Day 1
2	08-Apr	• Implementation a basic parallel algorithm for reduction • Discussion on the branch divergence problem	Day 2
3	09-Apr	• Implementation a basic parallel algorithm for reduction with shared memory • Discussion on the CUDA memory architecture and efficient memory usages	Day 3
4	10 - Apr	• Implementation and discussion on a problem of matrix multiplication • Introduction of the individual projects	Day 4
5	11 - Apr	• CUDA Visual Profiler • Selected individual project starts	Day 5
6	14 – Apr	Selected Project	Projects
7	15 – Apr	Selected Project	-
8	16 –Apr	Selected Project	-
9	17 – Apr	Selected Project / Project Demo / Presentation	-

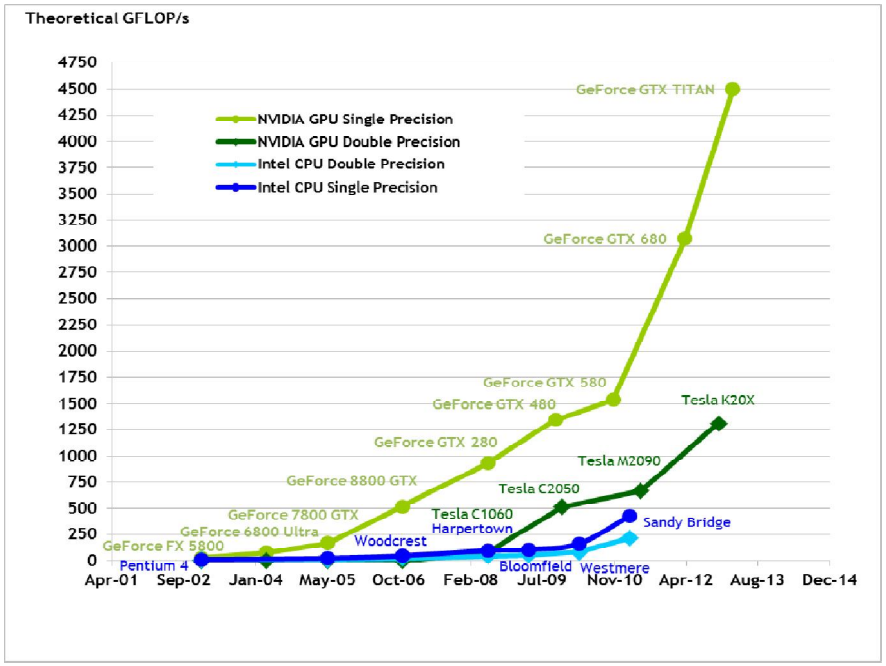
- **Core Timing: 10am to 5pm each day**
- For Day 1 to 4 : Present solutions to Tuan Tu Tran at the end of each day
- Project demo and presentation at the end of Day 9 (15 – 20 minutes)

Scheinkriterien

- Pass/Fail („unbenoteter Schein“)
- Regular participation (Anwesenheitskontrolle!)
- Show and Explain Solution for Day 1, 2, 3, 4
- Project presentation and demo at the end of Day 9

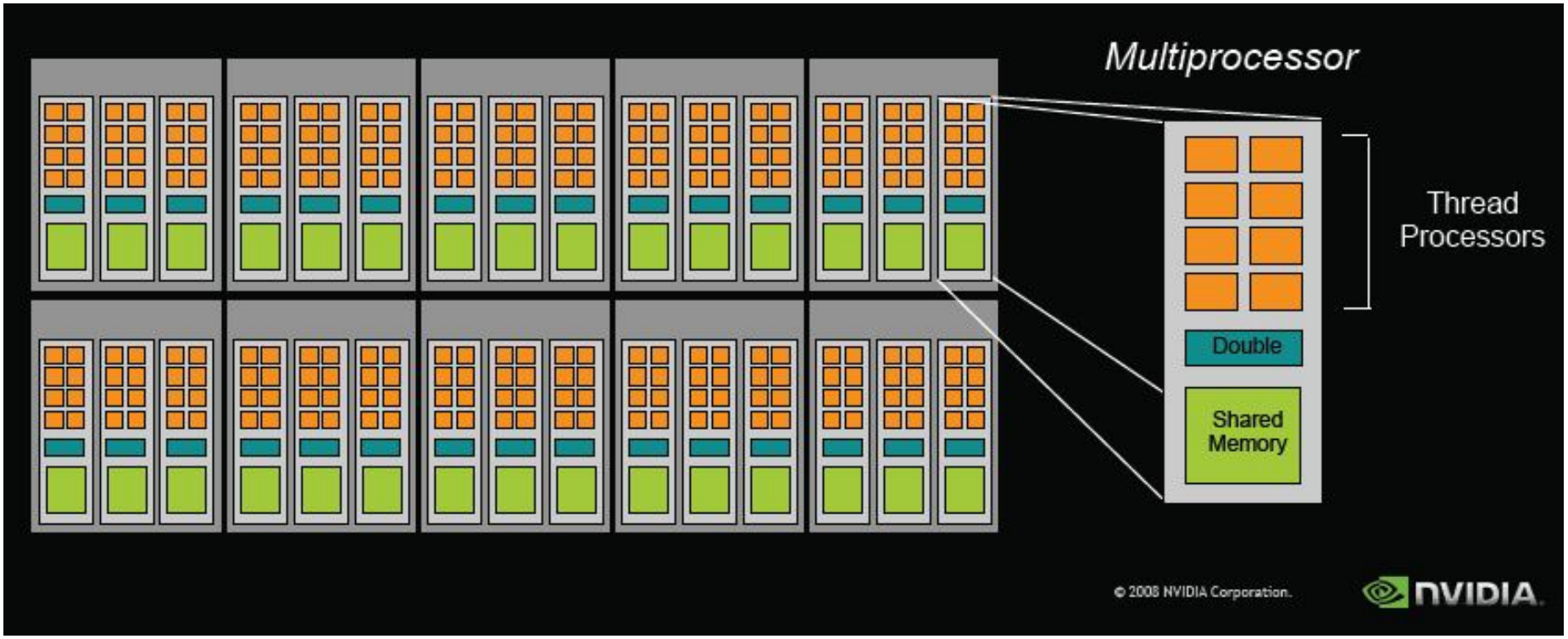
Terms

- What is GPGPU?
  - General-Purpose Computing on Graphics Processing Units
  - Using graphics hardware for non-graphics computations
- What is CUDA?
  - Compute Unified Device Architecture
  - Software architecture for managing data-parallel programming developed by NVIDIA
  - Extends C/C++
  - Supports NVIDIA GPUs from G80 series onwards including ***GeForce***, ***Quadro***, ***Tesla***
- Why GPGPU?



(source: NVIDIA's CUDA C Programming Guide, last updated 19/07/2013)

GPU Device Architecture

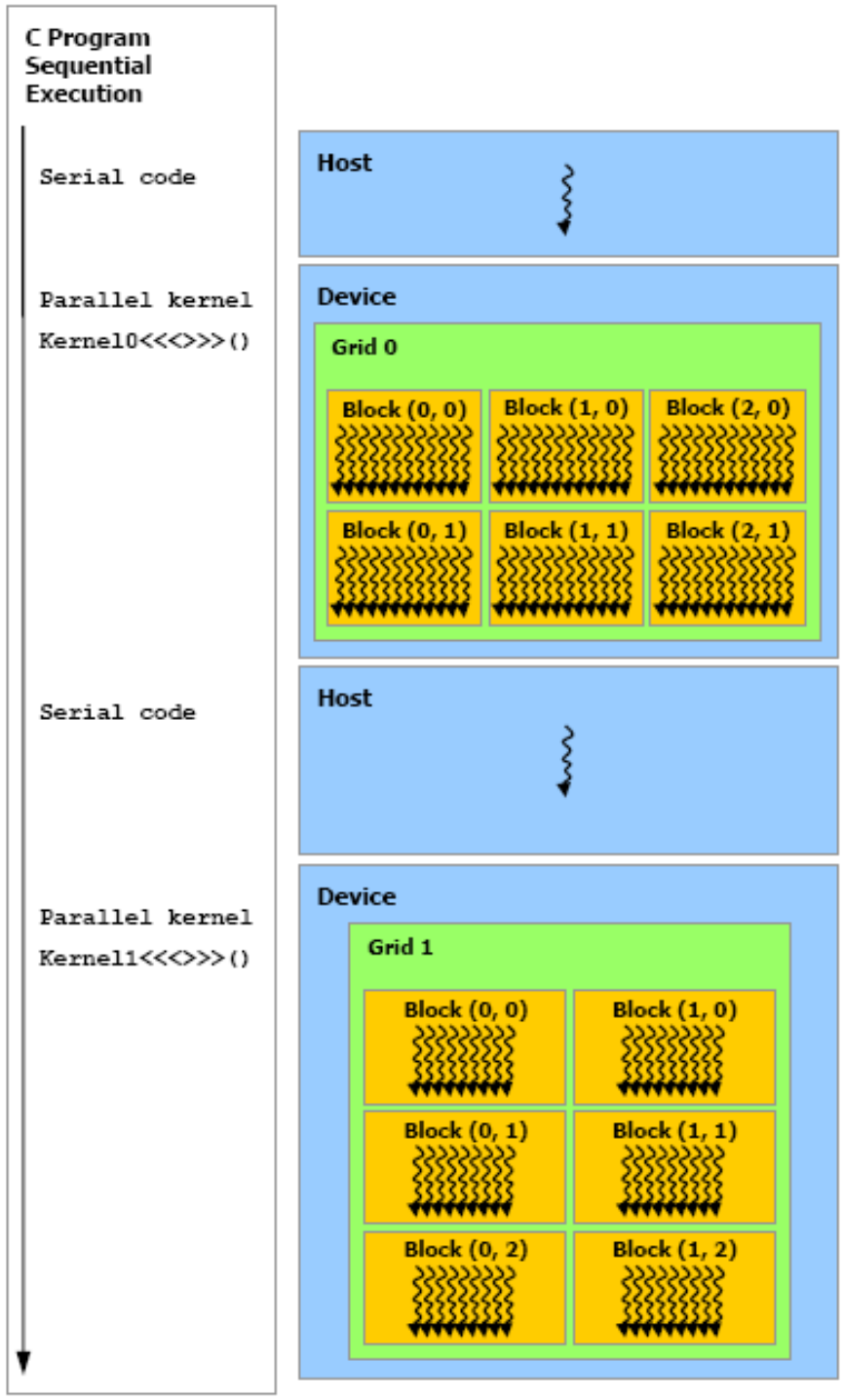


CUDA: Device + Hosts + Threads

- A compute ***device***
  - is a coprocessor to the CPU or ***host***
  - has its own DRAM (***device memory***)
  - runs many **threads in parallel**
  - is typically a **GPU**
- Data-parallel portions of an application are expressed as device ***kernels*** which run on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs many 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

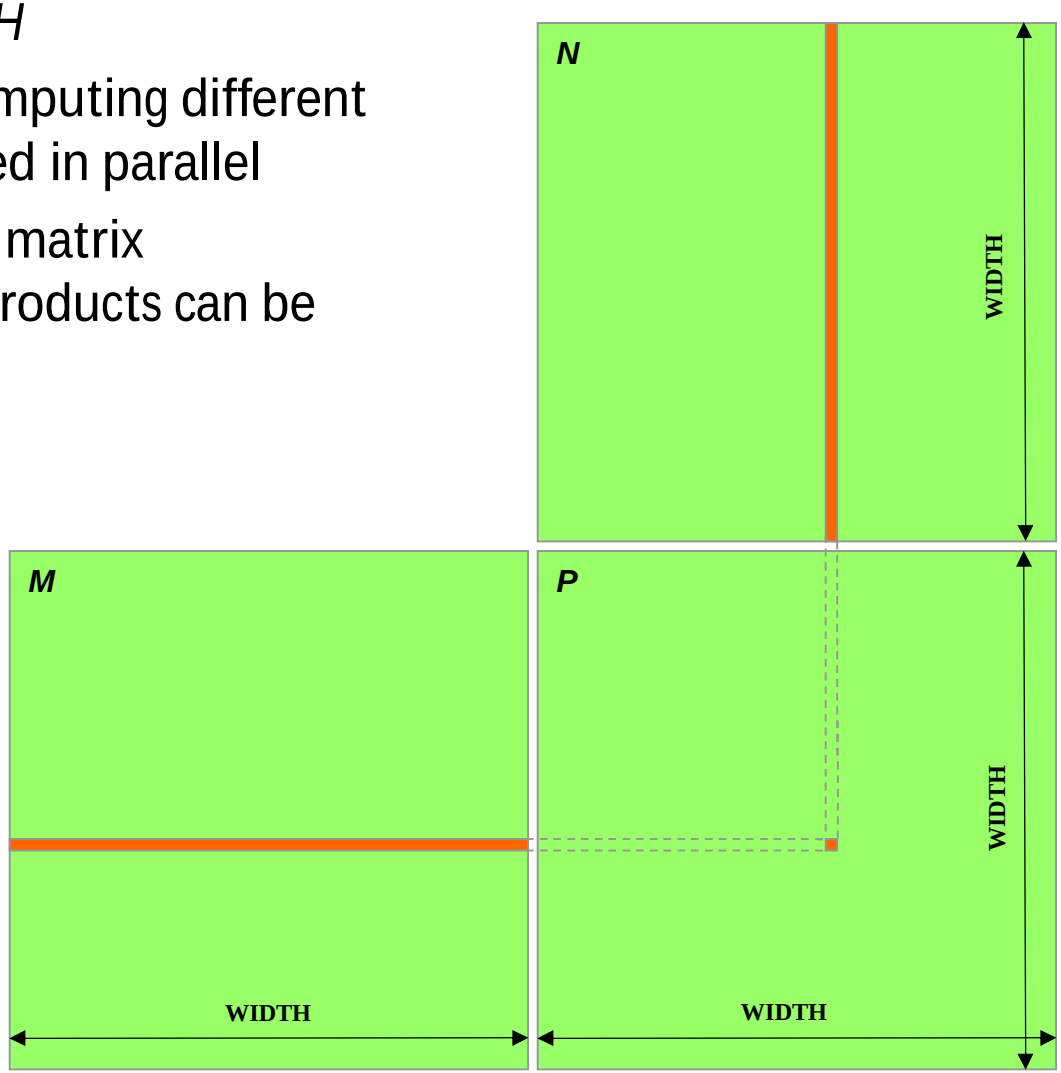
CUDA Program Structure

- Integrated **host + device** C program
  - Serial parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code



Data Parallelism: Square Matrix Multiplication

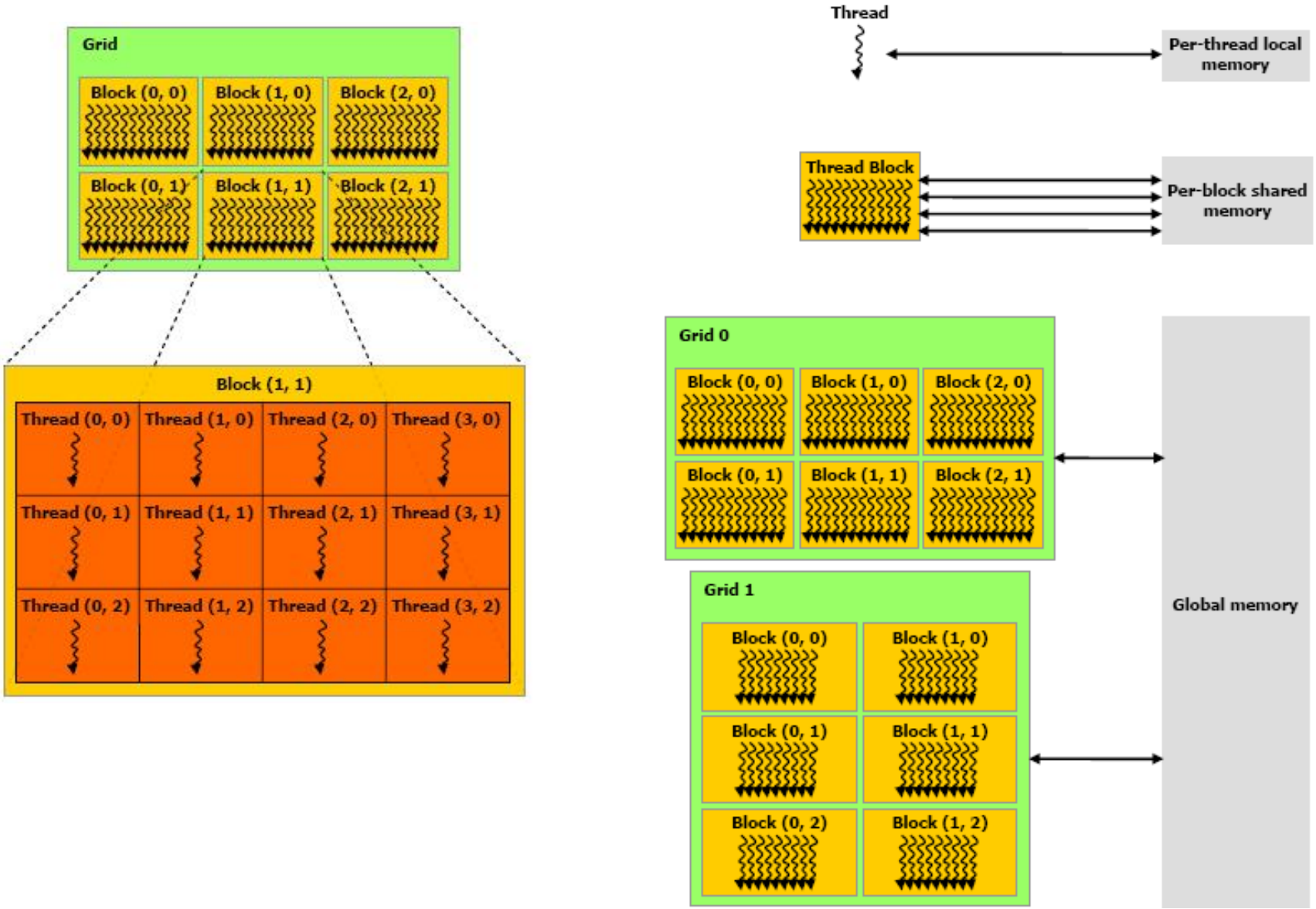
- *P* = *M*×*N* of size *WIDTH*×*WIDTH*
- Dot product operations for computing different elements in *P* can be performed in parallel
- For example, for a 1000×1000 matrix multiplication 1,000,000 dot products can be computed in parallel



Matrix Mult: Main function

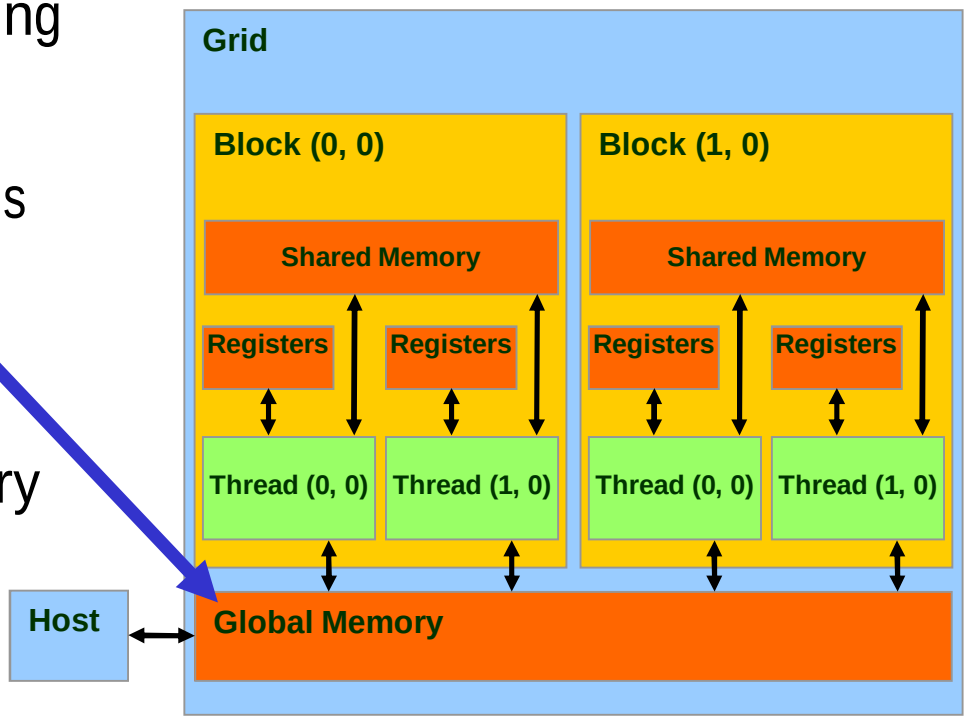
```
int main(void) {
1.   // Allocate and initialize matrices M, N, P
     //  transfer input matrices M and N from host to device
...
2.   // M×N on the device
      MatrixMultiplication(M, N, P, Width)

3.   // transfer the output matrix P from device to host
     // Free matrices M, N, P
...
return 0;
}
```

Thread and Memory Hierarchy



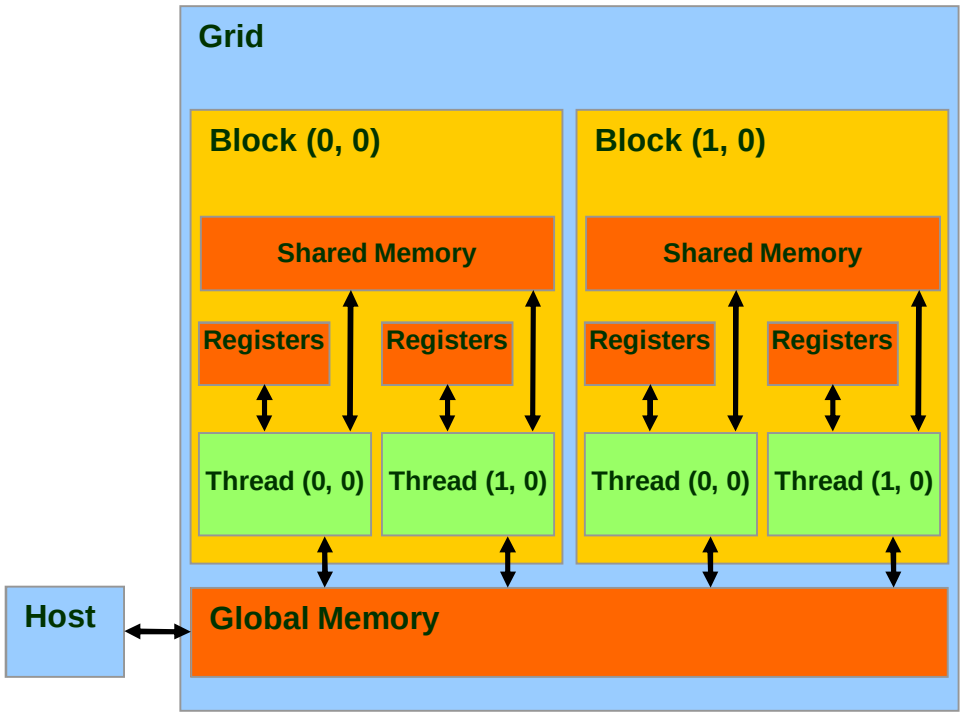
CUDA Memory Model Overview

- 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 global memory for now
  - Constant and texture memory for advanced usage



CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointe**r to the allocated object
    - **Size** of allocated object
- cudaFree()
  - Frees object from device Global Memory



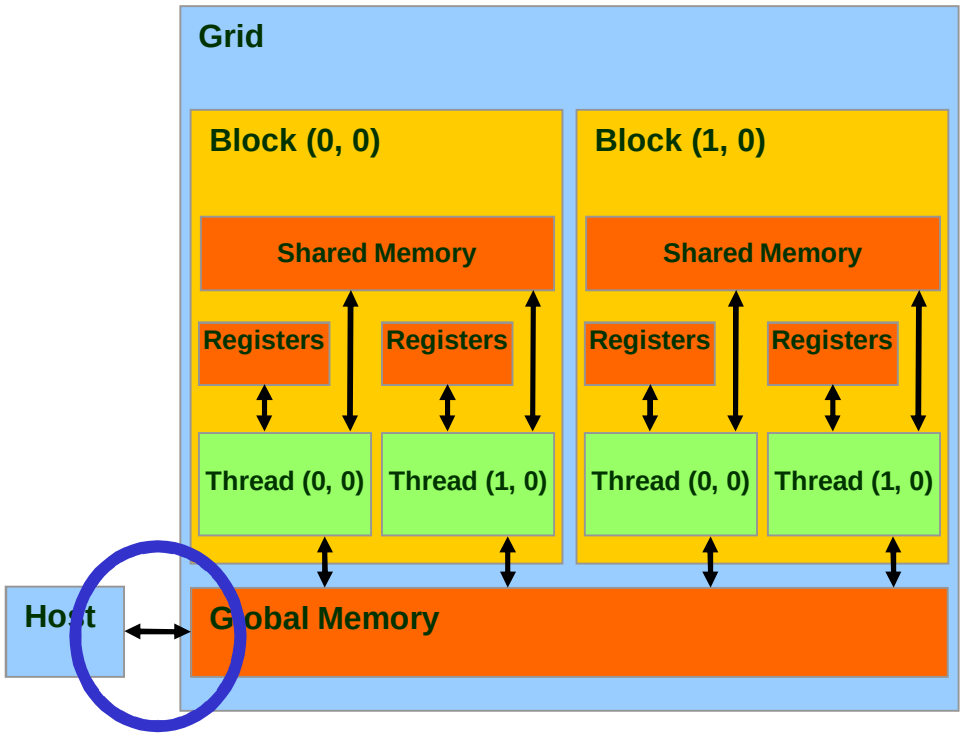
CUDA Device Memory Allocation

- Code example:
  - Allocate a 64×64 single precision float array
  - Attach the allocated storage to *Md*
  - "*d*" is often used to indicate a device data structure

```
TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires 4 parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - **Type of transfer**
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device



CUDA Host-Device Data Transfer

- Code example:
  - Transfer a 64×64 single precision float array
  - *M* is in host memory and *Md* is in device memory
  - *cudaMemcpyHostToDevice* and *cudaMemcpyDeviceToHost* are symbolic constants

cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

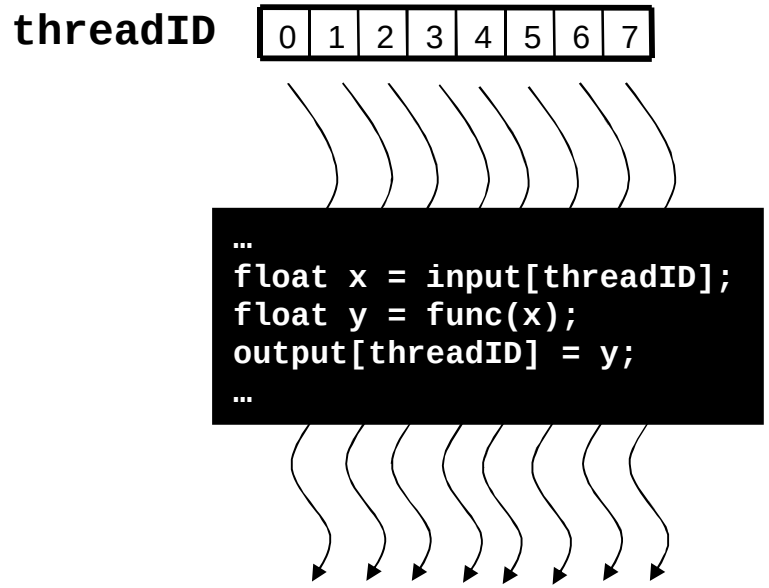
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);

Mat Mult: Host-side Code

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width) {
    int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;

1. // Allocate and Load M, N to device memory
     cudaMalloc(&Md, size);
     cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
     cudaMalloc(&Nd, size);
     cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

      // Allocate P on the device
     cudaMalloc(&Pd, size);

2.   // Kernel invocation code – to be shown later ….

3.    // Read P from the device
     cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
      // Free device matrices
     cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```

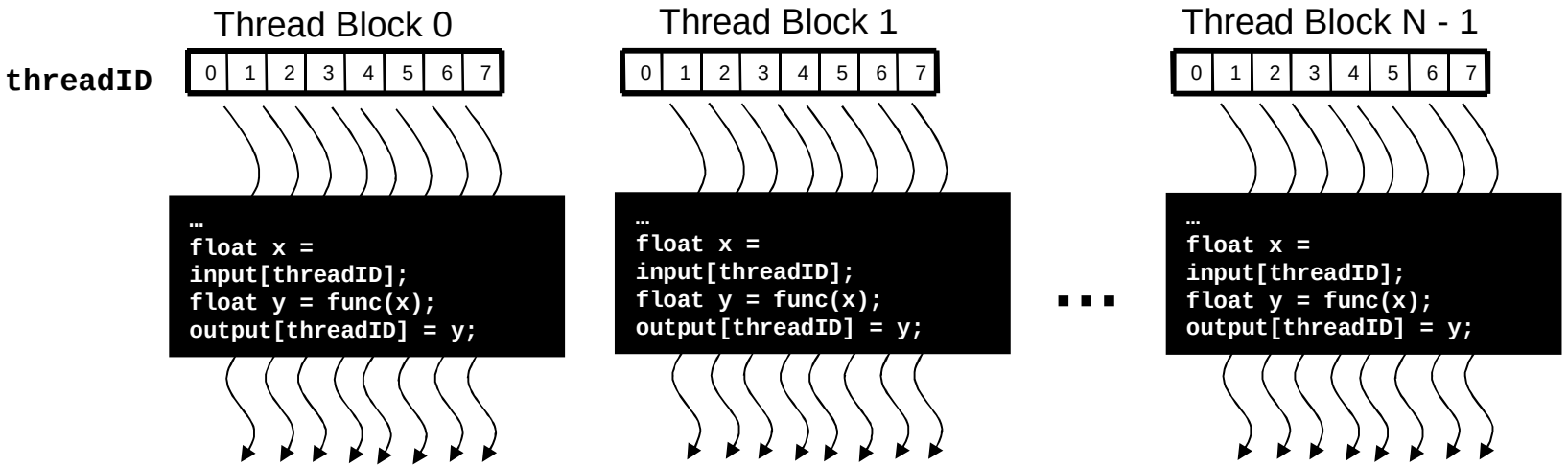
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



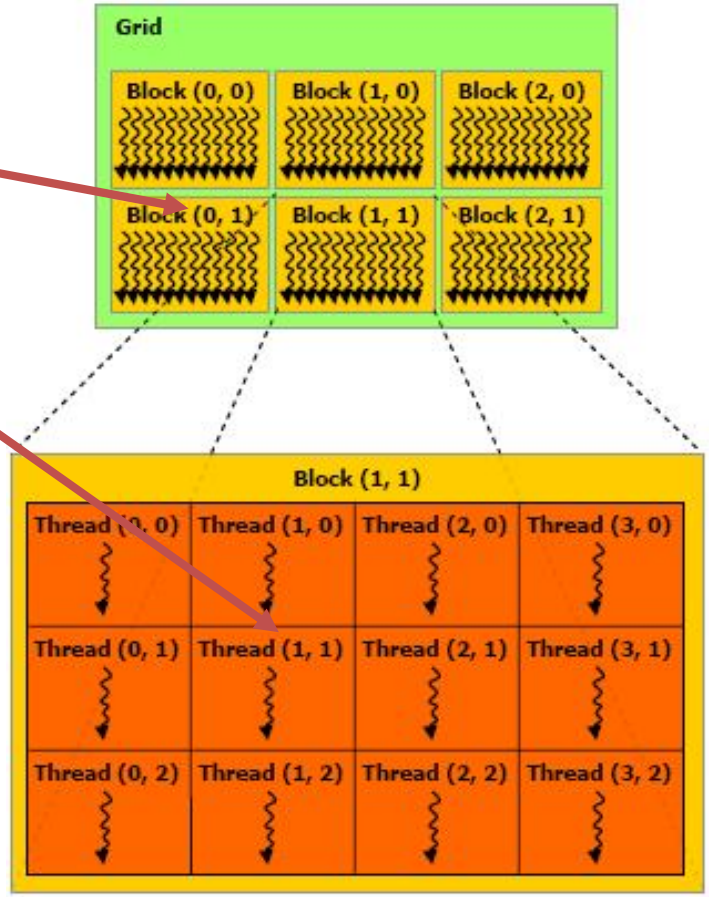
Thread Blocks

- Divide a monolithic thread array into multiple blocks
  - Threads within a block can cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate



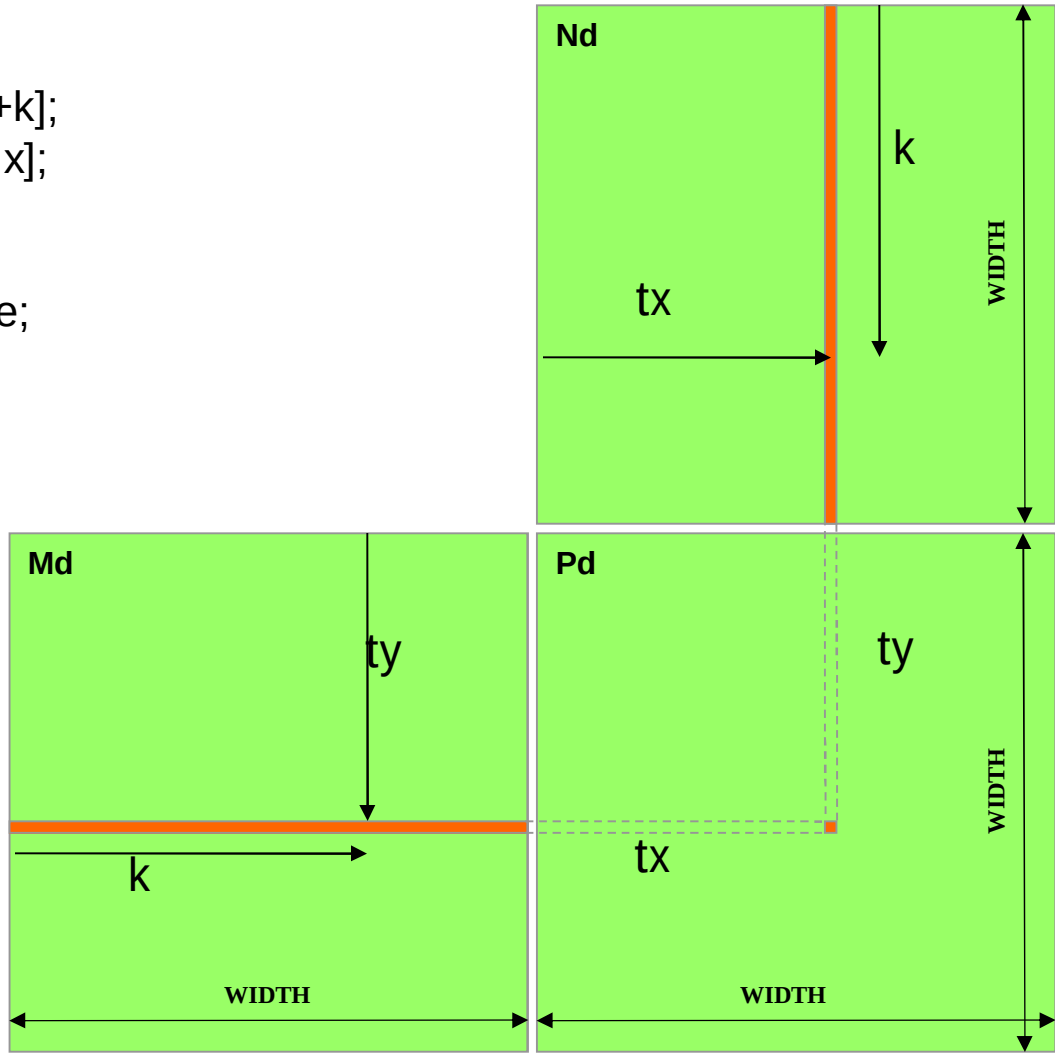
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Built-in variables
  - dim3 gridDim
    - Dimensions and size of grid in blocks
    - gridDim.x, gridDim.y
  - dim3 blockDim
    - Dimensions and size of block in threads:
    - blockDim.x, blockDim.y, blockDim.z
  - dim3 blockIdx
    - Block index within grid
    - blockIdx.x, blockIdx.y
  - dim3 threadIdx
    - Thread index within block
    - threadIdx.x, threadIdx.y, threadIdx.z



Mat Mult: Kernel with one thread block

```
 __global__ 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;
}
```



CUDA Function Declaration

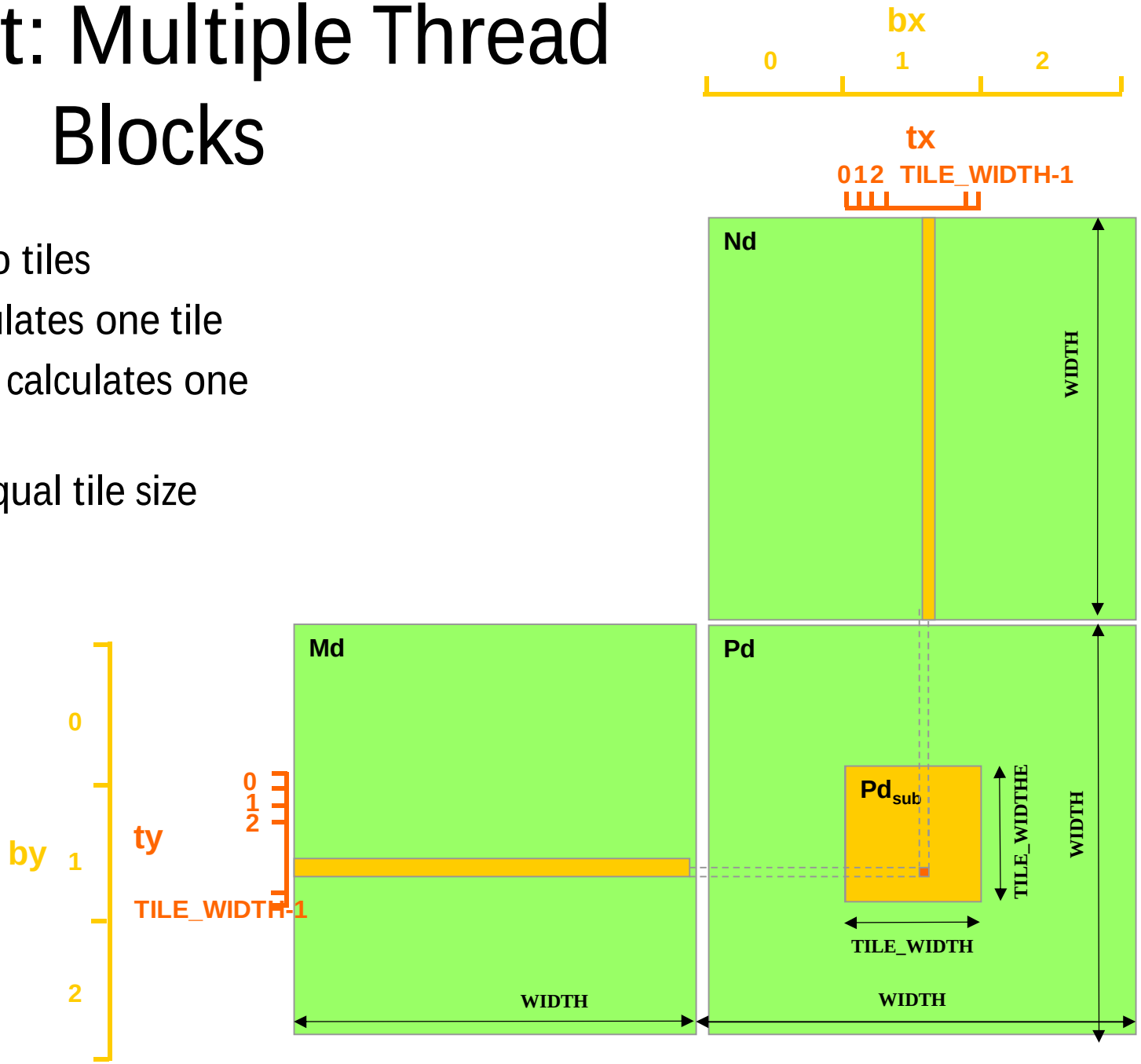
	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

Mat Mult: Kernel Invocation (Host-side Code)

```
// a kernel function must be called with an execution configuration
// Setup the execution configuration
   dim3 gridDim(1, 1);
   dim3 blockDim(Width, Width);

 // Launch the device computation threads!
 MatrixMulKernel<<<gridDim, blockDim>>>(Md, Nd, Pd, Width);
```

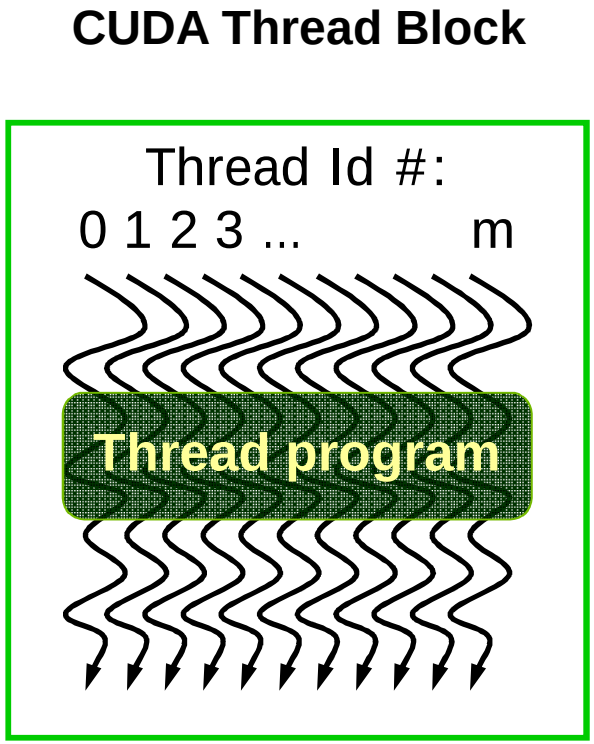
Mat Mult: Multiple Thread Blocks

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



CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **1024** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!



Vector Addition w/ one thread block

- Write a CUDA program for “Vector Addition”; i.e.
  - C[i] := A[i] + B[i] for vectors with *N* numbers
  - Write both the host and the device code
  - you can use a total of *N* CUDA threads
- Kernel and Kernel call with one thread block only
- Each CUDA thread performs one pair-wise addition

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Vector Addition w/ multiple thread blocks

- There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same Multi-processor
  - up to 1024 threads per block on the Fermi GPU
- Extending the previous **VecAdd()** example to handle multiple blocks
- A thread block size of 256 threads, although arbitrary in this case, is a common choice
- Thread blocks are required to execute independently
  - must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores
- Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses.
  - More precisely, one can specify synchronization points in the kernel by calling the **__syncthreads() intrinsic function;**
  - **__syncthreads()** acts as a barrier at which all threads in the block must wait before any is allowed to proceed

Vector Addition w/ multiple thread blocks

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

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Rules for efficient CUDA programming

1. Keep the GPU busy
   - by using a large number of thread blocks and a large number of threads per block
2. Make the kernel compute-bound
   - by re-using data using Shared Memory
3. Make memory accesses fast
   - by using Memory Coalescing to access global
   - By avoiding shared memory bank conflicts
4. Avoid Thread divergence within a warp
5. Avoid global synchronization

Question ?