

# 高性能计算的新发展

基于图形处理器的并行计算及**CUDA**编程

---

---

**Ying Liu, Associate Prof., Ph.D**

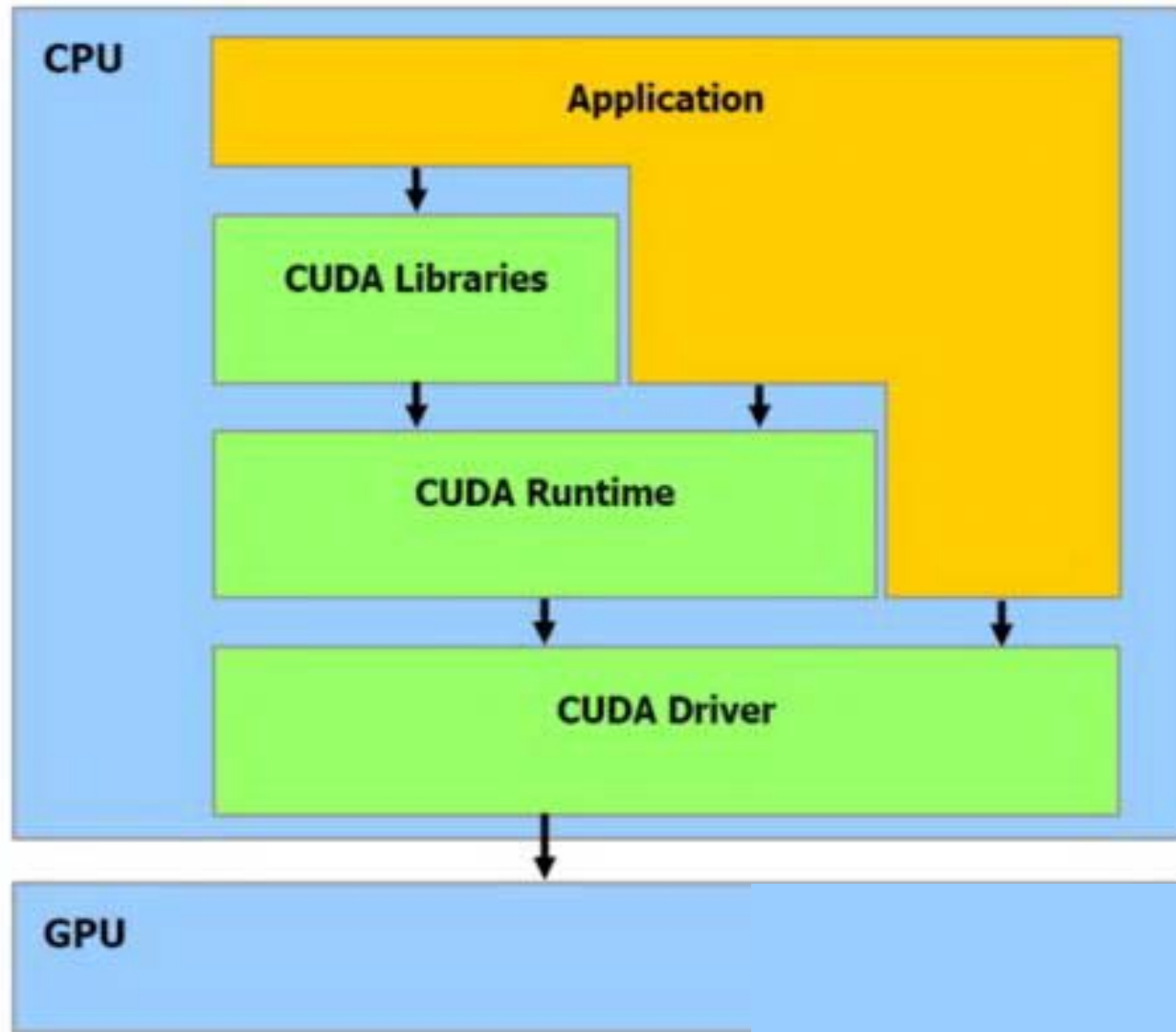
Graduate University of Chinese Academy of Sciences  
Research Center on Fictitious Economy and Data Science

# CUDA Programming Model

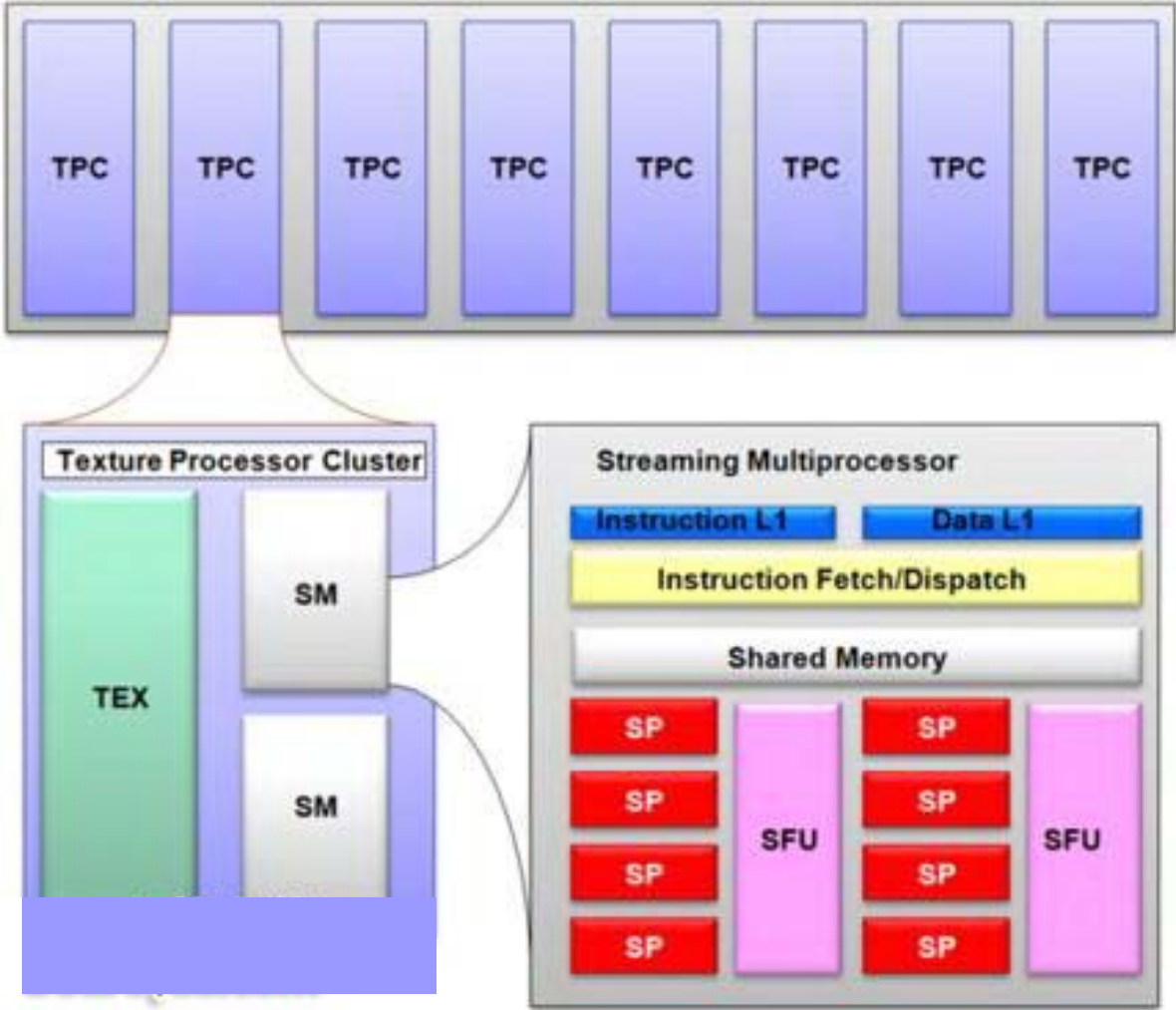
---

- **Compute Unified Device Architecture (CUDA)**
- Execution model: kernels, threads, blocks, and grids
- CUDA Basics
- A simple example: matrix multiplication
- CUDA Toolkit: libraries, compiler, debugger, emulator, etc.

# CUDA

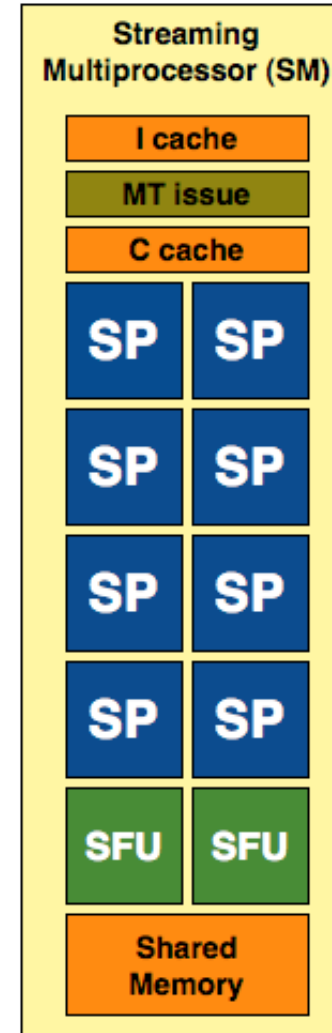


# G80 CUDA Mode



# Streaming Multiprocessor (SM)

- An array of SPs
  - 8 streaming processors
  - 2 Special Function Units (SFU)
    - Transcendental operations (e.g. sin, cos) and interpolation
  - A 16KB read/write shared memory
    - Not a cache, but a software-managed data store
  - Multithreading issuing unit
    - Dispatch instructions
  - Instruction cache
  - Constant cache



# CUDA Device

---

- 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** but can also be another type of parallel processing device
- **Kernel** — Data-parallel portions of an application which run on many threads

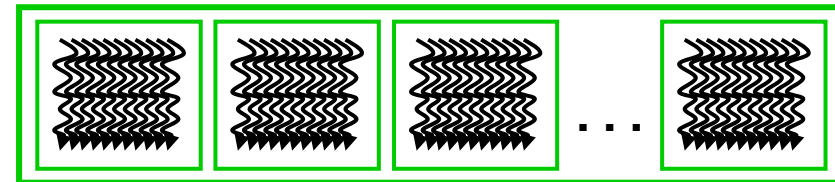
# CUDA

- Integrated host+device application C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)

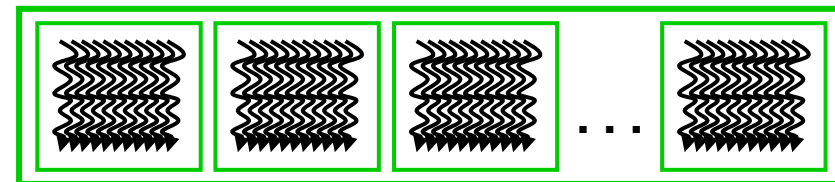
```
KernelA<<< nBlk, nTid >>>(args);
```



Serial Code (host)

Parallel Kernel (device)

```
KernelB<<< nBlk, nTid >>>(args);
```



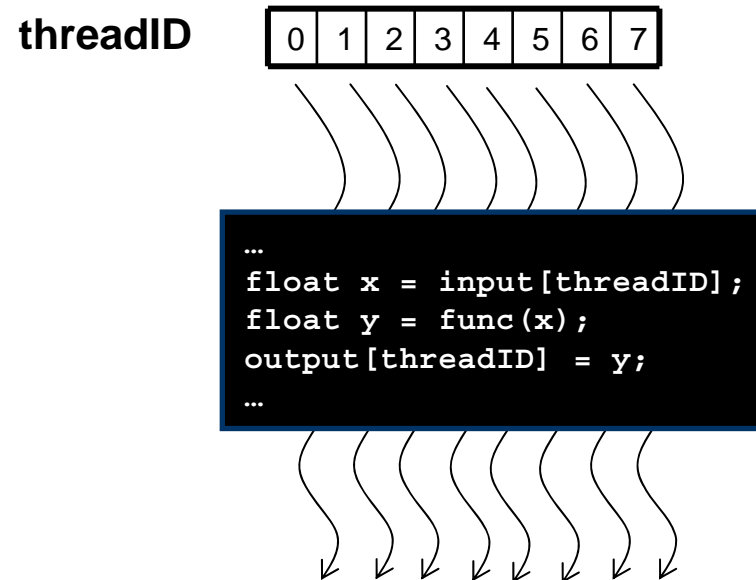
# CUDA Programming Model

---

- Compute Unified Device Architecture (CUDA)
- Execution model: kernels, threads, blocks, and grids
- CUDA Basics
- A simple example: matrix multiplication
- CUDA Toolkit: libraries, compiler, debugger, emulator, etc.

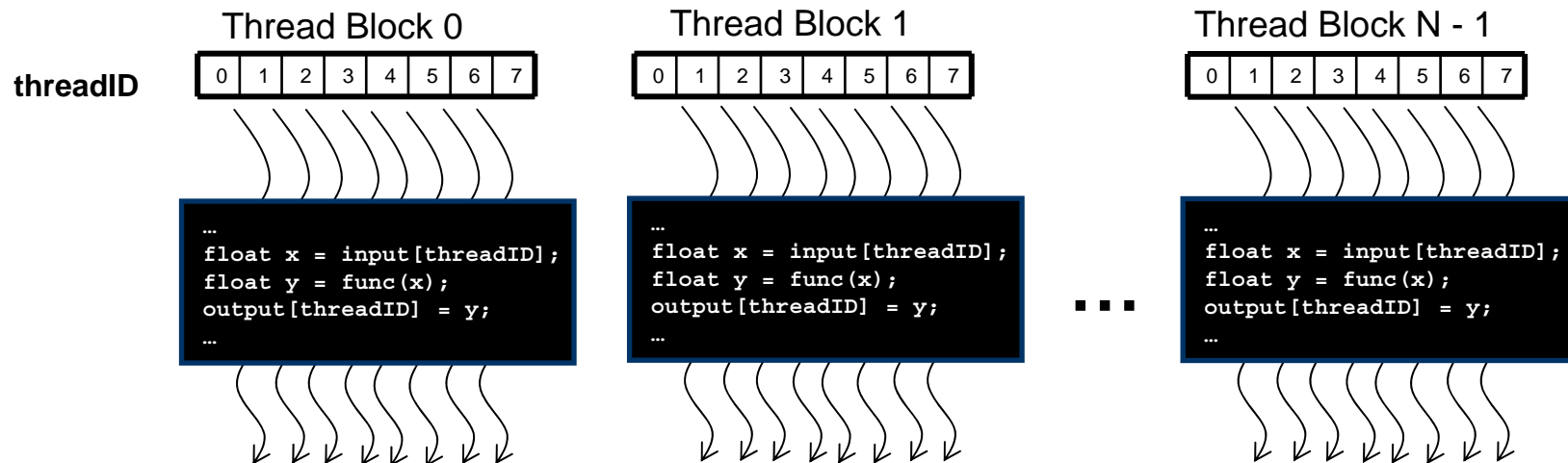
# Arrays of Parallel Threads

- 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

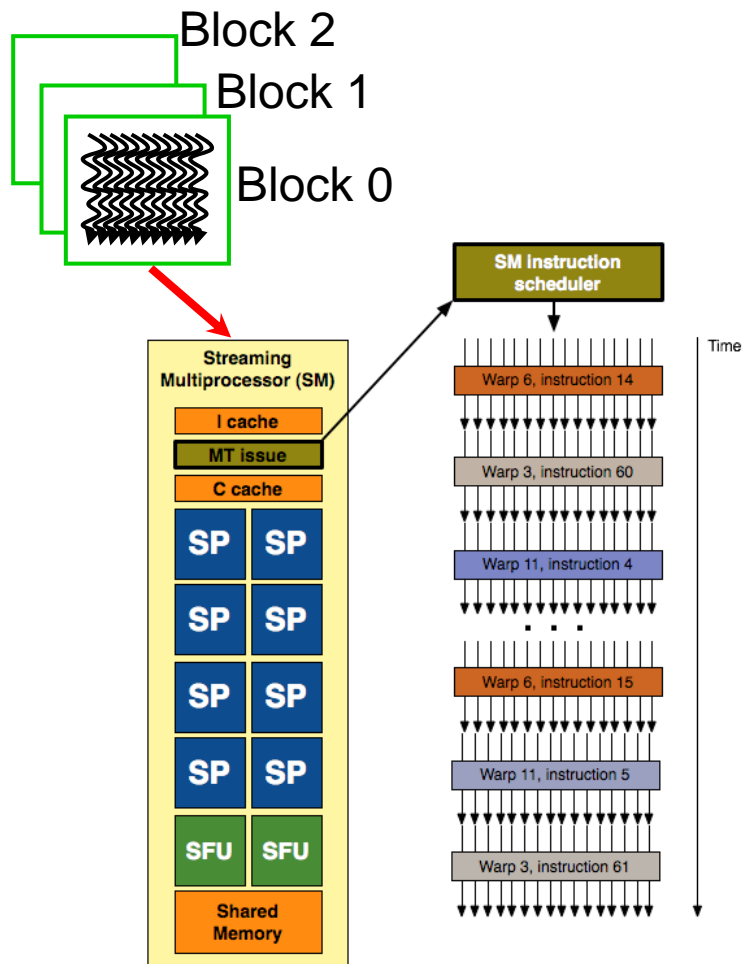


# Thread Blocks

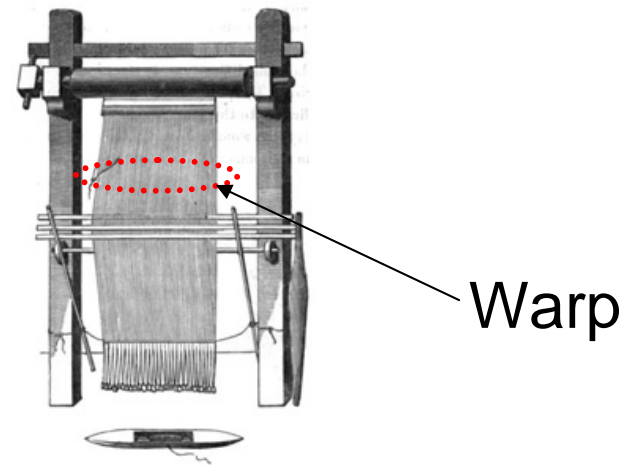
- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via *shared memory*, *atomic operations* and *barrier synchronization*
  - Threads in different blocks cannot cooperate



# Thread Execution

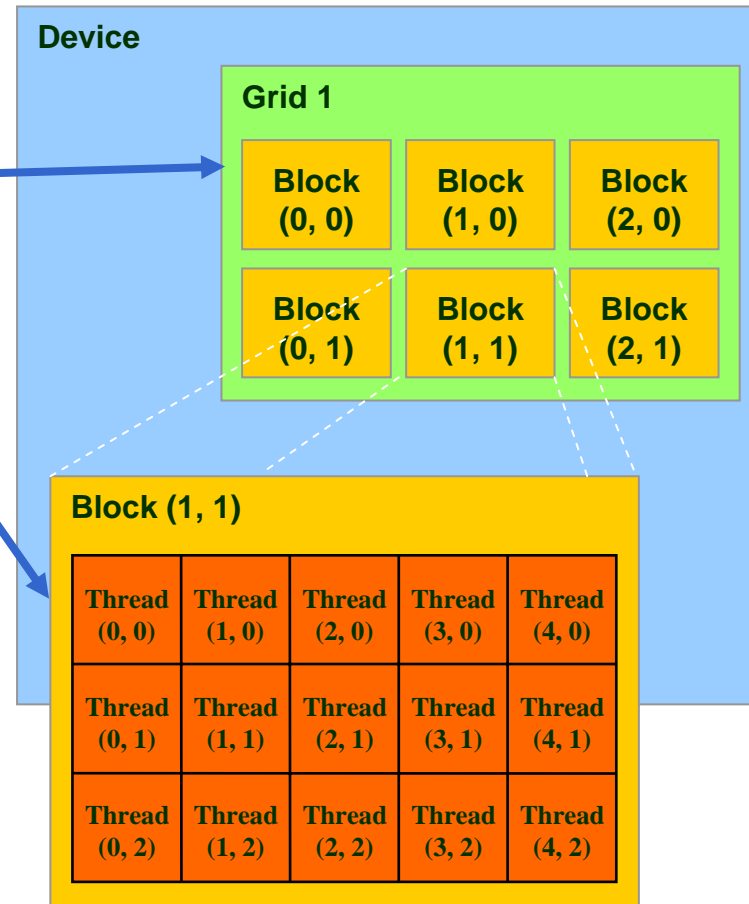


- A warp of 32 threads physically running on SM
  - Sharing instructions
  - 4 cycles for 1 warp instruction
  - Dynamically scheduled by SM
    - Executed when operands ready



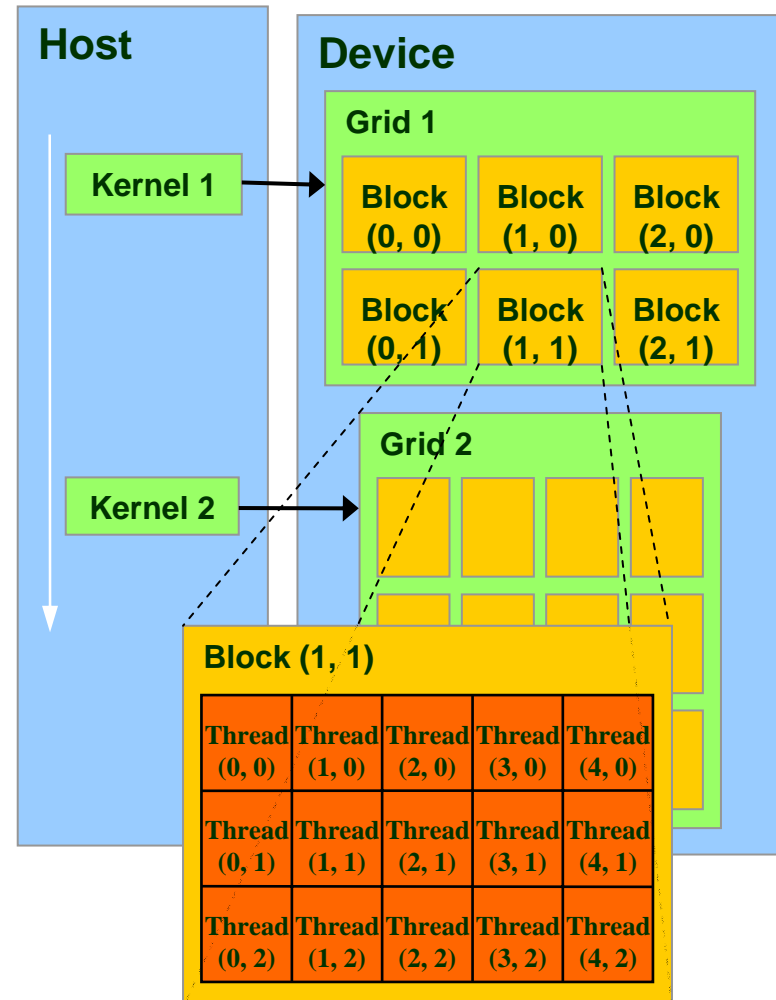
# Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplify memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...

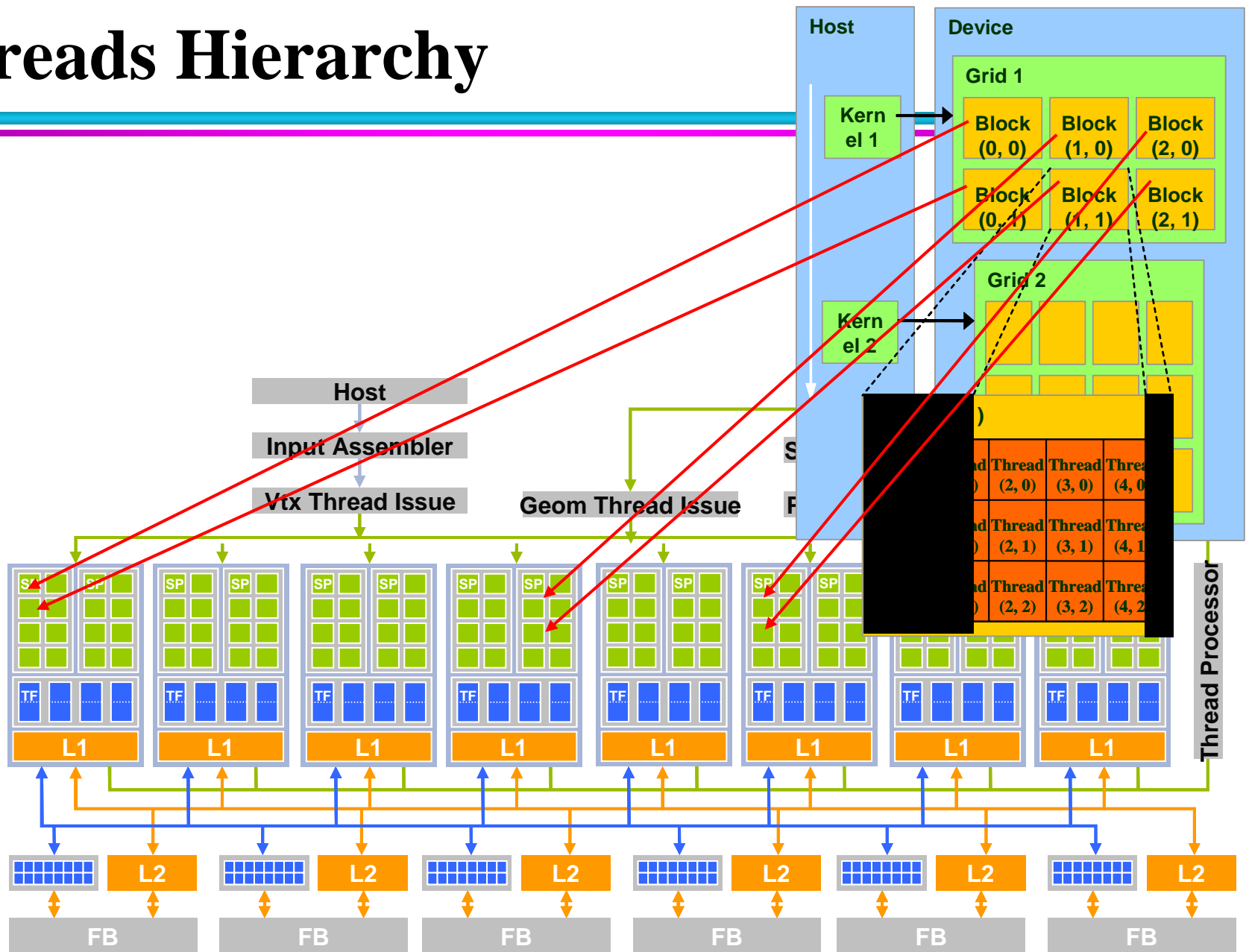


# Threads Hierarchy

- Thread: parallel execution
- Thread block
  - Cooperative Thread Array (CTA)
  - Synchronization between threads
  - Share data in shared memory
  - 1D, 2D, 3D
  - Max 512 threads
- Grid
  - A group of thread block
  - 1D, 2D, 3D
  - Share data in global memory
  - Dynamically scheduled at runtime
- Kernel
  - The part of code running on threads

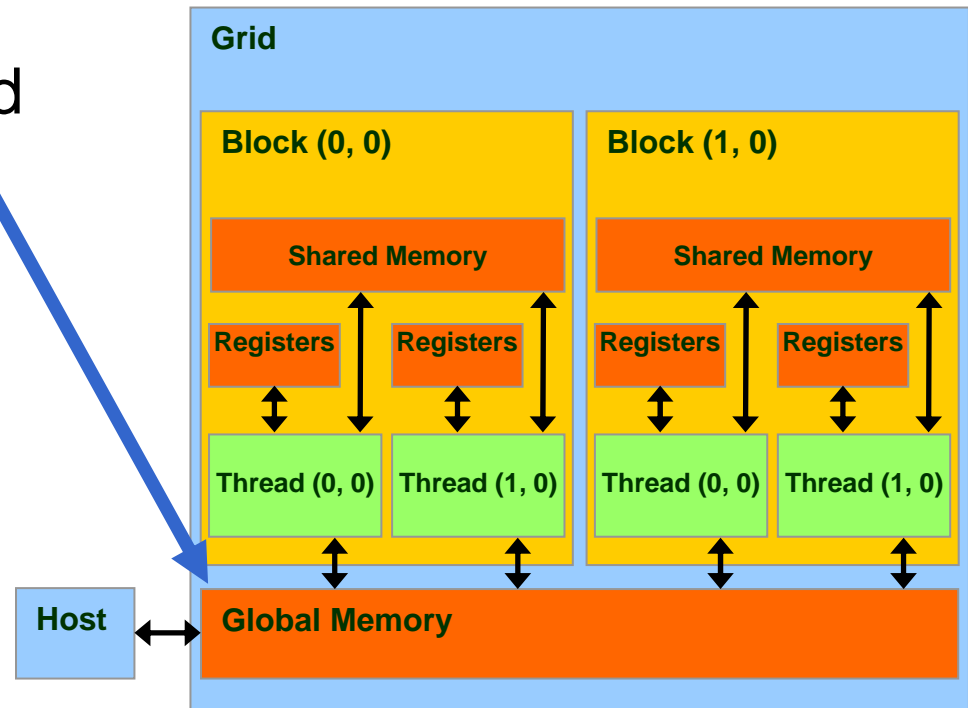


# Threads Hierarchy



# CUDA Memory Model

- Global memory
  - Main means of communicating R/W data between **host** and **device**
  - Contents visible to all threads
  - Long latency access



# CUDA Programming Model

---

- Compute Unified Device Architecture (CUDA)
- Execution model: kernels, threads, blocks, and grids
- **CUDA Basics**
- A simple example: matrix multiplication
- CUDA Toolkit: libraries, compiler, debugger, emulator, etc.

# CUDA Extends C

---

- **Declaration specs**

- global, device, shared, local, constant

- **Keywords**

- threadIdx, blockIdx

- **Intrinsics**

- \_\_syncthreads

- **Runtime API**

- Memory, symbol, execution management

- **Function launch**

```
__device__ float filter[N];
__global__ void convolve (float *image) {

    __shared__ float region[M];
    ...

    region[threadIdx] = image[i];

    __syncthreads()
    ...

    image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

# CUDA Variable Type Qualifiers

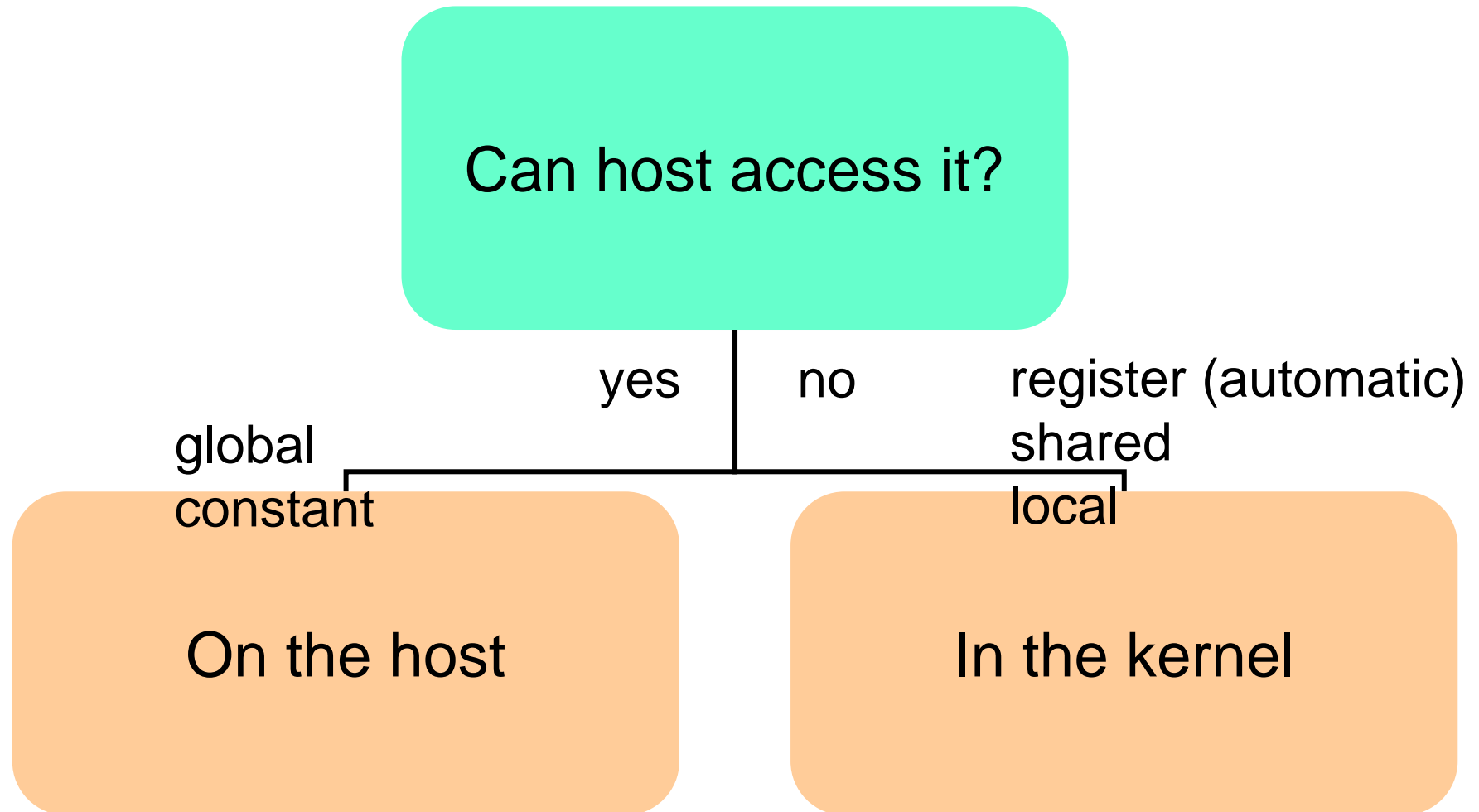
Variable declaration	Memory	Scope	Lifetime
<code>__device__ __local__ int LocalVar;</code>	local	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

# Where to Declare Variables?

---

---



# CUDA Variable Declarations

---

## ■ `__device__`

- Resides in the global memory
- Has the lifetime of an application
- Is accessible from all the threads within the grid
- Is accessible from the host through the runtime library
- Address of a `__device__` variable can only be used in device code

## ■ `__constant__`

- Resides in constant memory
- Has the lifetime of an application
- Is accessible from all the threads within the grid
- Is accessible from the host through the runtime library
- Cannot be assigned to from the device, only from the host through the runtime library

# CUDA Variable Declarations (Cont.)

---

## ■ `__shared__`

- Resides in the shared memory of a thread block
- Has the lifetime of the block
- Is only accessible from all the threads within the block
- Cannot have initialization as part of declaration
- Address of a `_device_` variable can only be used in device code

## ■ `Automatic variable`

- Declared with no qualifier
- Resides in a register or local memory
- Has the lifetime of the thread
- Thread private

# Variable Type Restrictions

---

- **Pointers** can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:  
`__global__ void KernelFunc(float* ptr)`
  - Obtained as the address of a global variable:  
`float* ptr = &GlobalVar;`

# Built-in Vector Types

---

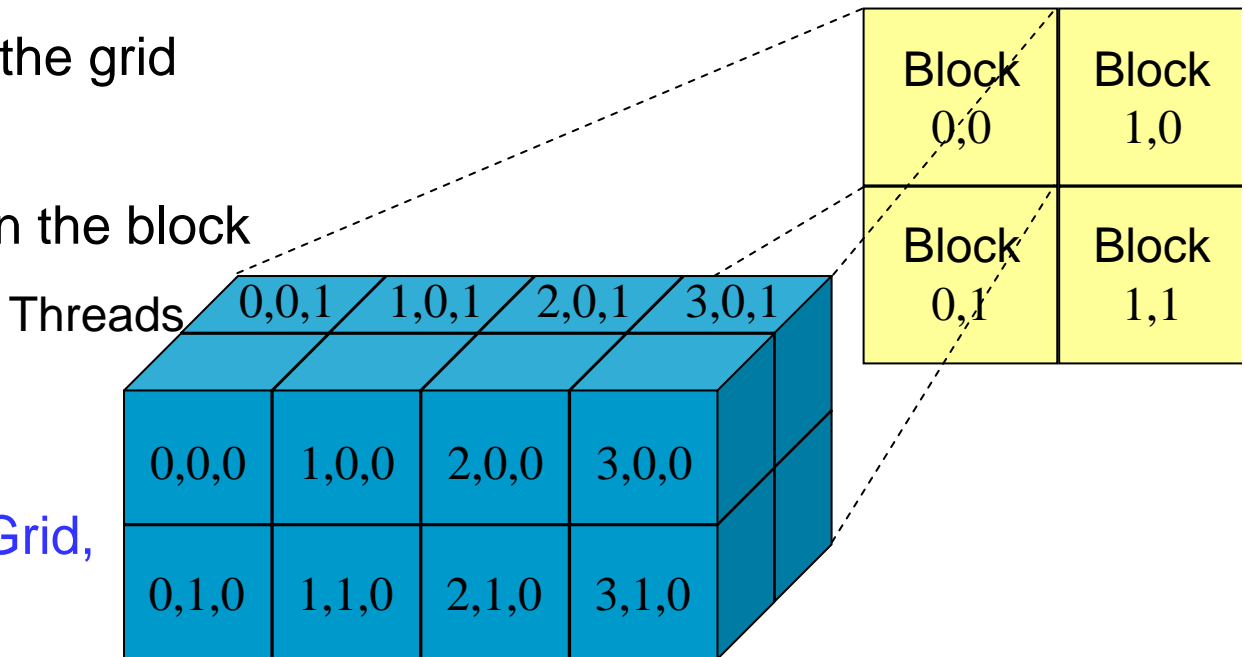
## ■ Built-in

- int1, int2, int3, int4, float1, float2, float3, float4, ...
- Define by a constructor function make\_<type name>
  - int4 make\_int4 (int w, int x, int y, int z)
  - int4 iv(1, 2, 3, 4)
    - iv.w = 1, iv.x = 2, iv.y = 3, iv.z = 4

# Built-in dim3 Type

- dim3 `gridDim`
  - Dimensions of the grid in blocks (`gridDim.z` unused)
- dim3 `blockDim`
  - Dimensions of the block in threads
- dim3 `blockIdx`
  - Block index within the grid
- dim3 `threadIdx`
  - Thread index within the block

```
dim3 dimGrid(2, 2)  
dim3 dimBlock(4, 2, 2)  
kernelFunction<<< dimGrid,  
dimBlock>>>(...
```



# CUDA Function Declarations

---

---

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

# CUDA Function Declarations (Cont.)

---

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

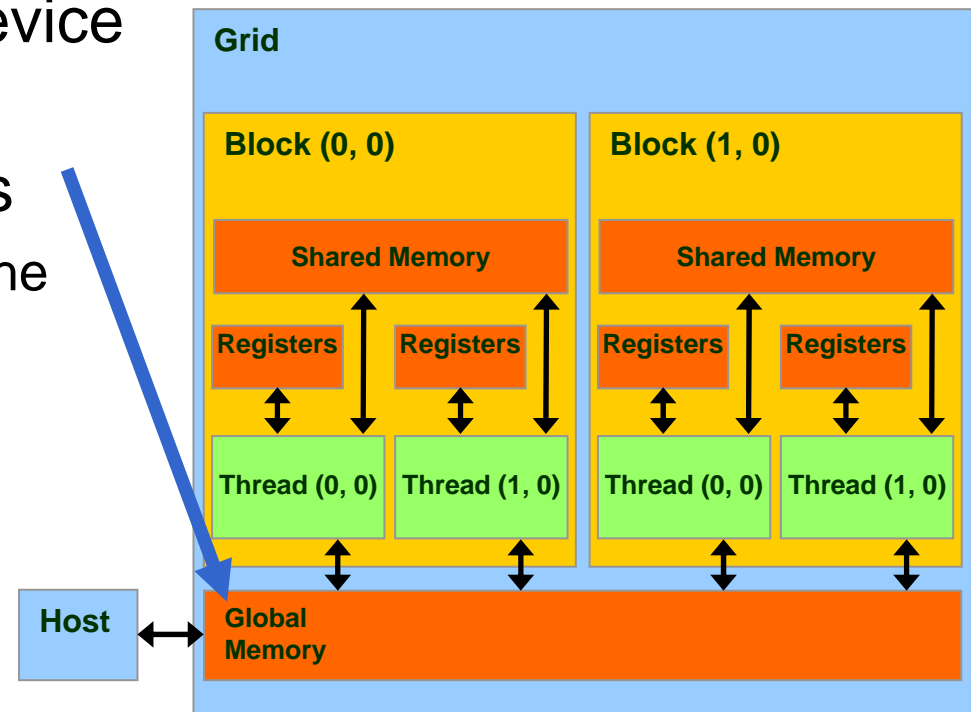
# Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);  
dim3  DimGrid(100, 50);           // 5000 thread blocks  
dim3  DimBlock(4, 8, 8);         // 256 threads per block  
size_t SharedMemBytes = 64;     // 64 bytes of shared  
    memory  
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes  
    >>>(...);
```

# CUDA Device Memory Allocation

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



# CUDA Device Memory Allocation (Cont.)

---

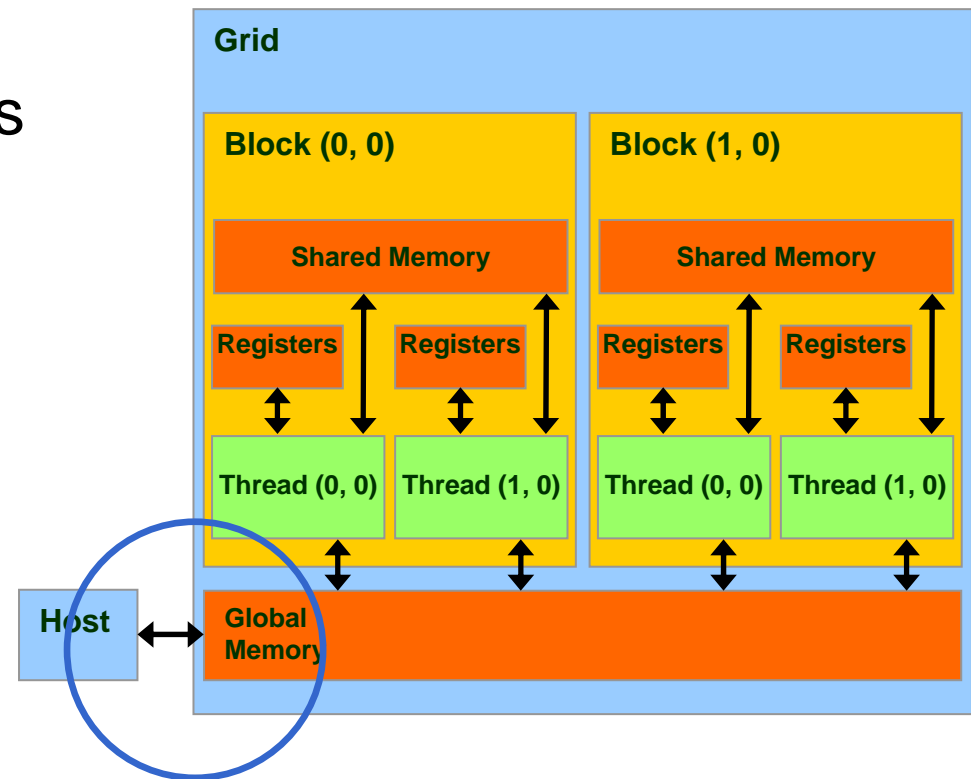
## ■ Example code:

- Allocate a 64 \* 64 single precision float array
- Attach the allocated storage to Md.elements
- “d” is often used to indicate a device data structure

```
TILE_WIDTH = 64;  
Matrix Md  
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);  
  
cudaMalloc((void**)&Md.elements, size);  
  
cudaFree(Md.elements);
```

# CUDA Host-Device Data Transfer

- `cudaMemcpy()`
  - memory data transfer
  - Requires four 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
- Asynchronous transfer



# CUDA Host-Device Data Transfer (Cont.)

---

- Example code :
  - Transfer a  $64 * 64$  single precision float array
  - M is in host memory and Md is in device memory
  - `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`

```
cudaMemcpy(Md.elements, M.elements, size,  
cudaMemcpyHostToDevice);
```

```
cudaMemcpy(M.elements, Md.elements, size,  
cudaMemcpyDeviceToHost);
```

# Threads Synchronization

---

- `void __syncthreads();`
  - Barrier
  - Synchronize all the threads within a block
  - Avoid race condition, memory contention

```
__shared__ float scratch[256];  
scratch[threadID] = begin[threadID];  
__syncthreads();  
int left = scratch[threadID - 1];
```

*Wait here till all the threads within this block reach this line*

# Dead-Lock with `__syncthreads`

---

- Dead-lock if
  - Some threads have val larger than threshold
  - And others not

```
__global__ void compute(...)  
{  
    // do some computation for val  
    if( val > threshold )  
        return;  
  
    __syncthreads();  
    // work with val & store it  
    return;  
}
```

# CUDA Programming Model

---

- Compute Unified Device Architecture (CUDA)
- Execution model: kernels, threads, blocks, and grids
- CUDA Basics
- **A simple example: matrix multiplication**
- CUDA Toolkit: libraries, compiler, debugger, emulator, etc.

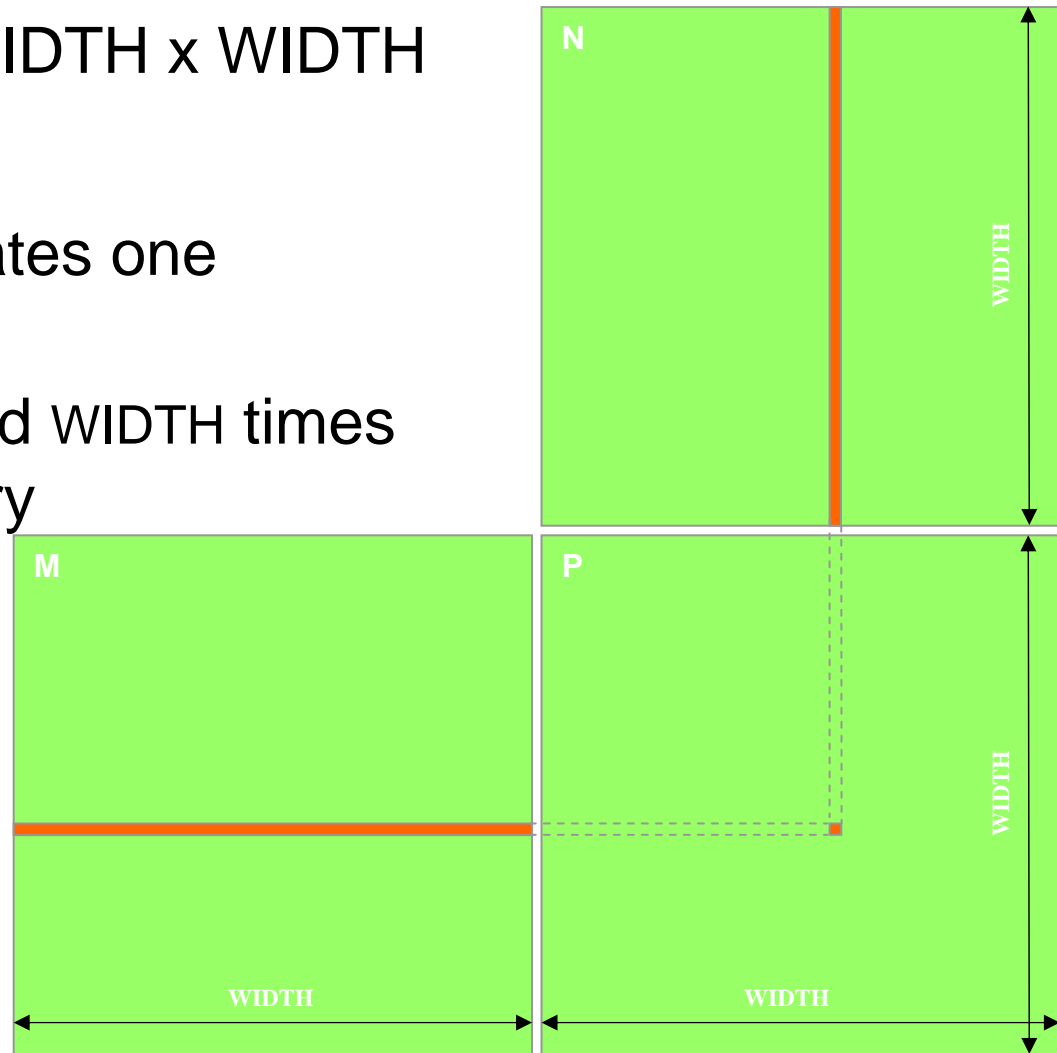
# A Simple Example: Matrix Multiplication

---

- Illustrate the basic features of memory and thread management in CUDA programs
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity
  - *Leave shared memory usage until later*

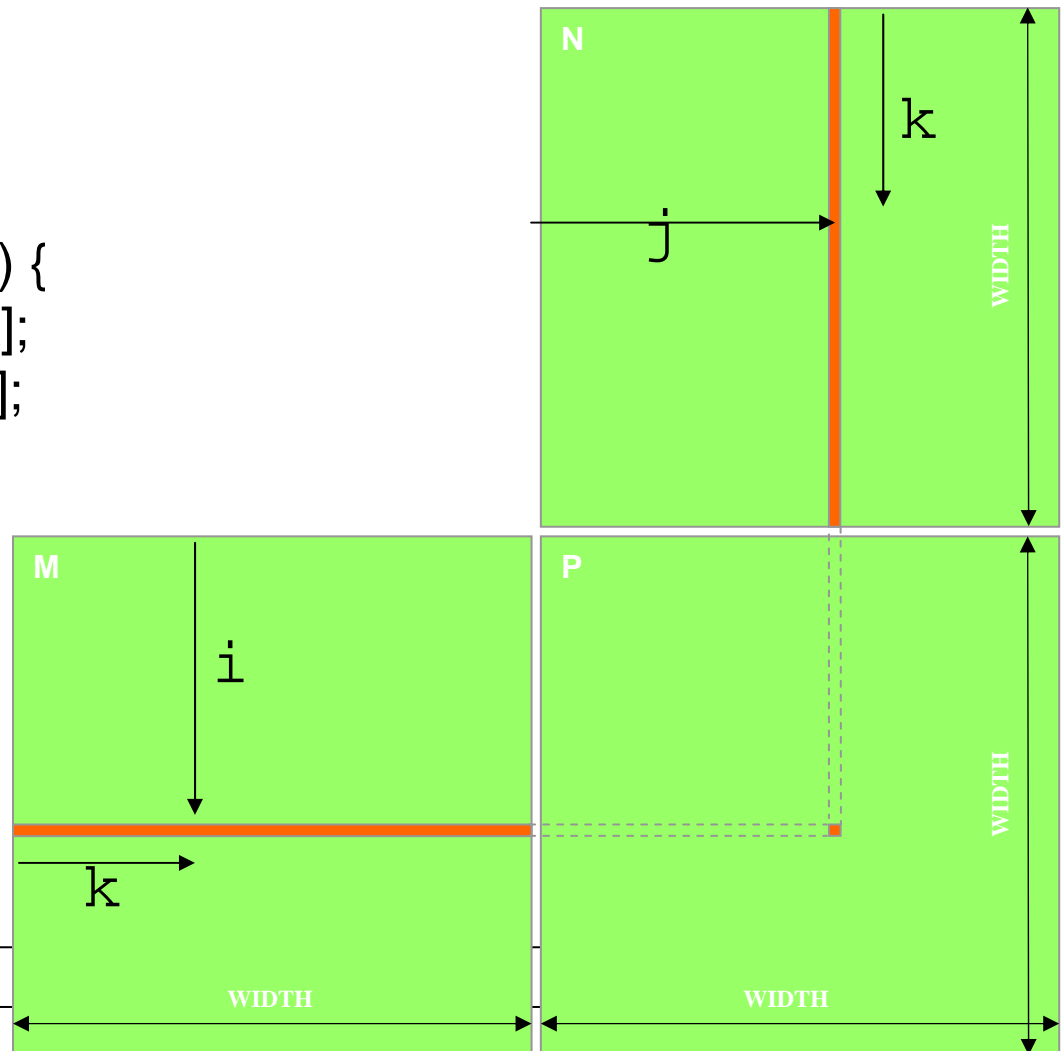
# Square Matrix Multiplication

- $P = M * N$  of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory



# Step 1: A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host
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;
        }
}
```



@ Ying Liu

## Step 2: Input Matrix Data Transfer (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);
```

## Step 3: Output Matrix Data Transfer (Host-side Code)

---

```
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);
}
```

# Step 4: Kernel Function

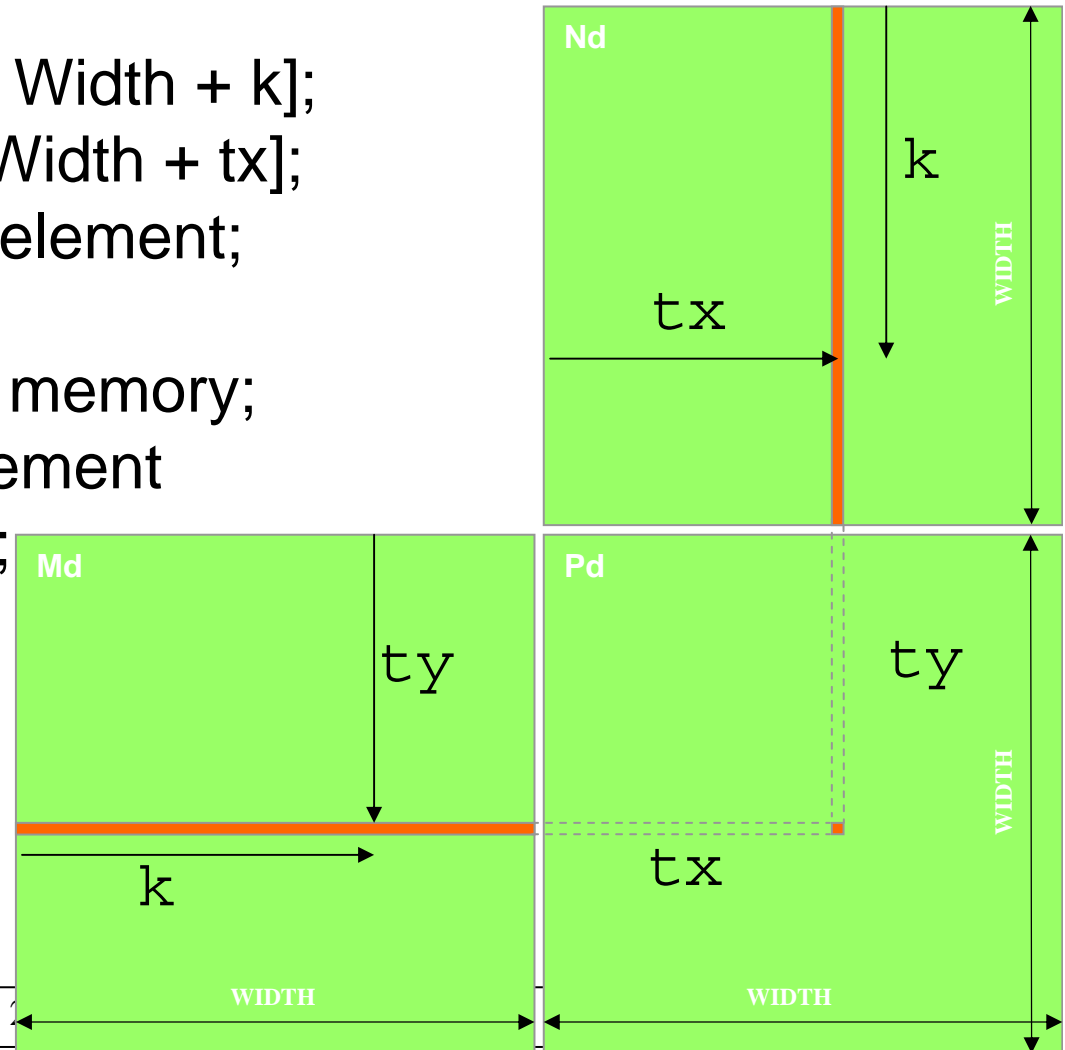
---

```
// Matrix multiplication kernel – per thread code
```

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd,  
int Width)  
{  
    // 2D Thread ID  
    int tx = threadIdx.x;  
    int ty = threadIdx.y;  
  
    // Pvalue is used to store the element of the matrix  
    // that is computed by the thread  
    float Pvalue = 0;
```

# Step 4: Kernel Function (Cont.)

```
for (int k = 0; k < Width; ++k)
{
    float Melement = Md[ty * Width + k];
    float Nelement = Nd[k * Width + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
Pd[ty * Width + tx] = Pvalue;
}
```



# Step 5: Kernel Invocation

## (Host-side Code)

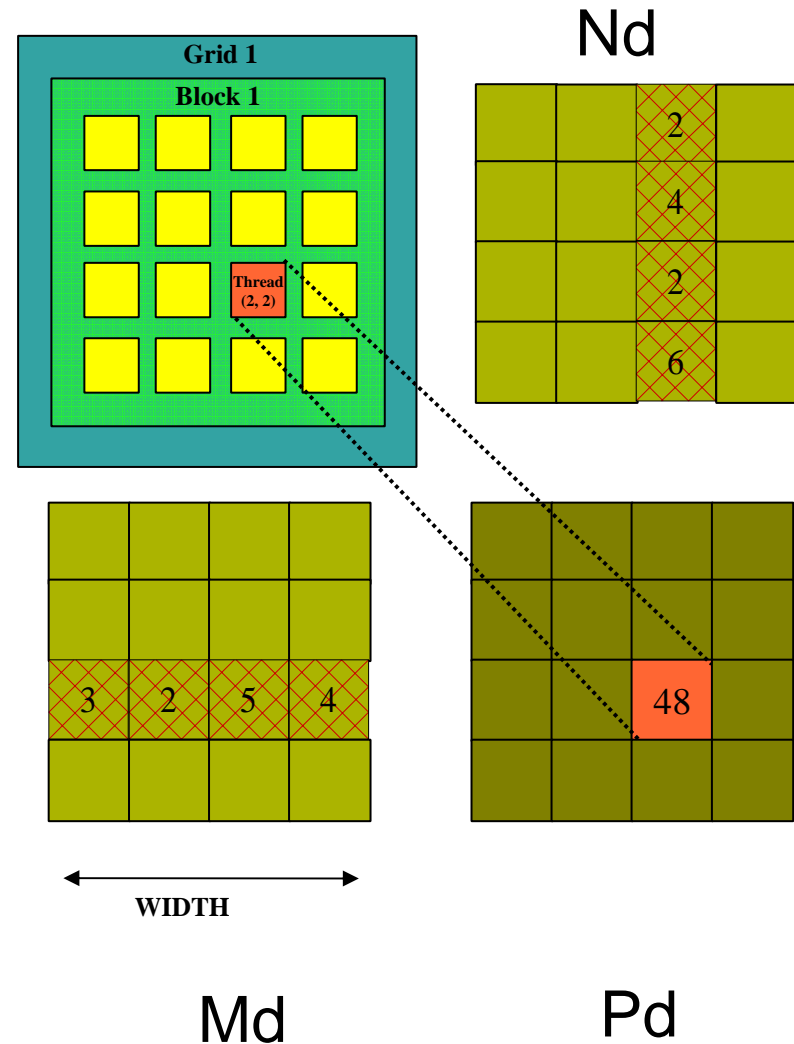
---

```
// Setup the execution configuration  
dim3 dimBlock(Width, Width);  
dim3 dimGrid(1, 1);
```

```
// Launch the device computation threads!  
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);
```

# Only One Thread Block Used

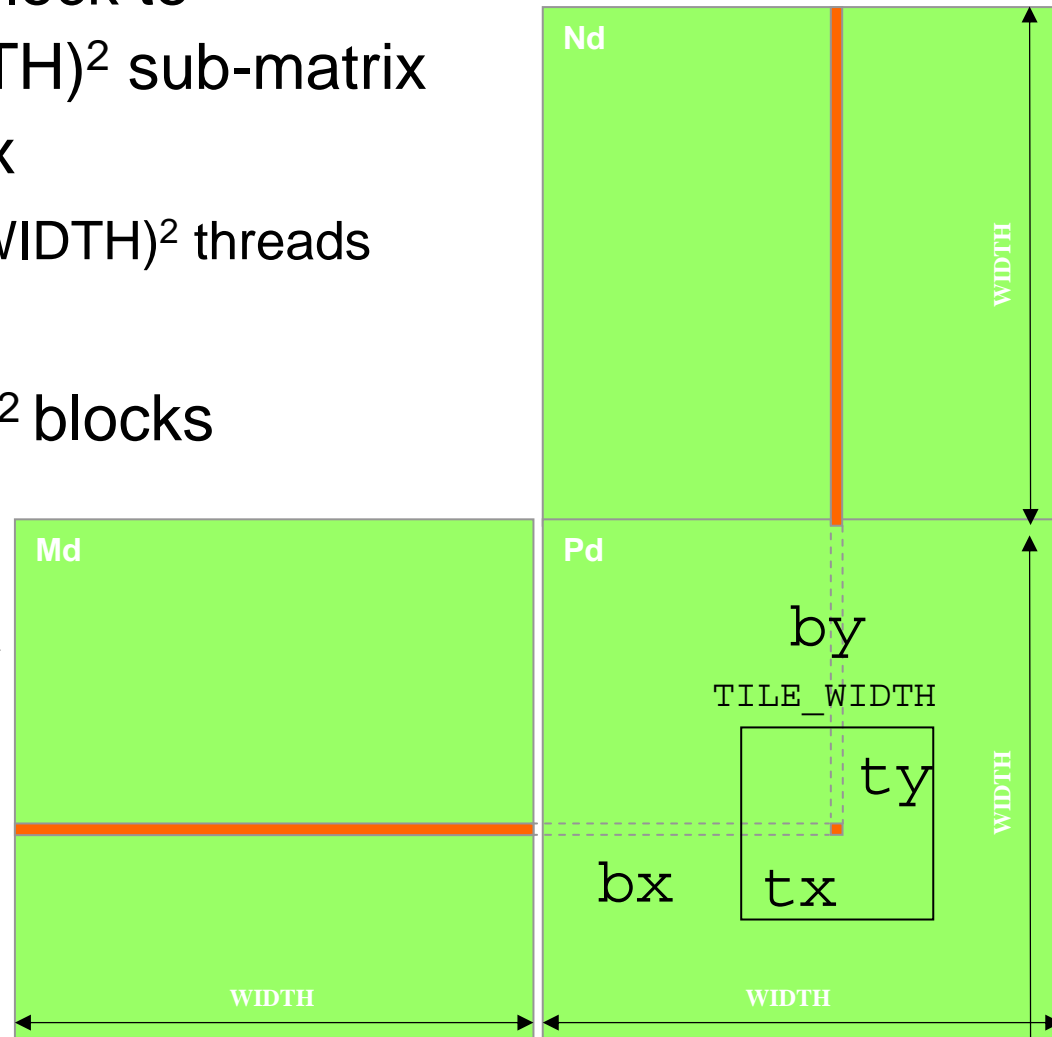
- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



# Step 6: Handling Arbitrary Sized Square Matrices

- Have each 2D thread block to compute a  $(\text{TILE\_WIDTH})^2$  sub-matrix (tile) of the result matrix
  - Each block has  $(\text{TILE\_WIDTH})^2$  threads
- Generate a 2D Grid of  $(\text{WIDTH}/\text{TILE\_WIDTH})^2$  blocks

*Need to put a loop around the kernel call for cases where  $\text{WIDTH}/\text{TILE\_WIDTH}$  is greater than max grid size (64K)!*



# CUDA Programming Model

---

- Compute Unified Device Architecture (CUDA)
- Execution model: kernels, threads, blocks, and grids
- CUDA basics
- A simple example: matrix multiplication
- **CUDA toolkit: libraries, compiler, debugger, emulator, etc.**

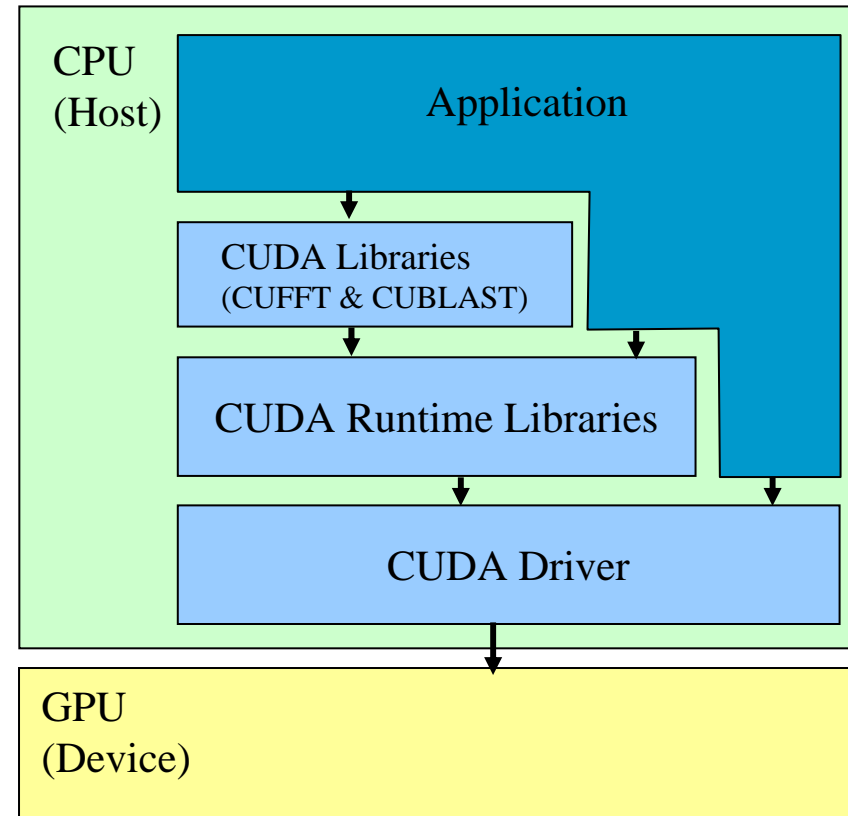
# Application Programming Interface

---

- It consists of:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library can be categorized into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
    - A host component to control and access one or more devices from the host
    - A device component providing device-specific functions

# CUDA Libraries

- CUBLAS
  - BLAS implementation
- CUFFT
  - FFT implementation



# CUBLAS

---

- An implementation of BLAS (Basic Linear Algebra Subprograms) on top of the CUDA driver
  - Allows access to the computational resources of NVIDIA GPUs
  - Self-contained APIs
  - Programmers not necessary not to interact with the CUDA driver directly
- How to use the library
  - Allocate matrices or vectors on device
  - Data padding
  - Call CUBLAS APIs
  - Deliver the result to the host

# CUFFT

---

- Very efficient data parallel implementation of Fast Fourier Transform (FFT)
- Supported features:
  - 1D, 2D and 3D transforms of complex to complex (C2C), real to complex (R2C) and complex to real (C2R)
  - Batched execution for doing multiple 1D transforms in parallel
  - 1D transform size up to 16M elements
  - 2D and 3D transform sizes in the range [2, 16384]
  - In-place and out-of-place transforms for real and complex data

# Common Runtime Component: Mathematical Functions

---

- pow, sqrt, cbrt, hypot
- exp, exp2, expm1
- log, log2, log10, log1p
- sin, cos, tan, asin, acos, atan, atan2
- sinh, cosh, tanh, asinh, acosh, atanh
- ceil, floor, trunc, round
- Etc.
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

# Device Runtime Component:

## Mathematical Functions

---

- Some mathematical functions (e.g. `sin(x)`) have a less accurate, but faster device-only version (e.g. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`
- Use flag “-use\_fast\_math” in compilation

# Device Runtime Component: Synchronization Function

---

- `void __syncthreads();`
- **Synchronizes all threads in a block**
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

# GPU Atomic Integer Operations

---

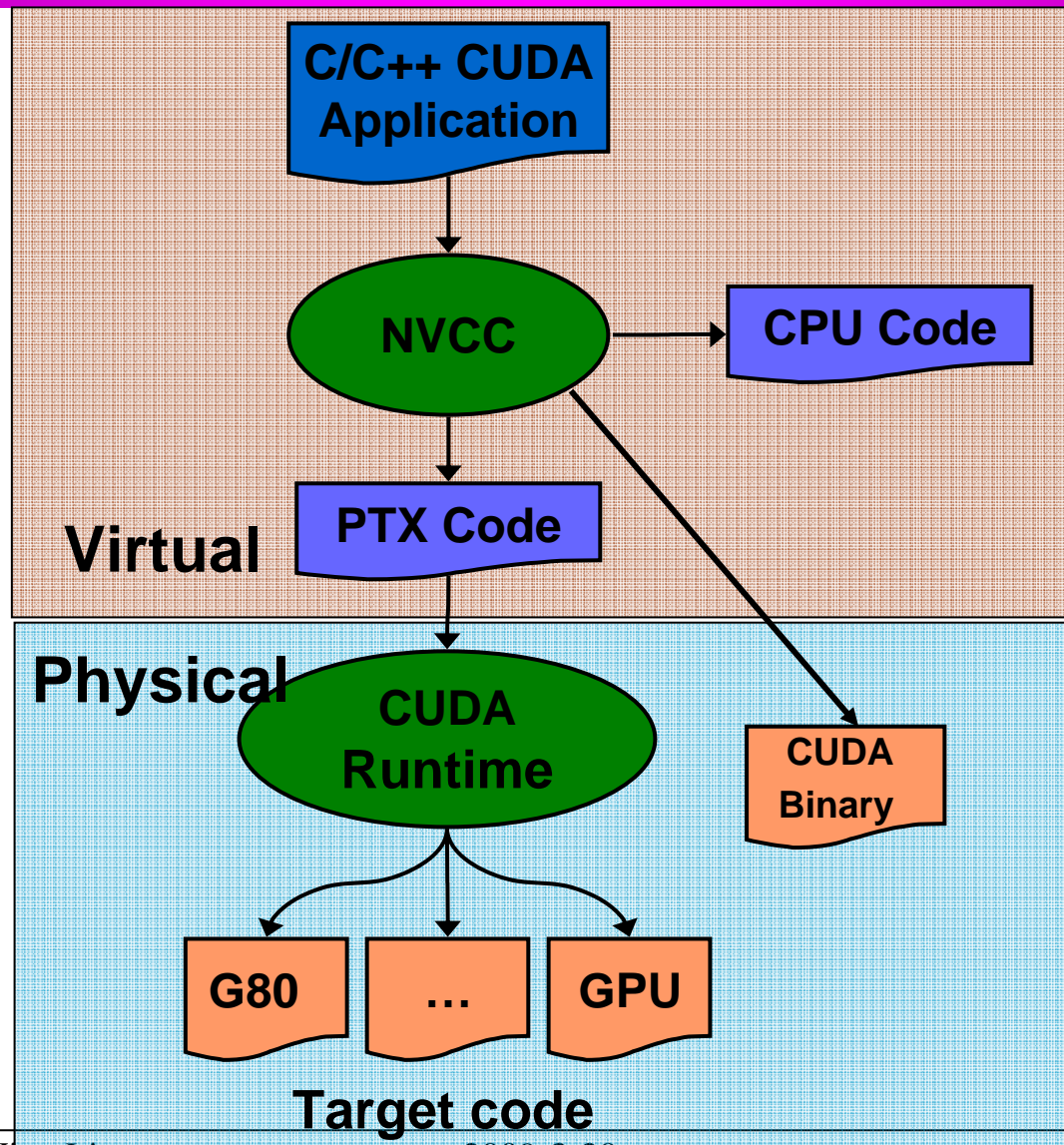
- Allow multiple threads to perform concurrent read-modify-write operations in global memory without conflicts (hardware with compute capability 1.1)
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap
- Useful in sorting, reduction operations and building data structures in parallel
- Add the option "-arch sm\_11" to the nvcc command line
- Hardware with compute capability 1.2 support atomic operations in shared memory

# Linking

---

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudaart`)
  - The CUDA core library (`cuda`)

# Compiling a CUDA Program



# Compilation

---

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++,...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime

# Debugging Using the Device Emulation Mode

---

- An executable compiled in `device emulation mode` (`nvcc -deviceemu`) runs completely on the host
  - Need the CUDA runtime library
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread
- Running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

# Device Emulation Mode Pitfalls

---

- The results of floating point operations may be slight different
- **Dereferencing** device **pointers** on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode