ECE/ME/EMA/CS 759
High Performance Computing
for Engineering Applications

Parallel Computing on the GPU
Execution Configuration
Elements of CUDA API

September 30, 2015

Quote of the Day

"What you're thinking is what you're becoming."

— Muhammad Ali

Before We Get Started

- Issues covered last time:
  - GPU computing
    - Generalities

- Today's topics

  - Parallel computing on GPU cards
    - Execution Configuration
    - CUDA API

- Assignment:
  - HW03 – due on Oct. 2 at 11:59 PM
  - HW04 – posted online later today and due on Oct. Oct. 7 at 11:59 PM

- Midterm Exam: 10/09 (Friday)
  - Review on Th 10/08, at 7:15 PM, room TBA

When Are GPUs Good?

- Ideally suited for data-parallel computing (SIMD)

- Moreover, you want to have high arithmetic intensity
  - Arithmetic intensity: ratio or arithmetic operations to memory operations

- You are off to a good start with GPU computing if you can do this…
  - Get the data on the GPU and keep it there
  - Give the GPU enough work to do
  - Focus on data reuse within the GPU to avoid memory bandwidth limitations

CUDA, Second Example

- Multiply, pairwise, two arrays of 3 million integers

```
int main(int argc, char* argv[])
{
    const int arraySize = 3000000; // 3,000,000 entries in each array
    int *hA, *hB, *hC;
    setupHost(&hA, &hB, &hC, arraySize);

    int *dA, *dB, *dC;
    setupDevice(&dA, &dB, &dC, arraySize);

    cudaMemcpy(dA, hA, sizeof(int) * arraySize, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, hB, sizeof(int) * arraySize, cudaMemcpyHostToDevice);

    const int threadsPerBlock = 512;
    const int blockSizeMultiplication = arraySize/threadsPerBlock + 1;
    multiply_ab<<<blockSizeMultiplication,threadsPerBlock>>>(dA,dB,dC,arraySize);
    cudaMemcpy(hC, dC, sizeof(int) * arraySize, cudaMemcpyDeviceToHost);

    cleanupHost(hA, hB, hC);
    cleanupDevice(dA, dB, dC);
    return 0;
}
```

CUDA, Second Example
[Cntd.]

```
__global__ void multiply_ab(int* a, int* b, int* c, int size)
{
   int whichEntry = threadIdx.x + blockIdx.x*blockDim.x;
   if( whichEntry<size )
      c[whichEntry] = a[whichEntry]*b[whichEntry];
}
```

```
void setupDevice(int** pdA, int** pdB, int** pdC, int arraySize)
{
    cudaMalloc((void**) pdA, sizeof(int) * arraySize);
    cudaMalloc((void**) pdB, sizeof(int) * arraySize);
    cudaMalloc((void**) pdC, sizeof(int) * arraySize);
}

void cleanupDevice(int *dA, int *dB, int *dC)
{
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
}
```

The Concept of Execution Configuration

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

```
__global__ void kernelFoo(...); // declaration

dim3   DimGrid(100, 50);          // 5000 thread blocks
dim3   DimBlock(4, 8, 8);         // 256 threads per block

kernelFoo<<< DimGrid, DimBlock>>>(...your arg list comes here…);
```

- Recall that any call to a kernel function is asynchronous
  - By default, execution on host doesn't wait for kernel to finish

Example

- The host call below instructs the GPU to execute the function (kernel) "**`foo`**" using 25,600 threads
  - Two arguments are passed down to each thread executing the kernel "`foo`"

```
foo<<<100,256>>>(pMyMatrixD, pMyVecD)
```

- In this execution configuration, the host instructs the device that it is supposed to run 100 blocks each having 256 threads in it

- The concept of block is important since it represents the entity that gets executed by an SM (stream multiprocessor)

More on the Execution Configuration

[Some CUDA Constraints]

- There is a limitation on the number of blocks in a grid:
  - The grid of blocks can be organized as a 3D structure: max of 65,535 by 65,535 by 65,535 grid of blocks (about 280,000 billion blocks)

- Threads in each block:
  - The threads can be organized as a 3D structure (x,y,z)
  - The total number of threads in each block cannot be larger than 1024
    - More on this 1024 number later

Execution Configuration: Dealing with Multiple Blocks

- Motivation: there is a limit on the number of threads squeezed in a block
  - As we saw, you can have up to 1024 threads in a block

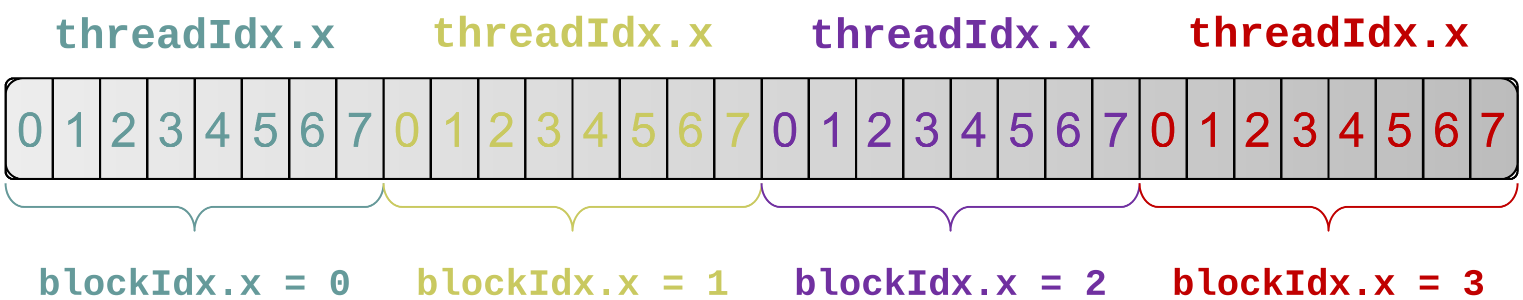
- Purpose of discussion: elaborate on a scenario when multiple blocks are needed and how this reflects into the array indexing scheme

- Lesson to be learned: Indexing no longer as simple as using only **`threadIdx.x`**
  - One will have to account for the size of the block as well

Example: Array Indexing

[Important to grasp: thread-to-task mapping]

- Consider indexing into an array, one thread accessing one element
- Assume you launch w/ `M=8` threads per block and the array is 32 entries long



- With M threads per block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

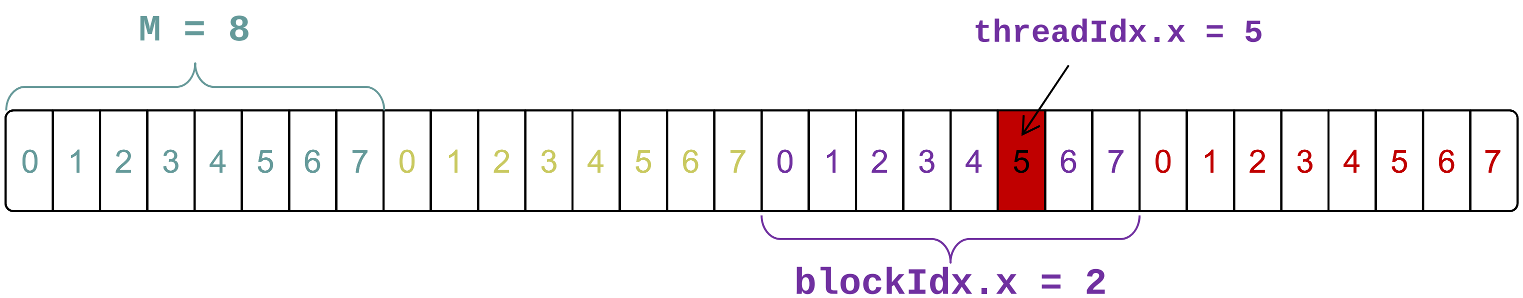
Size of the block of threads; i.e., **`blockDim.x`**

[NVIDIA]→

Example: Array Indexing

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

- What is the array entry that thread of index 5 in block of index 2 will work on?




```
int index = threadIdx.x + blockIdx.x * blockDim.x;
          =      5      +     2      * 8;
          = 21;
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

[NVIDIA]→

A Recurring Theme in CUDA Programming
[and in SIMD in general]

- Imagine you are one of many threads, and you have your thread index and block index
  - You need to figure out what the job you need to complete
    - Just like we did on previous slide where thread 5 in block 2 mapped into 21
  - One caveat: You have to make sure you actually need to do that work
    - In many cases there are threads, typically of large id, that need to do no work
    - Example: you launch two blocks with 512 threads but your array is only 1000 elements long. Then 24 threads at the end do nothing

Before Moving On…

[Some Words of Wisdom]

- In GPU computing you use as many threads as data items [tasks][jobs] you have to perform
  - This replaces the purpose in life of the “for” loop
  - Number of threads & blocks is established at run-time

- Number of threads = Number of data items [tasks][jobs]
  - It means that you’ll have to come up with a rule to match a thread to a data item[task][job] that this thread needs to process
  - Common source of errors and frustration in GPU computing
    - It never fails to deliver (frustration)

:-(

Review of Nomenclature…

- The HOST
  - This is your CPU executing the “master” thread
- The DEVICE
  - This is the GPU card, connected to the HOST through a PCIe connection
- The HOST (the master CPU thread) calls DEVICE to execute KERNEL
- When calling the KERNEL, the HOST also has to inform the DEVICE how many threads should each execute the KERNEL
  - This is called “defining the <u>execution configuration</u>”

Matrix Multiplication Example

Simple Example: Matrix Multiplication

- Purpose: Illustrate the basic features of memory and thread management in CUDA programs
- Quick remarks
  - We'll use only global memory
    - Shared memory usage discussion postponed later
  - Matrix will be of small dimension, job can be done using one block
  - We'll concentrate on two things:
    - Thread ID usage
    - Memory data transfer API between host and device

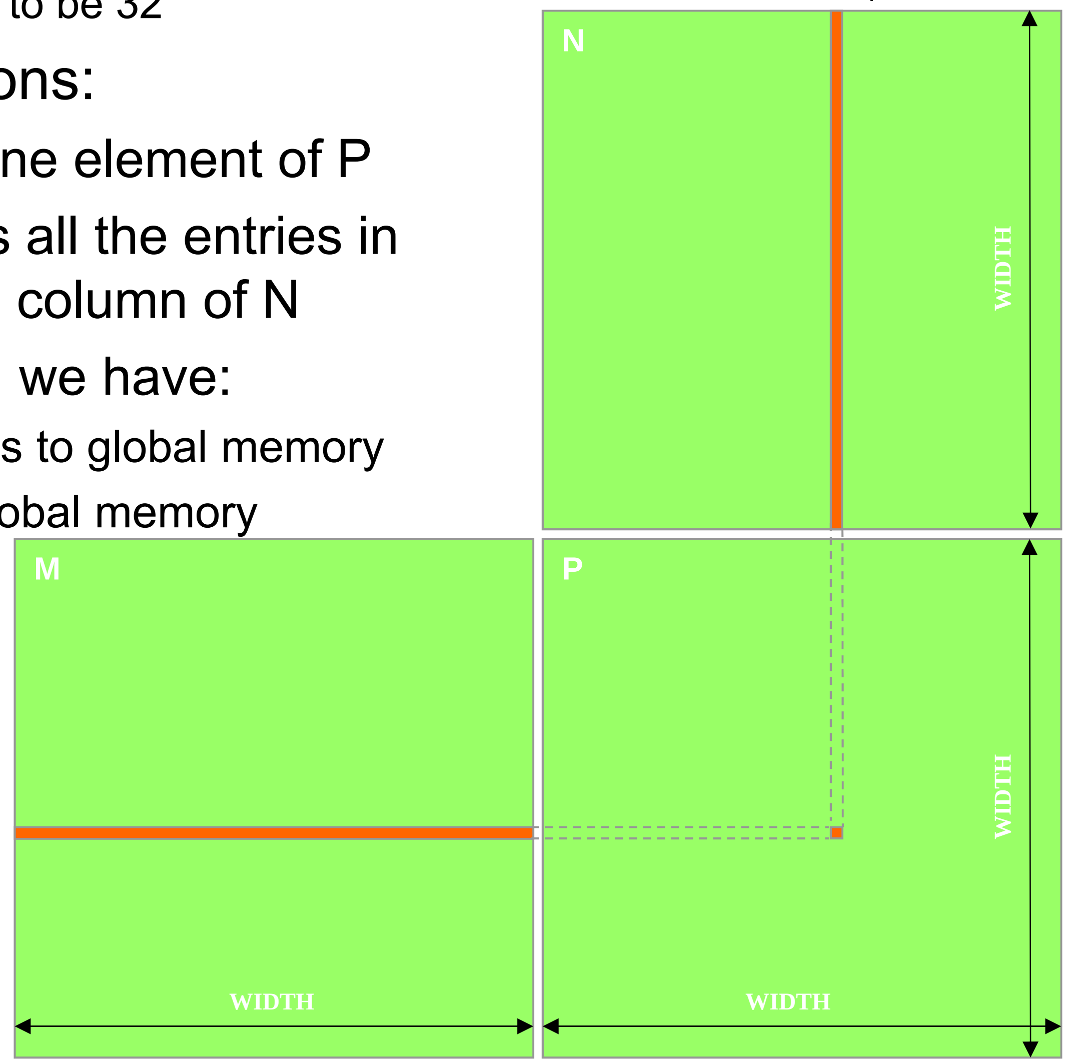
Matrix Data Structure

- The following data structure will come in handy
  - Purpose: store info related to a matrix
  - Note that the matrix is stored in row-major order in a one dimensional array pointed to by “elements”

```
// IMPORTANT - Matrices are stored in row-major order:
// M(row, col) = M.elements[row * M.width + col]

typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

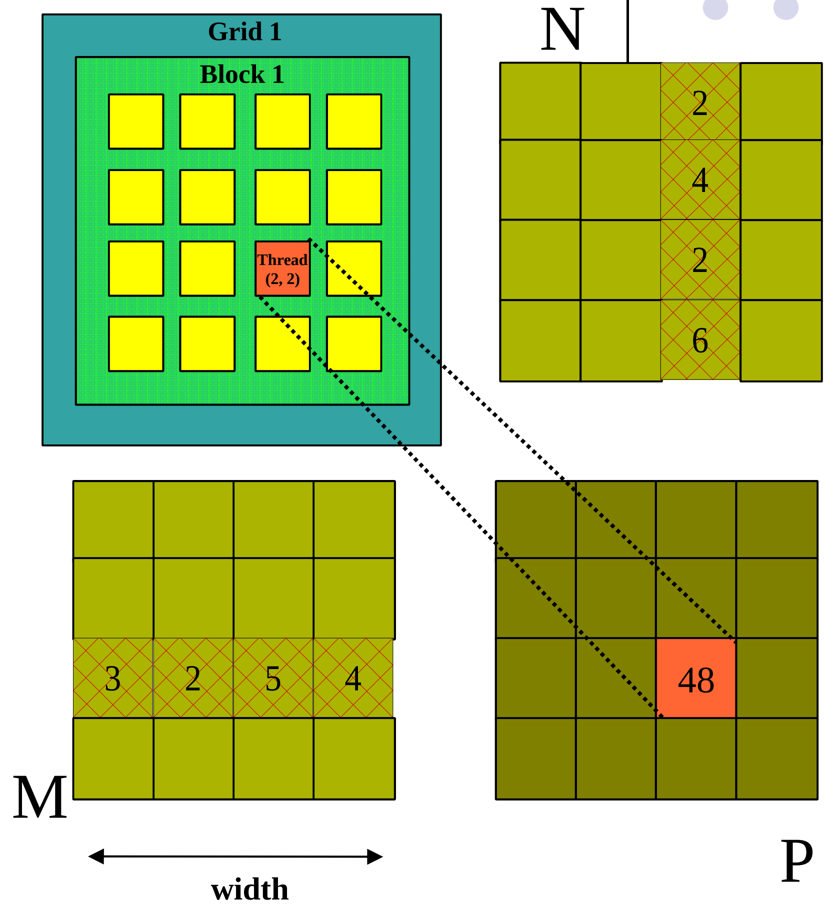
Square Matrix Multiplication Example

- Compute P = M * N
  - The matrices P, M, N are of size WIDTH x WIDTH
  - Assume WIDTH was defined to be 32
- Software Design Decisions:
  - One thread handles one element of P
  - Each thread accesses all the entries in one row of M and one column of N
  - Therefore, per thread, we have:
    - 2*WIDTH read accesses to global memory
    - One write access to global memory



Multiply Using One Thread Block

- One Block of threads computes matrix P
  - Each thread computes one element of P
- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1
    - Not that good, acceptable for now…
- Size of matrix limited by the number of threads allowed in a thread block



Matrix Multiplication: Sequential Approach, Coded in C

```
// Matrix multiplication on the (CPU) host in double precision;

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double accumulator = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];  //march along a row of M
                double b = N.elements[k * N.width + j];  //march along a column of N
                accumulator += a * b;
            }
            P.elements[i * N.width + j] = accumulator;
        }
    }
}
```

GPU Implementation
Step 1: Matrix Multiplication, Host-side. Main Program Code

```
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1.  No: 0 (everything is set to zero)
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);

   return 0;
}
```

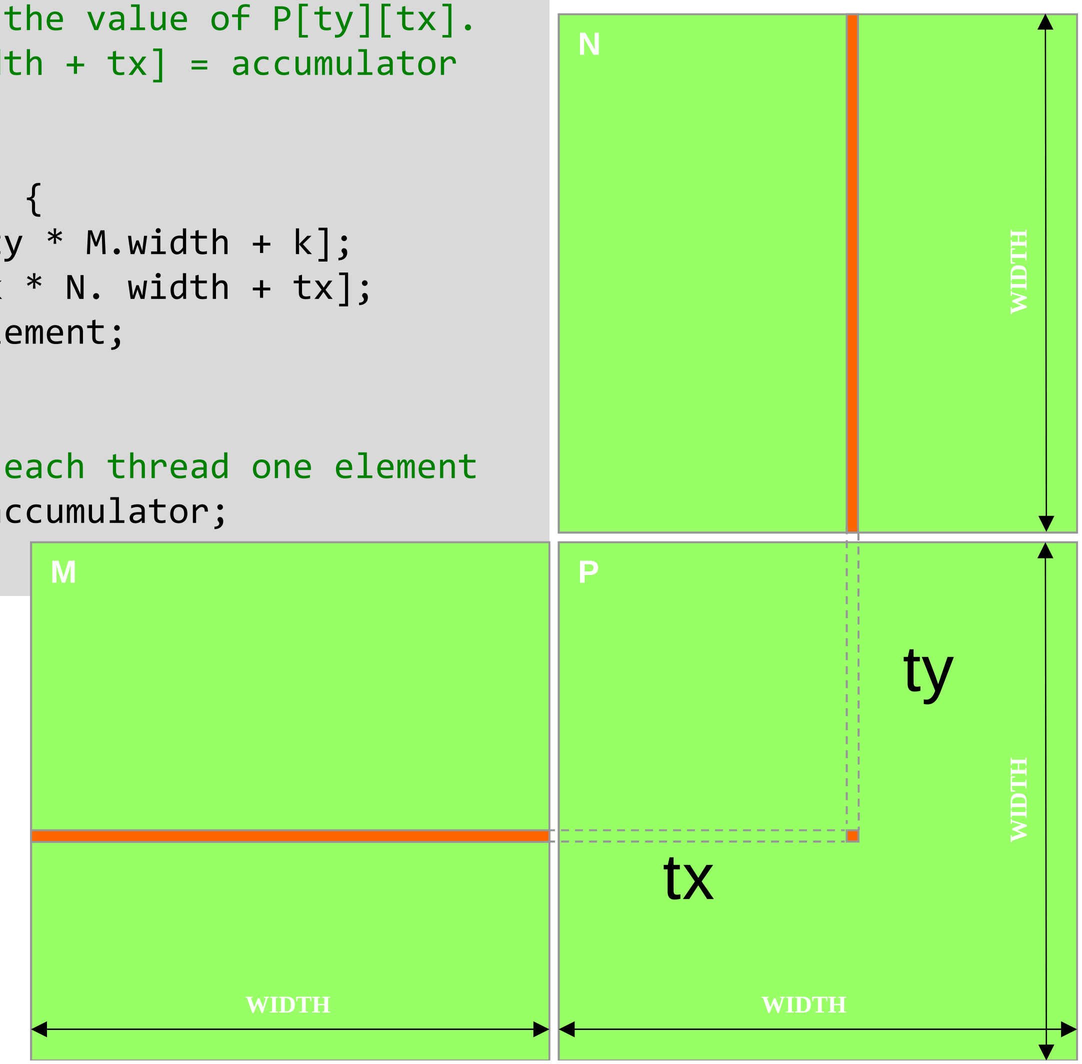
NOTE: `WIDTH=32`

Step 2: Matrix Multiplication
[host-side code]

```
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);

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

    // Launch the kernel on the device
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

Step 4: Matrix Multiplication- Device-side Kernel Function

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {
    // 2D Thread Index; computing P[ty][tx]…
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Computed value ends up storing the value of P[ty][tx].
    // That is, P.elements[ty * P. width + tx] = accumulator
    float accumulator = 0.0;

    for (int k = 0; k < M.width; ++k)  {
         float Melement = M.elements[ty * M.width + k];
         float Nelement = N.elements[k * N. width + tx];
         accumulator += Melement * Nelement;
    }

    // Write matrix to device memory; each thread one element
    P.elements[ty * P. width + tx] = accumulator;
}
```

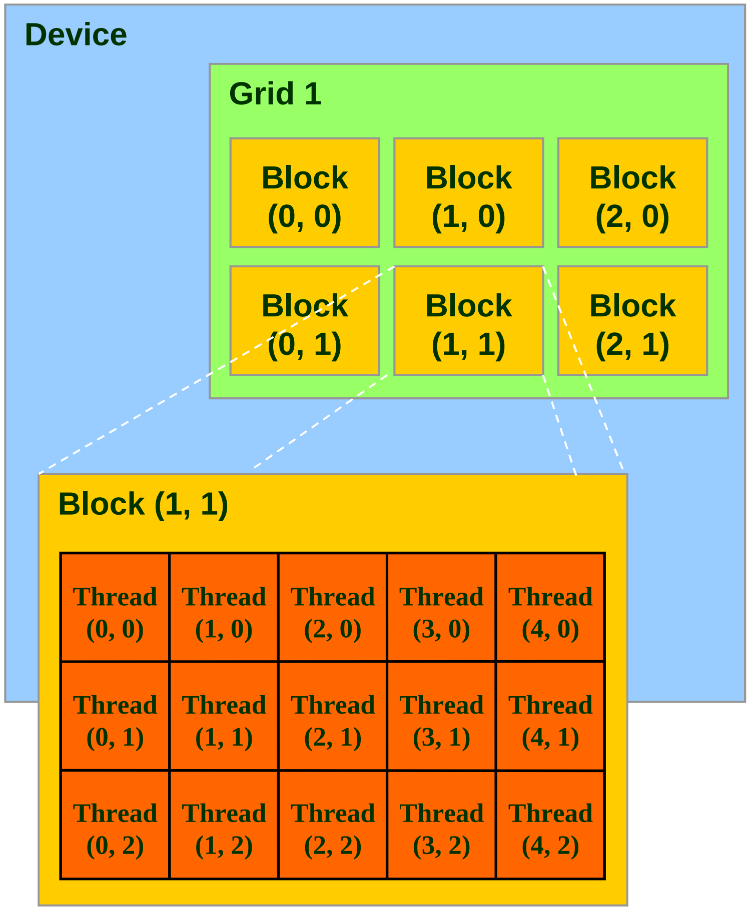


Step 4: Some Loose Ends

```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M) {
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost) {
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size, cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice) {
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size, cudaMemcpyDeviceToHost);
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

Block and Thread Index (Idx)

- Threads and blocks have indices
  - **Used by each thread the decide what data to work on (more later)**
  - Block Index: a triplet of `uint`
  - Thread Index: a triplet of `uint`
- Why this 3D layout?
  - Simplifies memory addressing when processing multidimensional data
    - Handling matrices
    - Solving PDEs on subdomains
    - …



A Couple of Built-In Variables

[in support of the SIMD parallel computing paradigm]

- It's essential for each thread to be able to find out the grid and block dimensions and its block index and thread index

- Each thread when executing a kernel has access to the following read-only built-in variables
  - `threadIdx` (`uint3`) – contains the thread index within a block
  - `blockDim` (`dim3`) – contains the dimension of the block
  - `blockIdx` (`uint3`) – contains the block index within the grid
  - `gridDim` (`dim3`) – contains the dimension of the grid
  - [`warpSize` (`uint`) – provides warp size, we'll talk about this later…]

Thread Index vs. Thread ID

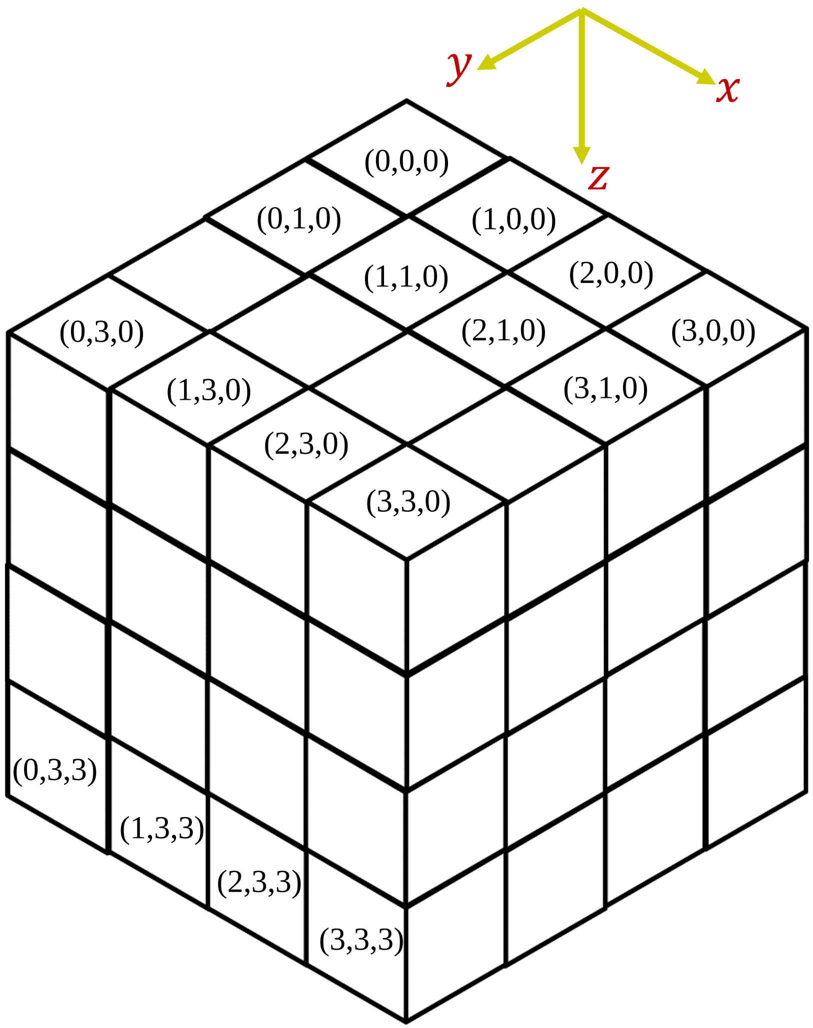
[important slide for (i) understanding how SIMD is supported in CUDA; and (ii) understanding later on the concept of "warp"]

- Each block organizes its threads in a 3D structure defined by its three dimensions: D_x, D_y, and D_z that you specify.
- A block cannot have more than 1024 threads $\Rightarrow$ $D_x \times D_y \times D_z \leq 1024$.
- Each thread in a block can be identified by a unique index (x, y, z), and

$$0 \leq x < D_x \qquad 0 \leq y < D_y \qquad 0 \leq z < D_z$$

- A triplet (x, y, z), called the thread index, is a high-level representation of a thread in the economy of a block. Under the hood, the same thread has a simplified and unique id, which is computed as $t_{id} = x + y * D_x + z * D_x * D_y$. You can regard this as a "projection" to a 1D representation. The concept of thread id is important in understanding how threads are grouped together in warps (more on "warps" later).
- In general, operating for vectors typically results in you choosing $D_y = D_z = 1$. Handling matrices typically goes well with $D_z = 1$. For handling PDEs in 3D you might want to have all three block dimensions nonzero.

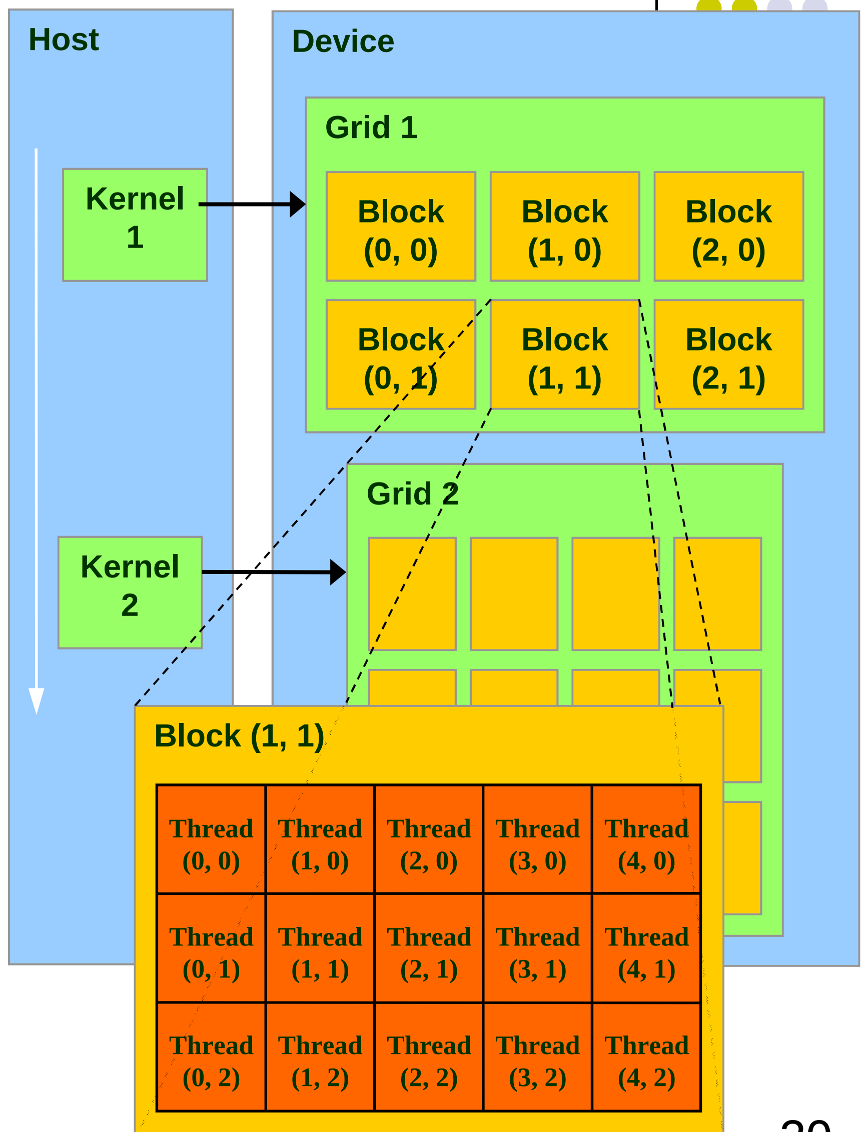
Example:
A CUDA block of dimension (4,4,4)



- Exam type questions:
  - How many threads apart are the threads of index (2,2,2) and (3,2,2)?
  - How many threads apart are the threads of index (2,2,2) and (2,3,2)?
  - How many threads apart are the threads of index (2,2,2) and (2,2,3)?
  - How many threads apart are the threads of index (2,2,2) and (3,3,3)?

Revisit - Execution Configuration: Grids and Blocks

- A kernel is executed as a grid of blocks of threads
  - All threads executing a kernel can access several device data memory spaces
- A block [of threads] is a collection of threads that can cooperate with each other by:
  - Synchronizing their execution
  - Efficiently sharing data through a low latency shared memory
- Check your understanding:
  - How was the grid defined for this pic?
    - I.e., how many blocks in X and Y directions?
  - How was a block defined in this pic?



[Sidebar]
Timing Your Application

- Timing support – part of the CUDA API
  - You pick it up as soon as you include `<cuda.h>`
  - Why it is good to use
    - Provides cross-platform compatibility
    - Deals with the asynchronous nature of the device calls by relying on events and forced synchronization
  - Reports time in miliseconds, accurate within 0.5 microseconds
    - From NVIDIA CUDA Library Documentation:
      - Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns `cudaErrorInvalidValue`. If either event has been recorded with a non-zero stream, the result is undefined.

Timing Example

~ Timing a GPU call ~

```
#include<iostream>
#include<cuda.h>

int main() {
    cudaEvent_t startEvent, stopEvent;
    cudaEventCreate(&startEvent);
    cudaEventCreate(&stopEvent);

    cudaEventRecord(startEvent, 0);

    yourKernelCallHere<<<NumBlk,NumThrds>>>(args);

    cudaEventRecord(stopEvent, 0);
    cudaEventSynchronize(stopEvent);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
    std::cout << "Time to get device properties: " << elapsedTime << " ms\n";

    cudaEventDestroy(startEvent);
    cudaEventDestroy(stopEvent);
    return 0;
}
```

The CUDA API

What is an API?

- Application Programming Interface (API)
  - “A set of functions, procedures or classes that an operating system, library, or service provides to support requests made by computer programs” (from Wikipedia)
  - Example: OpenGL, a graphics library, has its own API that allows one to draw a line, rotate it, resize it, etc.

- In this context, CUDA provides an API that enables you to tap into the computational resources of the NVIDIA’s GPUs
  - This replaced the old GPGPU way of programming the hardware
  - CUDA API exposed to you through a collection of header files that you include in your program

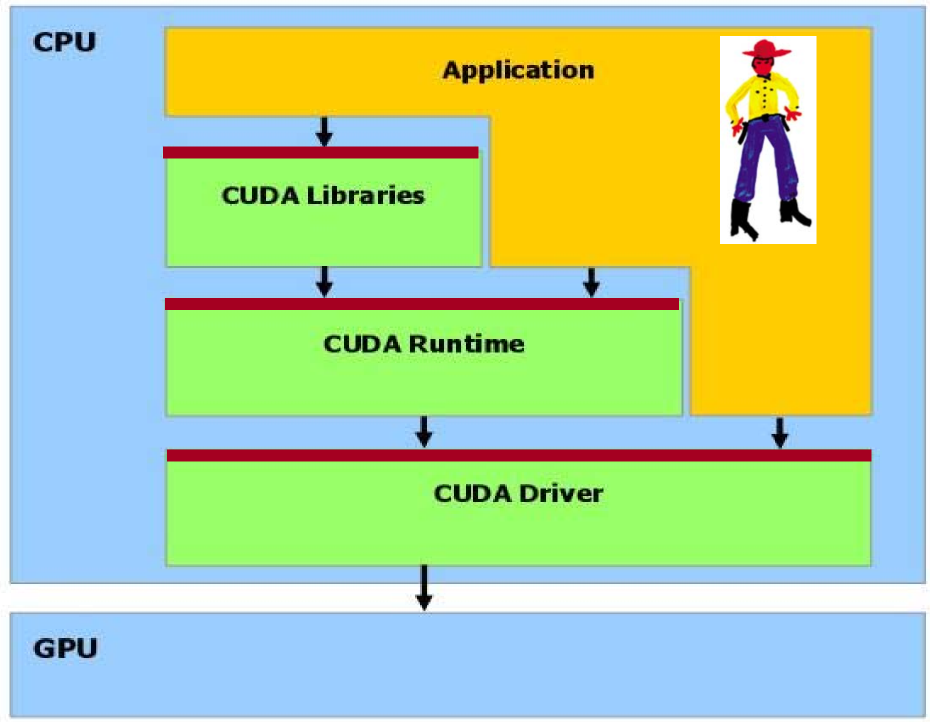
On the CUDA API

- Reading the CUDA Programming Guide you'll run into numerous references to the CUDA Runtime API and CUDA Driver API
  - Many time they talk about "CUDA runtime" and "CUDA driver". What they mean is CUDA Runtime API and CUDA Driver API
- CUDA Runtime API – is the friendly face that you can choose to see when interacting with the GPU. This is what gets identified with "C CUDA"
  - Needs **nvcc** compiler to generate an executable
- CUDA Driver API – low level way of interacting with the GPU
  - You have significantly more control over the host-device interaction
  - Significantly more clunky way to dialogue with the GPU, typically only needs a C compiler
- Almost everybody uses the CUDA Runtime API

Talking about the API: The C CUDA Software Stack

- Image at right indicates where the API fits in the picture

An API layer is indicated by a thick red line:



- NOTE: any CUDA runtime function has a name that starts with "cuda"
  - Examples: cudaMalloc, cudaFree, cudaMemcpy, etc.
- Examples of CUDA Libraries: CUFFT, CUBLAS, CUSP, thrust, etc.

Application Programming Interface (API) ~Taking a Step Back~

- CUDA runtime API: exposes a set of extensions to the C language
  - Spelled out in an appendix of "NVIDIA CUDA C Programming Guide"
  - There is many of them $\rightarrow$ Keep in mind the 20/80 rule

- CUDA runtime API:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library, which is split into:
    - A common component providing built-in vector types and a subset of the C runtime library available in both host and device codes
      - Callable both from device and host
    - A host component to control and access devices from the host
      - Callable from the host only
    - A device component providing device-specific functions
      - Callable from the device only

Language Extensions: Variable Type Qualifiers

	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

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`

- Automatic variables without any qualifier reside in a register
  - Except arrays, which reside in local memory (unless they are small and of known constant size)

Common Runtime Component

- “Common” above refers to functionality that is provided by the CUDA API and is common both to the device and host

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes

<u>Common</u> Runtime Component: Built-in Vector Types

- `[u]char[1..4]`, `[u]short[1..4]`, `[u]int[1..4]`, `[u]long[1..4]`, `float[1..4]`, `double[1..2]`
  - Structures accessed with `x`, `y`, `z`, `w` fields:

```
uint4 param;
int dummy = param.y;
```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions
  - You see a lot of it when defining the execution configuration of a kernel (any component left uninitialized assumes default value 1)

See Appendix B in
"NVIDIA CUDA C Programming Guide"

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 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`
- Some of these have hardware implementations
- By using the “-use_fast_math” flag, `sin(x)` is substituted at compile time by `__sin(x)`

```
>> nvcc -arch=sm_20 –use_fast_math foo.cu
```

Host Runtime Component

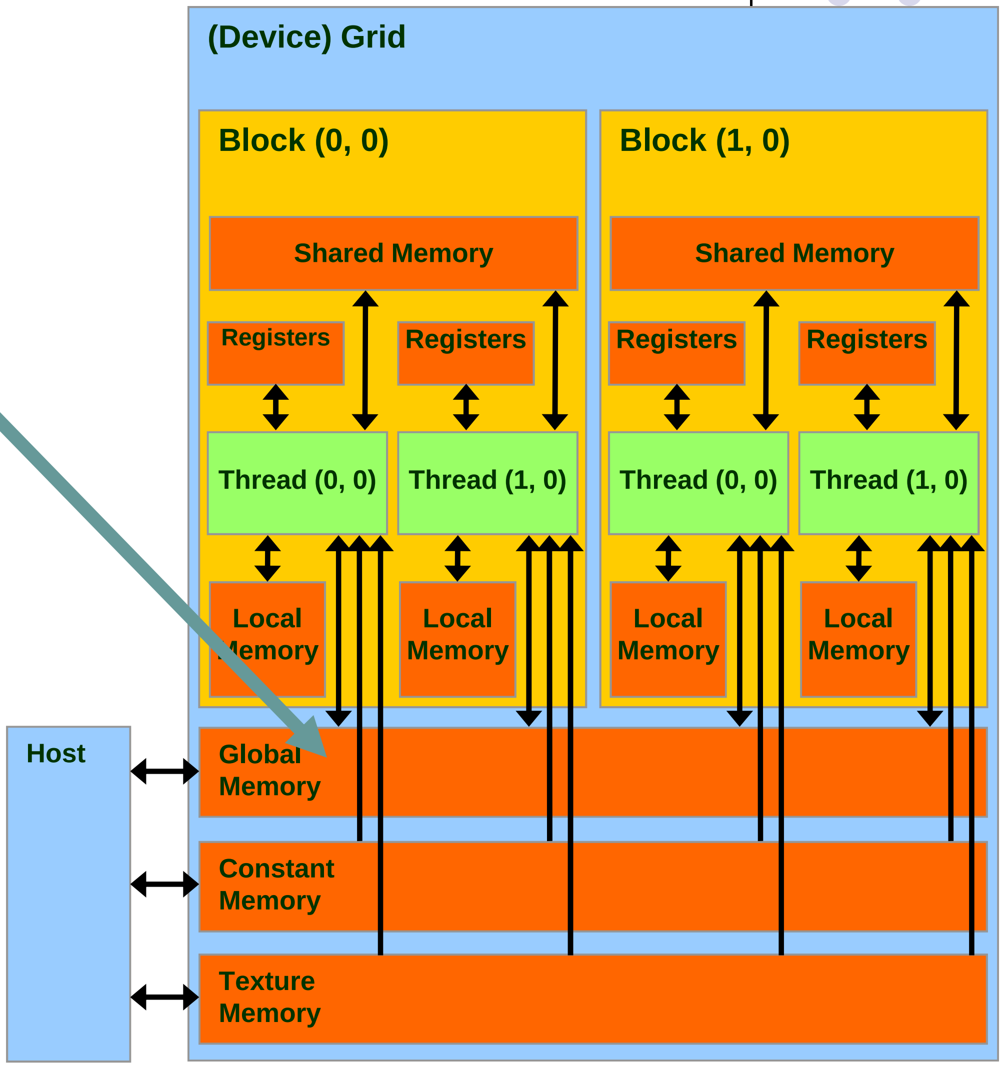
- Provides functions available only to the host to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Error handling

- Examples
  - Device memory allocation
    - `cudaMalloc()`, `cudaFree()`
  - Memory copy from host to device, device to host, device to device
    - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
  - Memory addressing – returns the address of a device variable
    - `cudaGetSymbolAddress()`

CUDA API: Device Memory Allocation

[Note: picture assumes two blocks, each with two threads]

- cudaMalloc()
  - Allocates object in the device <u>Global Memory</u>
  - 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



Example Use: A Matrix Data Type

```
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```

- NOT part of CUDA API
- Used in several code examples
  - 2 D matrix
  - Single precision float elements
  - `width` * `height` entries
  - Matrix entries attached to the pointer-to-float member called "`elements`"
  - Matrix is stored <u>row-wise</u>

Example
CUDA Device Memory Allocation (cont.)

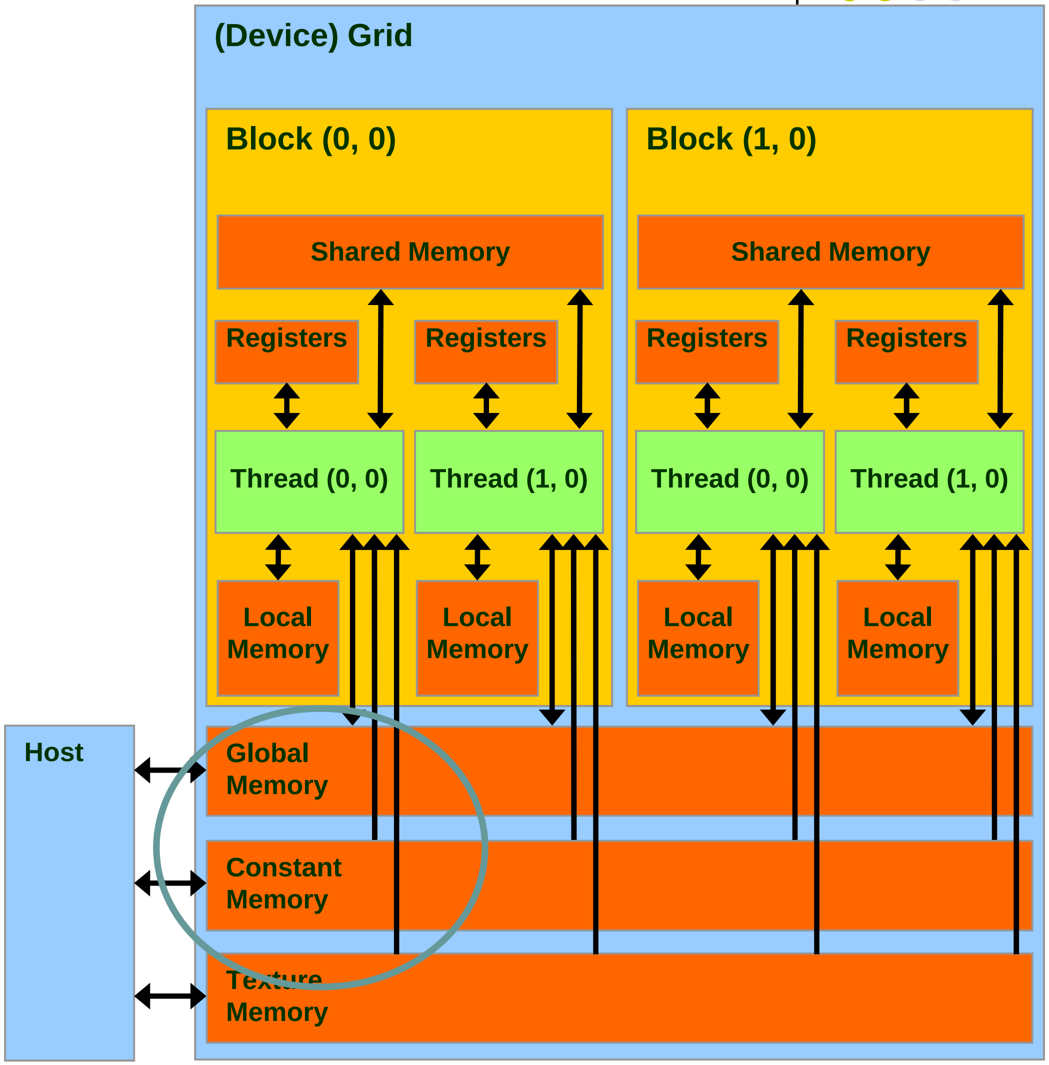
- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to **`Md.elements`**
  - "d" in "Md" is often used to indicate a device data structure

```
BLOCK_SIZE = 64;
Matrix Md;
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);
…
//use it for what you need, then free the device memory
cudaFree(Md.elements);
```

<u>Question</u>: why is the type of the first argument `(void **)` ?

CUDA Host-Device Data Transfer

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



CUDA Host-Device Data Transfer (cont.)

- 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.elements, M.elements, size, cudaMemcpyHostToDevice);

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