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

Array Indexing Example
Timing GPU Kernels
The CUDA API
The Memory Ecosystems

September 27, 2013

"There are three rules to follow when parallelizing large codes.
Unfortunately, no one knows what these rules are."
W. Somerset Maugham and Gary Montry

Before We Get Started…

- Last time
  - Covered the "execution configuration"
  - Discussion, thread index vs. thread ID
- Today
  - Example, working w/ large arrays
  - Timing a kernel execution
  - The CUDA API
  - The memory ecosystem
- Miscellaneous
  - Third assignment posted and due on Monday at 11:59 PM
    - Has to do with GPU computing
  - Read pages 56 through 73 of the primer: http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf
    - Please post suggestions for improvement

Example: Array Indexing

- Purpose of Example: see a scenario of how multiple blocks are used to index entries in an array

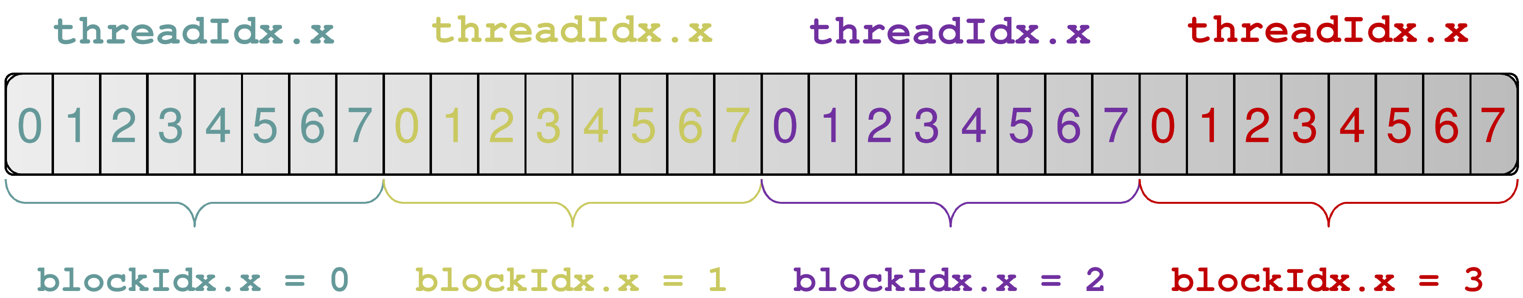
- First, recall this: there is a limit on the number of threads you can squeeze in a block (up to 1024 of them)

- Note: In the vast majority of applications you need to use many blocks (each of which contains the same number N of threads) to get a job done. This example puts things in perspective

Example: Array Indexing

[Important to grasp: shows thread to task mapping]

- No longer as simple as using only **`threadIdx.x`**
  - Consider indexing into an array, one thread accessing one element
  - Assume you have **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:

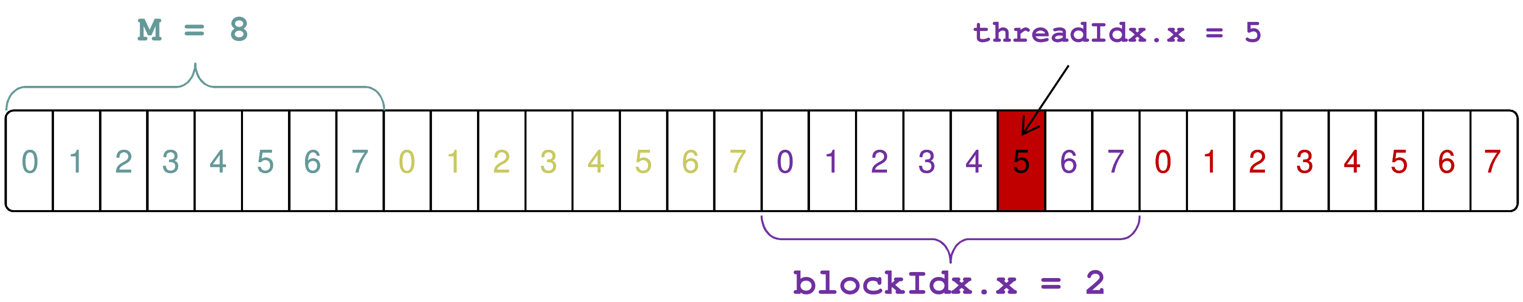
```
int index = threadIdx.x + blockIdx.x * M;
```

[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 will be the array entry that thread of index 5 in block of index 2 will work on?




```
int index = threadIdx.x + blockIdx.x * M;
          =       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 work you need to do is
    - Just like we did on previous slide, where thread 5 in block 2 mapped into 21
  - 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 launch 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)
  - It means that you’ll have to come up with a rule to match a thread to a data item (task) that this thread needs to process
  - Solid source of errors and frustration in GPU computing
    - It never fails to deliver (frustration)

:-(

[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 query of device 0 properties ~

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

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

    cudaEventRecord(startEvent, 0);

    cudaDeviceProp deviceProp;
    const int currentDevice = 0;
    if (cudaGetDeviceProperties(&deviceProp, currentDevice) == cudaSuccess)
        printf("Device %d: %s\n", currentDevice, deviceProp.name);

    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 is what replaced old GPGPU way of programming the hardware
  - CUDA API exposed to you through a collection of header files that you include in your program

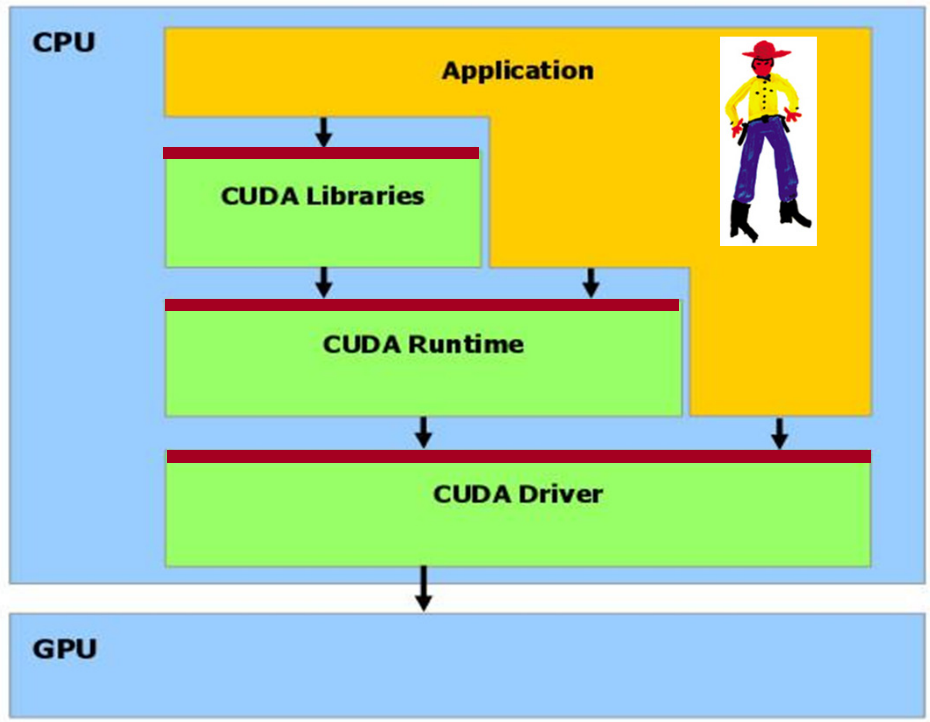
On the CUDA API

- 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 clunkier way to dialogue with the GPU, typically only needs a C compiler
- I don't anticipate any reason to use the CUDA Driver 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 → 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

Common 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

Host Runtime Component

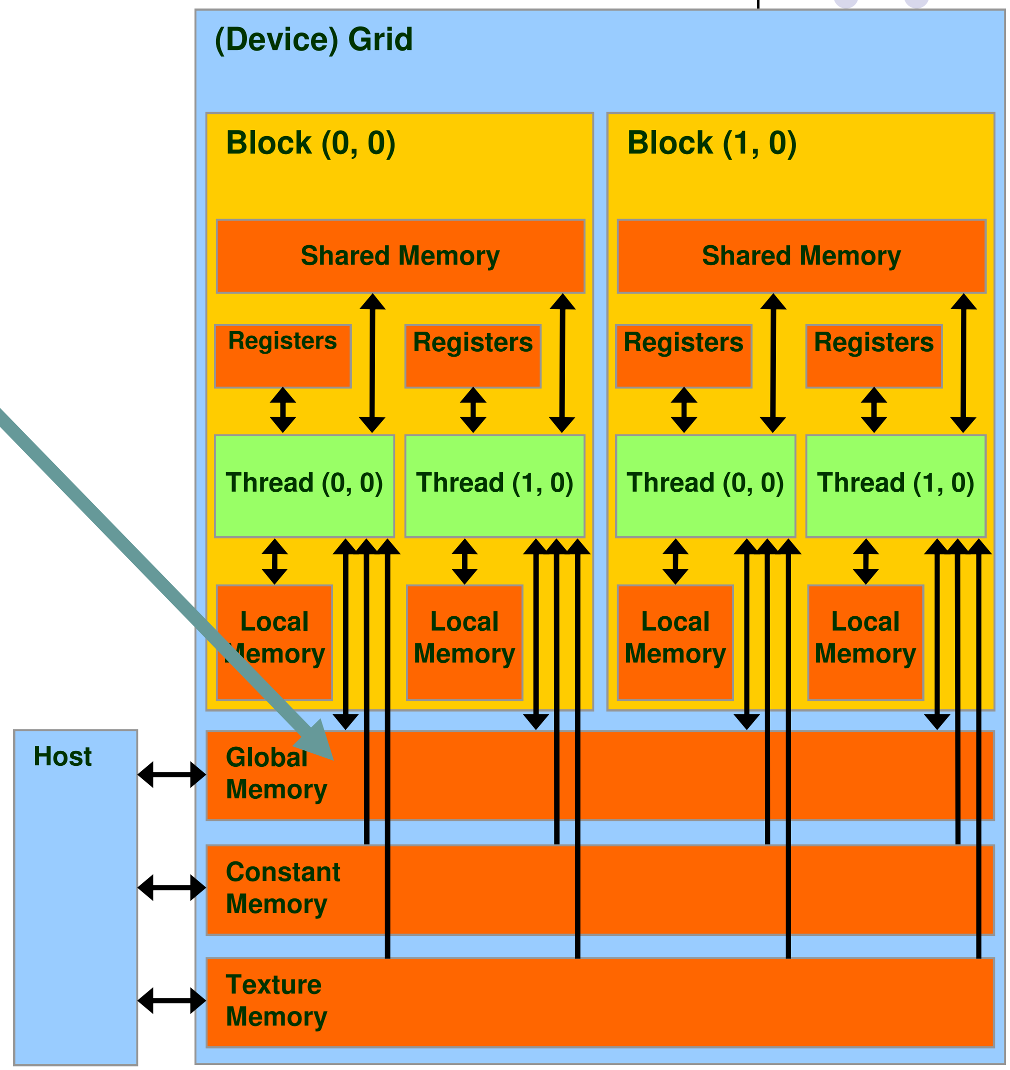
- Provides functions available only to the host to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Error handling

- 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.)

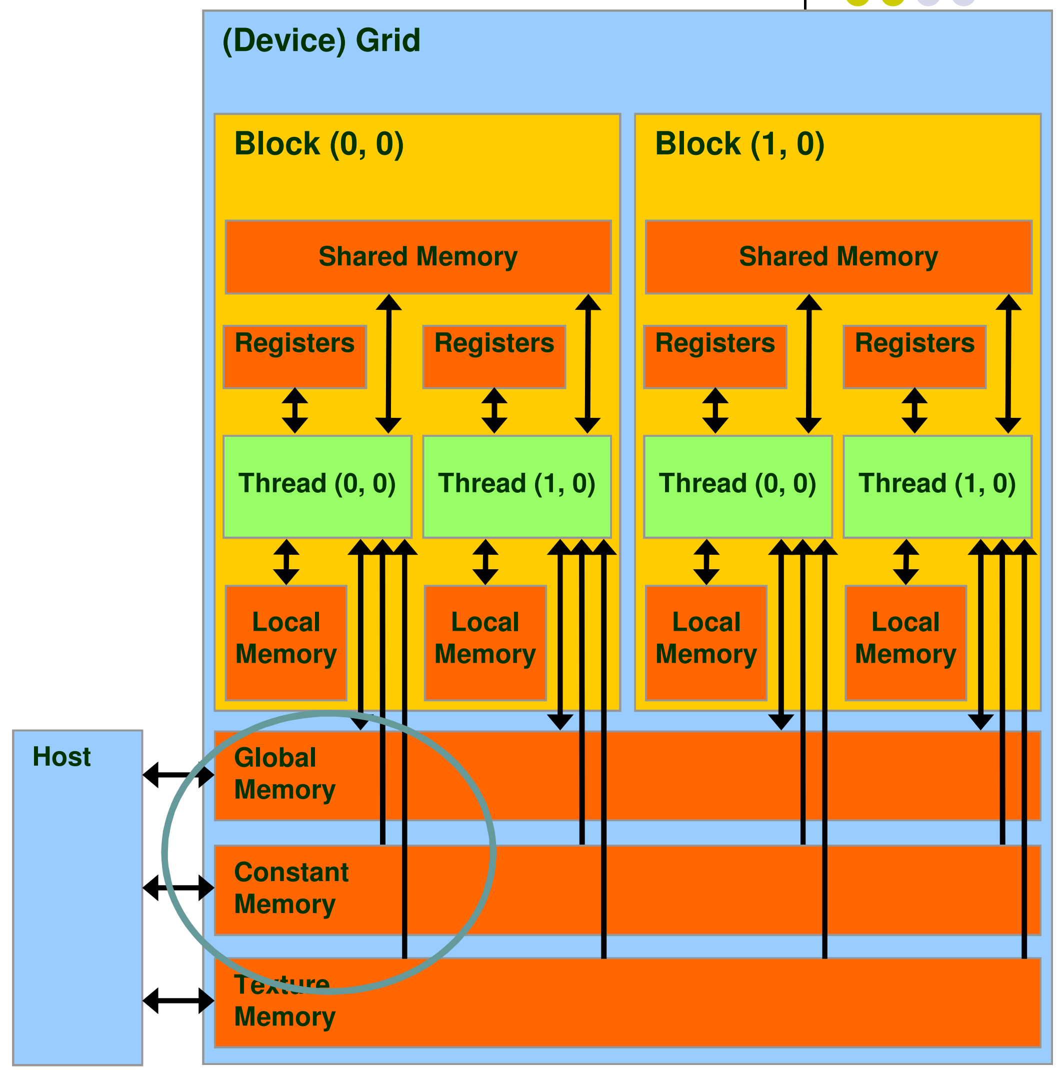
- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to **`Md.elements`**
  - "d" in "`Md`" is often used to indicate a device data structure

```
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

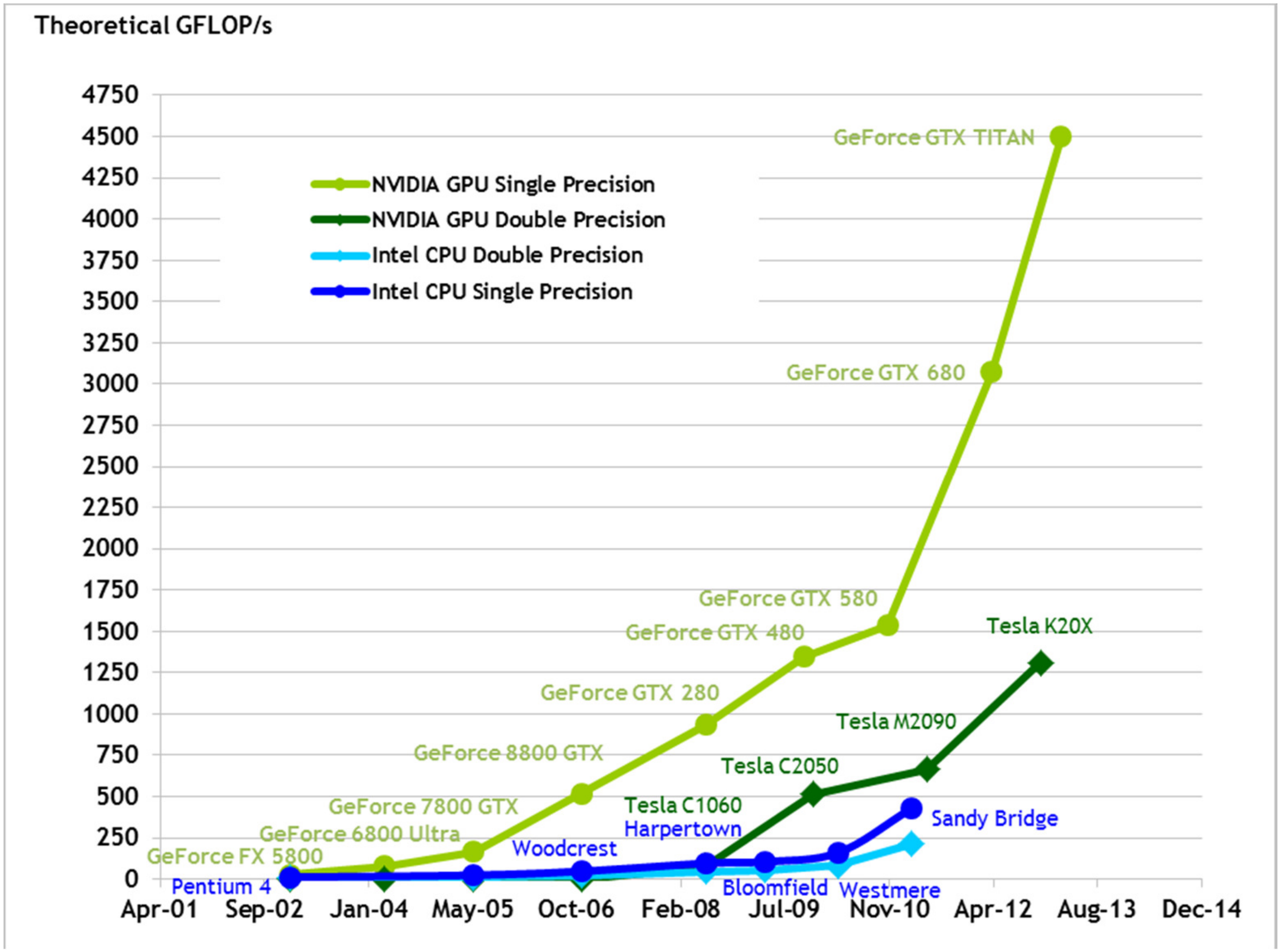
```
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);

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

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
```

CPU vs. GPU – Flop Rate (GFlops)

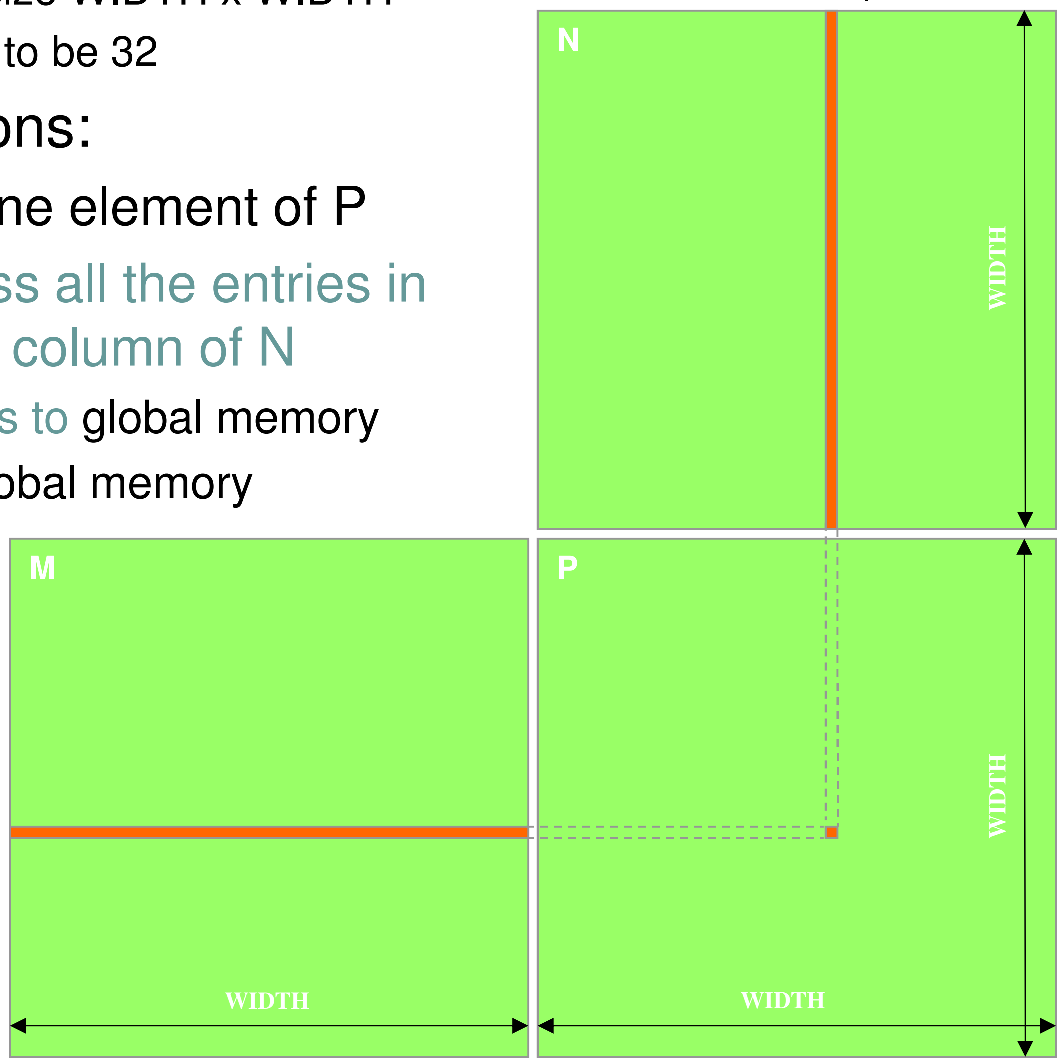


Simple Example: Matrix Multiplication

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Use only global memory (don't bring shared memory into picture yet)
  - Matrix will be of small dimension, job can be done using one block
  - Concentrate on
    - Thread ID usage
    - Memory data transfer API between host and device

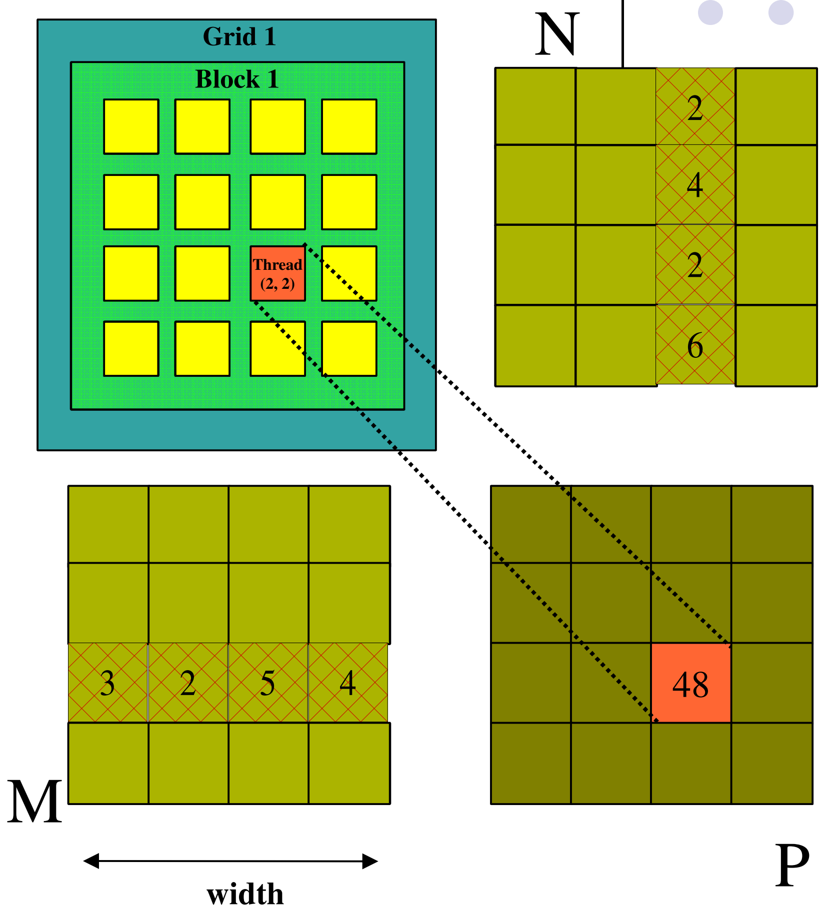
Square Matrix Multiplication Example

- 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 will access all the entries in one row of M and one column of N
    - 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: Traditional Approach, Coded in C

```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 sum = 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
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}
```

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

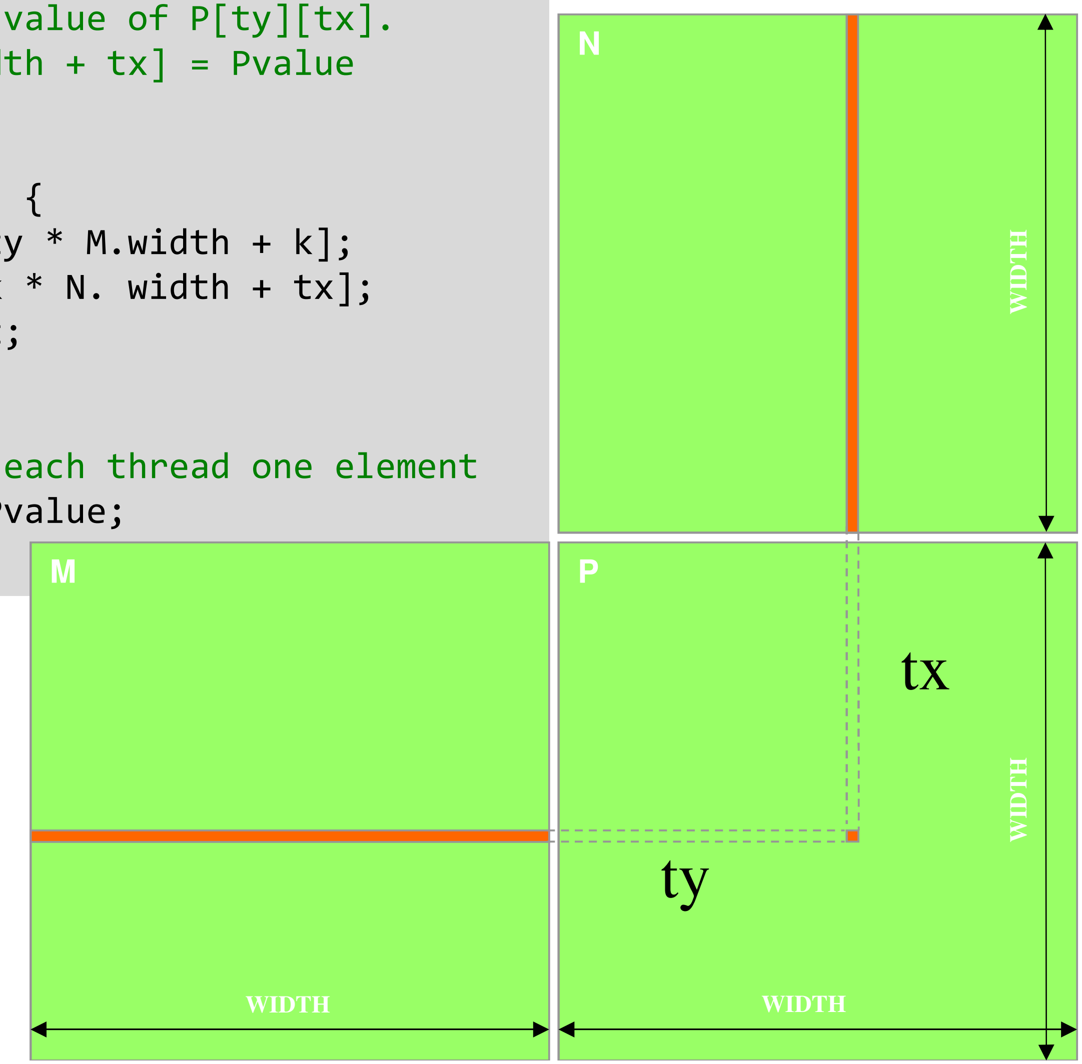
```
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1.  No: 0 (everything is set to zero)
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH,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

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;

    // Pvalue will end up storing the value of P[ty][tx].
    // That is, P.elements[ty * P. width + tx] = Pvalue
    float Pvalue = 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];
         Pvalue += Melement * Nelement;
    }

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



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);
}
```