

# CUDA Programming Introduction I

Nikos Hardavellas

Some slides/material from:  
UToronto course by Andreas Moshovos  
UIUC course by Wen-Mei Hwu and David Kirk  
UCSB course by Andrea Di Blas  
Universitat Jena by Waqar Saleem  
NVIDIA by Simon Green and many others

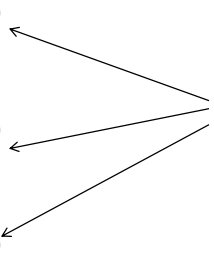
1

## GPU: bandwidth optimized – latencies are long

- A GPU ADD takes 24 GPU cycles
  - CPU ADD = 1 cycle (for a single core)
- The GPU cycle is ~4x of a CPU cycle
  - GTX680 vs. 3.8GHz Core i7
- In the time it takes to do 1 ADD in the GPU
  - CPU does ~100 ADDs (single)
  - **Need ~100 threads to break even with 1 CPU**
- 1000s of threads for GPU to be better

2

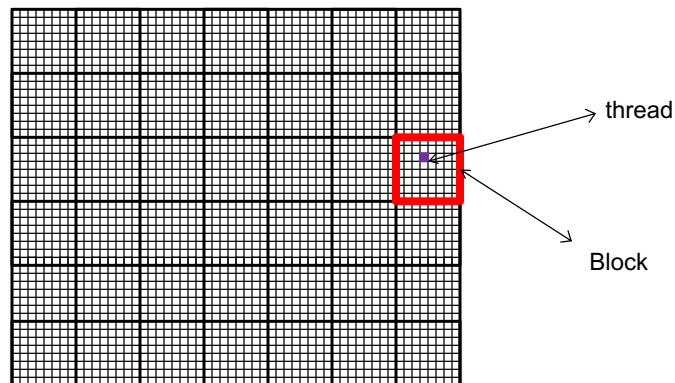
## Computation partitioning:

- At the highest level:
    - Think of computation as a series of loops:
      - for (i = 0; i < big\_number; i++)
        - a[i] = some function
      - for (i = 0; i < big\_number; i++)
        - a[i] = some other function
      - for (i = 0; i < big\_number; i++)
        - a[i] = some other function
- Kernels
- 

3

## Per Kernel Computation Partitioning

- Computation Grid: 2D Case

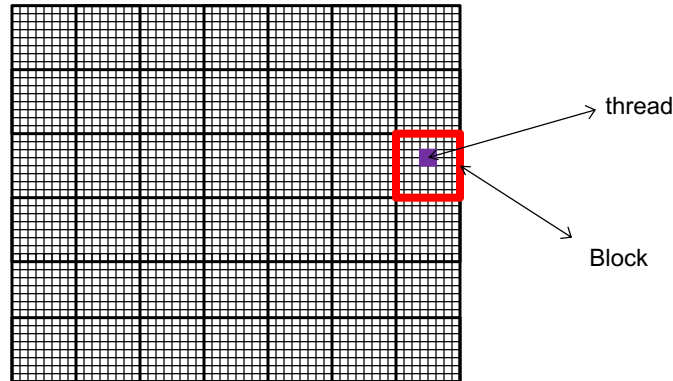


- Threads within a block can communicate/synchronize
  - Run on the same multiprocessor
- Threads across blocks can't communicate
  - Shouldn't touch each others data
  - Behavior undefined

4

## Per Kernel Computation Partitioning

- Computation Grid: 2D Case

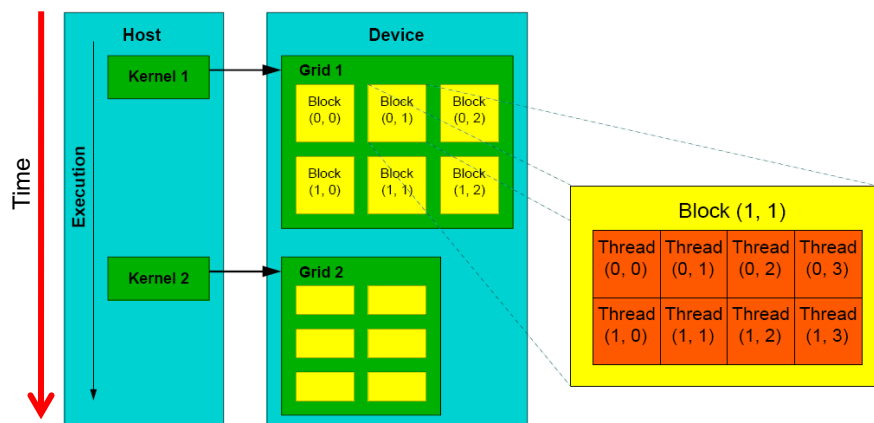


- One thread can process multiple data elements
- Other mappings are possible and often desirable
  - More on this when we talk about how to optimize for performance

5

## GBT: Grids of Blocks of Threads

Programmers view of data and computation partitioning



Why? Realities of integrated circuits:

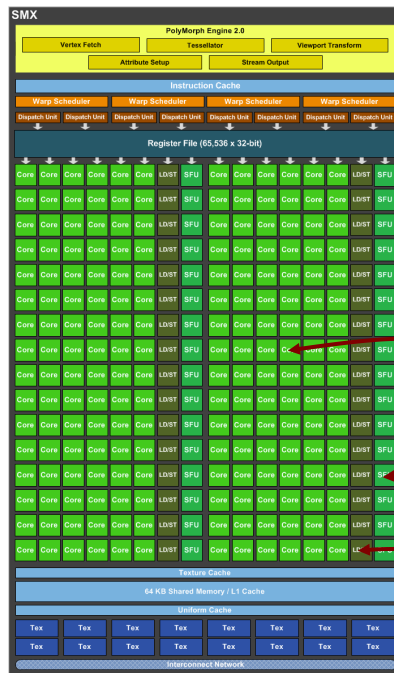
need to cluster computation and storage to achieve high speeds

Philosophy is:

We'll tell you about the hardware – you figure out how to make the best of it

6

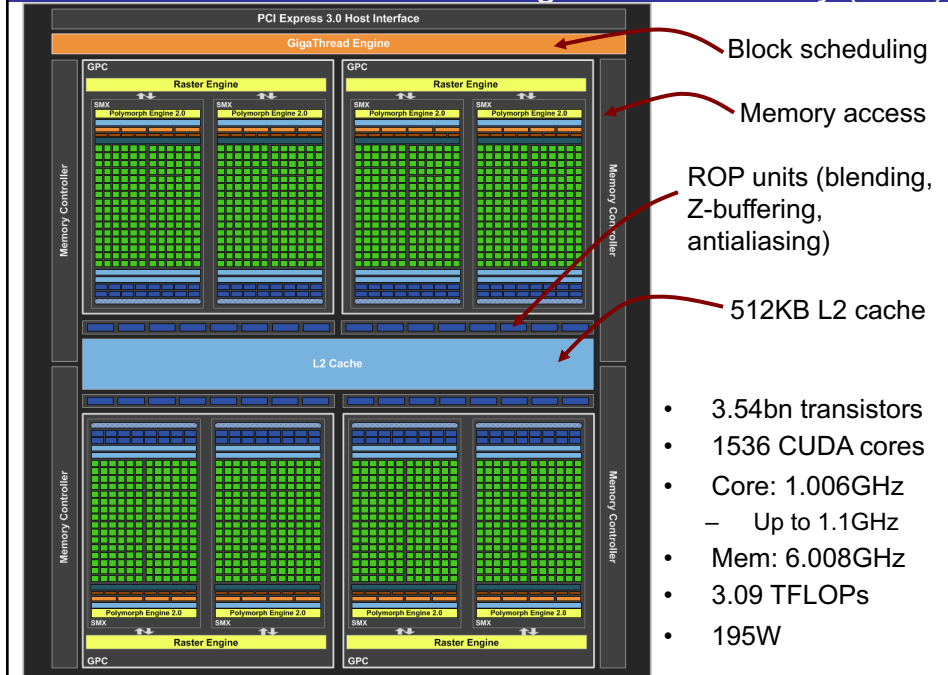
## GeForce GTX 680 – Streaming Multiprocessor



- **SM (a.k.a. SMX, SMP)**
  - Streaming Multiprocessor
  - Multi-threaded processor
    - 192 CUDA cores
    - 1 to 2048 threads active
  - Shared instruction fetch per 32 threads (warp)
  - Fundamental processing unit for CUDA thread block
- **SP (a.k.a. CUDA core)**
  - Streaming Processor
  - Scalar ALU for a single CUDA thread
- **SFU**
  - Special function unit
- **LDST**
  - Memory access unit

7

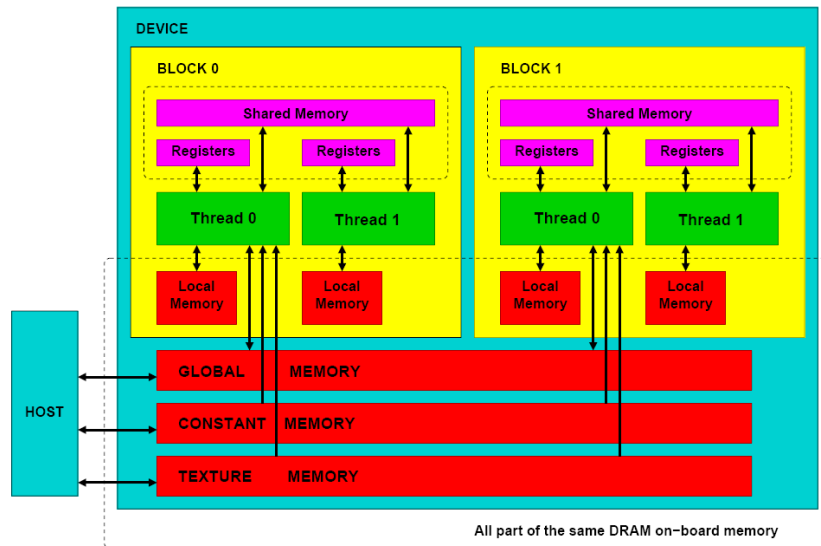
## GeForce GTX 680 – Streaming Processor Array (SPA)



- 3.54bn transistors
- 1536 CUDA cores
- Core: 1.006GHz
  - Up to 1.1GHz
- Mem: 6.008GHz
- 3.09 TFLOPs
- 195W

8

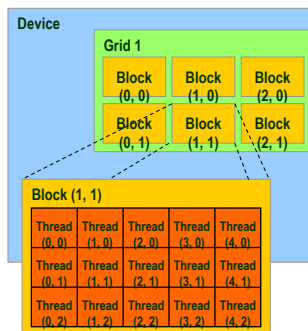
## Programmer's view: Memory Model



9

## Grids of Blocks of Threads: Dimension Limits (GTX680)

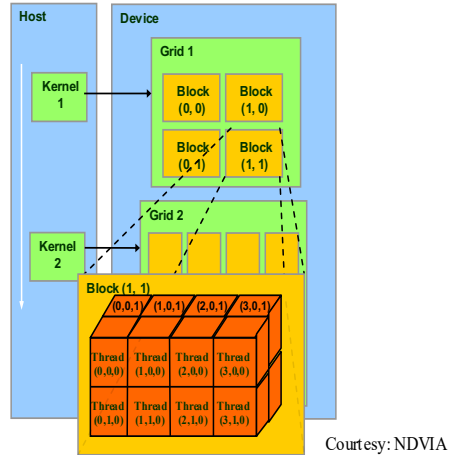
- Grid of Blocks **1D, 2D, or 3D**
  - Max grid dimensions:  
**2,147,483,647 x 65,535 x 65,535**
- Block of Threads: **1D, 2D, or 3D**
  - Max number of threads: **1024**
  - Max block dimension:  
**1024 x 1024 x 64**
- Limits apply to Compute Capability
  - **GTX680 = 3.0**
  - GTX280 = 1.3



10

## Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D, 2D, or 3D
  - Thread ID: 1D, 2D, or 3D
  - Combination is unique**
- Simplifies memory addressing when processing multidimensional data*
  - Convenience, not necessity**
- IDs and dimensions are accessible through predefined “variables”, e.g., `blockDim.x` and `threadIdx.x`

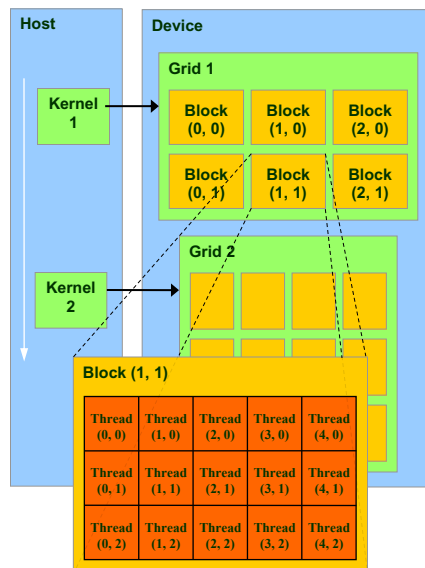


Courtesy: NDVIA

11

## Thread Batching

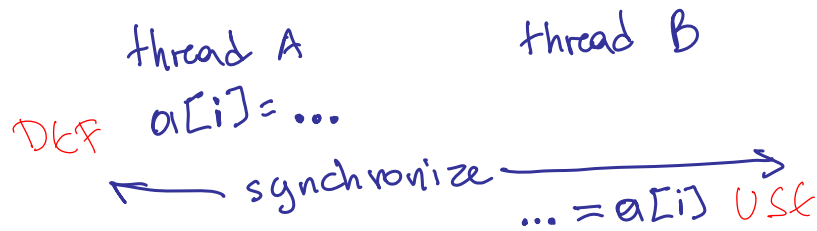
- A kernel is executed as a **grid of thread blocks**
  - All thread blocks **share** the same data memory space
    - But **cannot** communicate through it
- A **thread block**:
  - Threads that can cooperate with each other by:
    - Synchronizing their execution, for hazard-free shared memory accesses
    - Efficiently sharing data through a low latency **shared memory**
- Two threads from two different blocks cannot cooperate



12

## Thread Coordination Overview

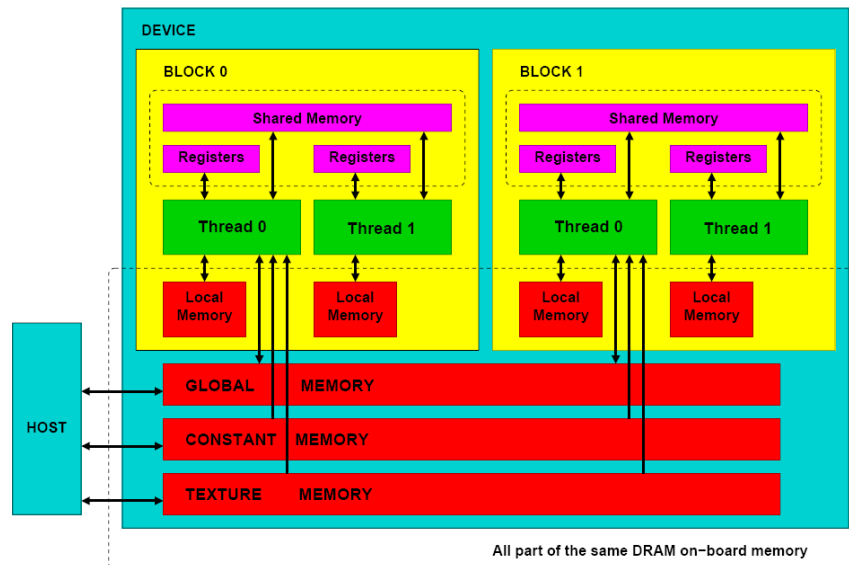
- Race-free access to data



Only across threads within the same block  
No communication across blocks

13

## Programmer's view: Memory Model: Thread vs. Host



Arrows show whether read and/or write is possible

14

#### Programmer's View: Memory Detail – Thread and Host

- Each thread can:
  - R/W per-thread **registers**
  - R/W per-thread **local memory**
  - R/W per-block **shared memory**
  - R/W per-grid **global memory**
  - Read only per-grid **constant memory**
  - Read only per-grid **texture memory**
- The host can R/W:
  - **global**, **constant**, and **texture** memories

15

#### Memory Model: Global, Constant, and Texture Memories

- **Global memory**
  - Main means of communicating R/W Data between **host** and **device**
  - Contents visible to all threads
  - Little locality – 3D graphics origin
- **Texture and Constant Memories**
  - Constants initialized by host
  - Contents visible to all threads

16



## Memory Model Summary

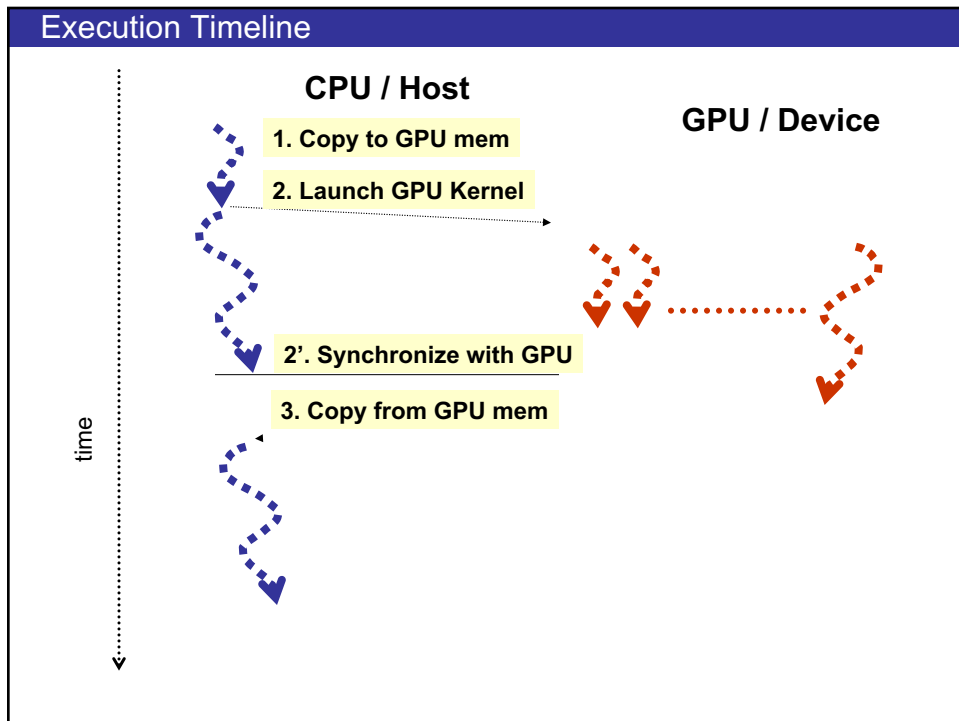
Memory	Location	Access	Scope
Local	off-chip	R/W	thread
Shared	on-chip	R/W	all threads in a block
Global	off-chip	R/W	all threads + host
Constant	off-chip	RO	all threads + host
Texture	off-chip	RO	all threads + host

17

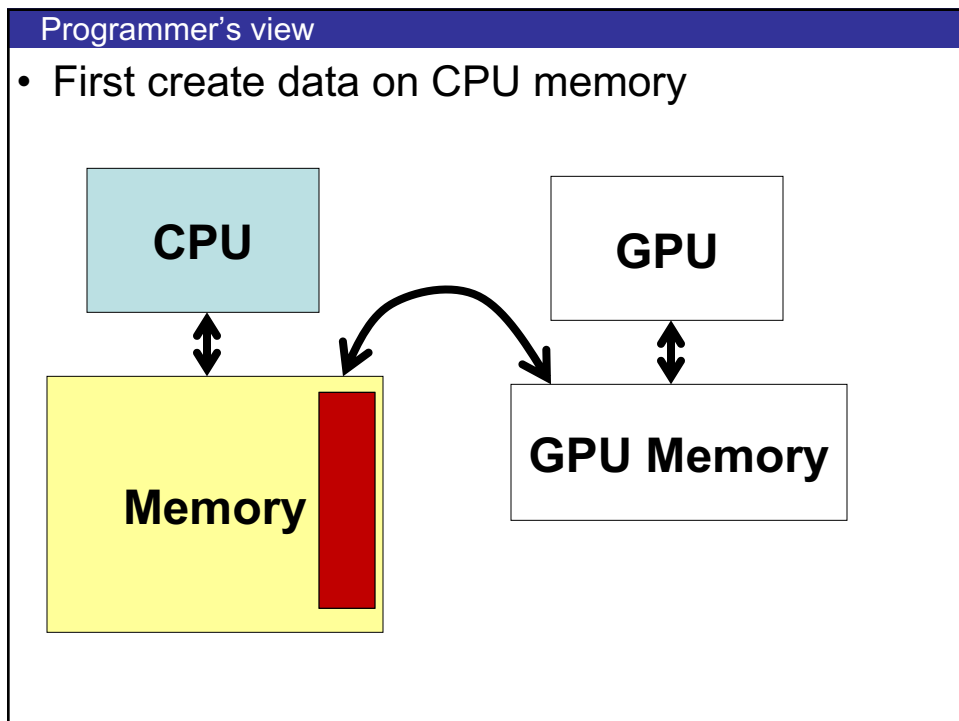
## Execution Model: Ordering

- Execution order is **undefined**
- Do not assume nor use:
  - block 0 executes before block 1
  - Thread 10 executes before thread 20
  - And **any** other ordering even if you can observe it
- Future implementations may break this ordering
- It's not part of the CUDA definition
- Why? More flexible hardware options

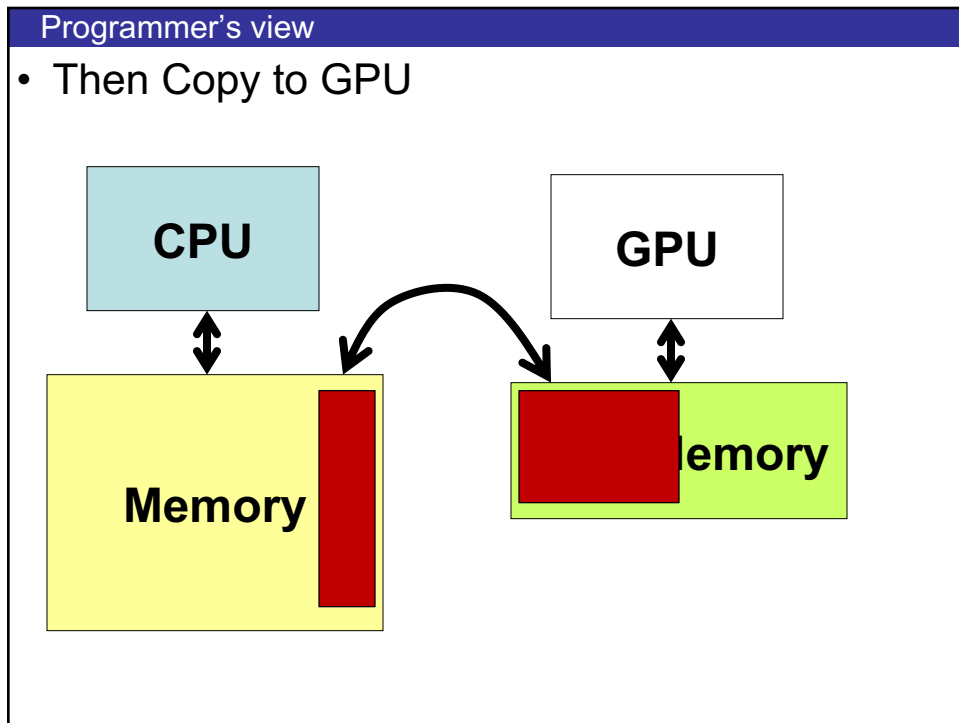
18



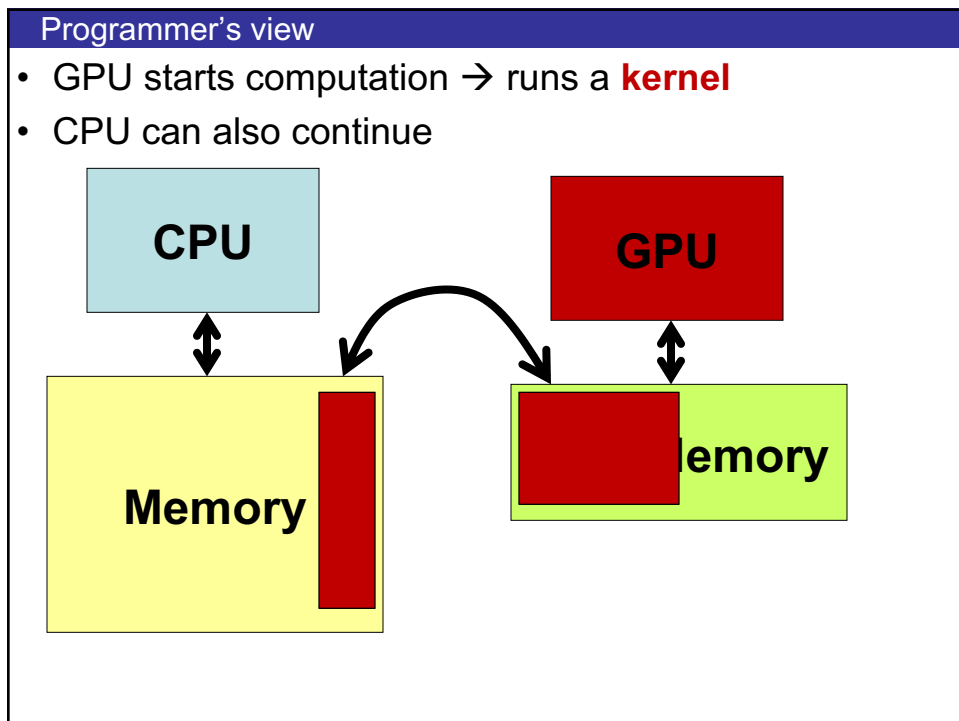
19



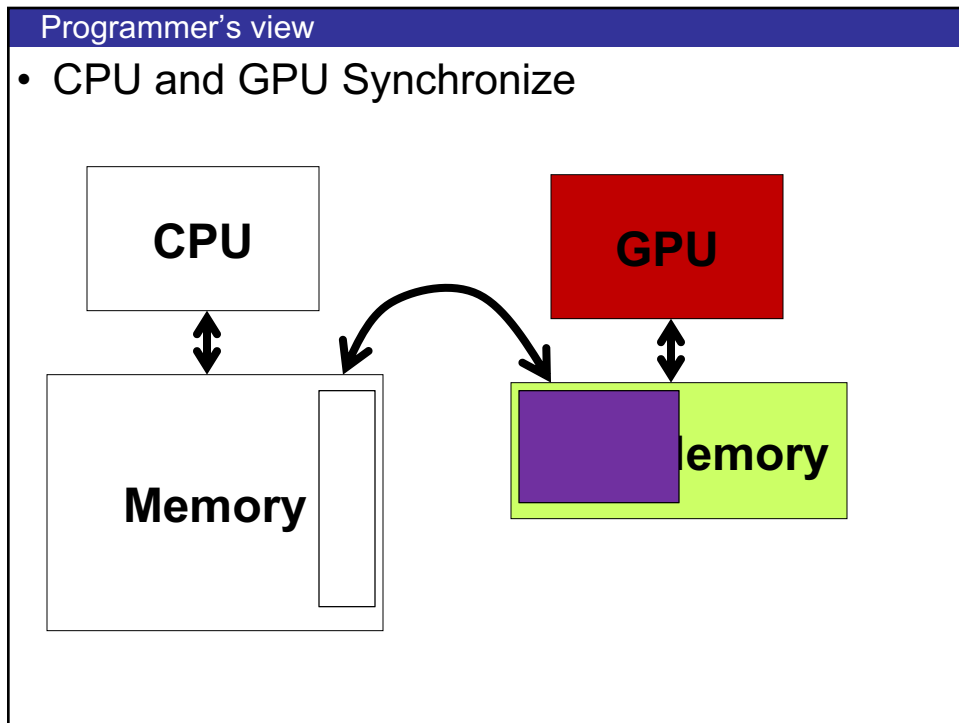
20



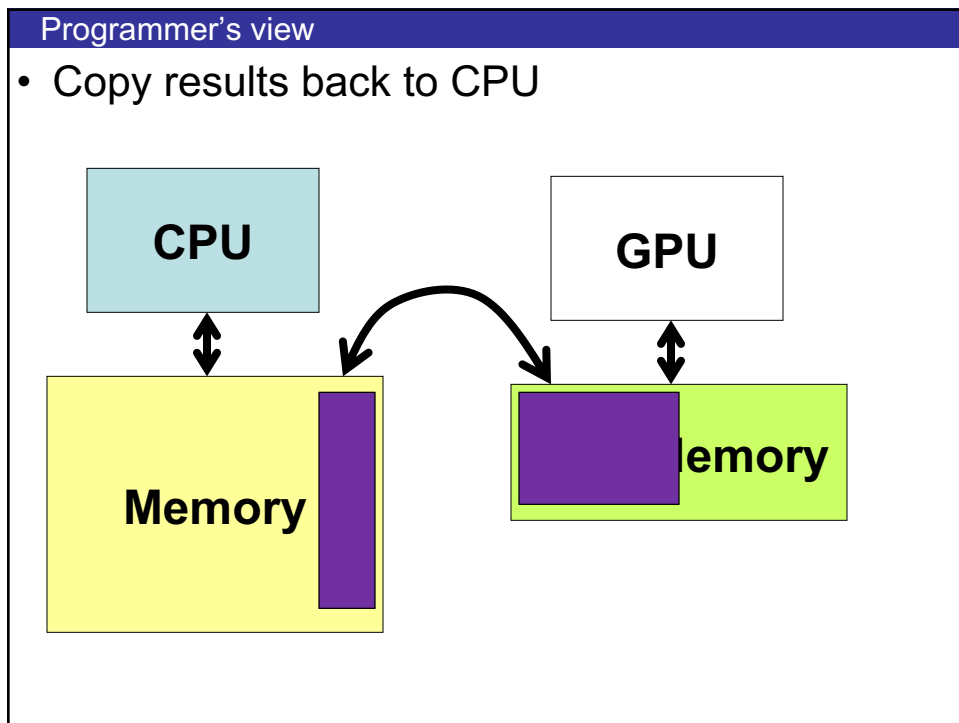
21



22



23



24

### Reasoning about CUDA call ordering

- GPU communication via `cuda...()` calls and kernel invocations
  - `cudaMalloc`, `cudaMemcpy`
- Asynchronous from the CPU's perspective
  - CPU places a request in a "CUDA" queue
  - requests are handled in-order
- Streams allow for multiple queues
  - Order within each queue honored
  - No order across queues
  - More on this much later on

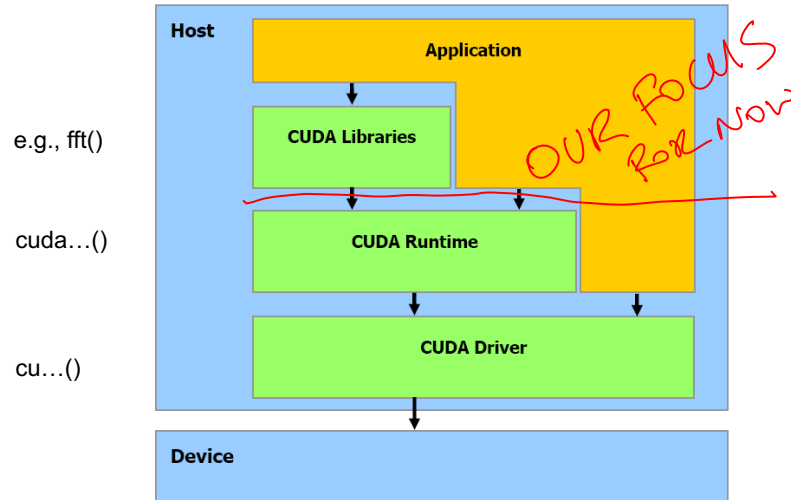
25

### Execution Model Summary (for your reference)

- **Grid of blocks of threads**
  - 1D/2D/3D grid of blocks (only 1D/2D in earlier versions)
  - 1D/2D/3D blocks of threads
- **All blocks are identical:**
  - same structure and # of threads
- **Block execution order is undefined**
- Same block threads:
  - can synchronize and share data fast (shared memory)
- Threads from different blocks:
  - Cannot cooperate
  - Communication through global memory
- Blocks do not migrate: execute on the same processor
- Several blocks may run over the same processor
- **Threads and Blocks have IDs**
  - Simplifies data indexing
  - Can be 1D, 2D, or 3D (threads)

26

## CUDA Software Architecture



27

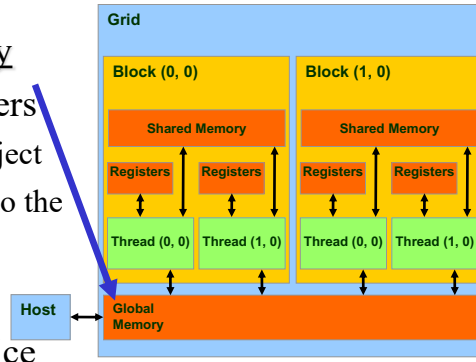
## CUDA API Highlights: Easy and Lightweight

- The API is an **extension to the ANSI C programming language**
  - ➡ Low learning curve
- The hardware is **designed to enable lightweight runtime and driver**
  - ➡ High performance

28

## CUDA Device Memory Allocation

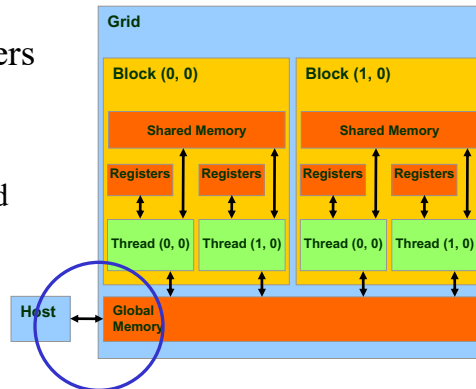
- `cudaMalloc()`
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Size of** allocated object
    - **Address of a pointer** to the allocated object
- `cudaFree()`
  - Frees object from device Global Memory
    - Pointer to freed object



29

## 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



30

### CUDA API: Example

```
int a[N];  
    for (i =0; i < N; i++)  
        a[i] = a[i] + x;
```

1. Allocate CPU Data Structure
2. Initialize Data on CPU
3. Allocate GPU Data Structure
4. Copy Data from CPU to GPU
5. Define *Execution Configuration*
6. Run Kernel
7. CPU synchronizes with GPU
8. Copy Data from GPU to CPU
9. De-allocate GPU and CPU memory

31

### My first CUDA Program / Skeleton

```
__global__ void arradd (float *a, float f, int N)  
{  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    if (i < N) a[i] = a[i] + f;  
}
```

**GPU**

```
int main()  
{  
    float h_a[N]; // 1. allocate cpu container  
    for (int i=0; i < N; i++) h_a[i] = (float) i; // 2. initialize
```

**CPU**

```
    float *d_a;  
    cudaMalloc ((void **) &d_a, SIZE); // 3. allocate GPU data structure
```

```
    cudaMemcpy (d_a, h_a, SIZE, cudaMemcpyHostToDevice)); // 4. copy cpu → gpu
```

```
    arradd <<< n_blocks, block_size >>> (d_a, 10.0, N); // 5 & 6. exec config & run
```

```
    cudaThreadSynchronize (); // 7. cpu & gpu synchronize
```

```
    cudaMemcpy (h_a, d_a, SIZE, cudaMemcpyDeviceToHost)); // 8. copy from gpu
```

```
    CUDA_SAFE_CALL (cudaFree (d_a)); // 9. de-allocate memory  
}
```

32



## 1. Allocate CPU Data container

```
float *ha;

main (int argc, char *argv[])
{
    int N = atoi (argv[1]);
    ha = (float *) malloc (sizeof (float) * N);

    ...
}
```

### No memory allocated on the GPU side

- Pinned memory allocation results in faster CPU to/from GPU copies
- But pinned memory cannot be **paged-out**
  - cudaMallocHost (...)

33

## 2. Initialize CPU Data (dummy)

```
float *ha;

int i;

for (i = 0; i < N; i++)
    ha[i] = i;
```

34

### 3. Allocate GPU Data container

```
float *da;
```

```
cudaMalloc ((void **) &da, sizeof (float) * N);
```

- Notice: no assignment side
  - NOT: `da = cudaMalloc (...)`
- Assignment is done internally:
  - That's why we pass `&da`
- Space is allocated in **Global Memory** on the GPU

35

### GPU Memory Allocation

- The host manages GPU memory allocation:
  - **cudaMalloc** (**void \*\*ptr, size\_t nbytes**)
    - Must explicitly cast to (**void \*\***)
      - `cudaMalloc ((void **) &da, sizeof (float) * N);`
  - **cudaFree** (**void \*ptr**);
    - `cudaFree (da);`
  - **cudaMemset** (**void \*ptr, int value, size\_t nbytes**);
    - `cudaMemset (da, 0, N * sizeof (int));`
- Check the **CUDA Reference Manual**

36

#### 4. Copy Initialized CPU data to GPU

```
float *da;  
float *ha;  
  
cudaMemcpy ((void *) da,           // DESTINATION  
            (void *) ha,           // SOURCE  
            sizeof (float) * N,    // #bytes  
            cudaMemcpyHostToDevice); // DIRECTION
```

37

#### Host/Device Data Transfers

- The host initiates all transfers:
- **cudaMemcpy**(void \*dst, void \*src,  
size\_t nbytes,  
enum cudaMemcpyKind direction)
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
  - cudaMemcpyHostToHost

38

## 5. Define Execution Configuration

- How many blocks and threads/block

```
// N = num elements
int threads_block = 64;

int blocks = N / threads_block;
if (blocks % N != 0)
    blocks ++;
```

- Alternatively:

```
blocks = (N + threads_block - 1) /
    threads_block;
```

39

## 6. Launch Kernel & 7. CPU/GPU Synchronization

- Instructs the GPU to launch `blocks x threads_block` threads:

```
darradd <<<blocks, threads_block>>>(da, 10f, N);
cudaThreadSynchronize (); // forces CPU to wait
```

- `darradd`: kernel name
- `<<< ... >>>` execution configuration
- `(da, x, N)`: arguments
  - Parameters are passed to the device via constant memory
  - Limited to 4 KB
  - No variable number of arguments

40

## Launch a Kernel with Multidimensional Blocks

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

```
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 >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

41

## CPU/GPU Synchronization

- CPU does not block on cuda...() calls
  - Kernel/requests are queued and processed in-order
  - Control returns to CPU immediately
- Good if there is other work to be done
  - e.g., preparing for the next kernel invocation
- Eventually, CPU must know when GPU is done
- Then it can safely copy the GPU results
- `cudaThreadSynchronize ()`
  - Block CPU until **all** preceding cuda...() and kernel requests have completed

42

## 8. Copy data from GPU to CPU & 9. DeAllocate Memory

```
float *da;
float *ha;

cudaMemCpy ((void *) ha,           // DESTINATION
            (void *) da,           // SOURCE
            sizeof (float) * N,     // #bytes
            cudaMemcpyDeviceToHost); // DIRECTION

cudaFree (da);

// display or process results here

free (ha);
```

43

## The GPU Kernel

```
__global__ darradd (float *da, float x, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

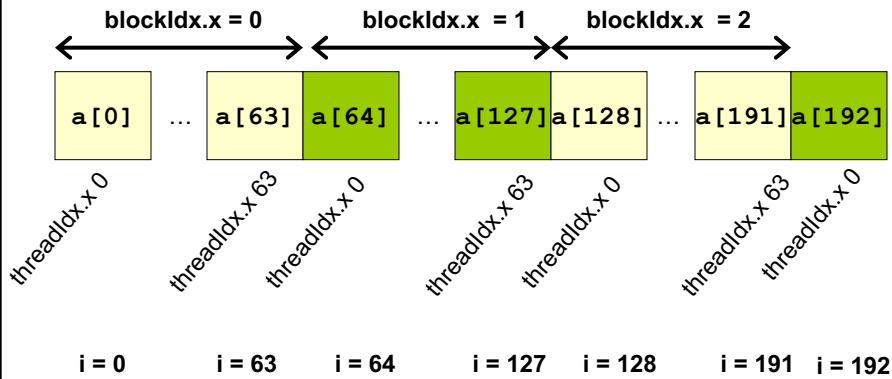
    if (i < N) da[i] = da[i] + x;
}
```

- **BlockIdx:** Unique Block ID.
  - Numerically ascending: 0, 1, ...
- **BlockDim:** Dimensions of Block = how many threads it has
  - BlockDim.x, BlockDim.y, BlockDim.z
  - Unused dimensions default to 0
- **ThreadIdx:** Unique per Block Index
  - 0, 1, ...
  - Per Block

44

### Array Index Calculation Example

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```



Assuming blockDim.x = 64

45

### Generic Unique Thread and Block Index Calculations #1

- **1D Grid / 1D Blocks:**

```
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x +
threadIdx.x;
```

- **1D Grid / 2D Blocks:**

```
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y +
threadIdx.y * blockDim.x + threadIdx.x;
```

- **1D Grid / 3D Blocks:**

```
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y *
blockDim.z + threadIdx.z * blockDim.y * blockDim.x +
threadIdx.y * blockDim.x + threadIdx.x;
```

- Source: <http://forums.nvidia.com/lofiversion/index.php?t82040.html>

46

## Generic Unique Thread and Block Index Calculations #2

- **2D Grid / 1D Blocks:**

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.x +
threadIdx.x;
```

- **2D Grid / 2D Blocks:**

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.y *
blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```

- **2D Grid / 3D Blocks:**

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.z *
blockDim.y * blockDim.x + threadIdx.z * blockDim.y *
blockDim.z + threadIdx.y * blockDim.x + threadIdx.x;
```

- **UniqueThreadIndex means unique per grid.**

47

## CUDA Function Declarations

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

- **\_\_global\_\_** defines a kernel function
  - Must return void
  - Can only call **\_\_device\_\_** functions
- **\_\_device\_\_** and **\_\_host\_\_** can be used together
  - Both versions of the code generated

48



### \_\_device\_\_ Example

- Add x to a[i] multiple times

```
__device__ float addmany (float a, float b, int count)
{
    while (count-->0) a += b;
    return a;
}

__global__ void darradd (float *da, float x, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N) da[i] = addmany (da[i], x, 10);
}
```

49

### Kernel and Device Function Restrictions

- **\_\_device\_\_** functions cannot have their address taken in host
  - e.g., `f = &addmany; *f(...);`
- For functions executed on the device:
  - Recursion is fine on newer gpu devices
    - `darradd (...)`

```
{
    darradd (...);
}
```
    - **Added support for \_\_device\_\_ on Fermi (2.x capability)**
    - **\_\_global\_\_ functions do not support recursion**
  - No static variable declarations inside the **\_\_device\_\_** function
    - `darradd (...)`

```
{
    static int cannot_have_this;
}
```
    - Actually you can, but with complex rules; see CUDA C++ Programming Guide
  - No variable number of arguments before 2.0
    - e.g., something like `printf (...)`
    - **`printf()` supported for \_\_global\_\_ & \_\_device\_\_ on capab. ≥ 2.0**

50

## My first CUDA Program / Skeleton

```
__global__ void arradd (float *a, float f, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) a[i] = a[i] + f;
}
```

**GPU**

```
int main()
{
    float h_a[N]; // 1. allocate cpu container
    for (int i=0; i < N; i++) h_a[i] = (float) i; // 2. initialize
```

**CPU**

```
    float *d_a;
    cudaMalloc ((void **) &d_a, SIZE); // 3. allocate GPU data structure
```

```
    cudaMemcpy (d_a, h_a, SIZE, cudaMemcpyHostToDevice); // 4. copy cpu → gpu
```

```
    arradd <<< n_blocks, block_size >>> (d_a, 10.0, N); // 5 & 6. exec config & run
```

```
    cudaThreadSynchronize (); // 7. cpu & gpu synchronize
```

```
    cudaMemcpy (h_a, d_a, SIZE, cudaMemcpyDeviceToHost); // 8. copy from gpu
```

```
    CUDA_SAFE_CALL (cudaFree (d_a)); // 9. de-allocate memory
}
```

51

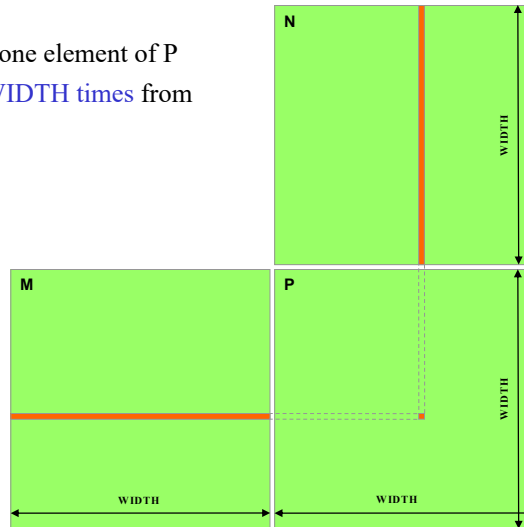
## A Simple Running Example: Matrix Multiplication

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

52

## Programming Model: Square Matrix Mult Example

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



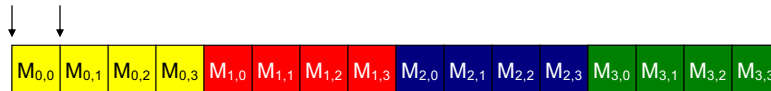
53

## Memory Layout of a Matrix in C

P:

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

M     $M + \text{sizeof}(M_{0,0})$

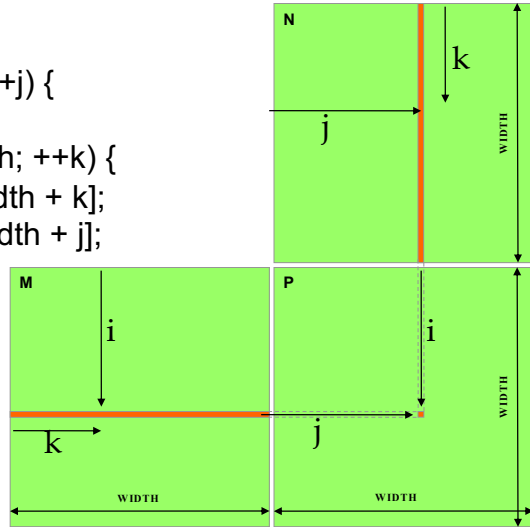


$P[i, j] = \text{Memory}[M + i * \text{width} + j]$   
 e.g.,  $P[2, 1] = \text{Memory}[M + 2 * 4 + 1]$

54

### Step 1: Matrix Mult; A Simple Host Version in C

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



55

### 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);
```

56

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

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

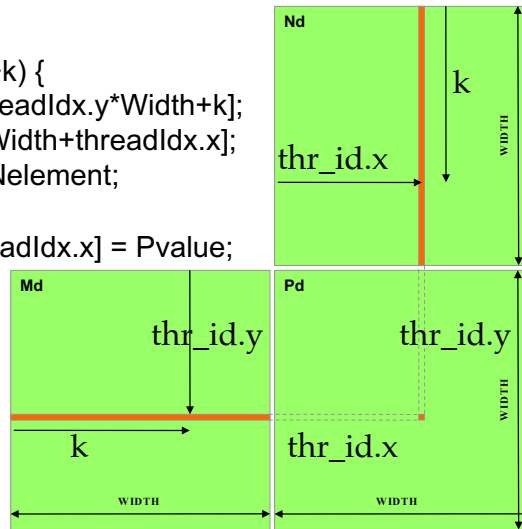
3. // Read P from the device
   cudaMemcpy(P,
             Pd,
             size,
             cudaMemcpyDeviceToHost);

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

57

### Step 4: Kernel Function (cont.)

```
// Matrix multiplication kernel – per thread code
__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;
}
```



58

## Step 5: Kernel Invocation (Host-side Code)

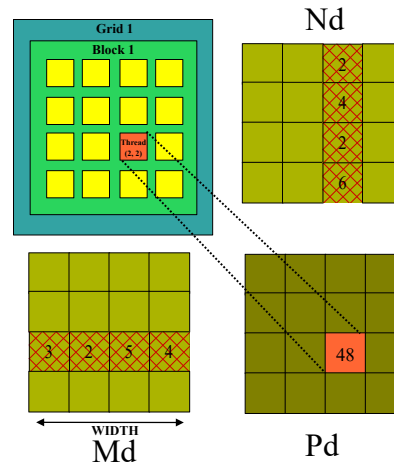
```
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(Width, Width);

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

59

## 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
  - Computation to off-chip memory access ratio close to 2:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



60