

CS 380 - GPU and GPGPU Programming

Lecture 13: GPU Compute APIs 2

Markus Hadwiger, KAUST

Reading Assignment #7 (until Oct 19)



Read (required):

- Read https://en.wikipedia.org/wiki/Instruction_pipelining
- CUDA NVCC doc (CUDA SDK: `CUDA_Compiler_Driver_NVCC.pdf`)
Read Chapters 1 – 3; Chapter 5; get an overview of the rest
- PTX Instruction Set Architecture 7.1 in CUDA SDK (`ptx_isa_7.1.pdf`)
Read Chapters 1 – 3; get an overview of Chapter 12;
browse through the other chapters to get a feeling for what PTX looks like
- Look at CUDA SASS in CUDA SDK: `CUDA_Binary_Uilities.pdf`, Chapter 4

Read (optional):

- Inline PTX Assembly in CUDA (CUDA SDK: `Inline_PTX_Assembly.pdf`)
- Dissecting GPU Architecture through Microbenchmarking:

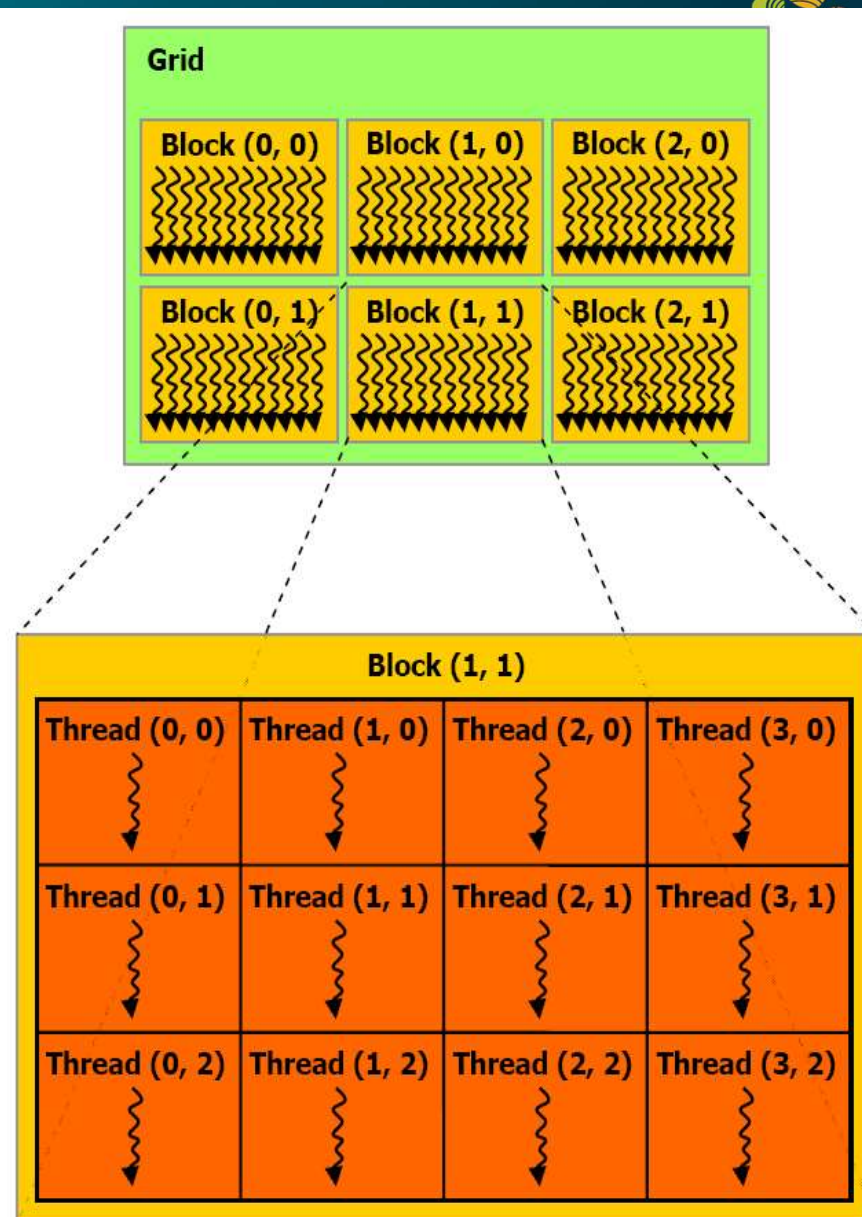
Volta: <https://arxiv.org/abs/1804.06826>

Turing: <https://arxiv.org/abs/1903.07486>

<https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9839-discovering-the-turing-t4-gpu-architecture-with-microbenchmarks.pdf>

CUDA Multi-Threading

- CUDA model groups threads into blocks; blocks into grid
- Execution on actual hardware:
 - Block assigned to SM (up to 8, 16, or 32 blocks per SM; depending on compute capability)
 - 32 threads grouped into warp



Execution Model

Software



Thread



Thread Block



Grid

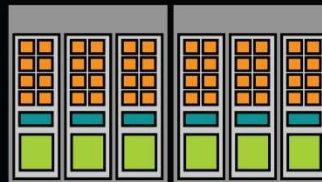
Hardware



Thread Processor



Multiprocessor



Device

Threads are executed by thread processors

Thread blocks are executed on multiprocessors

Thread blocks do not migrate

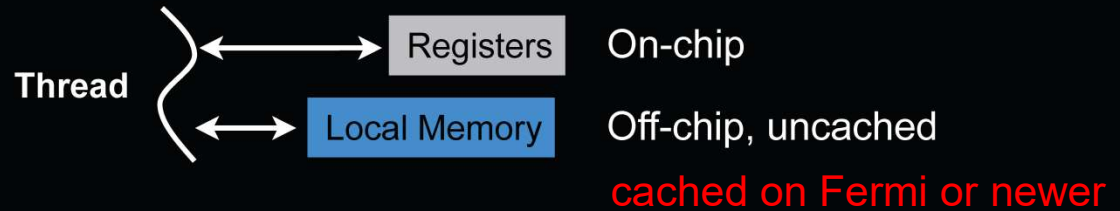
Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

A kernel is launched as a grid of thread blocks

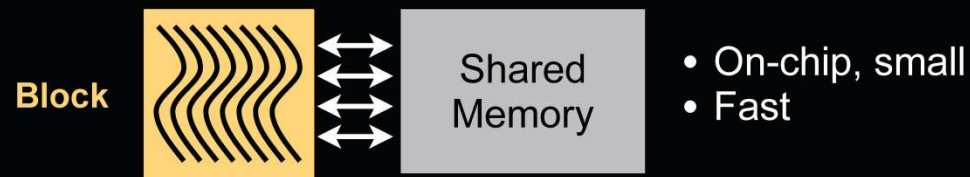
Only one kernel can execute on a device at one time

Kernel Memory Access

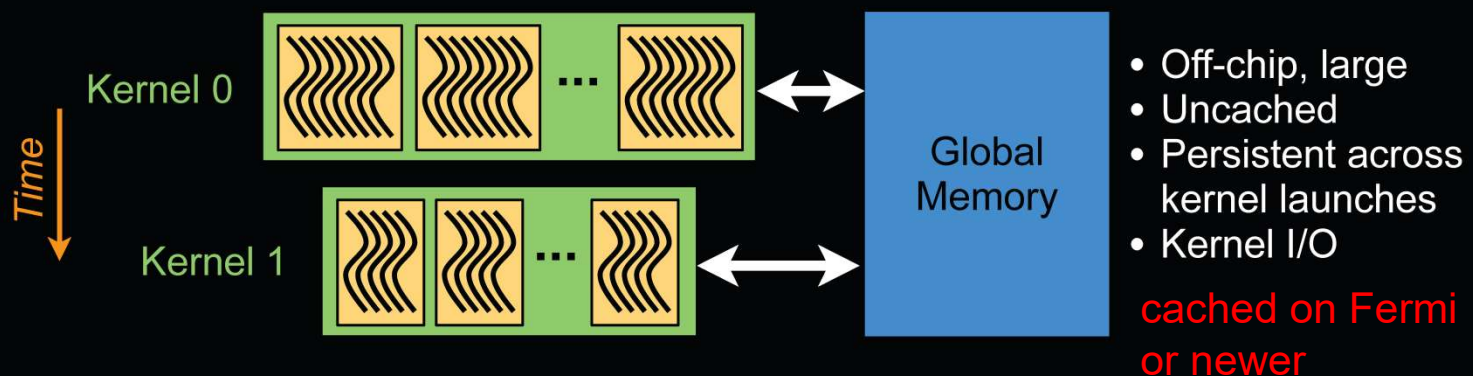
● Per-thread



● Per-block



● Per-device



Memory Architecture



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No *	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No *	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

* cached on Fermi or newer

(Memory) State Spaces



PTX ISA 7.1 (Chapter 5)

Name	Addressable	Initializable	Access	Sharing
<code>.reg</code>	No	No	R/W	per-thread
<code>.sreg</code>	No	No	RO	per-CTA
<code>.const</code>	Yes	Yes ¹	RO	per-grid
<code>.global</code>	Yes	Yes ¹	R/W	Context
<code>.local</code>	Yes	No	R/W	per-thread
<code>.param</code> (as input to kernel)	Yes ²	No	RO	per-grid
<code>.param</code> (used in functions)	Restricted ³	No	R/W	per-thread
<code>.shared</code>	Yes	No	R/W	per-CTA
<code>.tex</code>	No ⁴	Yes, via driver	RO	Context

Notes:

¹ Variables in `.const` and `.global` state spaces are initialized to zero by default.

² Accessible only via the `ld.param` instruction. Address may be taken via `mov` instruction.

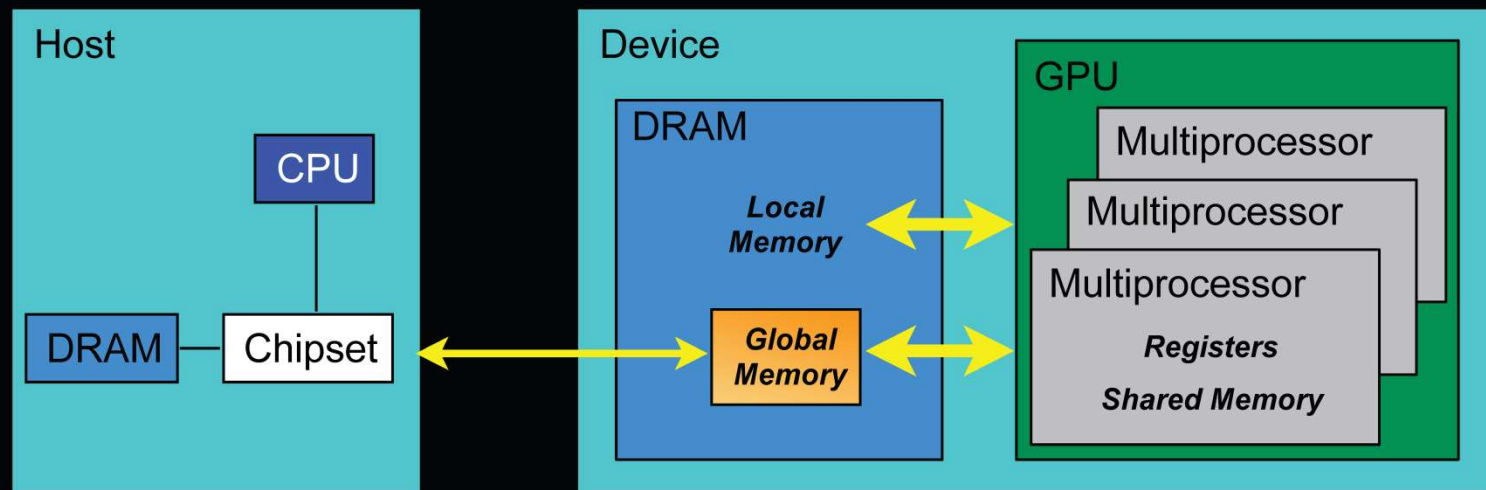
³ Accessible via `ld.param` and `st.param` instructions. Device function input and return parameters may have their address taken via `mov`; the parameter is then located on the stack frame and its address is in the `.local` state space.

⁴ Accessible only via the `tex` instruction.

Managing Memory

Unified memory space can be enabled on Fermi / CUDA 4.x and newer

- **CPU and GPU have separate memory spaces**
- **Host (CPU) code manages device (GPU) memory:**
 - Allocate / free
 - Copy data to and from device
 - Applies to *global* device memory (DRAM)



GPU Memory Allocation / Release

- `cudaMalloc(void ** pointer, size_t nbytes)`
- `cudaMemset(void * pointer, int value, size_t count)`
- `cudaFree(void* pointer)`

```
int n = 1024;  
int nbytes = 1024*sizeof(int);  
int *a_d = 0;  
cudaMalloc( (void**) &a_d,  nbytes );  
cudaMemset( a_d, 0, nbytes);  
cudaFree(a_d);
```


Data Copies

- **cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);**
 - **direction** specifies locations (host or device) of **src** and **dst**
 - Blocks CPU thread: returns after the copy is complete
 - Doesn't start copying until previous CUDA calls complete
- **enum cudaMemcpyKind**
 - **cudaMemcpyHostToDevice**
 - **cudaMemcpyDeviceToHost**
 - **cudaMemcpyDeviceToDevice**

Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Host

a_h

b_h

Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Host

a_h

b_h

Device

a_d

b_d

Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Host

a_h

b_h

Device

a_d

b_d

Data Movement Example

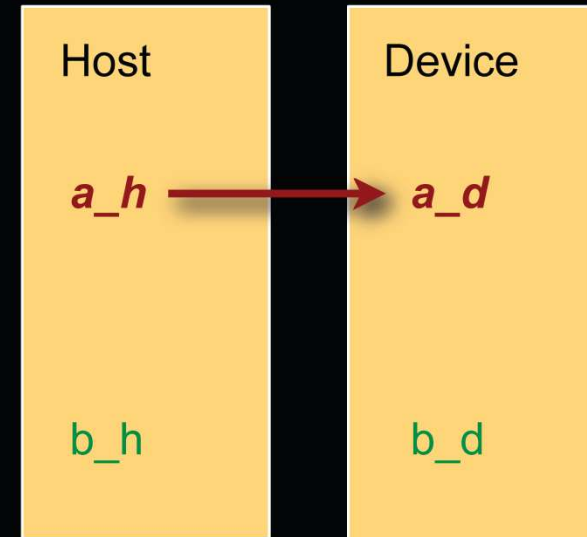
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```



Data Movement Example

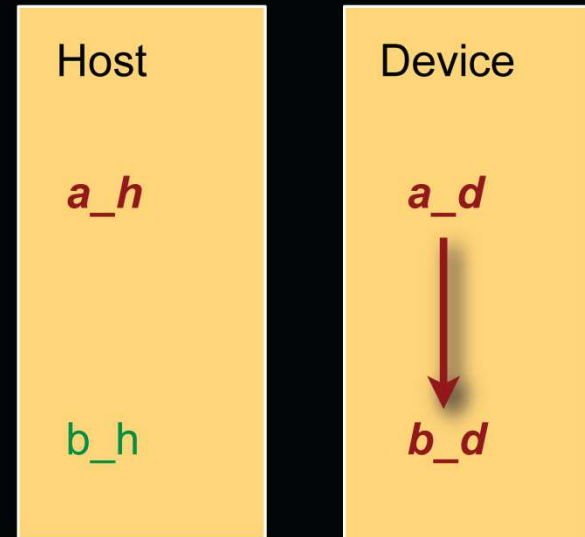
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```



Data Movement Example

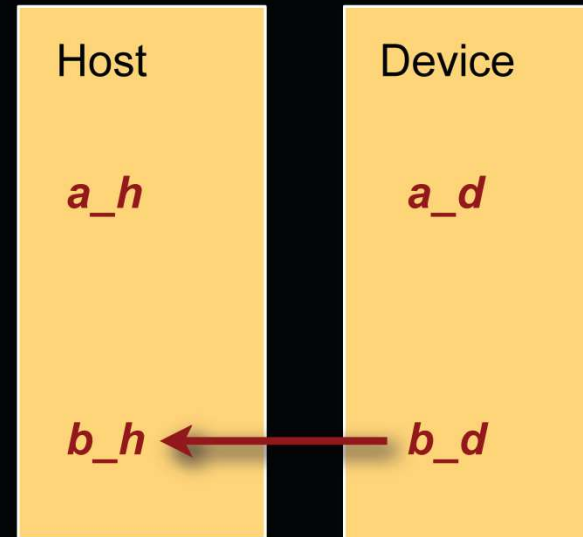
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```



Data Movement Example

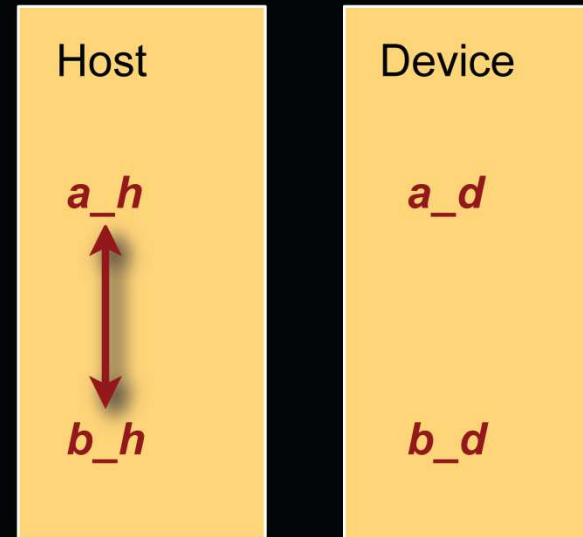
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```



Data Movement Example

```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Host

Device

Executing Code on the GPU


- **Kernels are C functions with some restrictions**

- Cannot access host memory **except: (*) and (**)**
- Must have **void** return type
- No variable number of arguments (“varargs”)
- **(Not recursive)** **recursion supported on `__device__` functions from cc. 2.x (i.e., basically on *all* current GPUs)**
- No static variables

- **Function arguments** automatically copied from host to device

(*) “unified memory programming” introduced with CUDA 6 (cc. 3.x +): allocate memory with `cudaMallocManaged()`; uses automatic migration

(**) also: mapped pinned (page-locked) memory (“zero-copy memory”) : allocate memory with `cudaMallocHost()`; beware of low performance!!

Note: UVA (“unified virtual addressing”; cc. 2.x +) is something different!! just pertains to unified pointers (see `cudaPointerGetAttributes()`, ...)  NVIDIA.

Function Qualifiers

- Kernels designated by function qualifier:
 - **__global__**
 - Function called from host and executed on device
 - Must return void
- Other CUDA function qualifiers
 - **__device__**
 - Function called from device and run on device
 - Cannot be called from host code
 - **__host__**
 - Function called from host and executed on host (default)
 - **__host__** and **__device__** qualifiers can be combined to generate both CPU and GPU code

Variable Qualifiers (GPU code)

- **__device__**
 - Stored in global memory (large, high latency, no cache)
 - Allocated with **cudaMalloc** (**__device__** qualifier implied)
 - Accessible by all threads
 - Lifetime: application
- **__shared__**
 - Stored in on-chip shared memory (very low latency)
 - Specified by execution configuration or at compile time
 - Accessible by all threads in the same thread block
 - Lifetime: thread block
- **Unqualified variables:**
 - Scalars and built-in vector types are stored in registers
 - What doesn't fit in registers spills to "local" memory

CUDA 6+: `__managed__` (with `__device__`) for managed memory (unified memory programming)

Launching Kernels

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(...)
```

- Execution Configuration (“<<< >>>”)

- **dG** - dimension and size of grid in blocks

- Two-dimensional: **x** and **y**
- Blocks launched in the grid: **dG.x * dG.y**

- **dB** - dimension and size of blocks in threads:

- Three-dimensional: **x**, **y**, and **z**
- Threads per block: **dB.x * dB.y * dB.z**

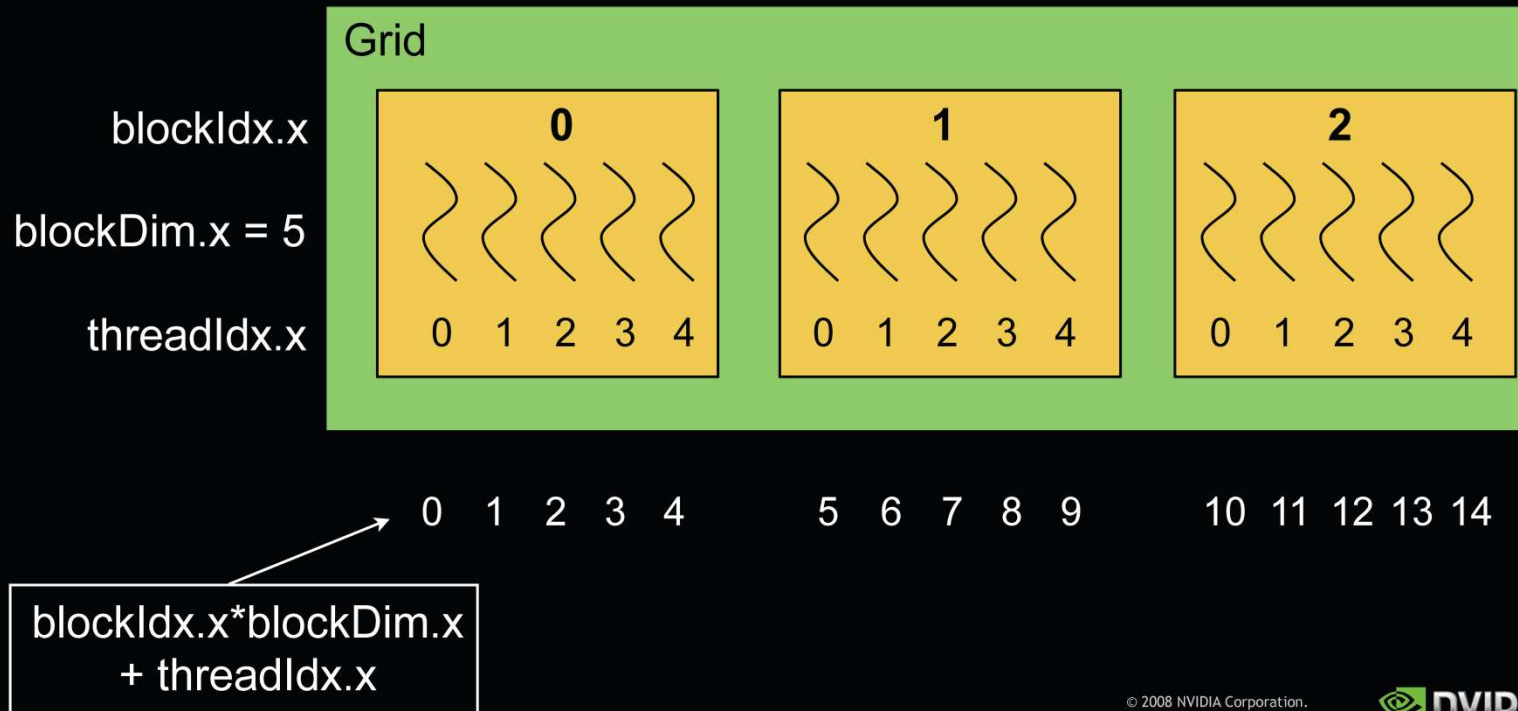
- Unspecified **dim3** fields initialize to 1

CUDA Built-in Device Variables

- All `__global__` and `__device__` functions have access to these automatically defined variables
 - `dim3 gridDim;`
 - Dimensions of the grid in blocks (at most 2D)
 - `dim3 blockDim;`
 - Dimensions of the block in threads
 - `dim3 blockIdx;`
 - Block index within the grid
 - `dim3 threadIdx;`
 - Thread index within the block

Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
 - Map from local thread ID (threadIdx) to a global ID which can be used as array indices



Increment Array Example

CPU program

```
void inc_cpu(int *a, int N)
{
    int idx;

    for (idx = 0; idx < N; idx++)
        a[idx] = a[idx] + 1;
}

int main()
{
    ...
    inc_cpu(a, N);
}
```

CUDA program

```
__global__ void inc_gpu(int *a, int N)
{
    int idx = blockIdx.x * blockDim.x
              + threadIdx.x;

    if (idx < N)
        a[idx] = a[idx] + 1;
}

int main()
{
    ...
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize) );
    inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```


Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is valuable
 - Share results to avoid redundant computation
 - Share memory accesses
 - Drastic bandwidth reduction
- Thread cooperation is a powerful feature of CUDA
- Cooperation between a monolithic array of threads is not scalable
 - Cooperation within smaller **batches** of threads is scalable

Host Synchronization

- **All kernel launches are asynchronous**
 - control returns to CPU immediately
 - kernel executes after all previous CUDA calls have completed
- **cudaMemcpy() is synchronous**
 - control returns to CPU after copy completes
 - copy starts after all previous CUDA calls have completed
- **cudaThreadSynchronize()**
 - blocks until all previous CUDA calls complete

CUDA 4.x or newer:
cudaDeviceSynchronize() and
cudaStreamSynchronize()

Host Synchronization Example

// copy data from host to device

```
cudaMemcpy(a_d, a_h, numBytes, cudaMemcpyHostToDevice);
```

// execute the kernel

```
inc_gpu<<<ceil(N/(float)blocksize), blocksize>>>(a_d, N);
```

// run independent CPU code

```
run_cpu_stuff();
```

// copy data from device back to host

```
cudaMemcpy(a_h, a_d, numBytes, cudaMemcpyDeviceToHost);
```

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
- **Allowed in conditional code only if the conditional is uniform across the entire thread block**

Synchronization

- Threads in the same block can communicate using shared memory
- No HW global synchronization function yet
- `__syncthreads()`
 - Barrier for threads only within the current block
- `__threadfence()`
 - Flushes global memory writes to make them visible to all threads

Plus newer sync functions, e.g., from compute capability 2.x:

**`__syncthreads_count()`, `__syncthreads_and/or()`,
`__threadfence_block()`, `__threadfence_system()`, ...**

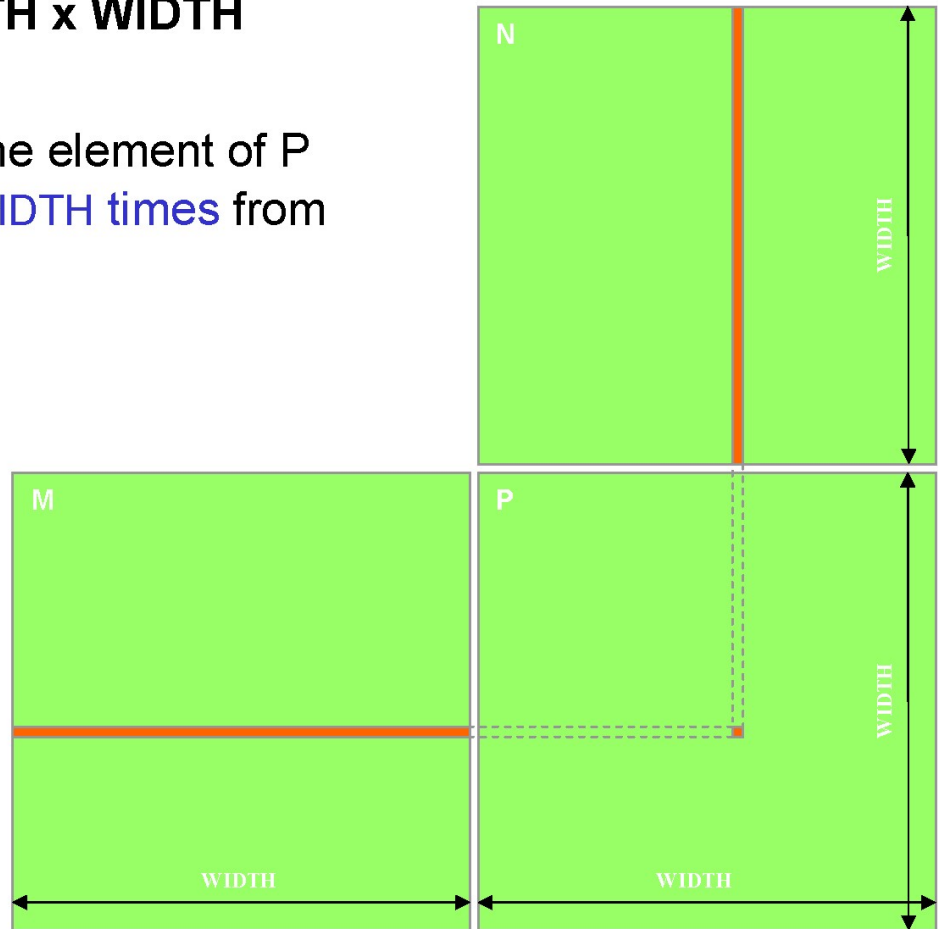
**Now: *Must* use versions with `_sync` suffix, because of
Independent Thread Scheduling (compute capability 7.x and newer)!**

Matrix-Matrix Multiplication

$$P=MN$$

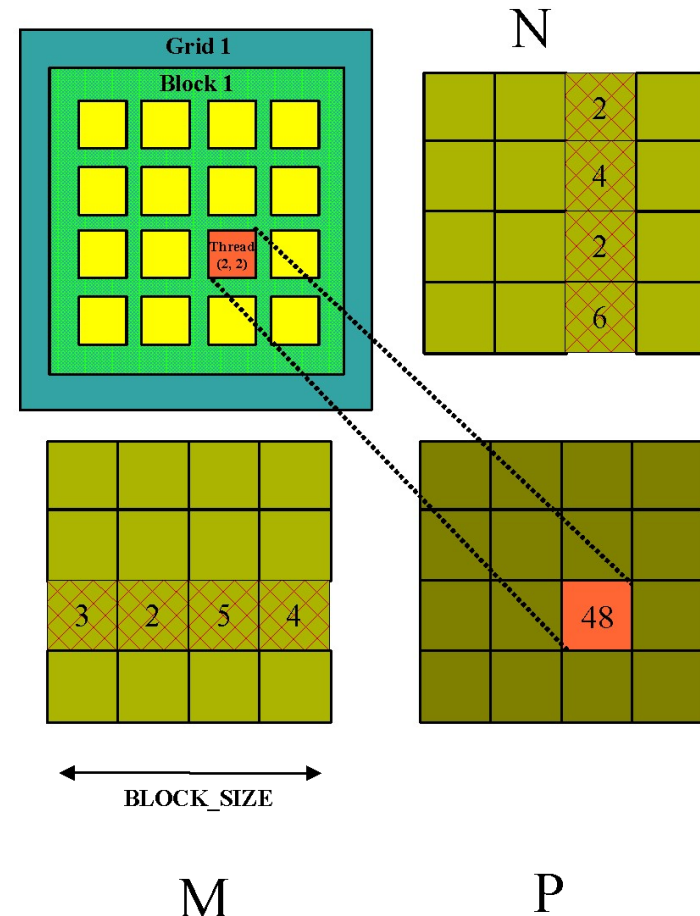
Programming Model: Square Matrix Multiplication

- $P = M * N$ of size $WIDTH \times WIDTH$
- Without tiling:
 - One **thread** handles one element of P
 - M and N are loaded $WIDTH$ times from 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 very high)
- **Size of matrix limited by the number of threads allowed in a thread block**

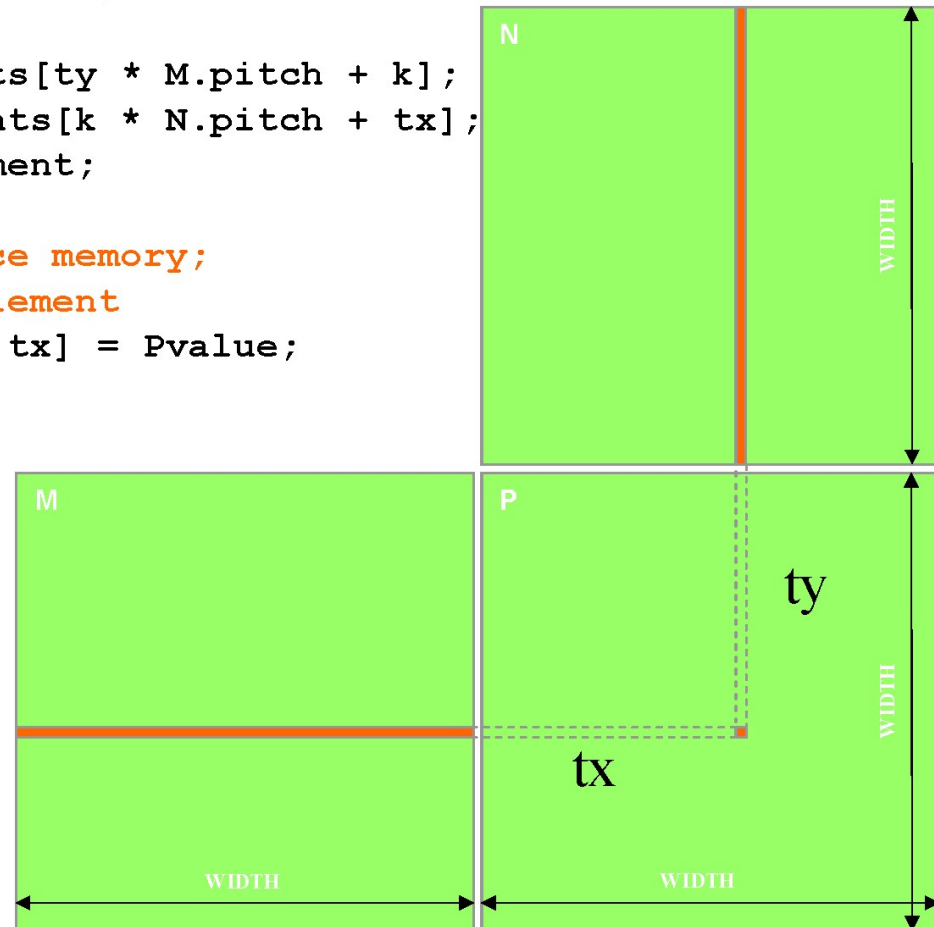


Matrix Multiplication

Device-Side Kernel Function (cont.)

...

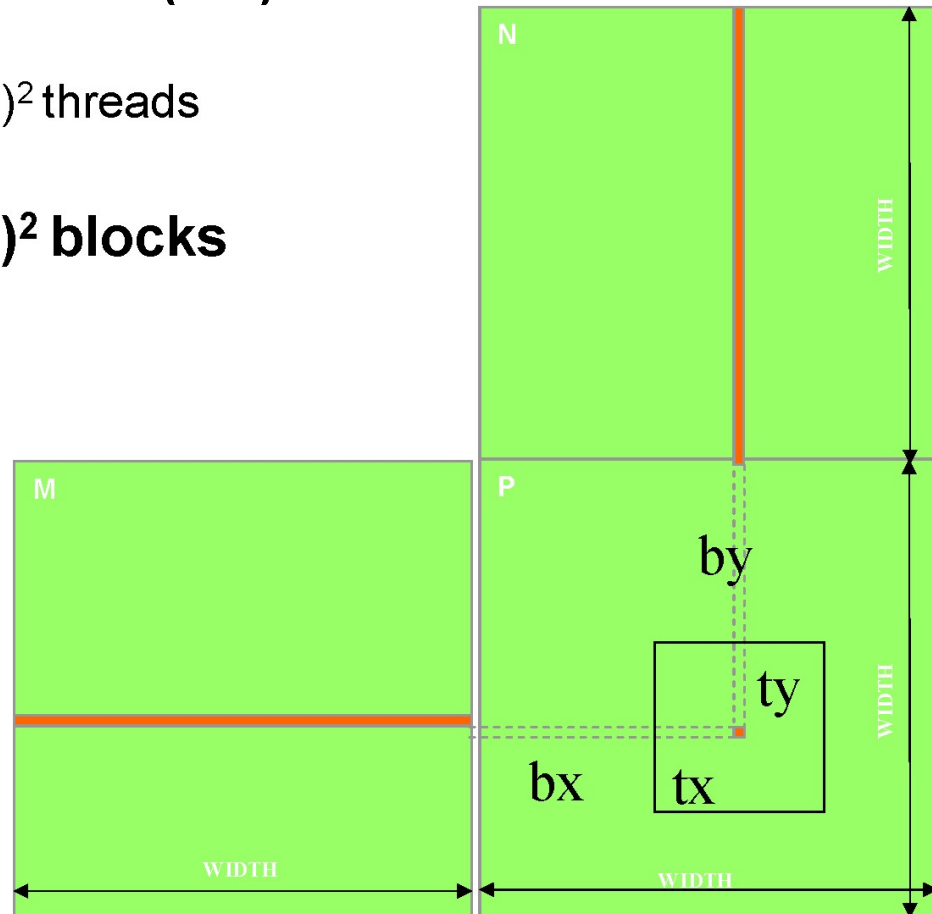
```
for (int k = 0; k < M.width; ++k)
{
    float Melement = M.elements[ty * M.pitch + k];
    float Nelement = Nd.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * blockDim.x + tx] = Pvalue;
}
```



Handling Arbitrary Sized Square Matrices

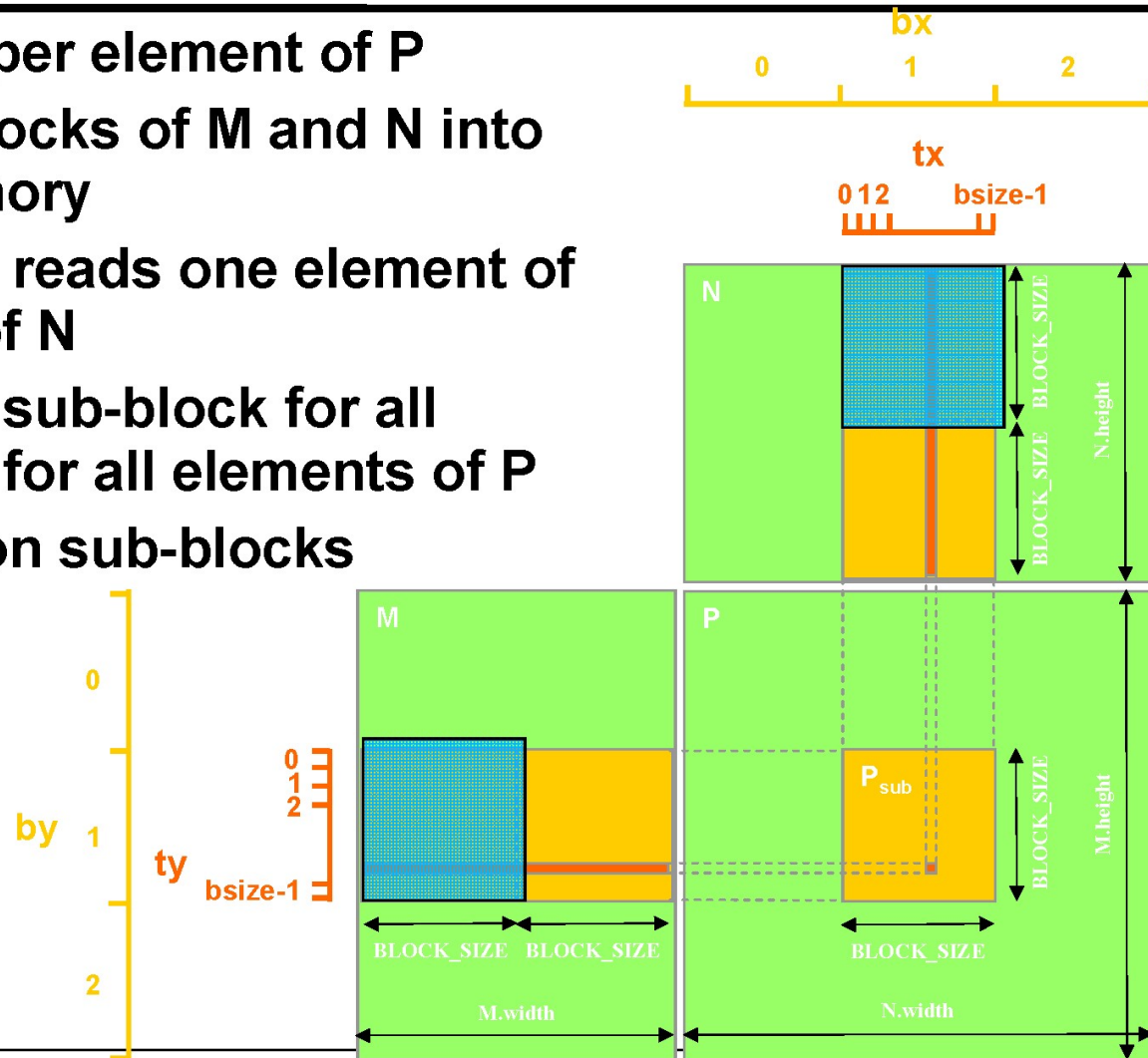
- Have each 2D thread block to compute a $(\text{BLOCK_WIDTH})^2$ sub-matrix (tile) of the result matrix
 - Each has $(\text{BLOCK_WIDTH})^2$ threads
- Generate a 2D Grid of $(\text{WIDTH}/\text{BLOCK_WIDTH})^2$ blocks

You still need to put a loop around the kernel call for cases where WIDTH is greater than Max grid size!



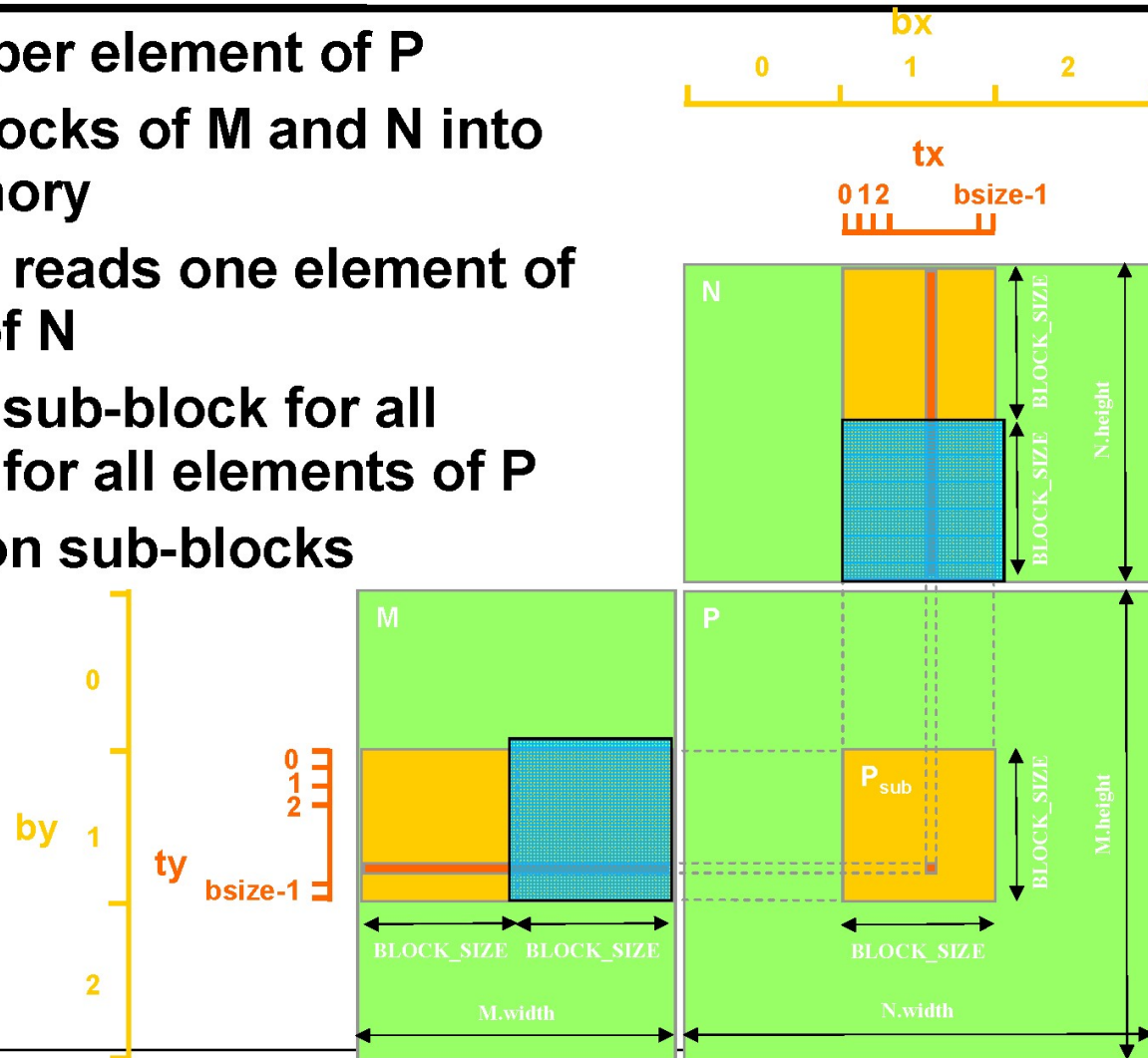
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



Example: Matrix Multiplication (1)



- Copy matrices to device; invoke kernel; copy result matrix back to host

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);

    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);
}
```

Example: Matrix Multiplication (2)



```
// Allocate C in device memory
Matrix d_C;
d_C.width = d_C.stride = C.width; d_C.height = C.height;
size = C.width * C.height * sizeof(float);
cudaMalloc((void**)&d_C.elements, size);

// Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

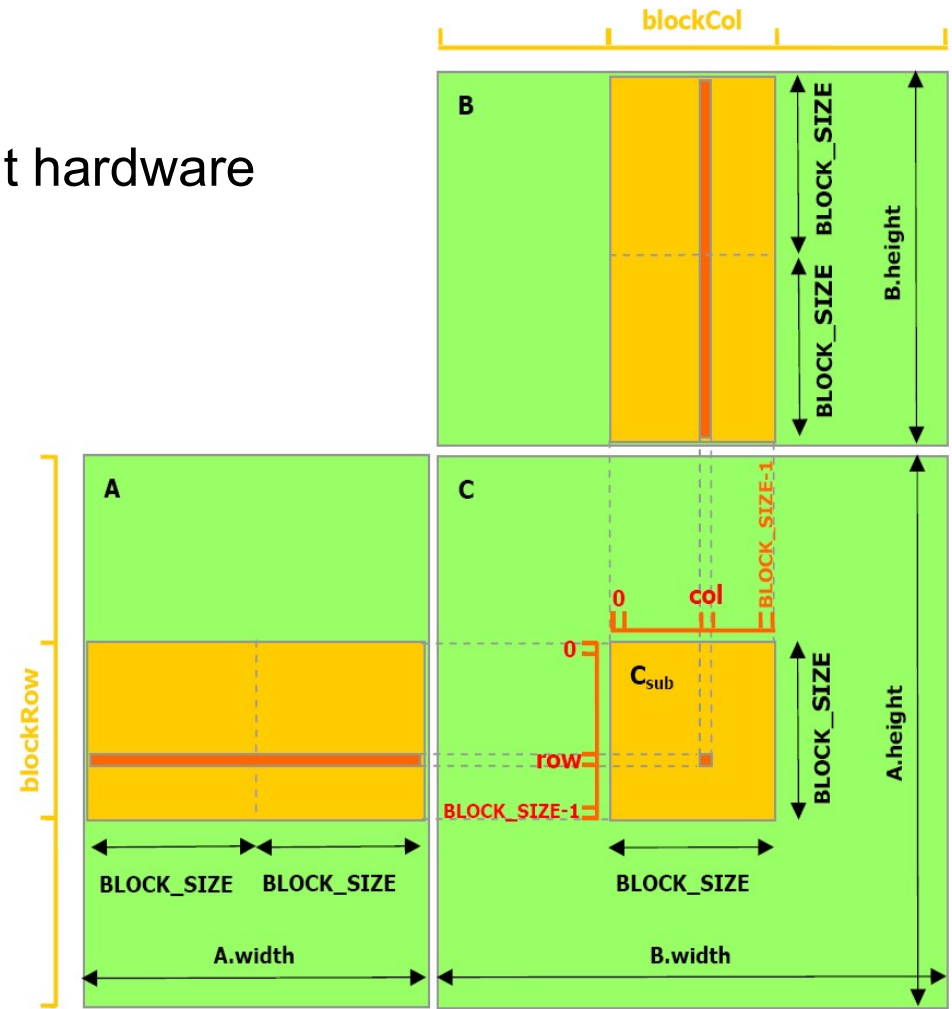
// Read C from device memory
cudaMemcpy(C.elements, d_C.elements, size,
           cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
}
```

Example: Matrix Multiplication (3)



- Multiply matrix block-wise
- Set BLOCK_SIZE for efficient hardware use, e.g., to 16 on cc. 1.x or 16 or 32 on cc. 2.x +
- Maximize parallelism
 - Launch as many threads per block as block elements
 - Each thread fetches one element of block
 - Perform row * column dot products in parallel



Example: Matrix Multiplication (4)



```
__global__ void MatrixMul( float *matA, float *matB, float *matC, int w )
{
    __shared__ float blockA[ BLOCK_SIZE ][ BLOCK_SIZE ];
    __shared__ float blockB[ BLOCK_SIZE ][ BLOCK_SIZE ];

    int bx = blockIdx.x; int tx = threadIdx.x;
    int by = blockIdx.y; int ty = threadIdx.y;

    int col = bx * BLOCK_SIZE + tx;
    int row = by * BLOCK_SIZE + ty;

    float out = 0.0f;
    for ( int m = 0; m < w / BLOCK_SIZE; m++ ) {

        blockA[ ty ][ tx ] = matA[ row * w + m * BLOCK_SIZE + tx ];
        blockB[ ty ][ tx ] = matB[ col + ( m * BLOCK_SIZE + ty ) * w ];
        __syncthreads();

        for ( int k = 0; k < BLOCK_SIZE; k++ ) {
            out += blockA[ ty ][ k ] * blockB[ k ][ tx ];
        }
        __syncthreads();
    }

    matC[ row * w + col ] = out;
}
```

Caveat: for brevity, this code assumes matrix sizes are a multiple of the block size (either because they really are, or because padding is used; otherwise guard code would need to be added)

Thank you.