Computer Graphics

– Cuda Programming –

Hendrik Lensch

Overview

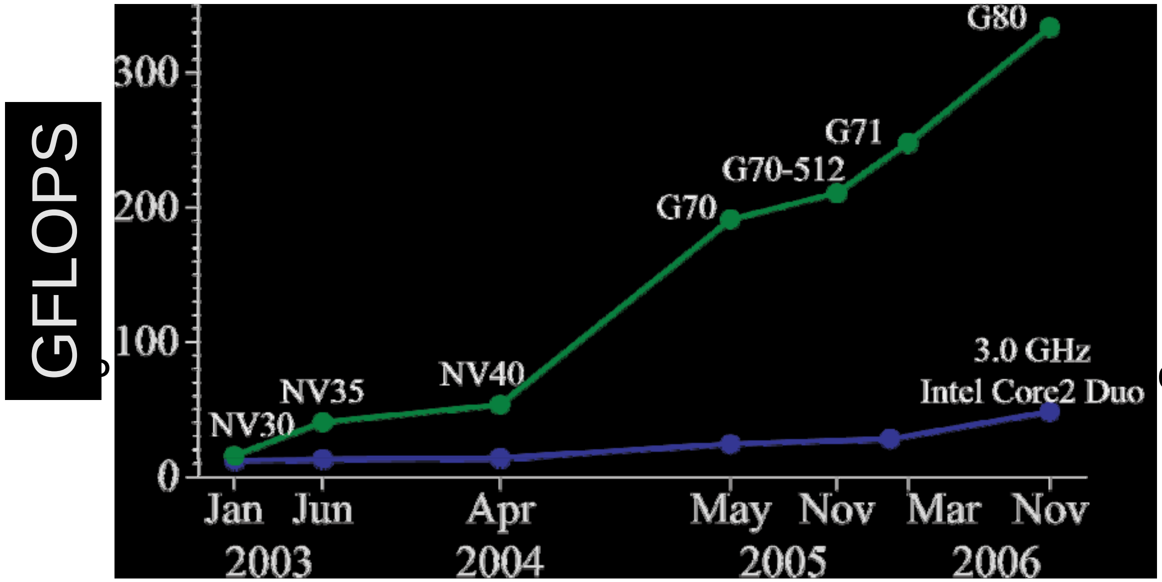
- **So far:**
  - OpenGL
  - Programmable Shader
- **Today:**
  - GPGPU via Cuda (general purpose computing on the GPU)
- **Next:**
  - Some Parallel Programming

Resources

- **Where to find Cuda and the documentation?**
  - http://www.nvidia.com/object/cuda_home.html
- **Lecture on parallel programming on the GPU by David Kirk (most of the following slides are copied from this course)**
  - http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html
- **On the Parallel Prefix Sum (Scan) algorithm**
  - http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/scan/doc/scan.pdf

Why Massively Parallel Processor

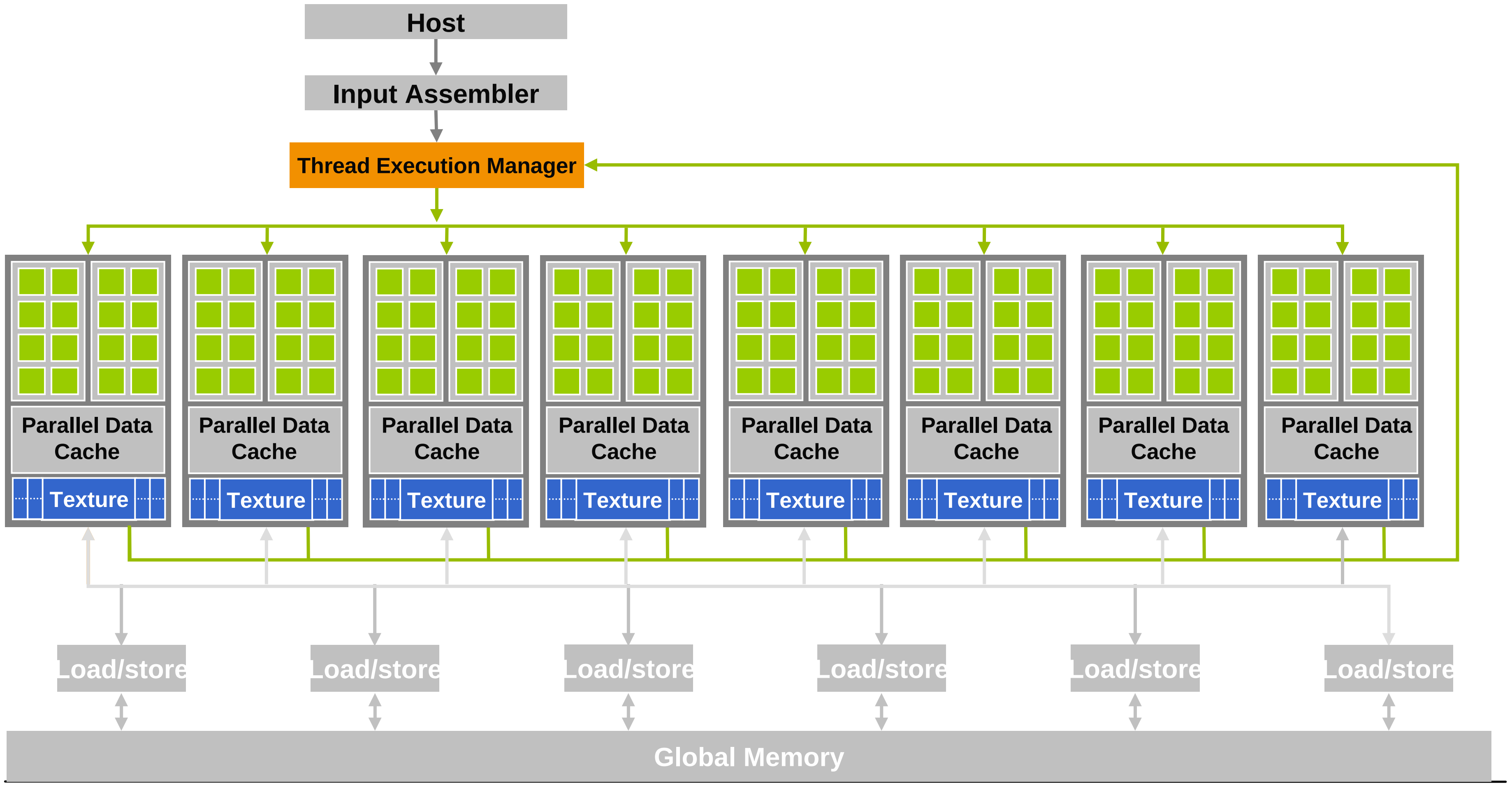
- **A quiet revolution and potential build-up**
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until last year, programmed through graphics API



G80 = GeForce 8800 GTX	
G71 = GeForce 7900 GTX	
G70 = GeForce 7800 GTX	
NV40 = GeForce 6800 Ultra	
NV35 = GeForce FX 5950 Ultra	
NV30 = GeForce FX 5800	

GeForce 8800

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU



Future Apps Reflect a Concurrent World

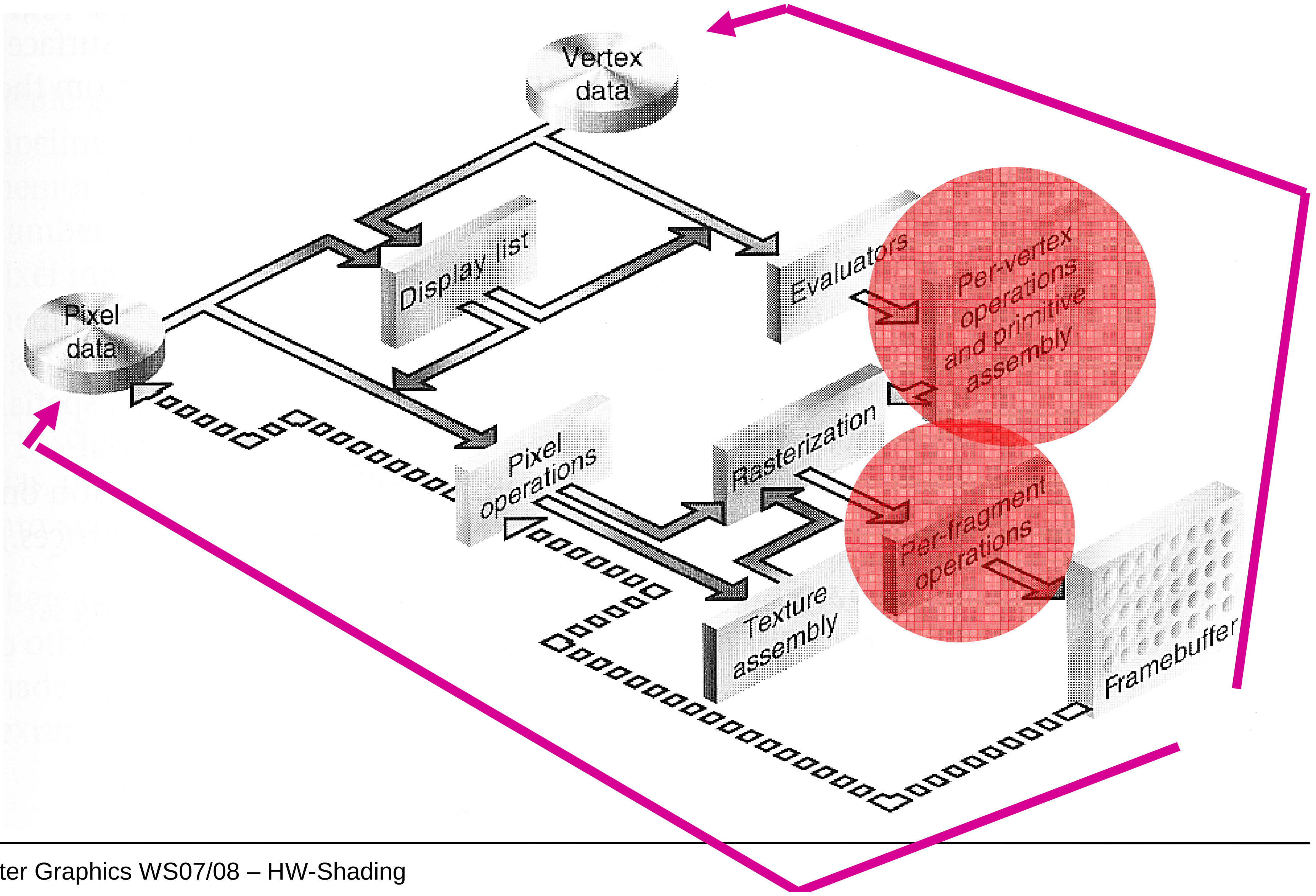
- **Exciting applications in future mass computing market have been traditionally considered "supercomputing applications"**
  - Molecular dynamics simulation, Video and audio coding and manipulation, 3D imaging and visualization, Consumer game physics, and virtual reality products
  - These "Super-apps" represent and model physical, concurrent world
- **Various granularities of parallelism exist, but…**
  - programming model must not hinder parallel implementation
  - data delivery needs careful management

What is GPGPU ?

- **General Purpose computation using GPU in applications other than 3D graphics**
  - GPU accelerates critical path of application
- **Data parallel algorithms leverage GPU attributes**
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- **Applications – see //GPGPU.org**
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

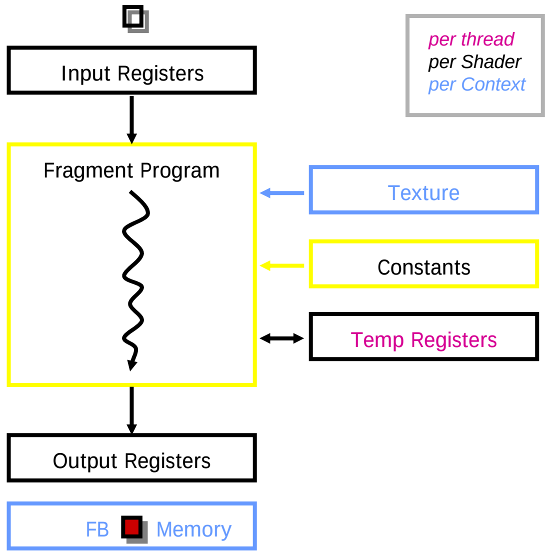
GPGPU

Multi-Pass Rendering



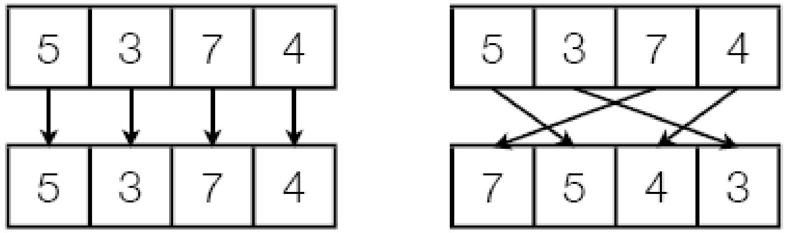
Previous GPGPU Constraints

- **Dealing with graphics API**
  - Working with the corner cases of the graphics API
- **Addressing modes**
  - Limited texture size/dimension
- **Shader capabilities**
  - Limited outputs
- **Instruction sets**
  - Lack of Integer & bit ops
- **Communication limited**
  - Between pixels
  - no Scatter a[i] = p



Traditional GPGPU

- **Standard Algorithm**
  - Set up OpenGL state
  - Draw a fullscreen quad
  - Shader program with textures as input to perform computation
  - Write result to framebuffer as a color
- **Limitations**
  - Requires non-graphics people to know a lot about graphics APIs
  - Computation power wasted on unnecessary graphics setup
  - Graphics API restricts input/output formats, integer/bit operations, branching/looping, etc.
  - Each fragment program must write to a single, predefined location: no way to **scatter** data



[from Jerry Talton]

CUDA

- **“Compute Unified Device Architecture”**
- **General purpose programming model**
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- **Targeted software stack**
  - Compute oriented drivers, language, and tools
- **Driver for loading computation programs into GPU**
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management
- **Not another graphics API**

Cuda

- **Compute Unified Device Architecture**
  - **Unified hardware and software specification for parallel computation**
  - **Simple extensions to C language to allow code to run on the GPU**
  - **Developed by and for NVIDIA (introduced with the GeForce 8800 series)**
  - **Much easier to use than ATI's Close To Metal hardware interface**
- **Benefits and Features**
  - **Application controlled SIMD program structure**
  - **Fully general load/store to GPU memory**
  - **Totally untyped (not limited to texture storage)**
  - **No limits on branching, looping, etc.**
  - **Full integer and bit instructions**
  - **Supports pointers**
  - **Explicitly managed memory down to cache level**
  - **No graphics code (although interoperability with OpenGL/D3D is supported)**

What is the GPU Good at?

- **The GPU is good at**

  data-parallel processing

  - The same computation executed on many data elements in parallel – low control flow overhead

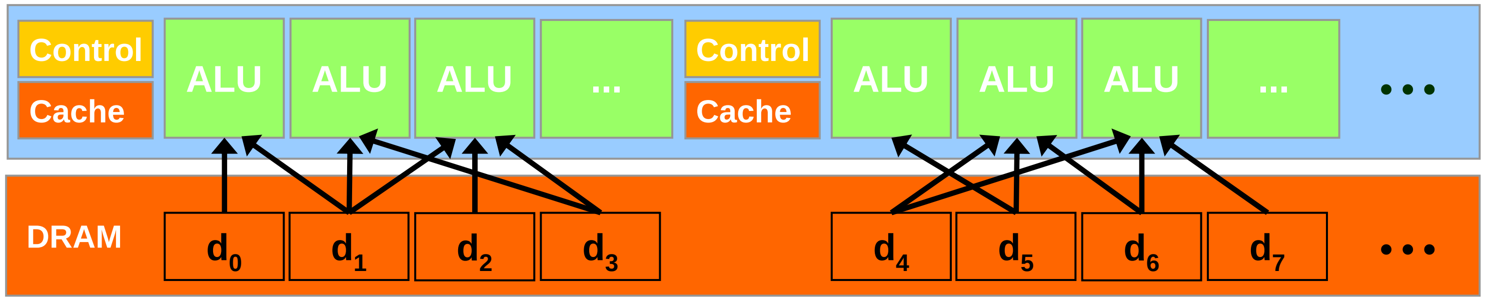
  with high SP floating point arithmetic intensity

  - Many calculations per memory access
  - Currently also need high floating point to integer ratio

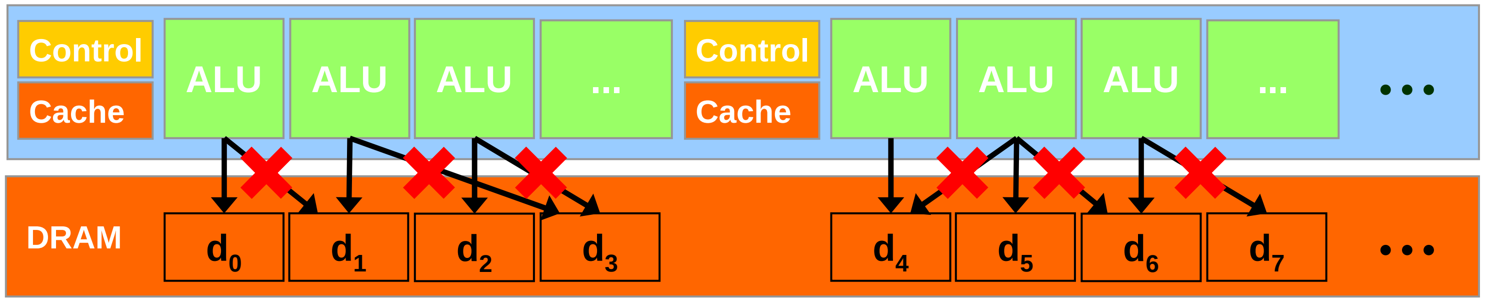
- **High floating-point arithmetic intensity and many data elements mean that memory access latency can be hidden with calculations instead of big data caches – Still need to avoid bandwidth saturation!**

Drawbacks of (legacy) GPGPU Model: Hardware Limitations

- **Memory accesses are done as pixels**
  - Only gather: can read data from other pixels



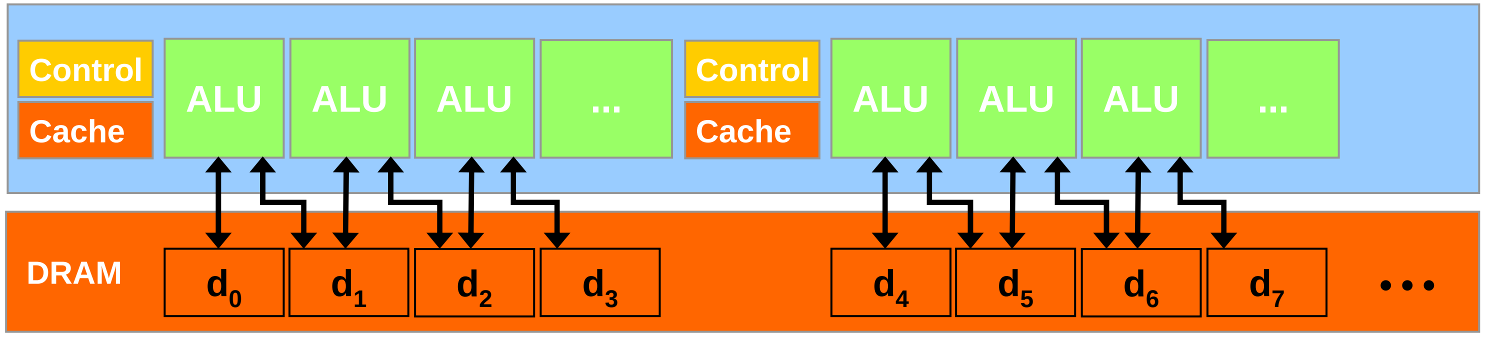
  - No scatter: (Can only write to one pixel)



→ Less programming flexibility

Drawbacks of (legacy) GPGPU Model: Hardware Limitations

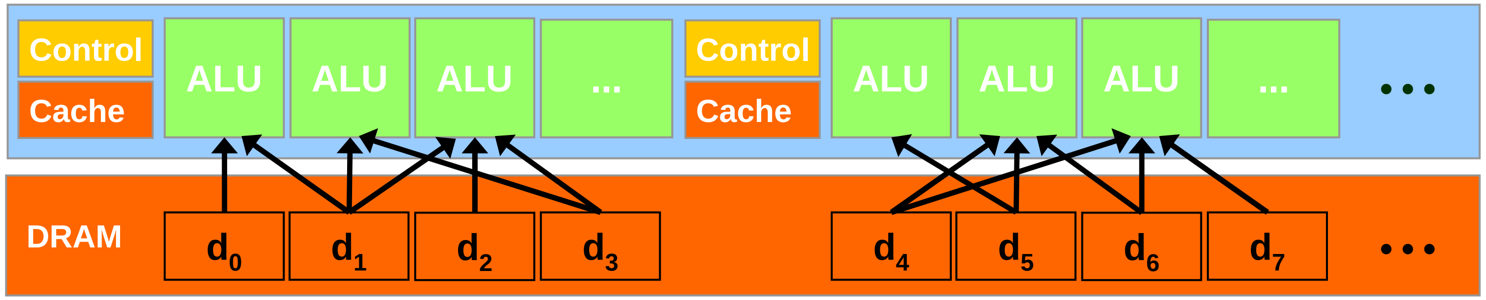
- **Applications can easily be limited by DRAM memory bandwidth**



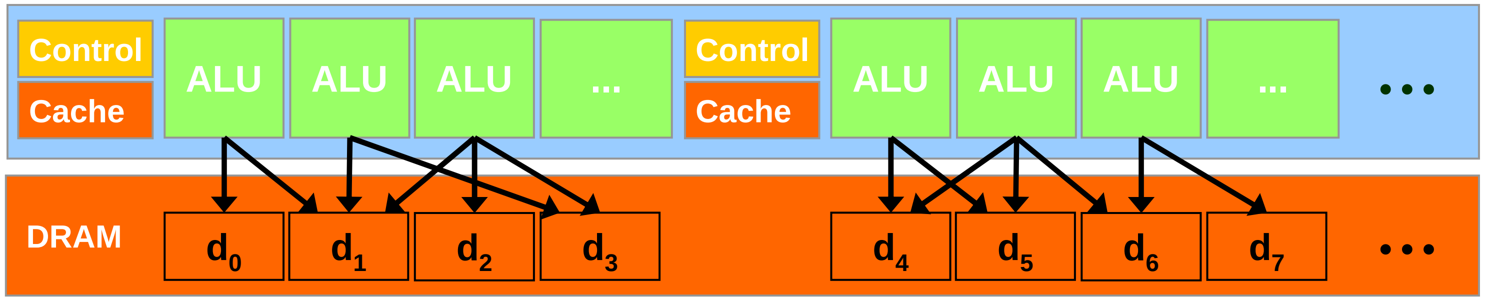
→ **Waste of computation power due to data starvation**

CUDA Highlights: Scatter

- **CUDA provides generic DRAM memory addressing**
  - Gather:



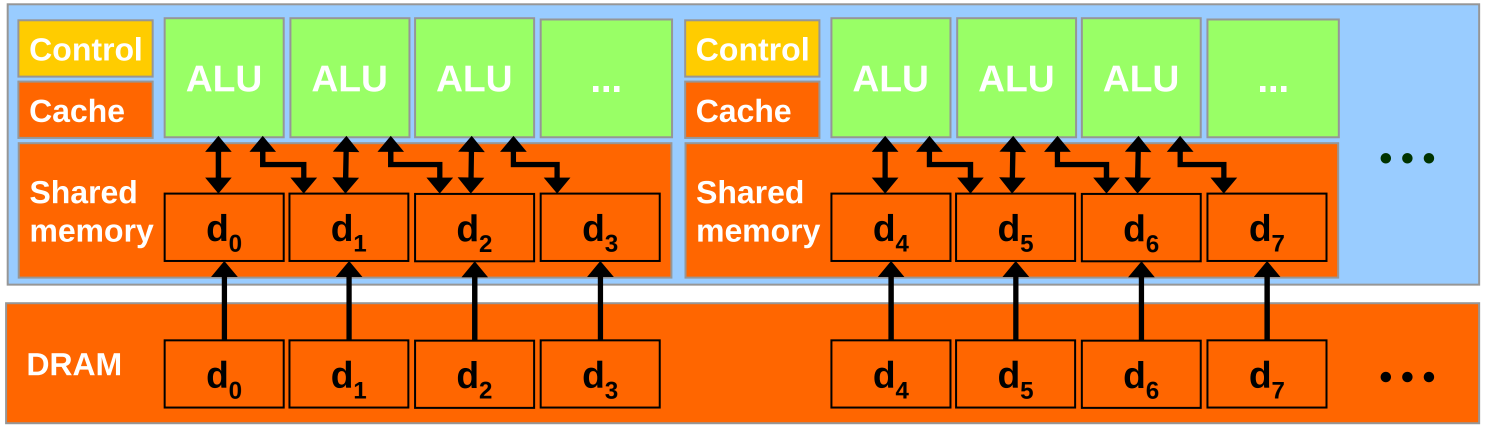
  - And scatter: no longer limited to write one pixel



→ **More programming flexibility**

CUDA Highlights: On-Chip Shared Memory

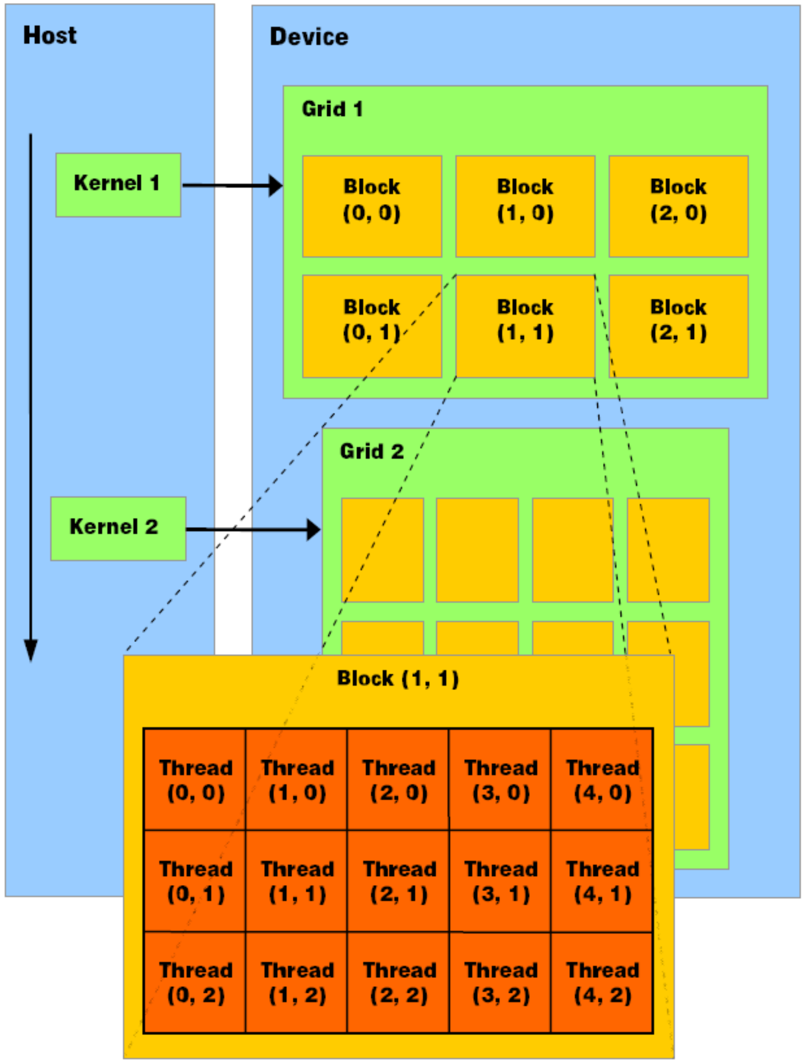
- **CUDA enables access to a parallel on-chip shared memory for efficient inter-thread data sharing**



$\rightarrow$ **Big memory bandwidth savings**

Programming Model

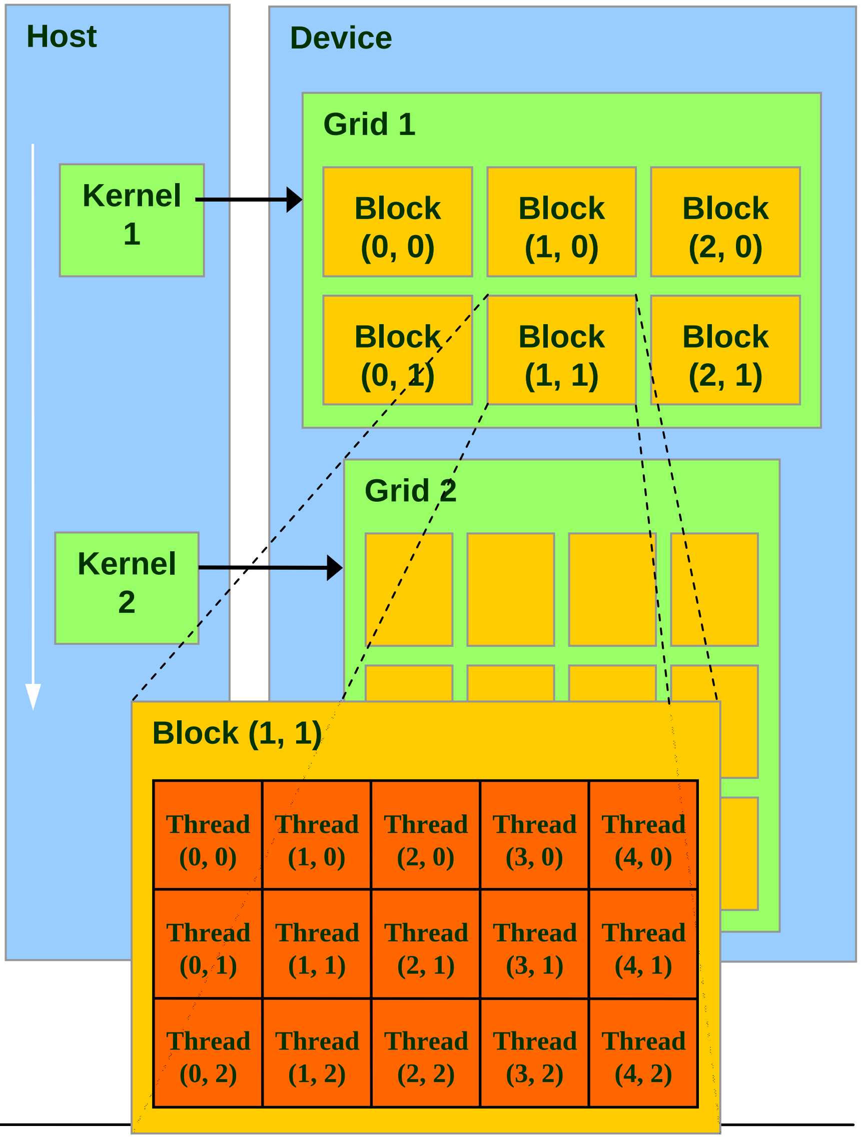
- **Programming Model**
  - The programmer writes a kernel (in C) for each task he or she wishes to perform
  - The application splits the data to be processed into grids of thread blocks
  - When a kernel is launched, each block is allocated to a single TP
  - Threads of a given block are time sliced onto SPs contained within that block's TP



Many problems have natural grid structure, but decomposing data into threads can be difficult in general

Thread Batching: Grids and Blocks

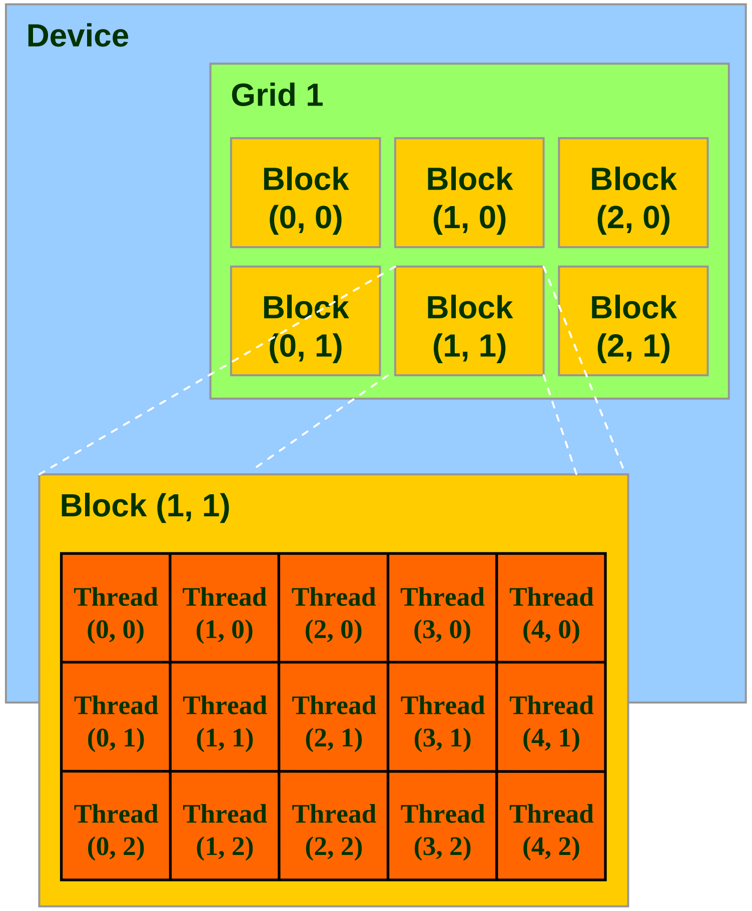
- **A kernel is executed as a grid of thread blocks**
  - All threads share data memory space
- **A thread block is a batch of threads that can cooperate with each other by:**
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency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



Courtesy: NDVIA

Block and Thread IDs

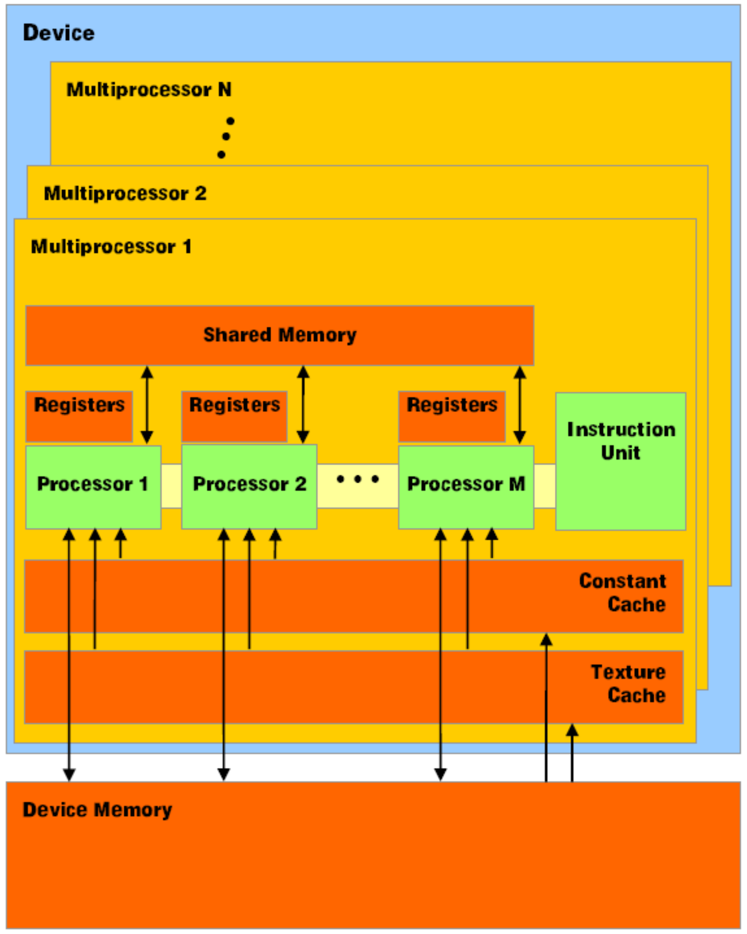
- **Threads and blocks have IDs**
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- **Simplifies memory addressing when processing multidimensional data**
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

Programming Model: Memory Spaces

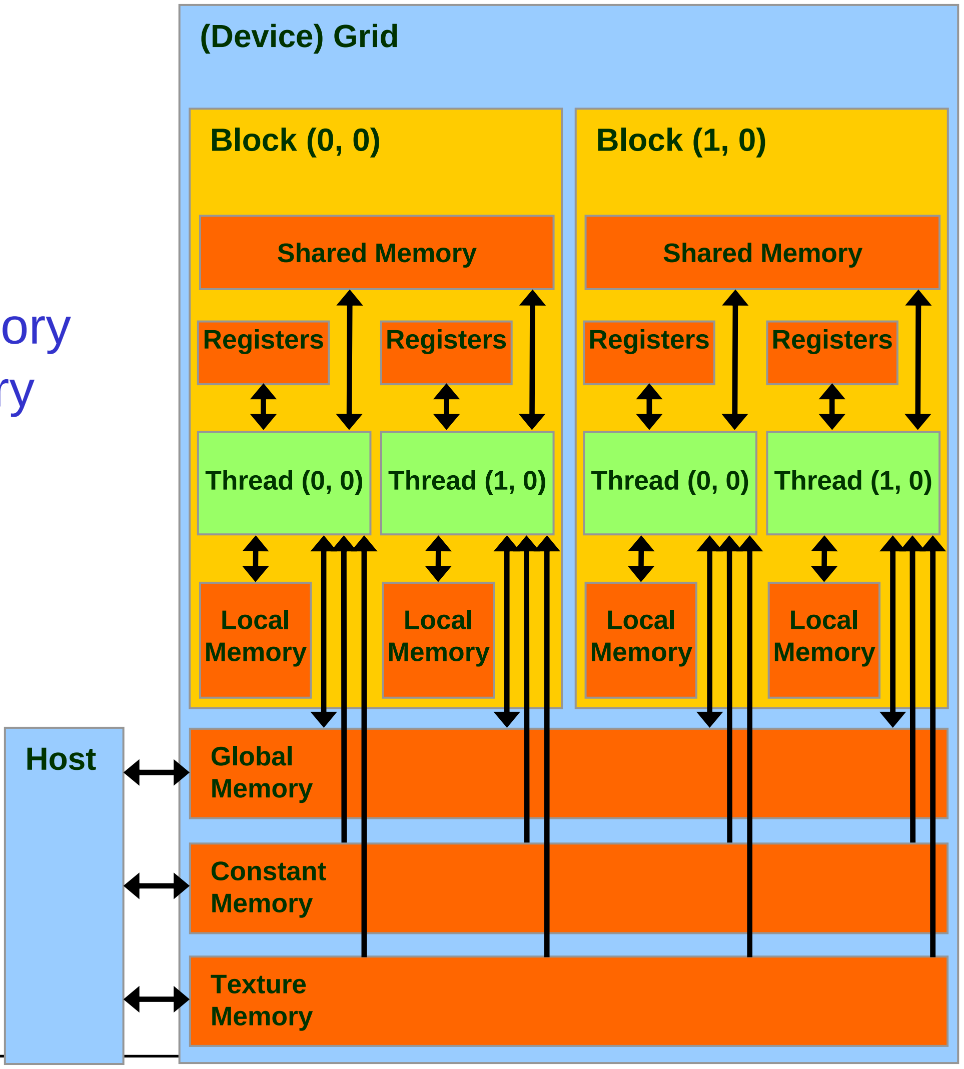
- **Global Memory**
  - **Read-write per-grid**
  - **Hundreds of MBs**
  - **Very slow (600 clocks)**
- **Texture Memory**
  - **Read-only per-grid**
  - **Hundreds of MBs**
  - **Slow first access, but cached**
  - **Built-in filtering, clamping**
- **Constant Memory**
- **Shared! Memory**
  - **Read-write per-block**
  - **16 KB per block**
  - **Very fast (4 clocks)**
- **Registers**
  - **Unique per thread**



CUDA Device Memory Space

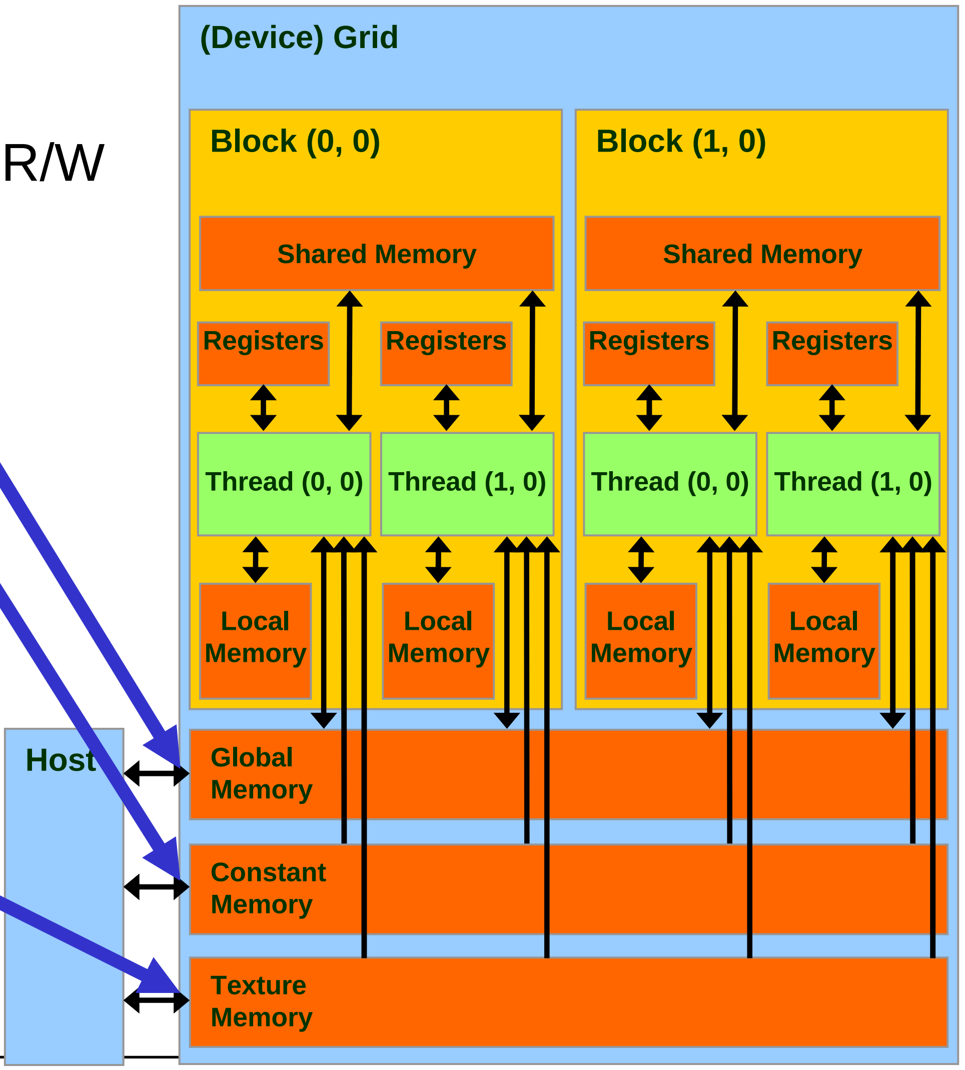
- **Each thread can:**
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- **The host can R/W global, constant, and texture memories**



Global, Constant, and Texture Memories (Long Latency Accesses)

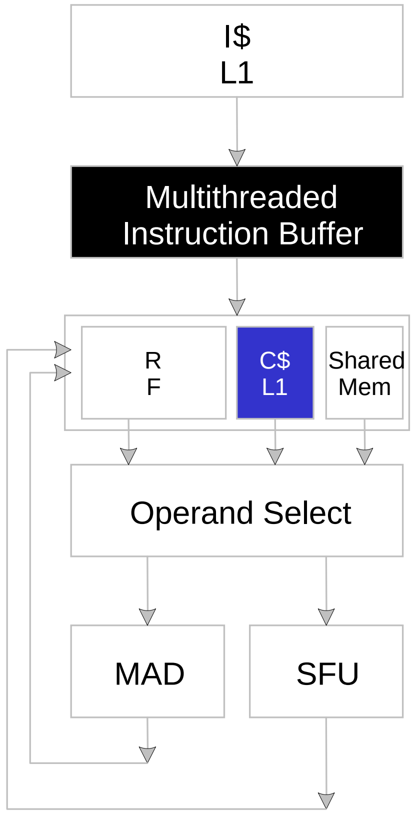
- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- **Texture and Constant Memories**
  - Constants initialized by host
  - Contents visible to all threads



Courtesy: NDVIA

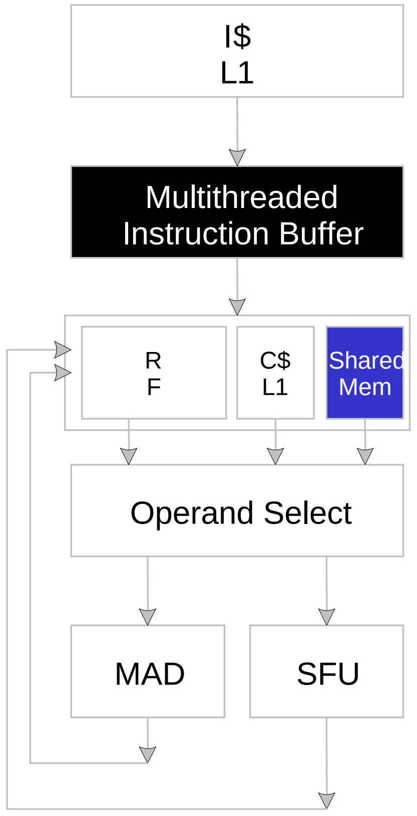
Constants

- Immediate address constants
- Indexed address constants
- Constants stored in DRAM, and cached on chip
  - L1 per SM
- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!



Shared Memory

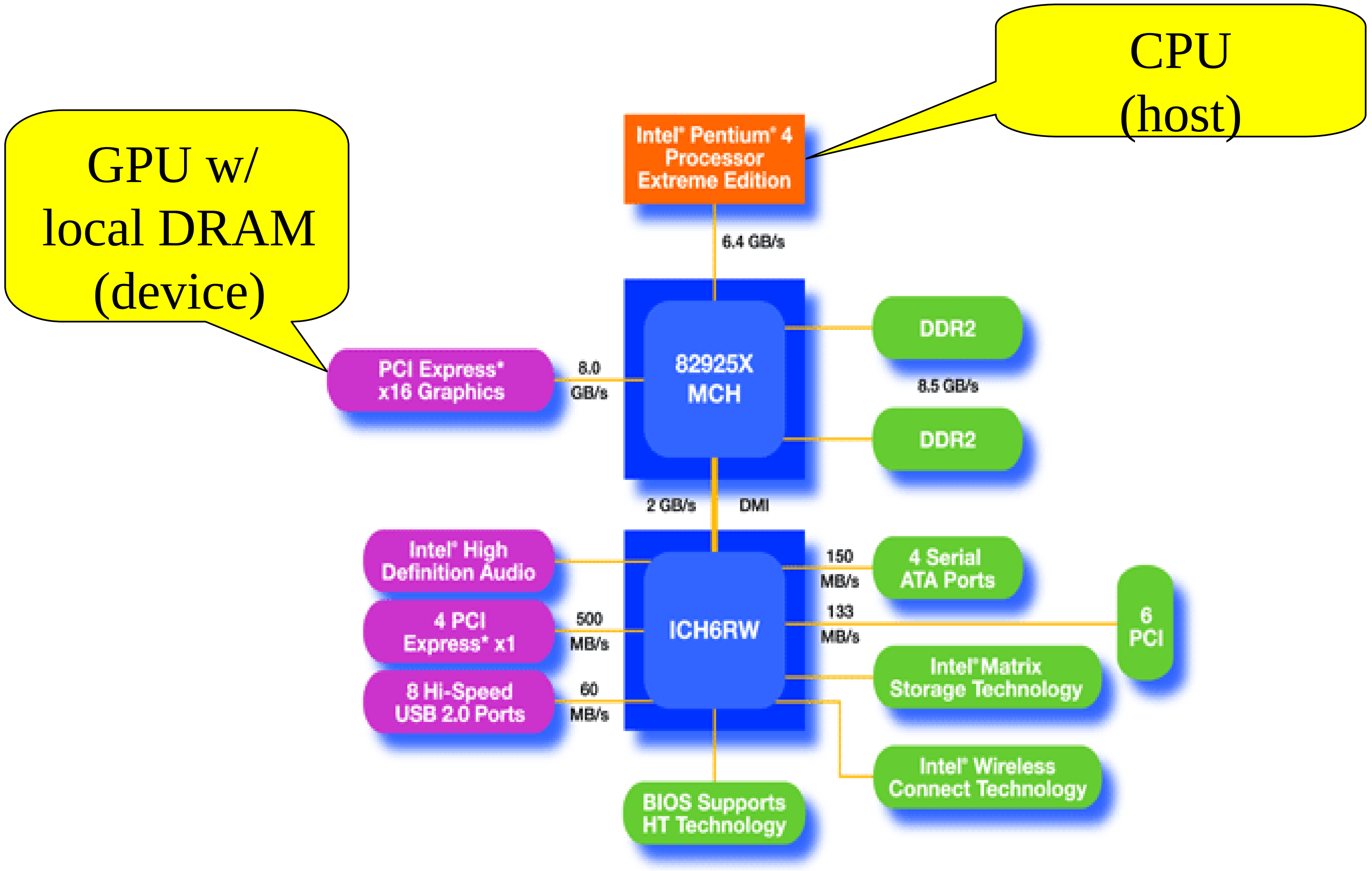
- Each SM has 16 KB of Shared Memory
  – 16 banks of 32bit words
- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  – read and write access
- Not used explicitly for pixel shader programs
  – we dislike pixels talking to each



Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

An Example of Physical Reality Behind CUDA


CPU (host)
GPU w/ local DRAM (device)
Intel® Pentium® 4 Processor Extreme Edition
6.4 GB/s
82925X MCH
PCI Express* x16 Graphics
8.0 GB/s
DDR2
8.5 GB/s
DDR2
2 GB/s
DMI
ICH6RW
Intel® High Definition Audio
4 PCI Express* x1
500 MB/s
8 Hi-Speed USB 2.0 Ports
60 MB/s
150 MB/s
4 Serial ATA Ports
133 MB/s
6 PCI
Intel® Matrix Storage Technology
Intel® Wireless Connect Technology
BIOS Supports HT Technology

CUDA Programming Model: A Highly Multithreaded Coprocessor

- **The GPU is viewed as a compute device that:**
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- **Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads**
- **Differences between GPU and CPU threads**
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Execution Model

- **Warps**
  - **Each block is split into SIMD groups of threads called** warps
  - **Warps are swapped in and out via thread scheduling**
  - **Threads within a warp execute in lock step**
  - **Threads are assigned to warps consecutively by their thread ID**
  - **Issue order of warps and blocks is undefined, but there are synchronization primitives**

- **Performance**
  - **Branches are predicated**
  - **Divergence within a warp should be avoided if possible**
  - **Memory coherence extremely important**
  - **Always try to read/write in a coalesced manner**

Application Programming Interface

- **The API is an**
  extension to the C programming language
- **It consists of:**
  - Language extensions
    - To target portions of the code for execution on the device
    - Two stage compilation (e.g. nvcc + gcc)
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
    - A host component to control and access one or more devices from the host
    - A device component providing device-specific functions

- **Function Quantifiers**
  - **`__device__` callable on the GPU from the GPU**
  - **`__global__` callable on the GPU from the CPU**
  - **`__host__` callable on the CPU from the CPU**
- **Variable Quantifiers**
  - **`__device__` global memory on the GPU**
  - **`__constant__` constant memory on the GPU**
  - **`__shared__` shared per-block memory on the GPU**
- **Built-in Variables**
  - **`gridDim, blockDim` gives dimensions of grids and blocks in kernel**
  - **`blockIdx, threadIdx` gives index of block and thread in kernel**
- **Built-in Vector Types**
  - **`float2, float3, float4`, etc.**

Extended C

- Declspecs
  - **global, device, shared, local, constant**
- Keywords
  - **threadIdx, blockIdx**
- Intrinsics
  - **__syncthreads**
- Runtime API
  - **Memory, symbol, execution management**
- Function launch

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

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


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

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

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

CUDA Function Declarations (cont.)

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

Calling a Kernel Function – Thread Creation

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

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

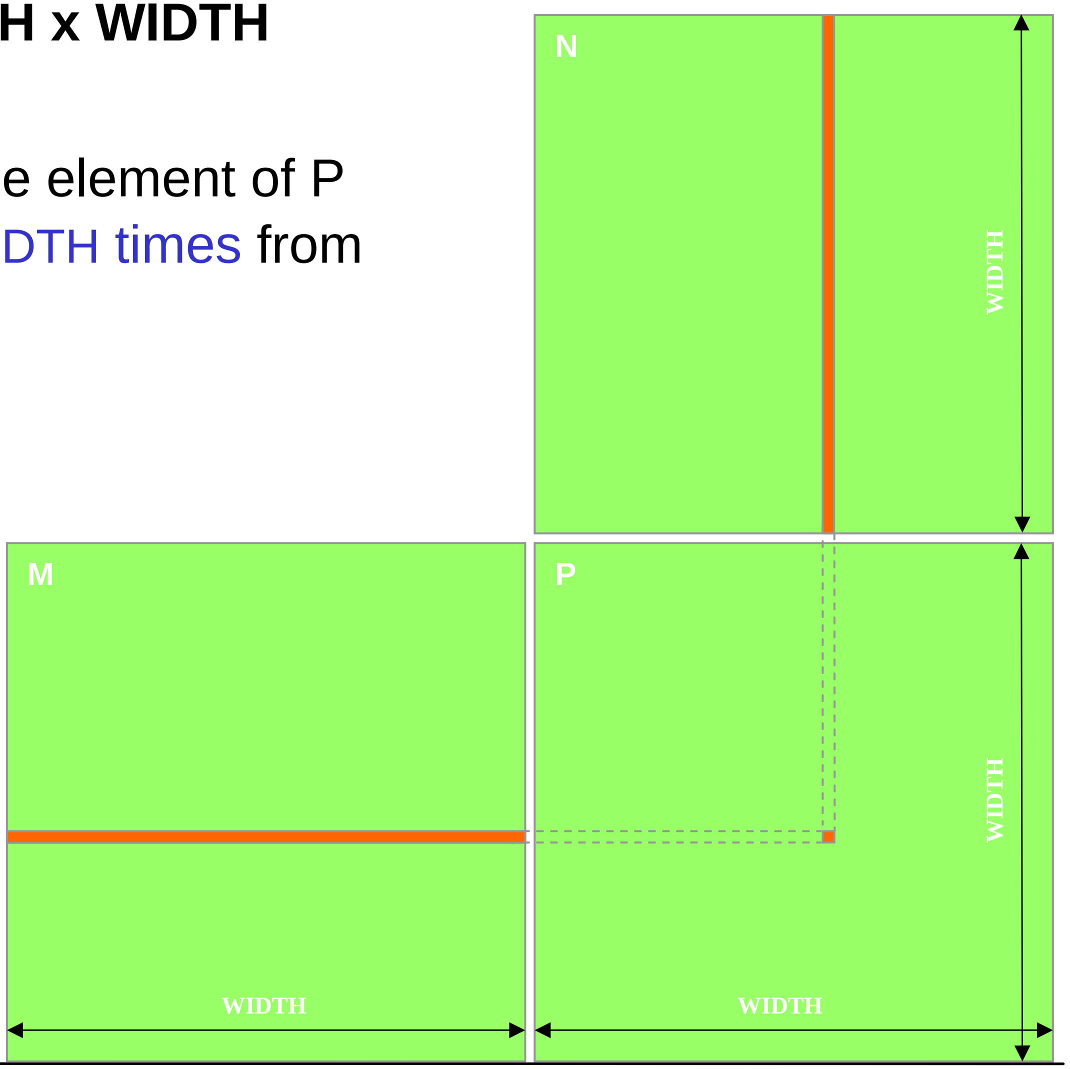
- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

A Simple Running Example: Matrix Multiplication

- **A straightforward 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

Programming Model: Square Matrix Multiplication

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



Step 1: Matrix Data Transfers

```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size, cudaMemcpyDeviceToHost);
...
// Free device memory
cudaFree(Md.elements);
```

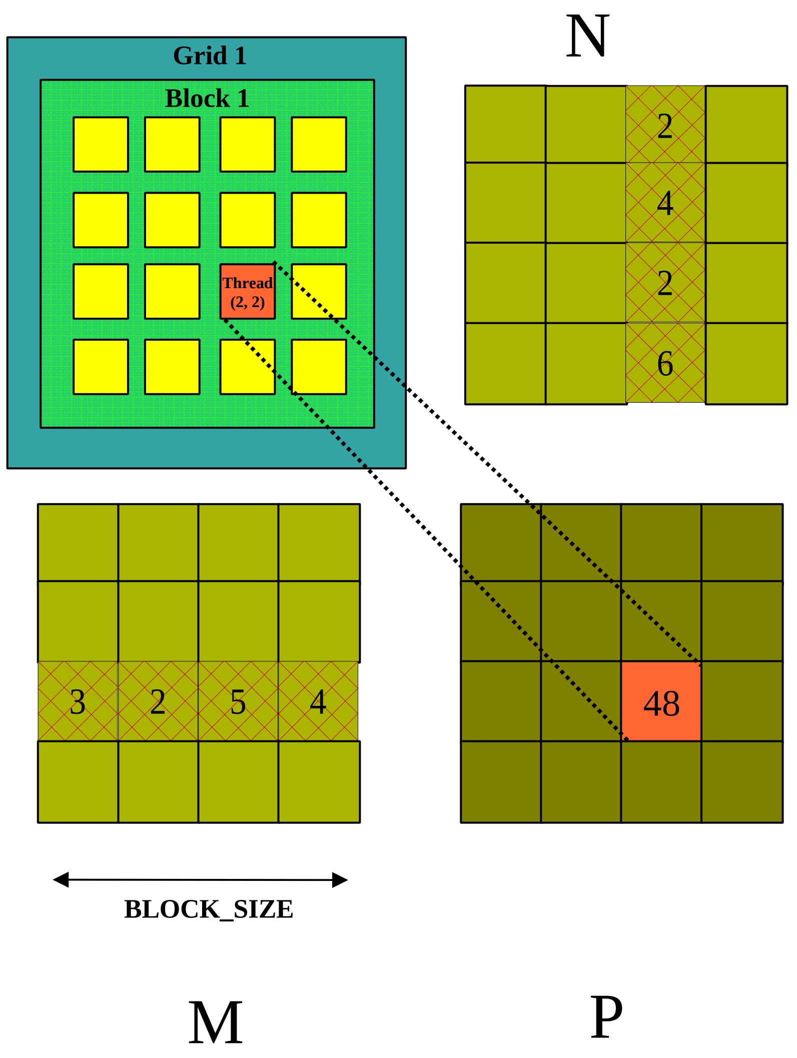
Step 2: Matrix Multiplication A Simple Host Code in C

```
// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal

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

Multiply Using One Thread Block

- **One Block of threads compute 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**



Step 3: Matrix Multiplication Host-side Main Program Code

```
int main(void) {
// Allocate and initialize the matrices
    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 3: Matrix Multiplication Host-side code

```
// Matrix multiplication on the device
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);
    CopyToDeviceMatrix(Pd, P); // Clear memory
```

Step 3: Matrix Multiplication Host-side Code (cont.)

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

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

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

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

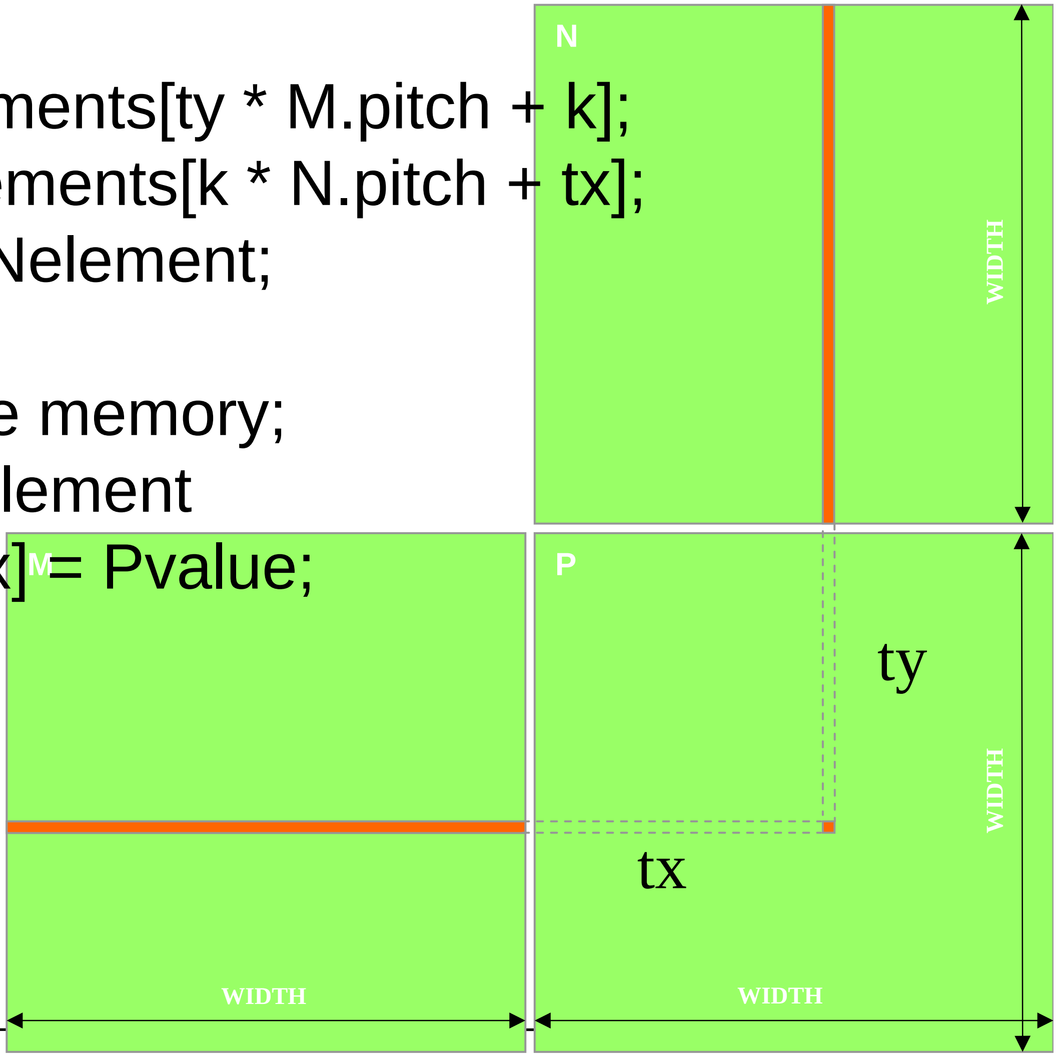
Step 4: Matrix Multiplication Device-side Kernel Function

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

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Step 4: 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 * P.pitch + tx] = Pvalue;
}
```



Step 5: 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;
}

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

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

Step 5: Some Loose Ends (cont.)

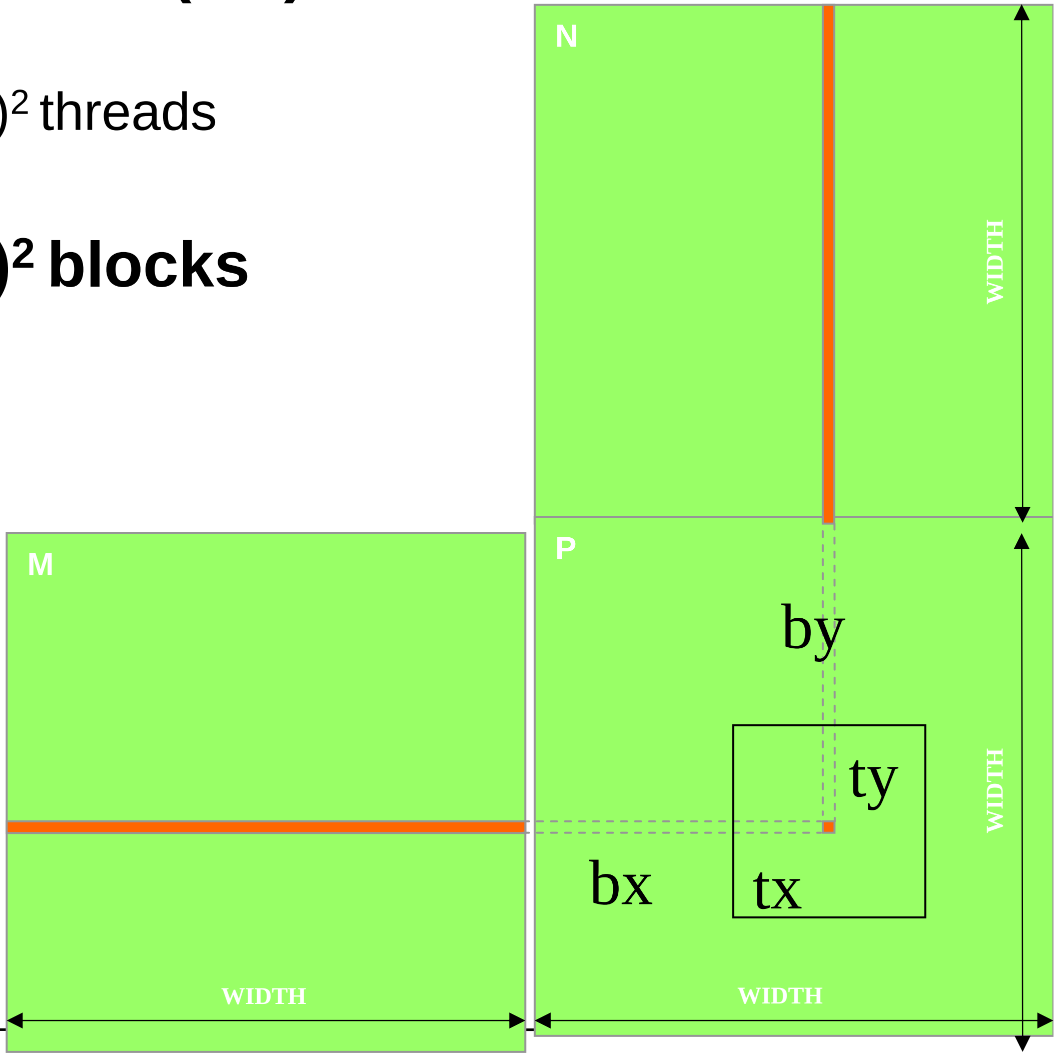
```
// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
   int size = Mhost.width * Mhost.height * sizeof(float);
   cudaMemcpy(Mdevice.elements, Mhost.elements, size,
            cudaMemcpyHostToDevice);
}

// 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);
}
```

Step 6: Handling Arbitrary Sized Square Matrices

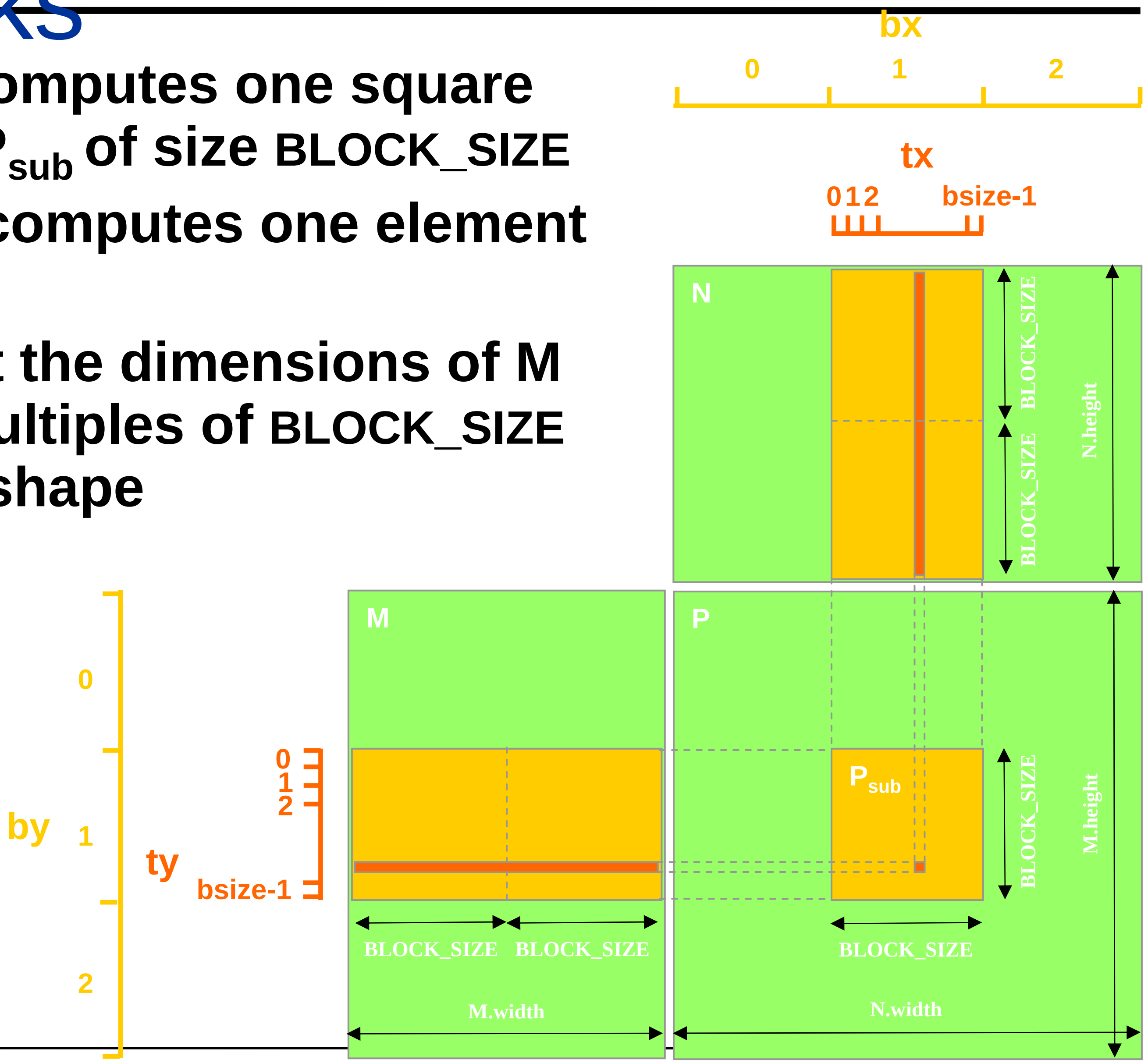
- **Have each 2D thread block to compute a (BLOCK_WIDTH)^2 sub-matrix (tile) of the result matrix**
  - Each has (BLOCK_WIDTH)^2 threads
- **Generate a 2D Grid of (WIDTH/BLOCK_WIDTH)^2 blocks**

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



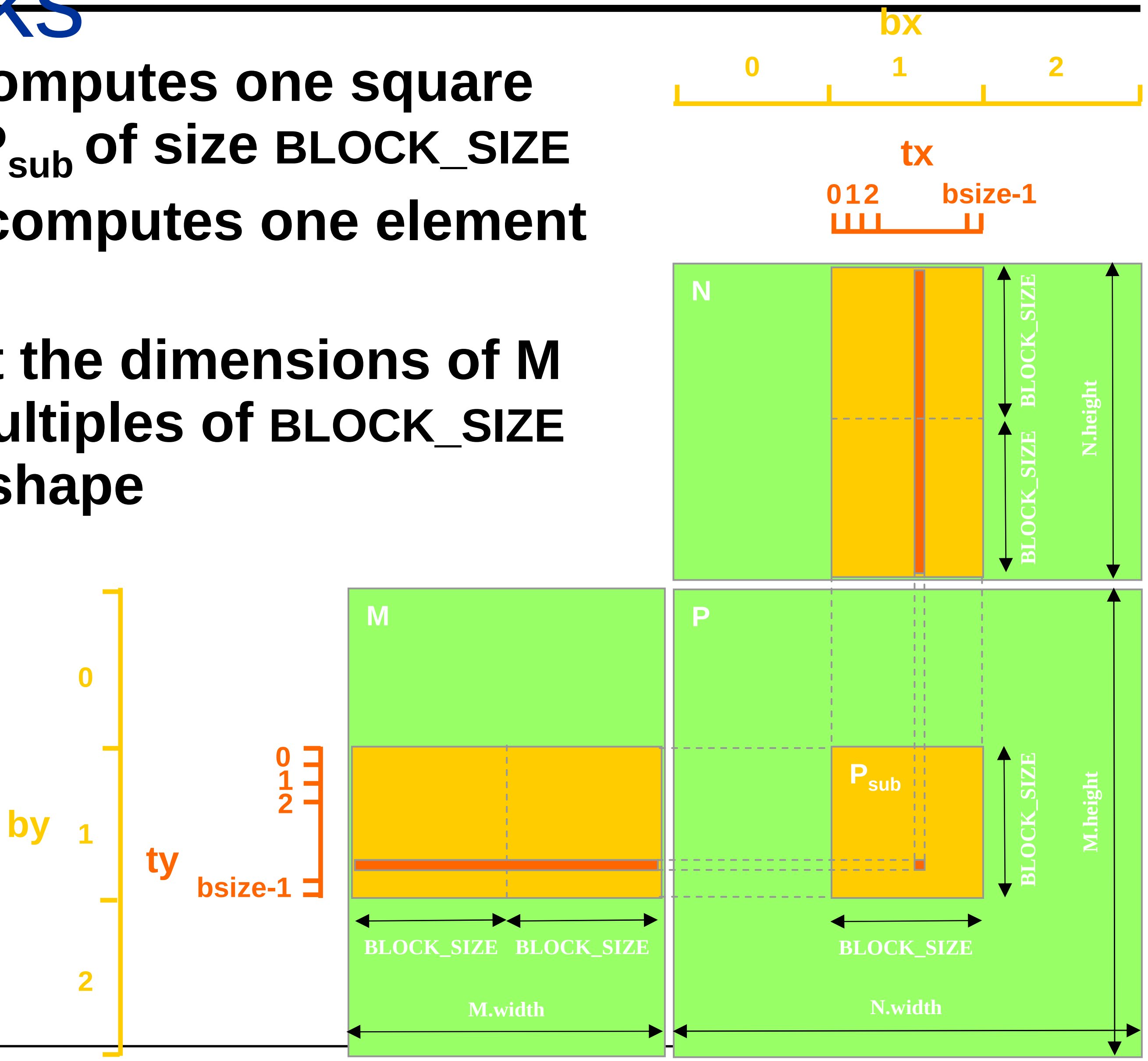
Multiply Using Several Blocks

- One block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- One thread computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape



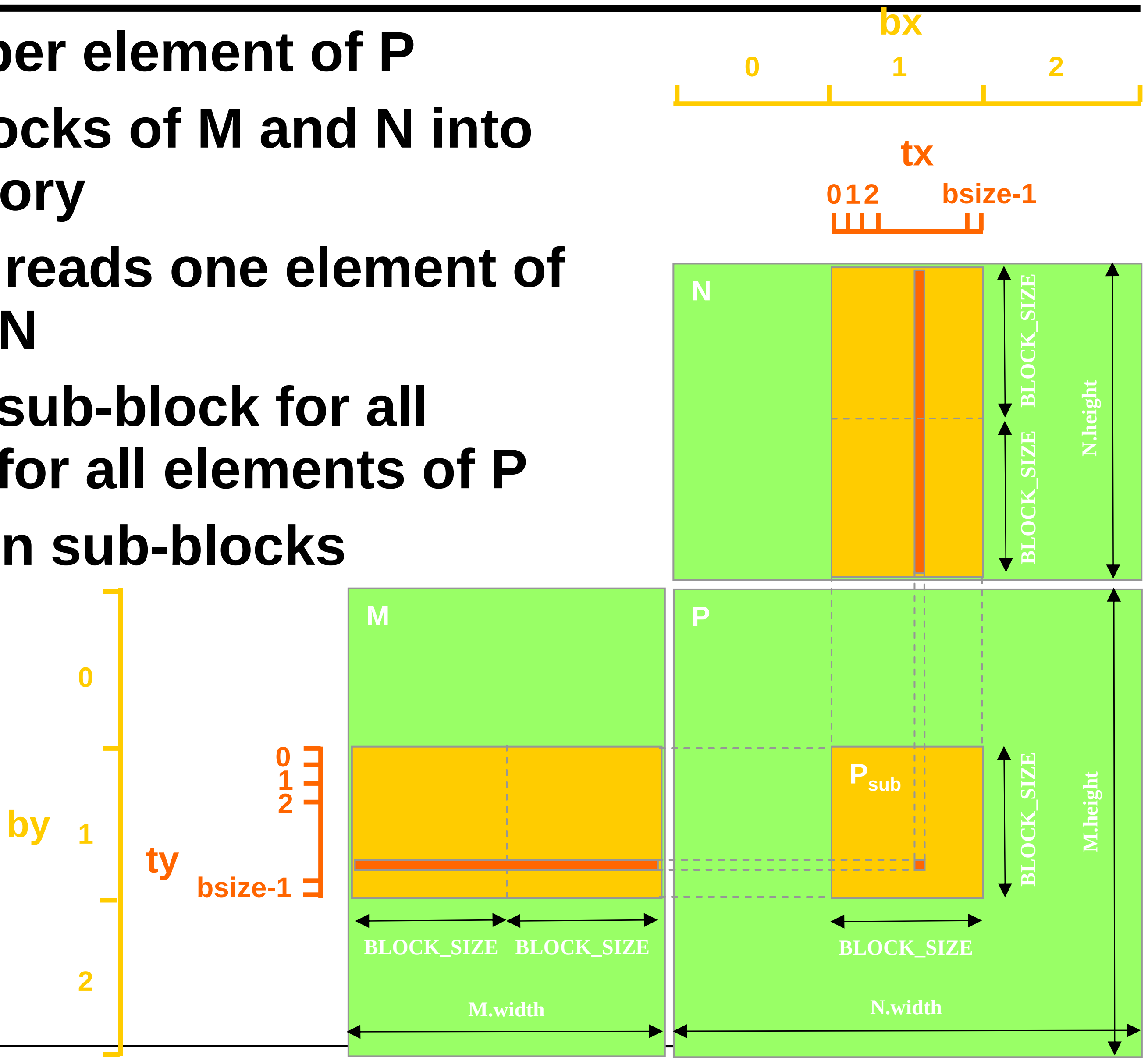
Multiply Using Several Blocks

- One block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- One thread computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape



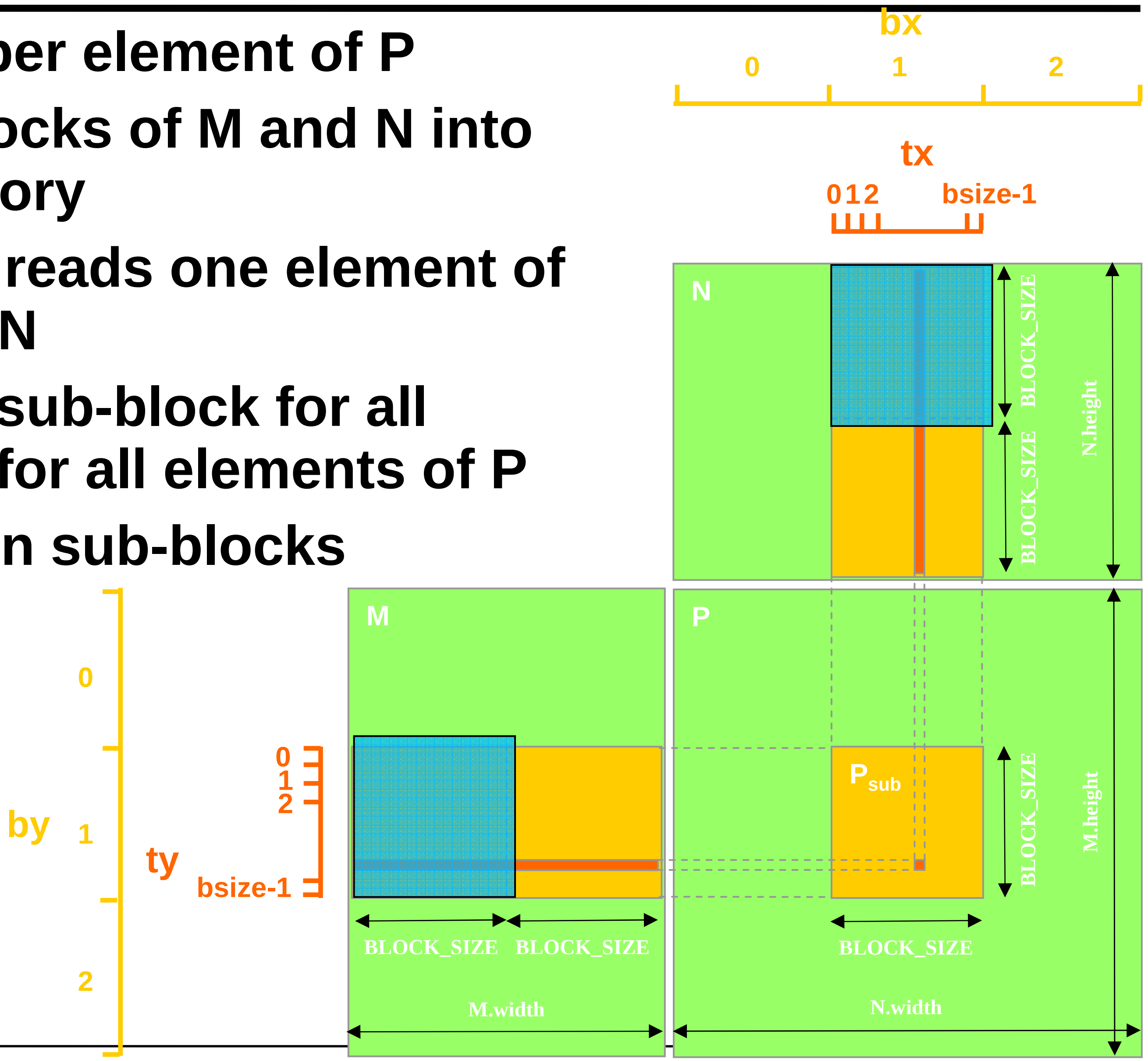
Multiply Using Several Blocks - Idea

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



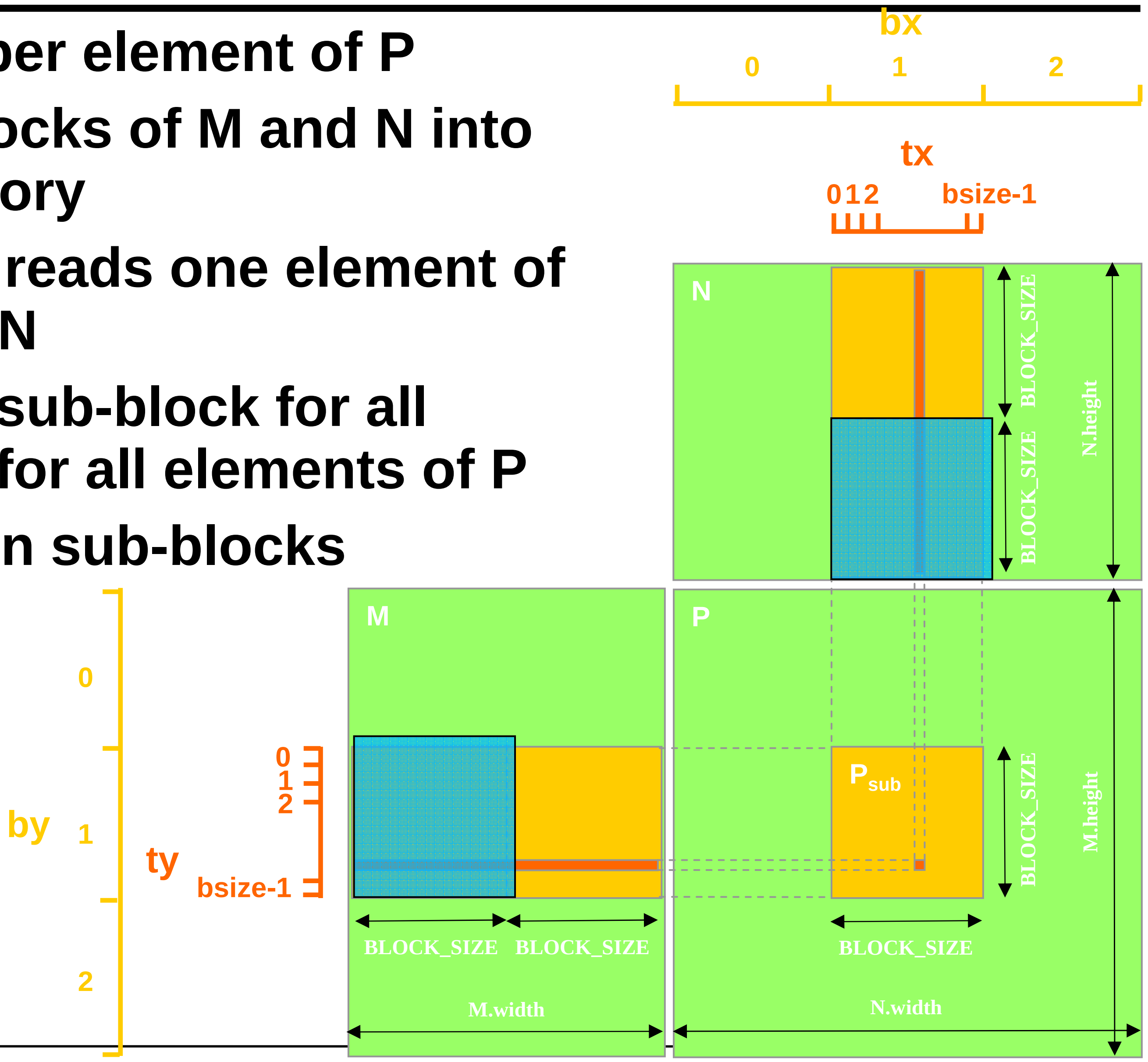
Multiply Using Several Blocks - Idea

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



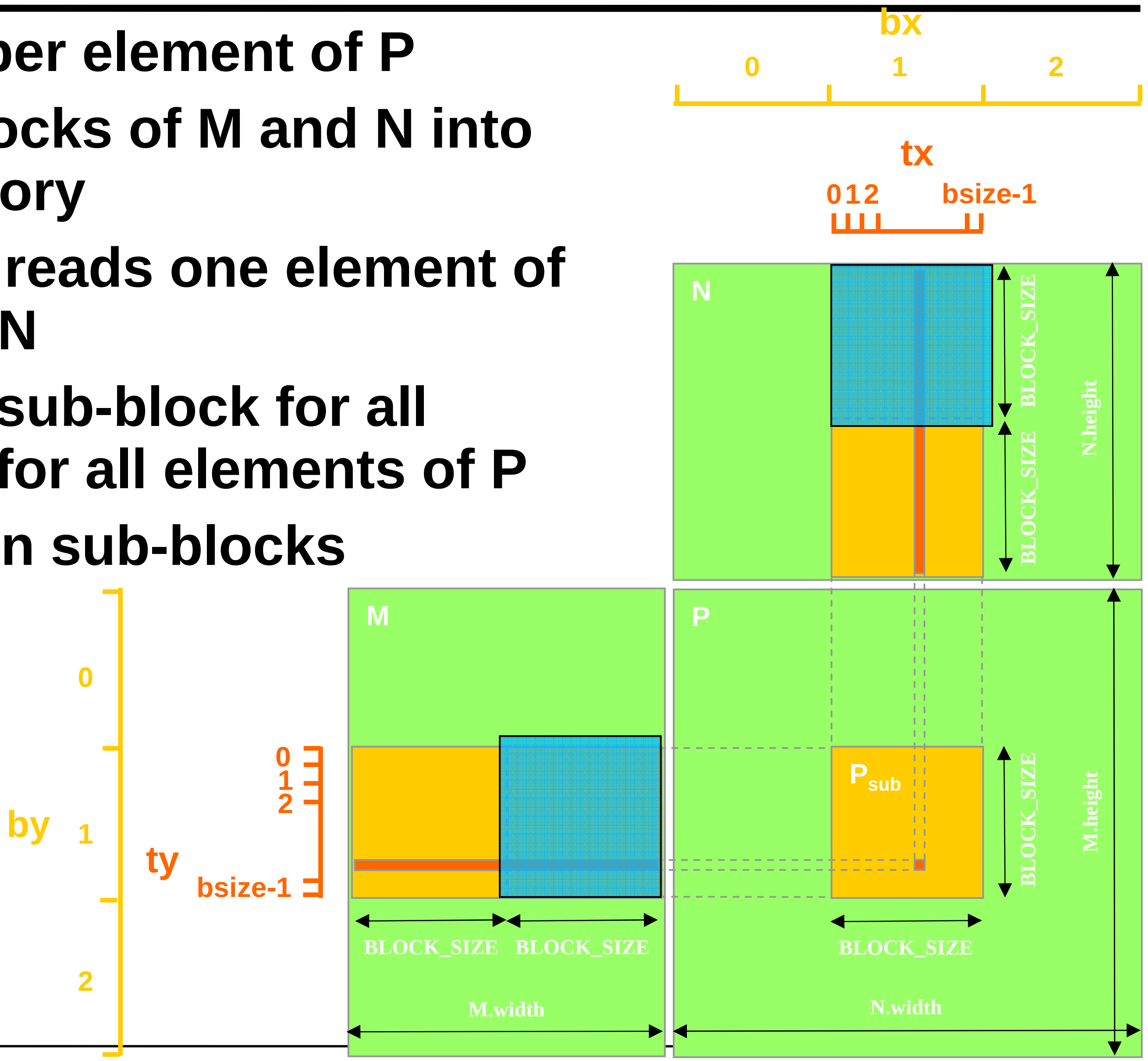
Multiply Using Several Blocks - Idea

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



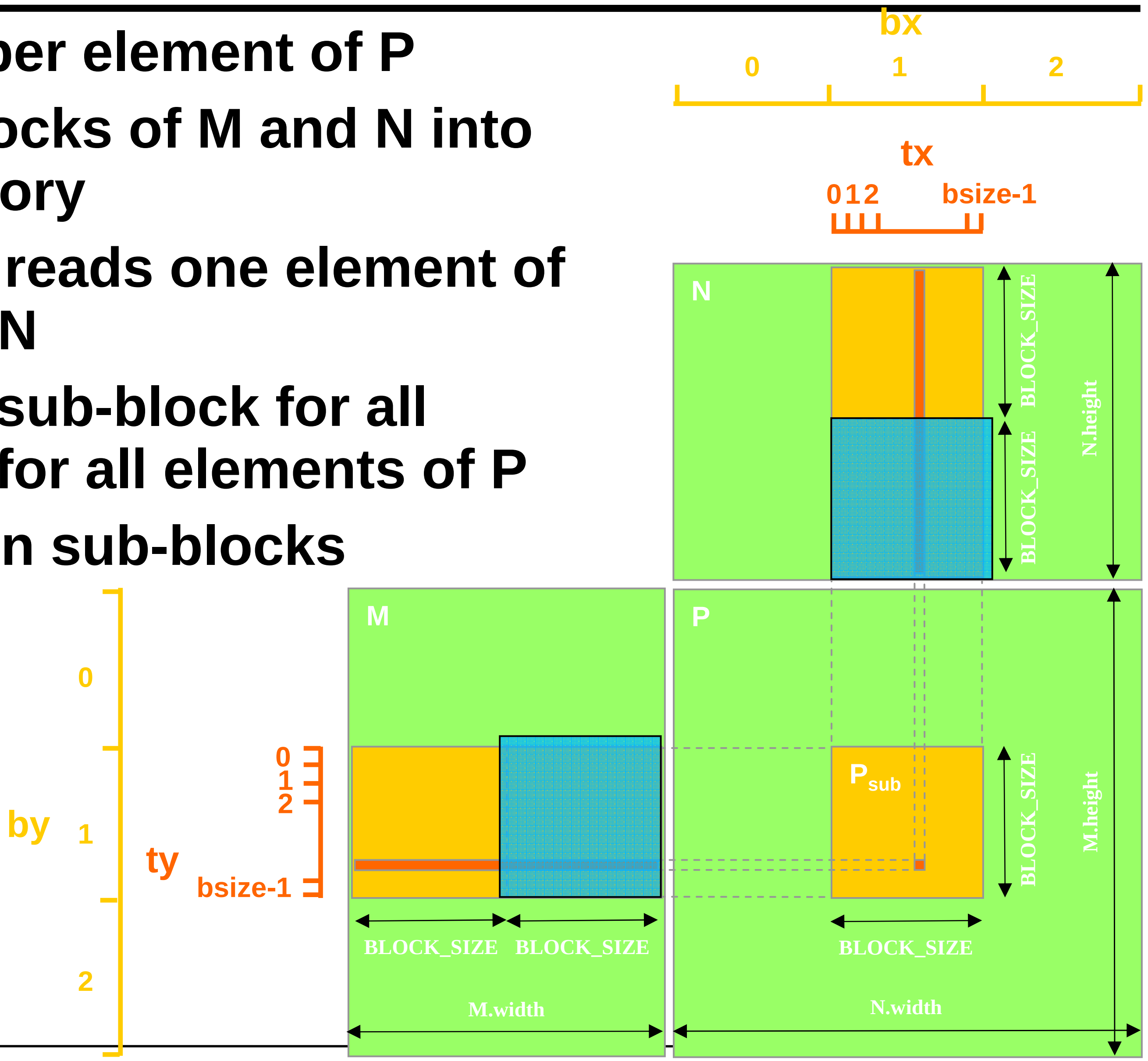
Multiply Using Several Blocks - Idea

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



Matrix Multiplication Kernel with Shared Mem

```
__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
   int bx = blockIdx.x;  int by = blockIdx.y;  //Block index
   int tx = threadIdx.x;  int ty = threadIdx.y; // Thread index

   // Index of the first sub-matrix of A processed by the block
   int aBegin = wA * BLOCK_SIZE * by;
  // Index of the last sub-matrix of A processed by the block
   int aEnd   = aBegin + wA - 1;

   // Step size used to iterate through the sub-matrices of A
   int aStep  = BLOCK_SIZE;
  // Index of the first sub-matrix of B processed by the block
   int bBegin = BLOCK_SIZE * bx;
  // Step size used to iterate through the sub-matrices of B
   int bStep  = BLOCK_SIZE * wB;

   // Csub is used to store the element of the block sub-matrix
   // that is computed by the thread
   float Csub = 0;

   // Loop over all the sub-matrices of A and B
   // required to compute the block sub-matrix
  for (int a = aBegin, b = bBegin;
          a <= aEnd;
          a += aStep, b += bStep) {
    // Declaration of the shared memory array As used to
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load the matrices from device memory to shared
    //  memory; each thread loads one element of each matrix
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];

    __syncthreads(); // to make sure the matrices are loaded

    // Multiply the two matrices together; each thread
    // computes one element of the block sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k)
       Csub += AS(ty, k) * BS(k, tx);

    // Make sure that the preceding computation is done
    // before loading two new sub-matrices of A and B
    __syncthreads();
  }
  // Write the block sub-matrix to device memory;
  // each thread writes one element
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
```