PMPP 2015/16

CUDA Programming (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

GCC Graphics, Capture and Massively Parallel Computing

(Preliminary) Course Schedule

TECHNISCHE UNIVERSITÄT DARMSTADT

you are here →

12.10.2015	Introduction to PMPP
13.10.2015	Lecture CUDA Programming 1
19.10.2015	Lecture CUDA Programming 2
20.10.2015	Lecture CUDA Programming 3
26.10.2015	Introduction Final Projects, Exercise 1 assigned
27.10.2015	Questions and Answers (Q&A)
02.11.2015	Lecture, Final Projects assigned, Ex. 1 due, Ex. 2 assigned
03.11.2015	Questions and Answers (Q&A)
09.11.2015	Lecture, Exercise 2 due
10.11.2015	Lecture
16.11.2015	Questions and Answers (Q&A)
17.11.2015	Questions and Answers (Q&A)
23.11.2015	1st Status Presentation Final Projects
24.11.2015	1st Status Presentation Final Projects (continued)

(Preliminary) Course Schedule

30.11.2015	
01.12.2015	
07.12.2014	
08.12.2014	
14.12.2014	
15.12.2014	
	Christmas Break
11.01.2015	2nd Presentation Status Final Projects (1)
12.01.2015	2nd Presentation Status Final Projects (2)
…	
01.02.2015	
02.02.2015	
08.02.2015	final Presentation Status Final Projects (1)
09.02.2015	final Presentation Status Final Projects (2)

Today's Topics

- review of CUDA and GPU features
- shared memory introduction
- more about the programming model
- extension of simple matrix multiplication example to incorporate shared memory

Review: CUDA Programming Model

- the GPU (graphics processing unit) 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 running 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

Review: Blocks and Grids of Threads

- corresponding code in helloWorld.cu

```
#define BLOCK_SIZE 16
#define GW (5 * BLOCK_SIZE)


dim3 threads(BLOCK_SIZE);
dim3 grid(GW / BLOCK_SIZE);
helloWorld<<<grid, threads>>>();
```

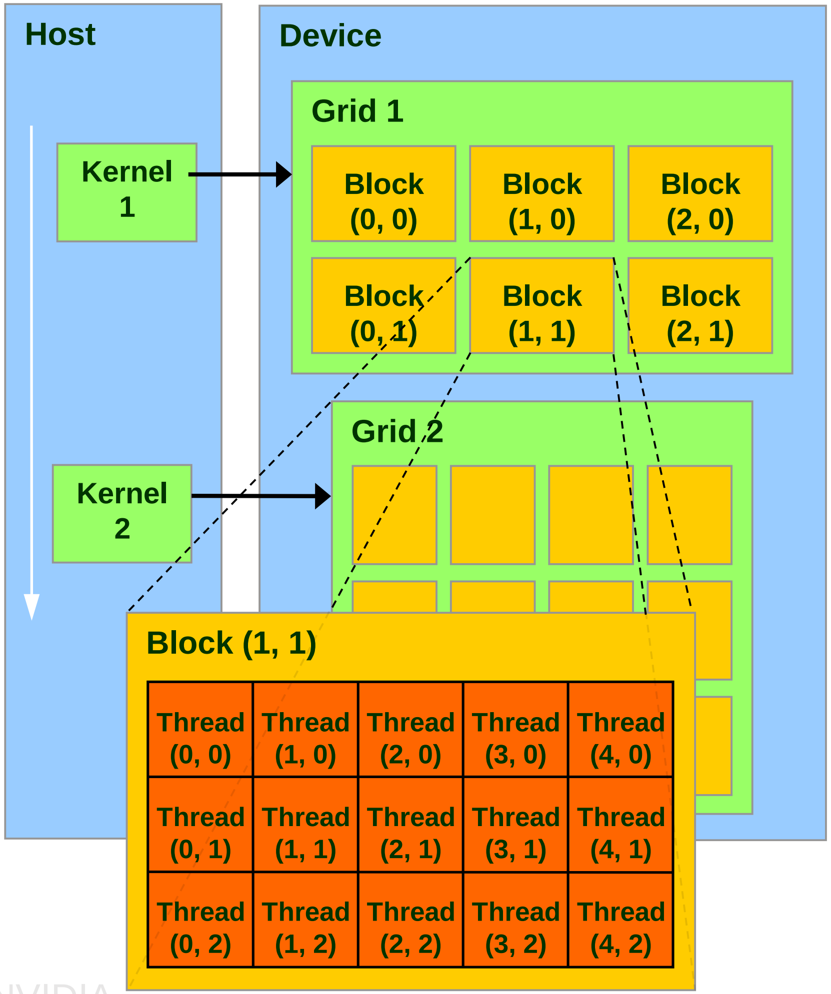
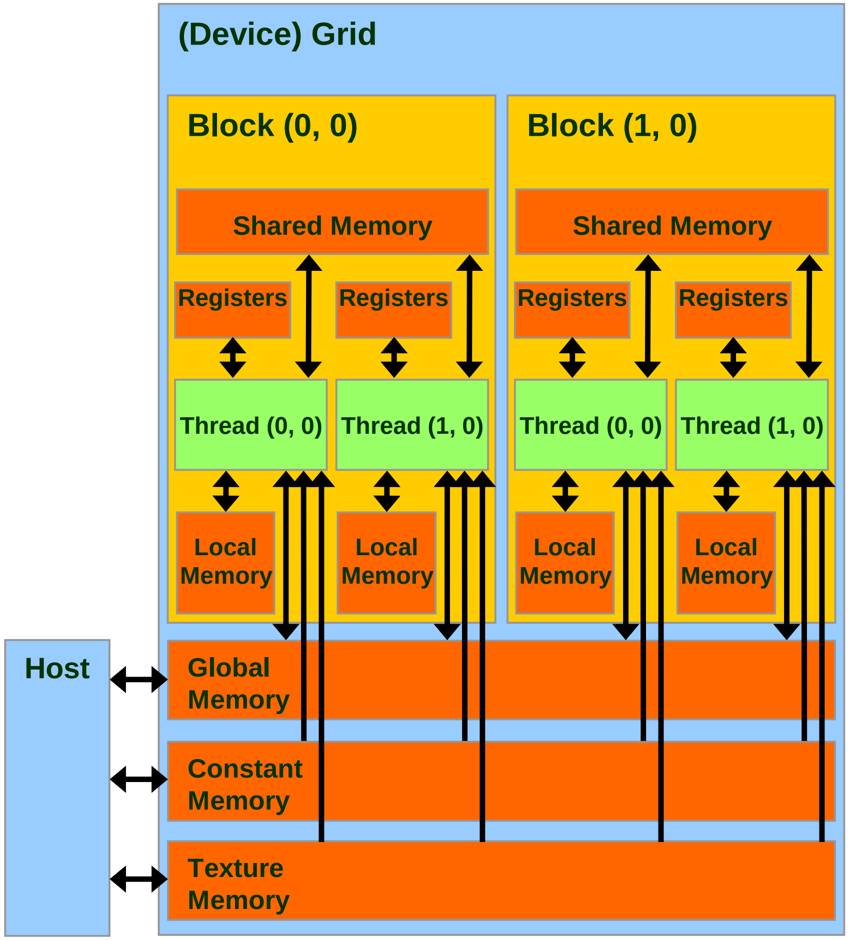


image source: NVIDIA

Review: Different Memory Types

- 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 memory
  - R/W constant memory
  - R/W texture memory



* Can use surface memory for R/W access

image source: NVIDIA

Review: Different Memory Types

- long latency access
  - 400-600 clock cycles
- 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
  - cached access
  - texture has built-in interpolation

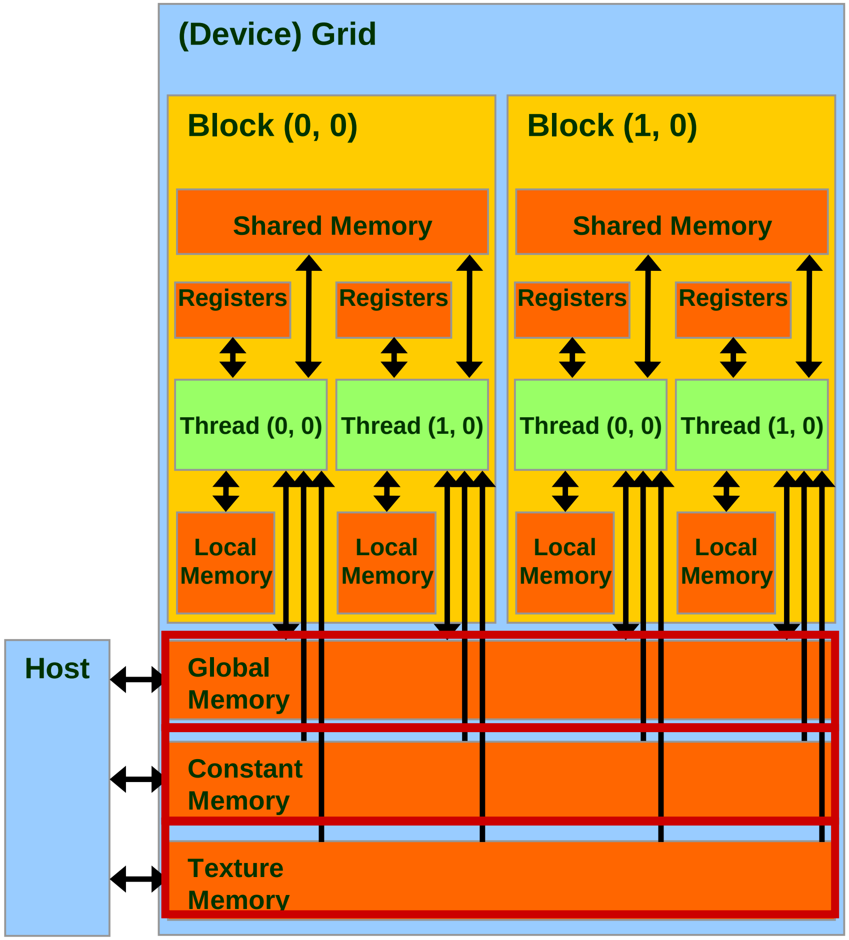


image source: NVIDIA

Review: Simple Matrix Multiplication

- one block of threads compute matrix P
  - one element of P per thread
- each thread
  - loads a row of matrix M
  - loads a column of matrix N
  - perform one multiply and addition per 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

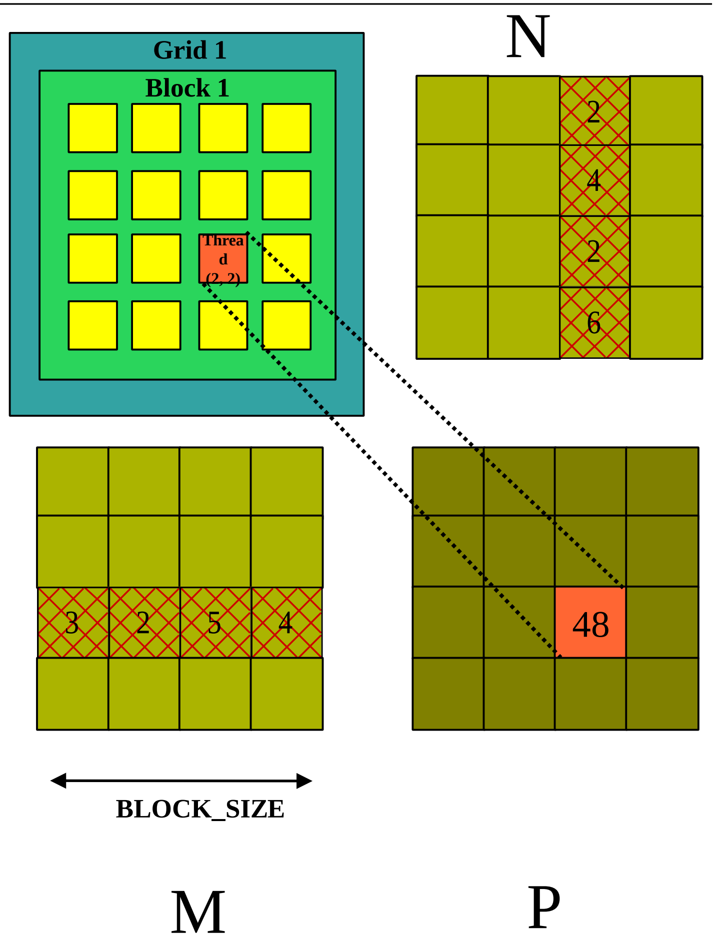


image source: NVIDIA

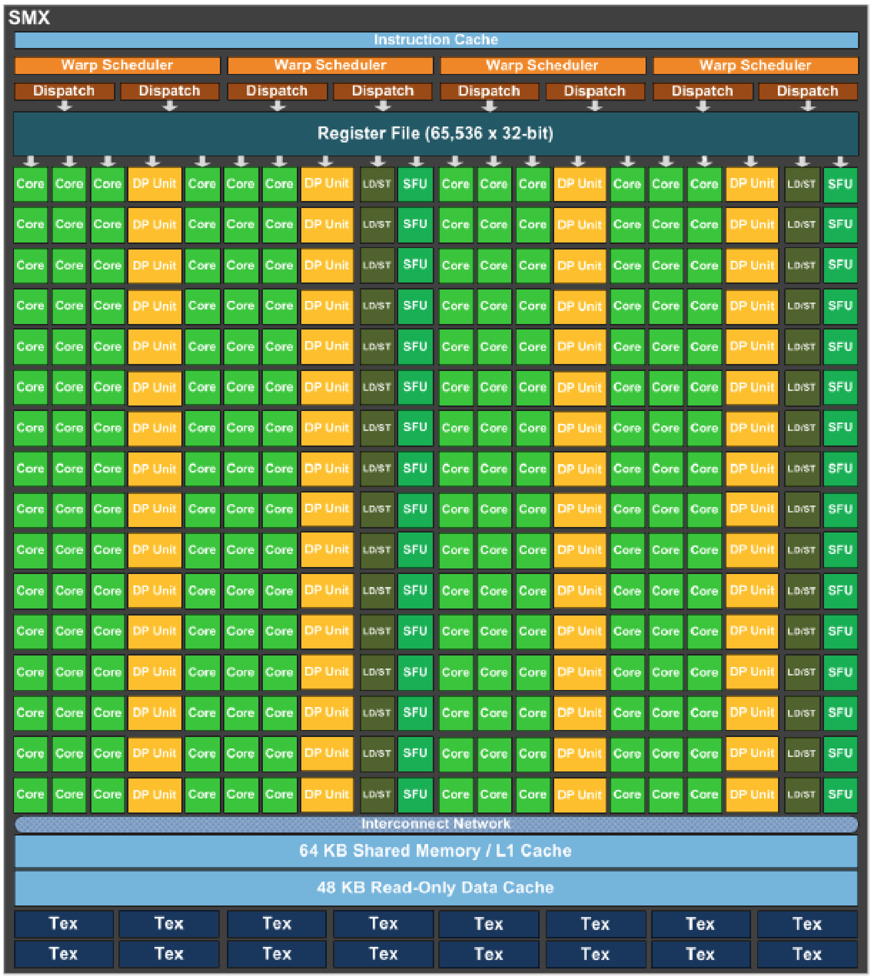
Review: CUDA Function Declarations

- function pointer to `__device__` functions are supported for compute capability 2.x and higher
- for functions executed on the device:
  - recursion only on cards with compute capability 2.x and higher
  - no static variable declarations inside the function
  - no variable number of arguments

GK110 Architecture

image source: NVIDIA

GK110 Thread Computing Pipeline



- Each SMX consists of 192 CUDA cores
  - FP Unit – IEEE 754-2008
  - INT Unit
- 32 Load/Store Units
- 32 Special Function Units
  - sin, cosine, reciprocal, and square root in HW
- 64 Dynamic Parallelism Units

image source: NVIDIA

A Set of SIMT Multiprocessors

- each SMX is a set of CUDA cores with a Single Instruction Multiple Thread (SIMT) architecture
  - shared instruction unit
  - same program counter per set of 32 threads (warp)
  - thread divergence handled in HW

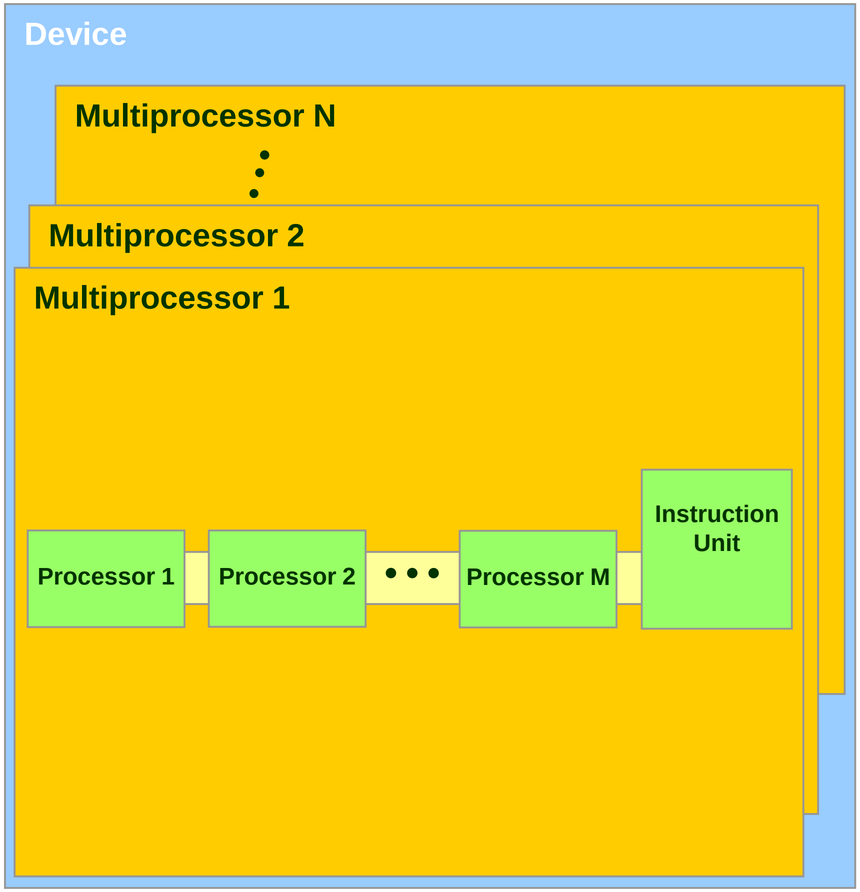


image source: NVIDIA

Memory Architecture (Simplified)

- local, global, constant, and texture spaces are regions of device memory
- each multiprocessor has:
  - a set of 32-bit registers per processor (two registers are combined for double values)
  - on-chip shared memory where the shared memory space resides
  - read-only constant cache to speed up access to the constant memory space
  - read-only texture cache to speed up access to the texture memory space

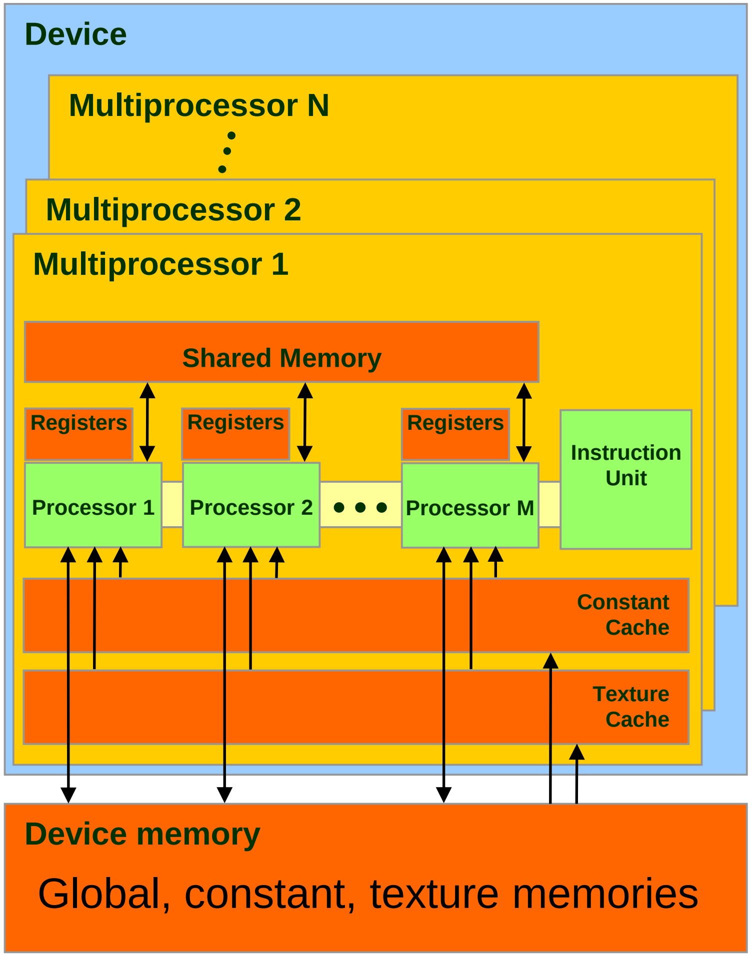


image source: NVIDIA

What is the GPU Good at?

TECHNISCHE UNIVERSITÄT DARMSTADT

- GPU is good at data-parallel processing with high single precision floating point arithmetic intensity
  - the same computation executed on many data elements in parallel – low control flow overhead
  - many calculations per memory access
  - newer generations also have strong double precision capabilities

- 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!

Review: Execution Model

- each thread block of a grid is executed by one streaming multiprocessor
  - shared memory space resides in the on-chip shared memory
- a streaming multiprocessor can execute multiple blocks concurrently
  - shared memory and registers are partitioned among the threads of all concurrent blocks
  - decreasing shared memory usage (per block) and register usage (per thread) increases number of blocks that can run concurrently
- each thread block is split into warps
  - each gets executed by one streaming multiprocessor (SM)

Threads, Warps, Blocks

- there are (up to) 32 threads in a warp
  - only less than 32 when there are fewer than 32 total threads
- there are (up to) 32 warps in a block
- each block (and thus, each warp) executes on a single multiprocessor

- K20X has 14 multiprocessors
- at least 14 blocks required to "fill" the device
  - more is better to hide latencies, …
  - if resources (registers, thread space, shared memory) allow, more than 1 block can occupy each multiprocessor

➔ actual numbers may vary, see https://developer.nvidia.com/cuda-gpus

More Terminology Review

- device = GPU = set of multiprocessors
- multiprocessor = set of processors & shared memory
- kernel = GPU program
- grid = array of thread blocks that execute a kernel
- thread block = group of SIMT threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	Yes (GF100 and newer)	Read/write	One thread
Shared	On-chip	N/A – resident	Read/write	All threads in a block
Global	Off-chip	Yes (GF100 and newer)	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Access Times

- register, shared memory
  - On-chip, single cycle
- local memory, global memory
  - DRAM, cached
  - slow when cache-miss (400 – 800 cycles)
  - single cycle when cache hit
- constant memory, texture memory
  - DRAM, cached, 1…10s…100s of cycles
  - depending on cache locality

Common Programming Pattern

- local and global memory reside in device memory (DRAM)
  - much slower access than shared memory
- a profitable way of performing computation on the device is to block data to take advantage of fast shared memory
  - partition data into subsets that fit into shared memory
  - handle each data subset with one thread block

Common Programming Pattern

- general approach
- load a data subset from global memory to shared memory
  - using multiple threads to exploit memory-level parallelism
- perform the computation on the subset from shared memory
  - each thread can efficiently multi-pass over any data element
- copying results from shared memory to global memory

Application Programming Interface

- the API is an extension to the C programming language consisting of:

- language extensions
    - to target portions of the code for execution on the device

- 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

Language Extensions: Variable Type Qualifiers

	Memory	Scope	Lifetime
__device__ int GlobalVar;	global	grid	application
__shared__ int SharedVar;	shared	block	block
__constant__ int ConstantVar;	constant	grid	application

- automatic variables without any qualifier reside in a register
  - Warning: array with random access pattern resides in local memory

Caveat: Shared Memory and Volatile

- compiler may re-use read results from shared memory unless a **`__syncthread()`** is executed
- if other threads modified the shared memory location, this will not be visible unless variable is marked volatile (or you sync the threads):
  `volatile __shared__ int foo;`

```
// myArray is an array of non-zero integers // located in global or shared memory
__global__ void MyKernel(int *result) {
    int tid = threadIdx.x;
    int ref1 = myArray[tid];
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid]; // Will not work as expected!
    result[tid] = ref1 * ref2;
}
```

Variable Type Restrictions

- pointers can point to memory allocated or declared in global or shared memory
  - allocated in the host and passed to the kernel
    ```
    __global__ void KernelFunc(float *ptr)
    ```
  - obtained as the address of a global variable
    ```
    float *ptr = &GlobalVar;
    ```
  - obtained as the address of a shared variable
    ```
    float *ptr = &SharedVar;
    ```

Language Extensions: Built-in Variables

- **`dim3 gridDim;`**
  - dimensions of the grid in blocks (gridDim.z unused on older hardware)
- **`dim3 blockDim;`**
  - dimensions of the block in threads
- **`dim3 blockIdx;`**
  - block index within the grid
- **`dim3 threadIdx;`**
  - thread index within the block

Application Programming Interface

- the API is an extension to the C programming language consisting of:
- language extensions
  - to target portions of the code for execution on the device
- 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

Common Runtime Component (CRC) Built-in Vector Types

- **`[u]char[1..4]`**
- **`[u]short[1..4]`**
- **`[u]int[1..4]`**
- **`[u]long[1..4]`**
- **`float[1..4]`**
- **`double2`** (only available on devices with compute capability >=1.3)
  - structures accessed with x, y, z, w fields

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

- **`dim3`**
  - based on uint3
  - used to specify dimensions

CRC: 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`
- …

- when executed on the host, a given function uses the C runtime implementation if available
- these functions are only supported for scalar types, not vector types

Application Programming Interface

- the API is an extension to the C programming language consisting of:

- language extensions
  - to target portions of the code for execution on the device

- 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

Host Runtime Component (HRC)

- provides functions to deal with:
  - device management (including multi-device systems)
  - memory management
  - error handling

- initialized the first time a runtime function is called

- a host thread can invoke device code on only one device at a time
  - multiple host threads required to efficiently run on multiple devices

HRC: Memory Management

TECHNISCHE UNIVERSITÄT DARMSTADT

- device memory allocation
  - **`cudaMalloc(),cudaFree()`**
- memory copy from host to device, device to host, device to device and host to host
  - **`cudaMemcpy(),cudaMemcpy2D(),cudaMemcpyToSymbol(), cudaMemcpyFromSymbol(), ...`**
- memory addressing
  - **`cudaGetSymbolAddress()`**

Application Programming Interface

- the API is an extension to the C programming language consisting of:

- language extensions
  - to target portions of the code for execution on the device

- 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

Device Runtime Component (DRC): Mathematical Functions

TECHNISCHE UNIVERSITÄT DARMSTADT

- some mathematical functions such as **`sin(x)`** have a less accurate, but faster device-only version (e.g. **`__sin(x)`**)
  - **`__pow`**
  - **`__log`, `__log2`, `__log10`**
  - **`__exp`**
  - **`__sin`, `__cos`, `__tan`**
- can also be enforced via compiler switch -use_fast_math
  - Better to explicitly use the functions above though due to general precision loss when using fast math

DRC: Synchronization Function

- **`void __syncthreads();`**
- synchronizes all threads in a block
- once all threads have reached this point, execution resumes normally
  - warning: all non-terminated threads must reach this point
  - device emulation mode provides error message
- used to avoid RAW/WAR/WAW hazards when accessing shared or global memory
- allowed in conditional constructs only if the conditional is uniform across the entire thread block
  - otherwise not reached by all threads

DRC: Memory Fence

- `void __threadfence();`
- memory fence for access to global and shared memory
- guaranties modification to be visible for all threads
    - Programming Guide B.5 – Memory Fence Functions
- used to avoid RAW/WAR/WAW hazards

- `volatile variable`
- enforce memory read instruction
    - Rarely needed
    - Programming Guide for further details

Compiling, Linking, …

- some general information on workflow
- assumes familiarity with basic C/C++ compilation and linking

Compilation

- any source file containing CUDA language extensions must be compiled with nvcc
- nvcc is a compiler driver
  - works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- nvcc can output:
  - either C code
    - must be compiled with the rest of the application using another tool
  - or object code directly

Linking

- any executable with CUDA code requires two dynamic libraries:
  - the CUDA runtime library `cudart`
  - the CUDA core library `cuda`
  - later versions of CUDA support static libraries

Back to Matrix Multiplication …

- second look on the problem
- higher performance using shared memory implementation to decrease access to global memory

Recall Step 4: Device-side Kernel Function 2

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

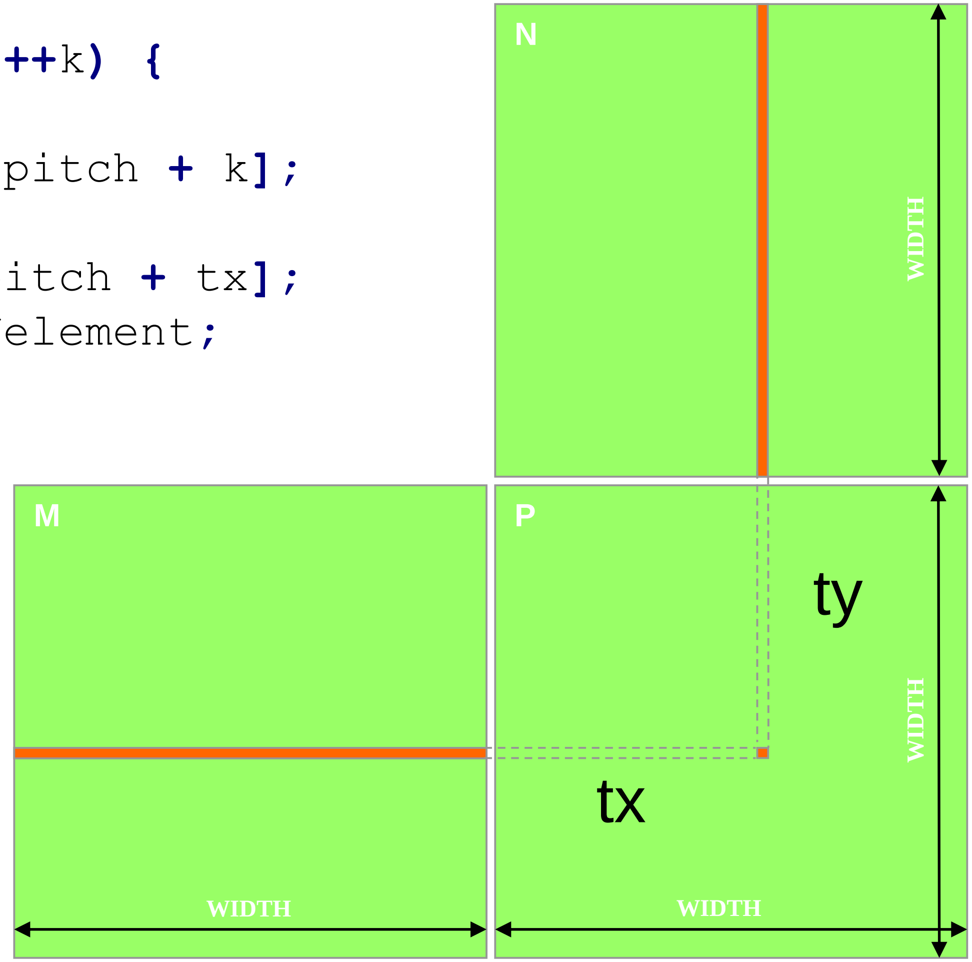


image source: NVIDIA

How about performance on K20X?

- all threads access global memory for their input matrix elements
  - two memory accesses (8 bytes) per floating point fused multiply-add
  - 8B/s of memory bandwidth/2FLOPS
  - 250 GB/s limits the code at 62.5 GFLOPS
- need to drastically cut down memory accesses to get closer to the peak 3.95 TFLOPS

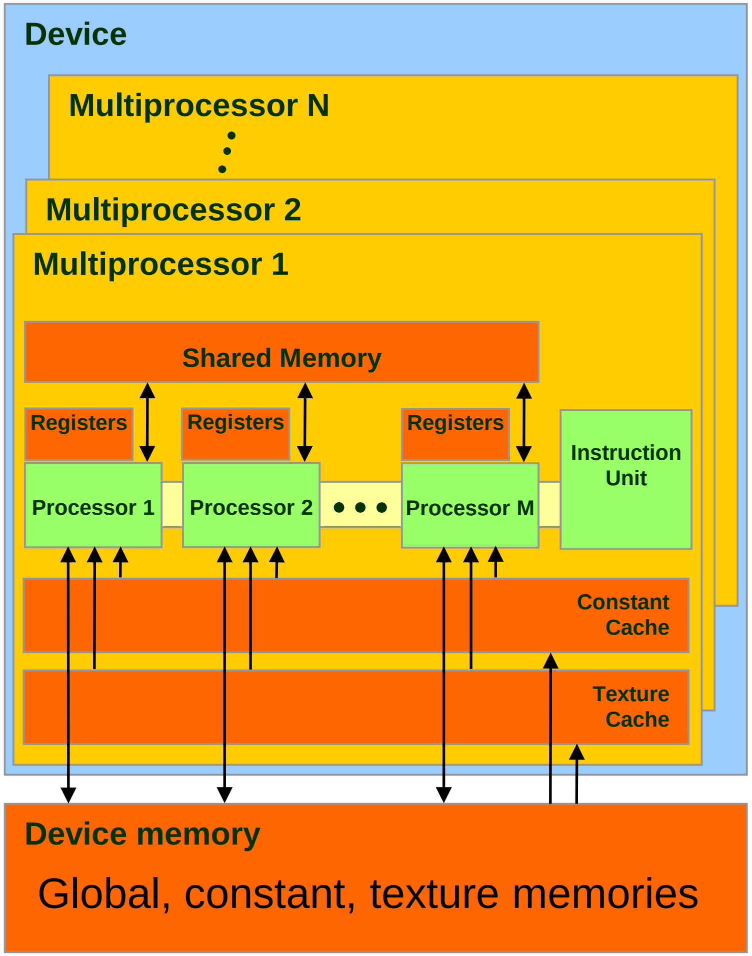


image source: NVIDIA

Idea: Use Shared Memory to Reuse Data from Global Memory

- each input element is read by WIDTH threads.
- if we load each element into shared memory and have several threads use the local version, we can drastically reduce the memory bandwidth
  - tiled algorithms

Tiled Multiply Using Thread 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

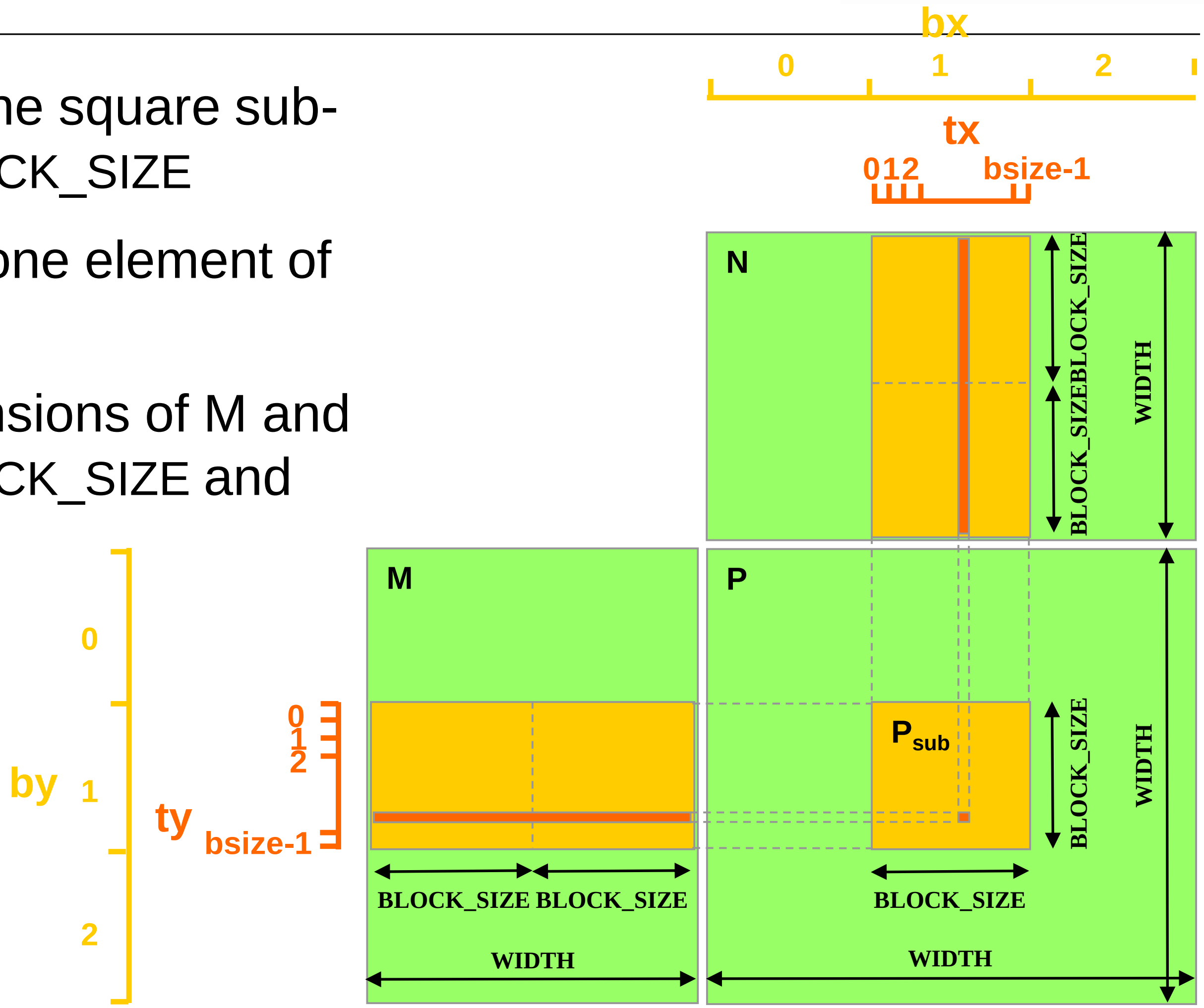


image source: NVIDIA

Shared Memory Usage

- At BLOCK_SIZE 16 each multiprocessor has up to 48KB shared memory and allows 16 active blocks on GK110
  - each thread block uses 2*256*4B = 2KB of shared memory
  - can have up to 24 thread blocks actively executing due to shared memory
  - clamped to 16 because of block limit
  - for BLOCK_SIZE = 16, this allows up to 16*512 = 8K pending loads

- the next BLOCK_SIZE 32 leads to 2*32*32*4B= 8KB shared memory usage per thread block, allowing up to 6 thread blocks active at the same time

Note:
Numbers are different for different shared memory configurations

First-order Size Considerations

TECHNISCHE UNIVERSITÄT DARMSTADT

- each thread block should have a minimum size of 192 threads (usually)
  - BLOCK_SIZE of 16 gives 16*16 = 256 threads
- we need at least 14 thread blocks (on K20X)
  - 1024*1024 P matrix gives 64*64 = 4096 thread blocks
- each thread block performs
  - 2*256 = 512 float loads from global memory
  - for 256 * 16 = 4096 fused mul/add operations.
  - memory bandwidth is a less limiting factor
    - Still only at about 1000 GFLOPS

First-order Size Considerations

- each thread block should have a minimum size of 192 threads (usually)
  - BLOCK_SIZE of 32 gives 32*32 = 1024 threads
- we need at least 14 thread blocks (on K20X)
  - 1024*1024 P matrix gives 32*32 = 1024 thread blocks
- each thread block performs
  - 2*1024 = 2048 float loads from global memory
  - for 1024 * 32 = 32768 fused mul/add operations.
  - memory bandwidth even less limiting factor
    - Still only 2 TFLOPS of the theoretical 3.95 TFLOPS

Kernel Execution Configuration

```
// Setup the execution configuration
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(N.width  / dimBlock.x,
             M.height / dimBlock.y);
```

- for very large N and M dimensions, one will need to add another level of blocking and execute the second-level blocks sequentially
  - see assignment

CUDA Code – Kernel Overview

```
// Block index
int bx = blockIdx.x; int by = blockIdx.y;
// Thread index
int tx = threadIdx.x; int ty = threadIdx.y;

// Pvalue stores the element of the block sub-matrix
// that is computed by the thread
float Pvalue = 0;

// Loop over all the sub-matrices of M and N
// required to compute the block sub-matrix
for (int m = 0; m < M.width / BLOCK_SIZE; ++m) {
    // code from the next few slides
}
```

Tiled Multiply Using Thread 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

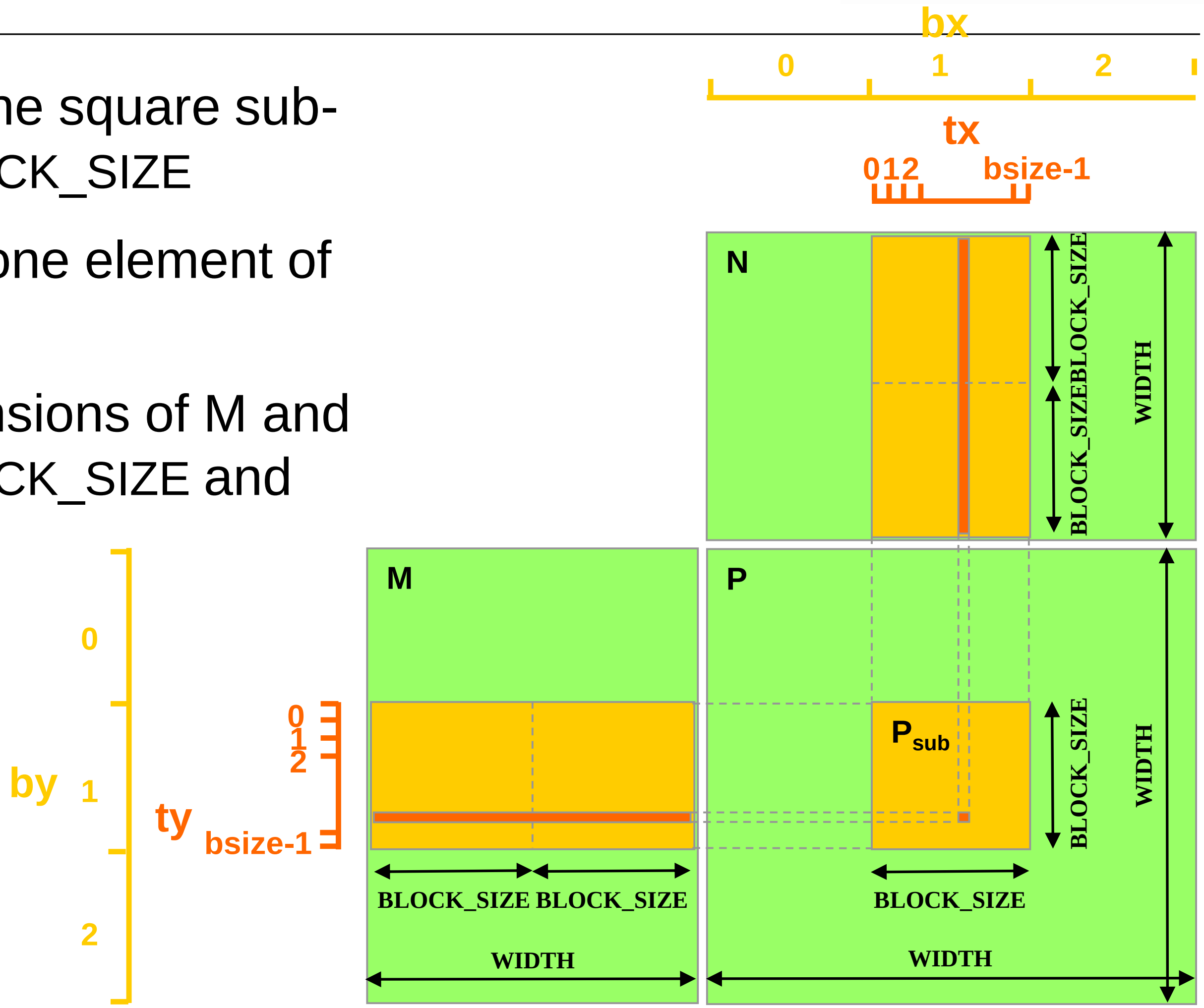


image source: NVIDIA

Load Data to Shared Memory

```
// Get a pointer to the current sub-matrix Msub of M
Matrix Msub = GetSubMatrix(M, m, by);

// Get a pointer to the current sub-matrix Nsub of N
Matrix Nsub = GetSubMatrix(N, bx, m);

__shared__ float Ms[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Ns[BLOCK_SIZE][BLOCK_SIZE];

// each thread loads one element of the sub-matrix
Ms[ty][tx] = GetMatrixElement(Msub, tx, ty);

// each thread loads one element of the sub-matrix
Ns[ty][tx] = GetMatrixElement(Nsub, tx, ty);
```

Tiled Multiply Using Thread 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

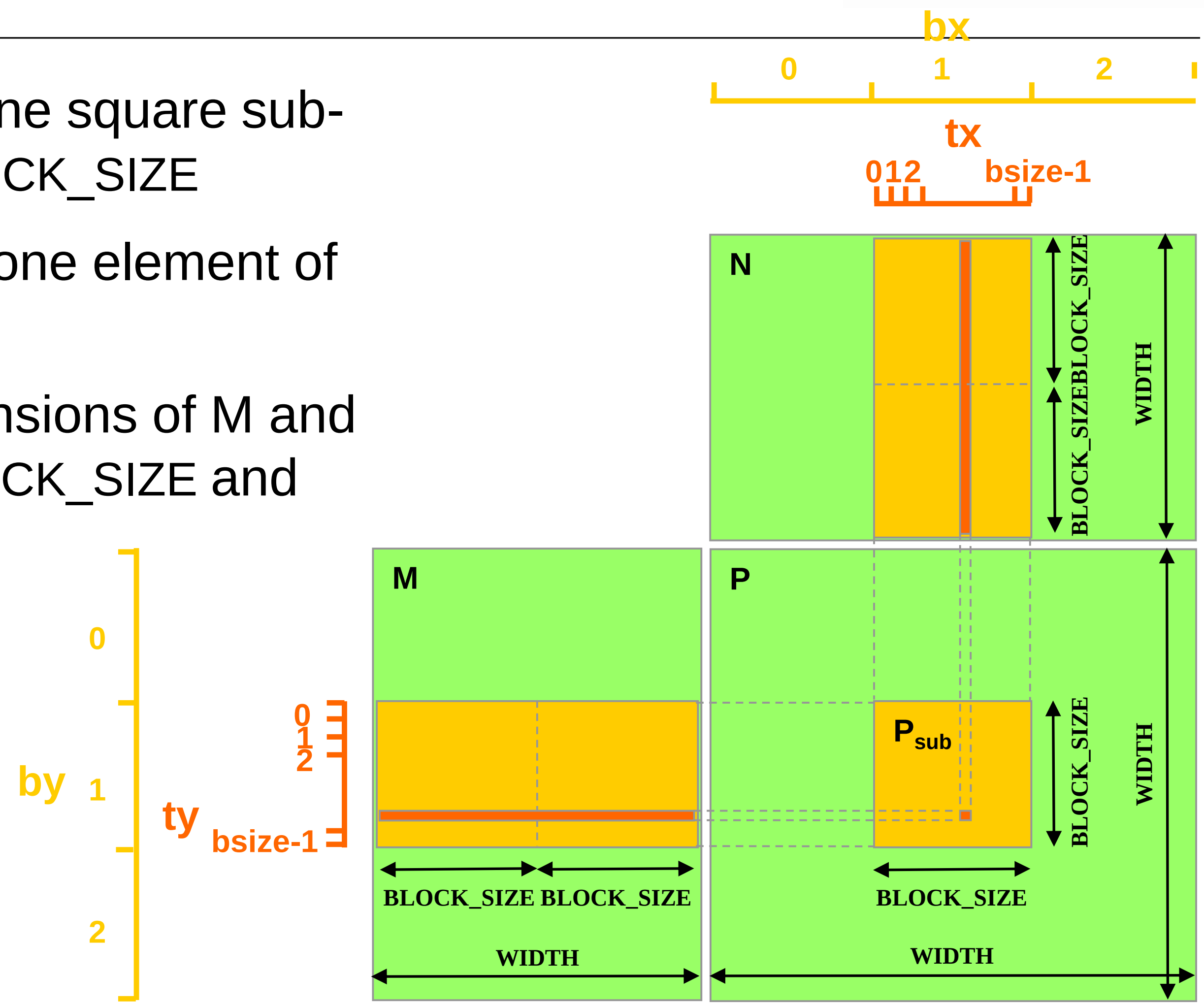


image source: NVIDIA

Compute Result

```
// Synchronize to make sure the sub-matrices are
// loaded before starting the computation
__syncthreads();

// each thread computes one element of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of M and N in the next iteration
__syncthreads();
```

This may cause issues with bank conflicts but more about this tomorrow.

CUDA Code - Save Result

```
// Get a pointer to the block sub-matrix of P
Matrix Psub = GetSubMatrix(P, bx, by);

// Write the block sub-matrix to device memory;
// each thread writes one element
SetMatrixElement(Psub, tx, ty, Pvalue);
```

Assignment

- read Chapters 3 (selected parts), 4+5 of the CUDA Programming Guide (Version 7.5)
  - available online at NVIDIA’s web site and via the course web page
- read Paper
  - Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU