COSC 407
Intro to Parallel Computing

Topic 14: Scheduling, Warps and memory

Outline

Previously:

- Kernel Launch Configuration: nD grids/blocks
- CUDA limits
- Thread Cooperation
- Running Example: Matrix Multiplication

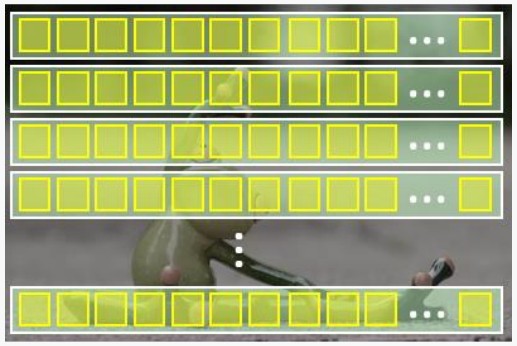
Today:

- Tiling (Improving Performance of Matrix Multiplication)
- CUDA Scalability
- Thread Scheduling on the H/W: Thread Lifecycle
- zero-overhead and latency tolerance
- GPU limits
- CUDA Memories Types (and Performance)

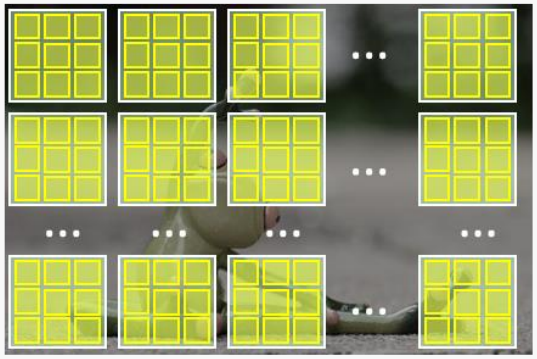
Question

Assume we have a 100x100 image. Which of the following choices is an ***invalid*** choice?

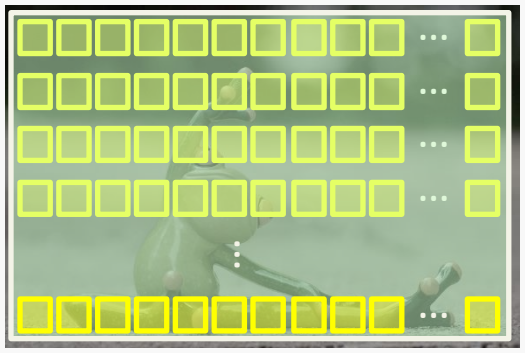
a) Many 1D blocks



b) Many 2D blocks



c) One big 2D block

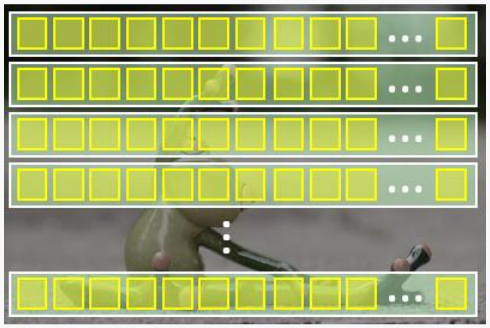


d) all are valid

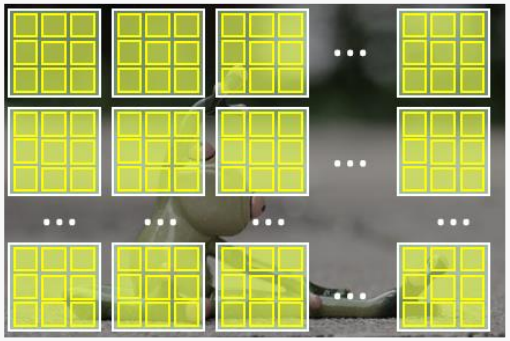
Question

- Assume we have a 32x32 image. Which of the following choices is the ***least*** preferred?

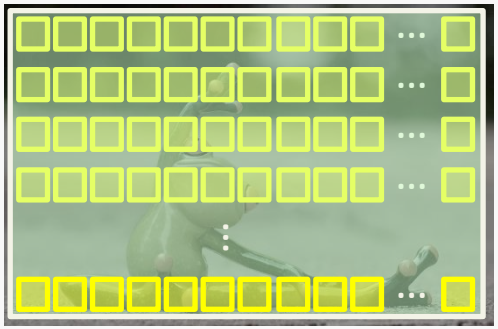
a) Many 1D blocks



b) Many 2D blocks



c) One big 2D block



d) they are all the same

Question

How many stars (*) we will be printed?

```
__global__ void foo(){
     printf("*");
}
 int main(){
     dim3 blockSize(2,2,1);
     foo<<<3, blockSize>>>();
     cudaDeviceSynchronize();
     return 0;
 }
```

```
dim3 blockSize(2,2,1);
foo<<<3, blockSize>>>();
```

A. 1
B. 3
C. 4
D. 12
E. Error

```
foo<<<3, dim3(2,2,1)>>>();
```

Question

How many blocks and threads are used?
*(Note: how many "threads" **in total, not "threads per block"**)*

```
dim3 gridSize(5, 10, 2);
dim3 blockSize(4, 4);
foo<<< gridSize , blockSize >>>(...);
```

A. 100 blocks, 16 threads
B. 100 blocks, 1600 threads
C. 100 blocks, 116 threads
D. 17 blocks, 8 threads
E. 17 blocks, 136 threads

Question

We can use 1,000,000 threads to compute a matrix product of 1000x1000 using the following

```
Multiply<<<1,1000000>>>(…);
```

A. True
B. False

Question

We can use 1 block with with 1,024x1,024 threads in it.

A. True

B. False

Question

We can use 1024 blocks each with 1,024 threads.

A. True
B. False

Question

The data of the matrix d_M is stored in memory as:

```
__global__
void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int
width)
{
        ...
}
```

A. a 2D grid
B. a one dimensional array following the column-major convention
C. a one dimensional array following the row-major convention
D. random memory locations to minimize buffer overflow vulnerabilities

Question

The statement

```
MatrixMulKernel<<<dimGrid,dimBlock>>>(d_M, d_N, d_ P, Width)
```

A. Copies the data of the matrices `d_M`, `d_N`, and `d_P` to the device memory
B. Reserves the memory space on the device
C. Launches a kernel execution on the device
D. Does both (a) and (c)
E. Does all (a), (b), and (c)

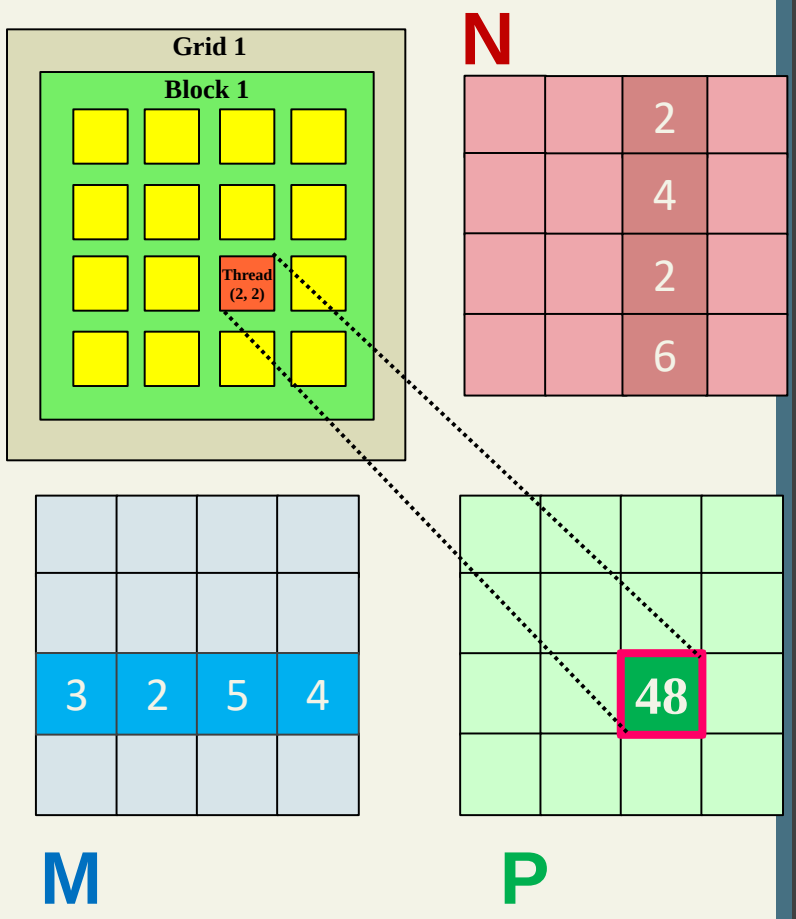
Parallel Code: Using *One* Block

Basic Idea

- Only **ONE** block used to compute the output matrix P
- Each thread computes one element of P as follows:
  - 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 and stores the result on an off-chip memory (DRAM)

Limitation:

- Size of P is limited to 32x32
  - i.e. the number of threads allowed in a thread block.



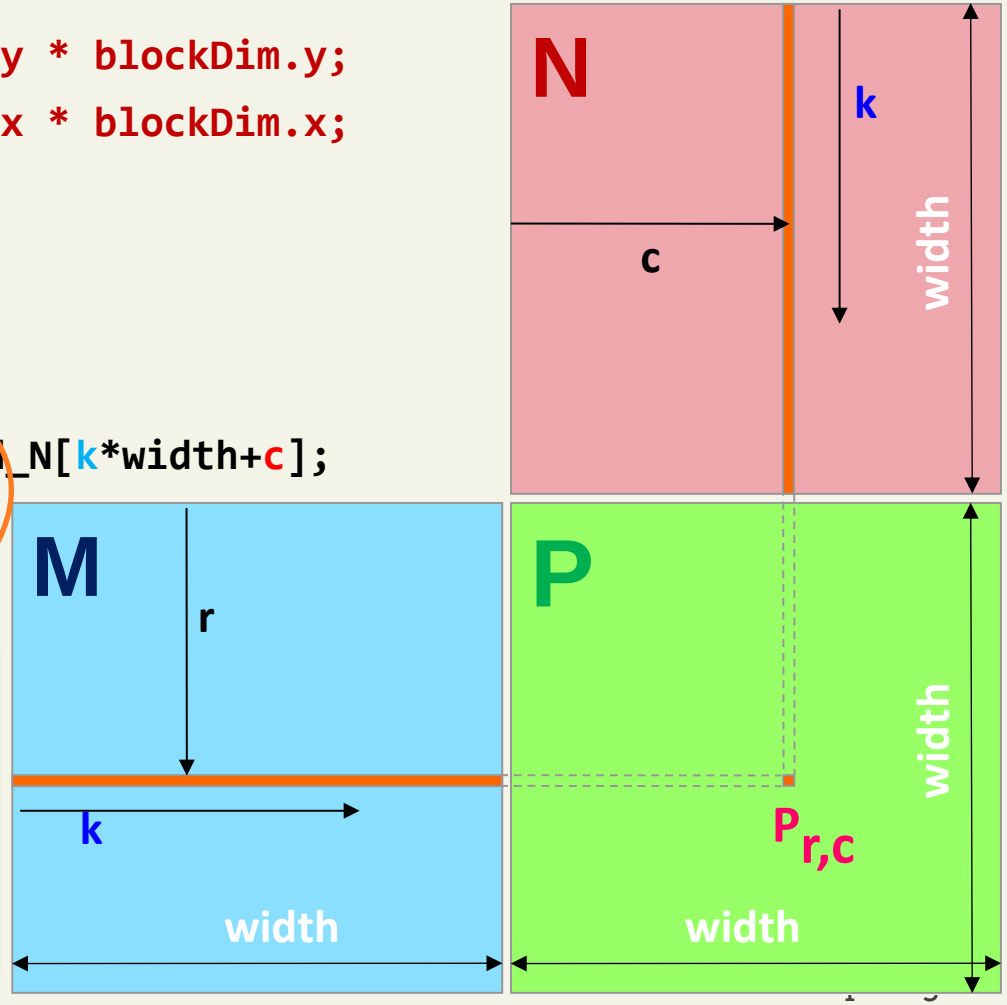
Parallel: Kernel – *One* Block

```
// Matrix multiplication kernel – each thread computes one P element
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int width){
    //find index of P_{r,c} element
    int r = threadIdx.y + blockIdx.y * blockDim.y;
    int c = threadIdx.x + blockIdx.x * blockDim.x;
    //compute P's element
    if(r<width && c<width){
      float value = 0;
      for (int k=0; k<width; k++)
        value += d_M[r*width+k] * d_N[k*width+c];
      d_P[r*width+c] = value;
  }
}
```

Also ok to use

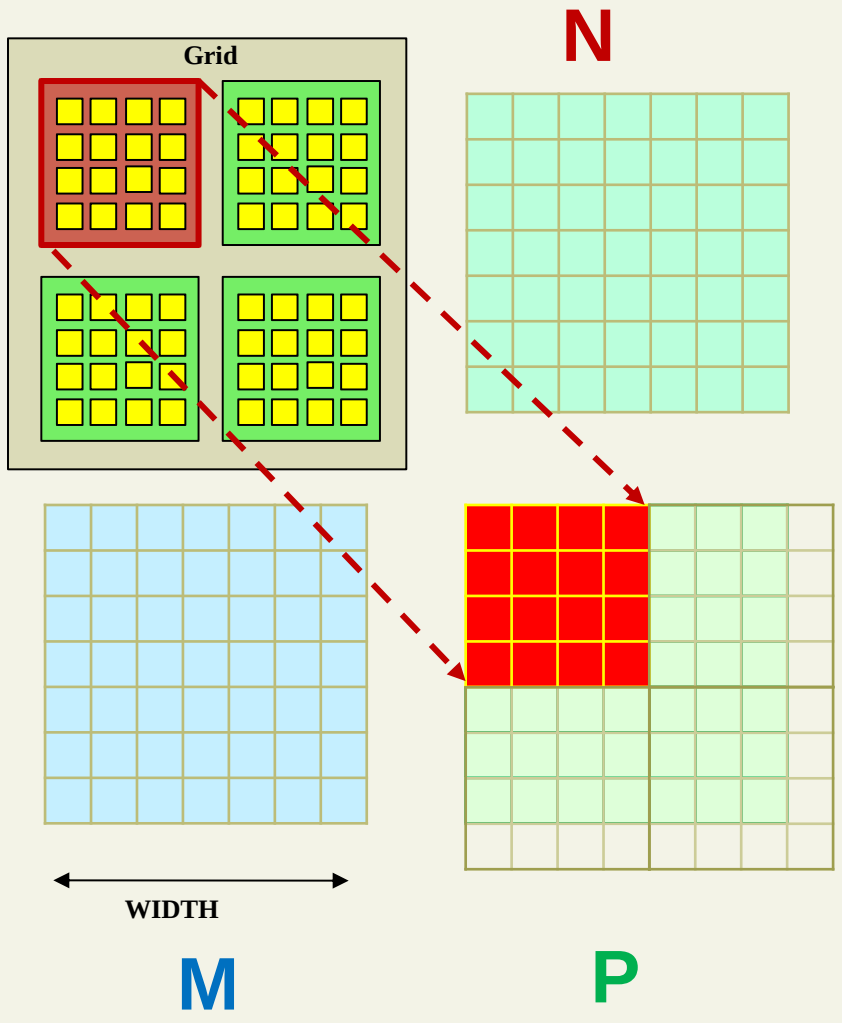
```
int r = threadIdx.y;
int c = threadIdx.x;
```

But it is better to use the general formula (here, we use only one block, and thus blockIdx = 0). ***WHY BETTER?***



Using *Multiple* Blocks

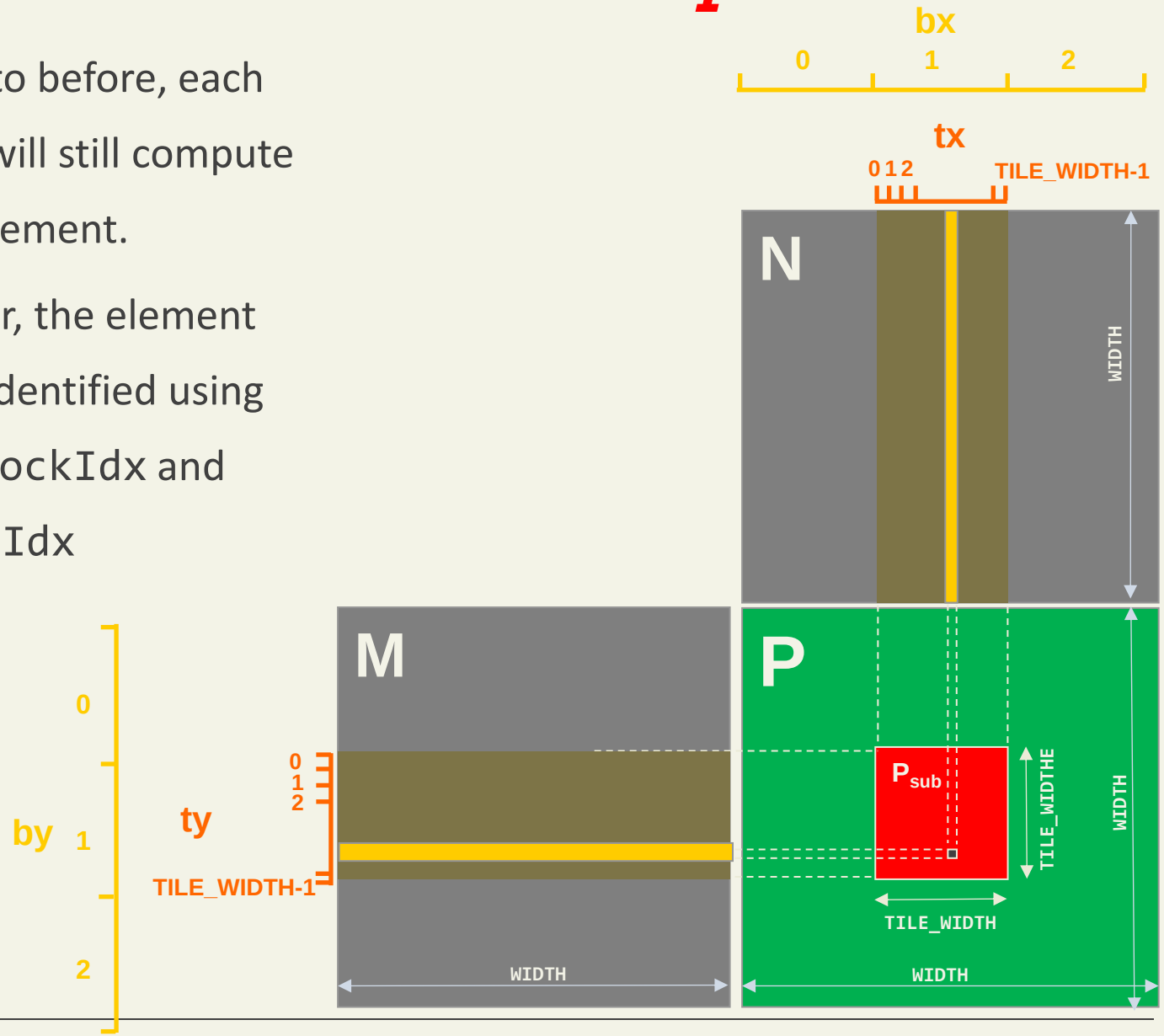
- We saw that using only **one block has a serious limitation**: size of matrix limited by 1024.
- Also, you are not fully using your GPU
- **Solution**: use multiple blocks
  - We shall apply the method explained previously



Slide materials based on, 2007-2010, ECE 408, University of Illinois, Urbana-Champaign
© David Kirk/NVIDIA and Wen-mei W. Hwu

Parallel Code: *Multiple* Blocks

- Similar to before, each thread will still compute one P element.
- However, the element will be identified using both `blockIdx` and `threadIdx`

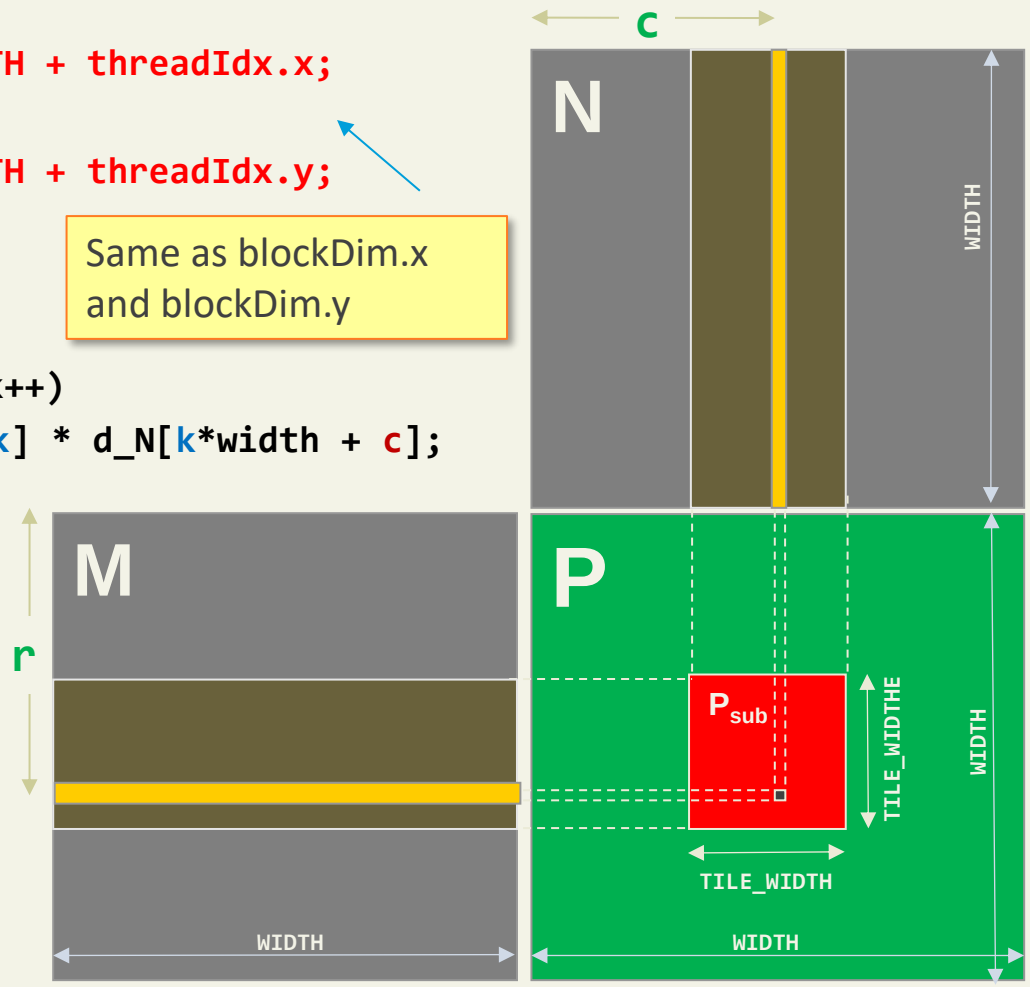


Parallel: Kernel - *Multiple* Blocks

```
__global__ void MatrixMul(float* d_M,float* d_N,float* d_P,int width) {
    //find index of P[r,c] element
    int r = blockIdx.x * TILE_WIDTH + threadIdx.x;
    //TILE_WIDTH = TILE_HEIGHT
    int c = blockIdx.y * TILE_WIDTH + threadIdx.y;
    //compute P[r,c] element
    if(r<width && c<width){
      float value = 0;
      for (int k = 0; k < width; k++)
           value += d_M[r*width + k] * d_N[k*width + c];
      d_P[r*width + c] = value;
    }
}
```

Same as blockDim.x and blockDim.y

The kernel is the same as before.. The difference is in the host code!



Parallel : Host - *Multiple* Blocks

- The host code will be the same as before except that we need to ***setup the kernel launch configuration***

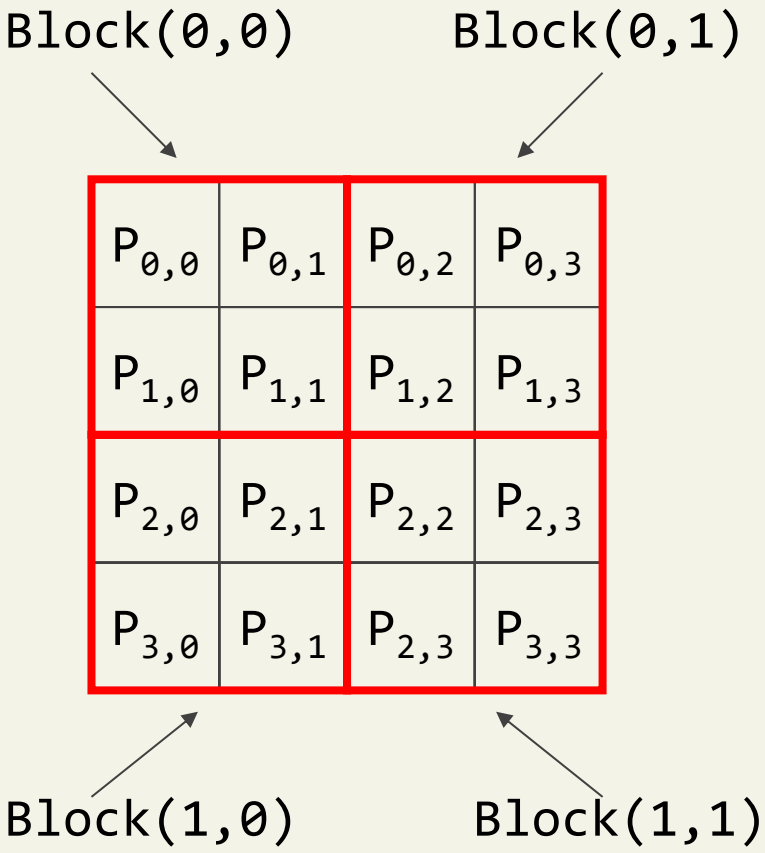
```
//block dimensions - how many threads per block (our choice but must be <1024)
int TILE_WIDTH = TILE_HEIGHT = n;          // e.g. n = 32
dim3 blockSize(TILE_WIDTH, TILE_HEIGHT);
//grid dimensions (how many blocks are required to cover the whole matrix P)
int nblocks_x = 1+(WIDTH-1)/TILE_WIDTH;    // WIDTH = # elements along x
int nblocks_y = 1+(WIDTH-1)/TILE_HEIGHT;   // HEIGHT = WIDTH
dim3 gridSize(nblocks_x, nblocks_y);
//launch the kernel
MatrixMul<<<gridSize, blockSize>>>(d_M, d_N, d_P, WIDTH);
```

Parallel : *Multiple* Blocks - Details

- To illustrate how the algorithm works, assume
  - TILE_WIDTH = blockDim.x = blockDim.y = 2
  - Each block has 4 threads

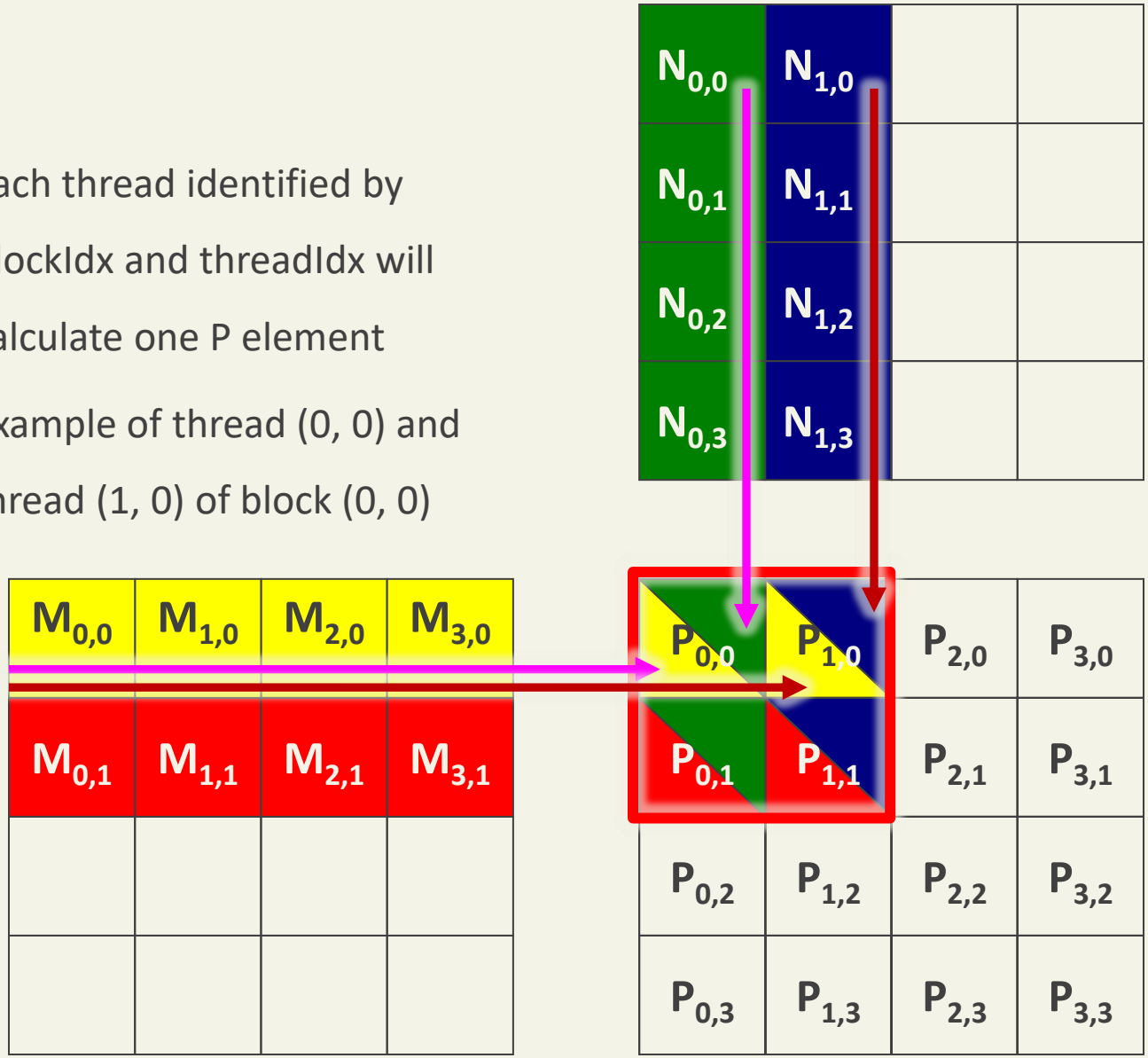
- If WIDTH = 4, how many blocks?
  - WIDTH / TILE_WIDTH = 2
  - Use 2* 2 = 4 blocks

- How to identify element $P_{x,y}$?
  - x = TILE_WIDTH * bx + tx
  - y = TILE_WIDTH * by + ty



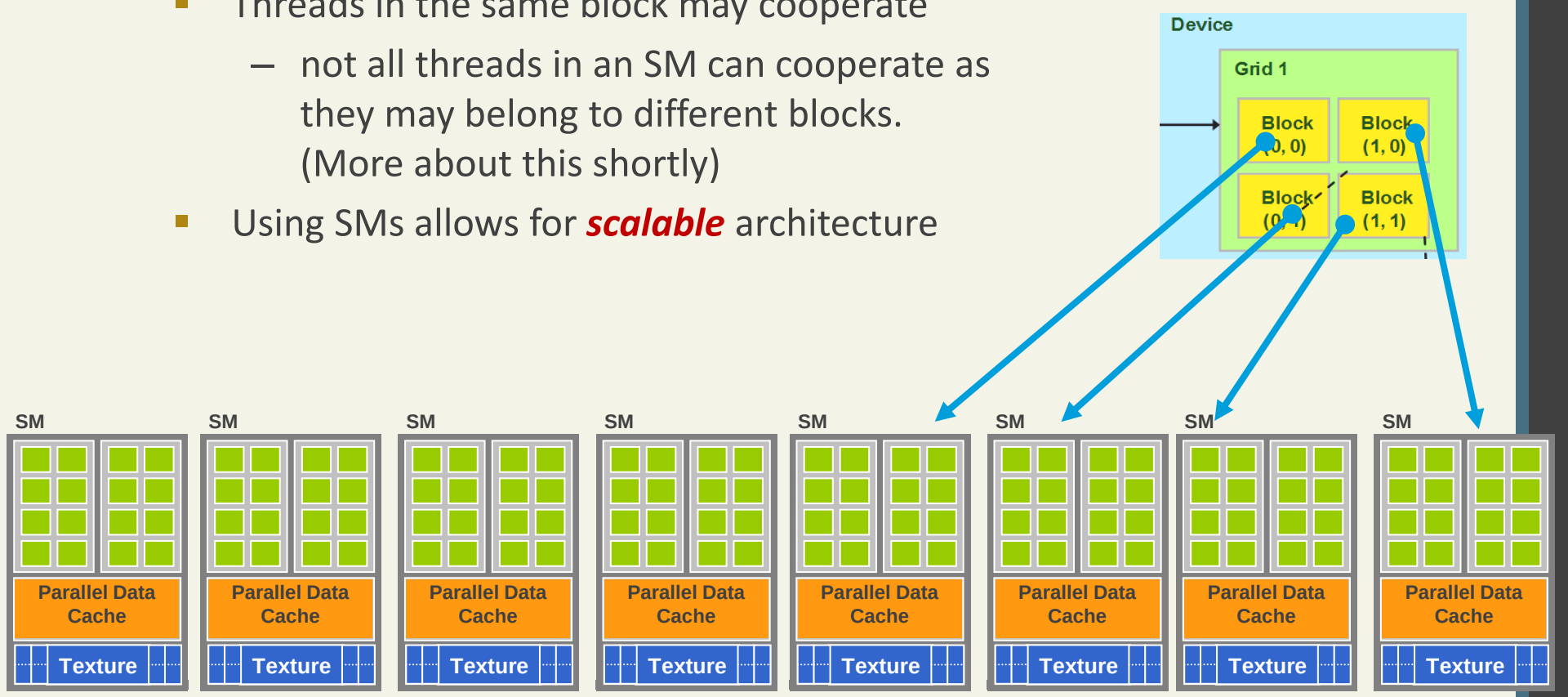
Parallel : *Multiple* blocks - Details

- Each thread identified by blockIdx and threadIdx will calculate one P element
- Example of thread (0, 0) and thread (1, 0) of block (0, 0)



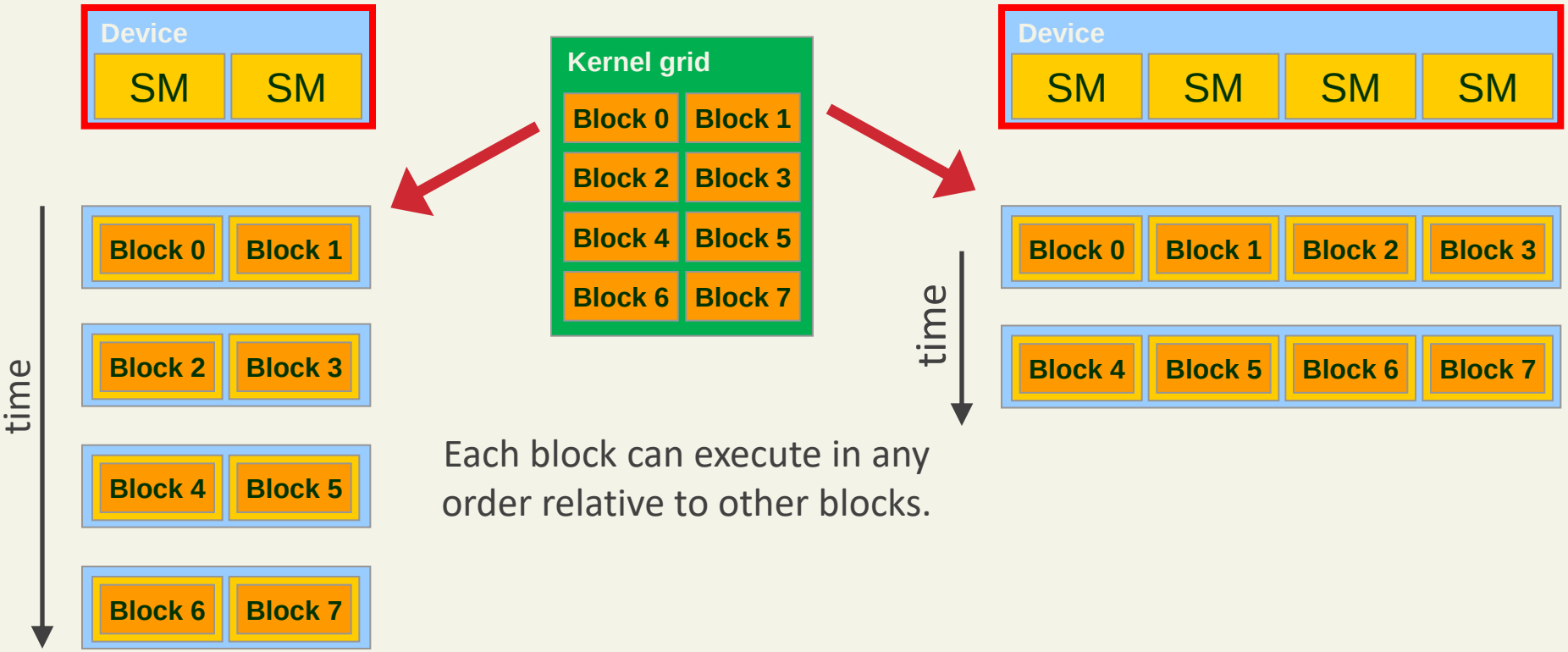
Transparent Scalability

- The GPU is responsible for assigning thread blocks to SMs
  - A block must be assigned to exactly one SM.
  - An SM can run more than one thread block
- Threads in the same block may cooperate
  - not all threads in an SM can cooperate as they may belong to different blocks. (More about this shortly)
- Using SMs allows for ***scalable*** architecture



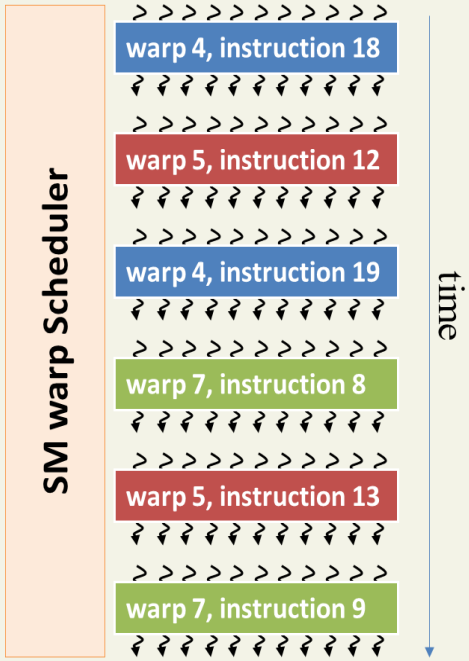
Transparent Scalability

- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors



Warps

1. Blocks are assigned to SMs (as explained before)
2. Each SM splits threads in its blocks into **Warps**.
   - Groups of threads known as warps in SIMT fashion (execute same instruction)
   - Warps are the **scheduling units of SM**
   - **Thread IDs** within a warp are consecutive and increasing:
     - Warp 0 starts with Thread ID 0
   - Size of the warp is implementation specific
     - Generally **# of threads in a warp (32) = # of SPs** in SM
   - The warp scheduler of SM decides which of the warp gets prioritized during issuance of instructions.

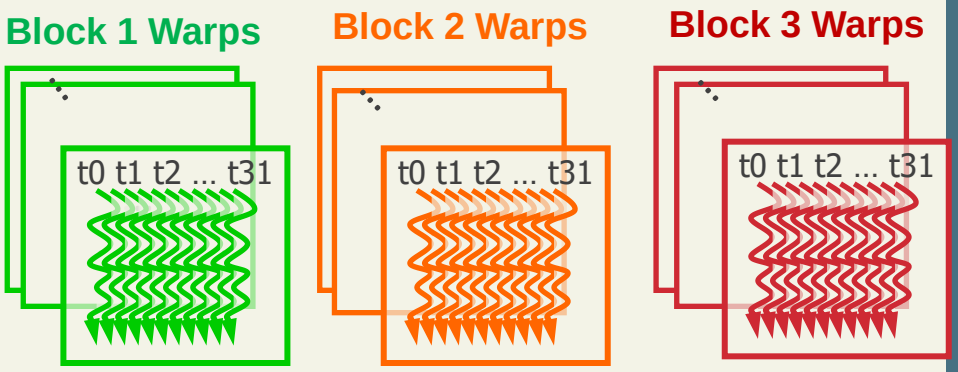


- **DO NOT rely on any ordering between warps**
  - If there are any dependencies between threads, you must synchronize them to get correct results (more on this later).

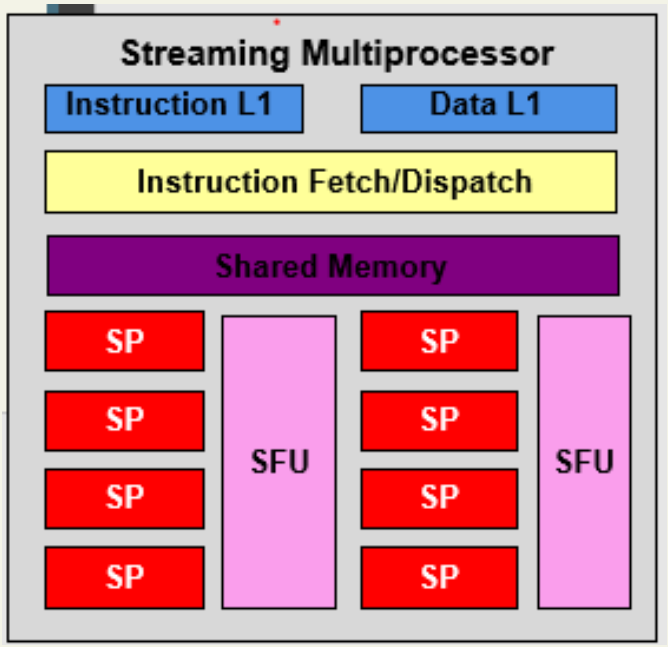
- Warps are not part of the CUDA specification, but
  - Can help optimize the performance in particular devices (discussed later)

Thread Scheduling

- **Each block** is executed as subsequent Warps
  - All threads in a single warp execute in parallel
  - A warp in an SM runs in parallel with Warps in other SMs



- **Question**: Consider a GPU with warp = 32 threads
  - if 3 blocks are assigned to an SM and each block has 256 threads, **how many Warps are there in an SM**?
  - Each Block is divided into 256 / 32 = 8 Warps
  - There are 8 * 3 = 24 Warps



Thread Life Cycle on the HW

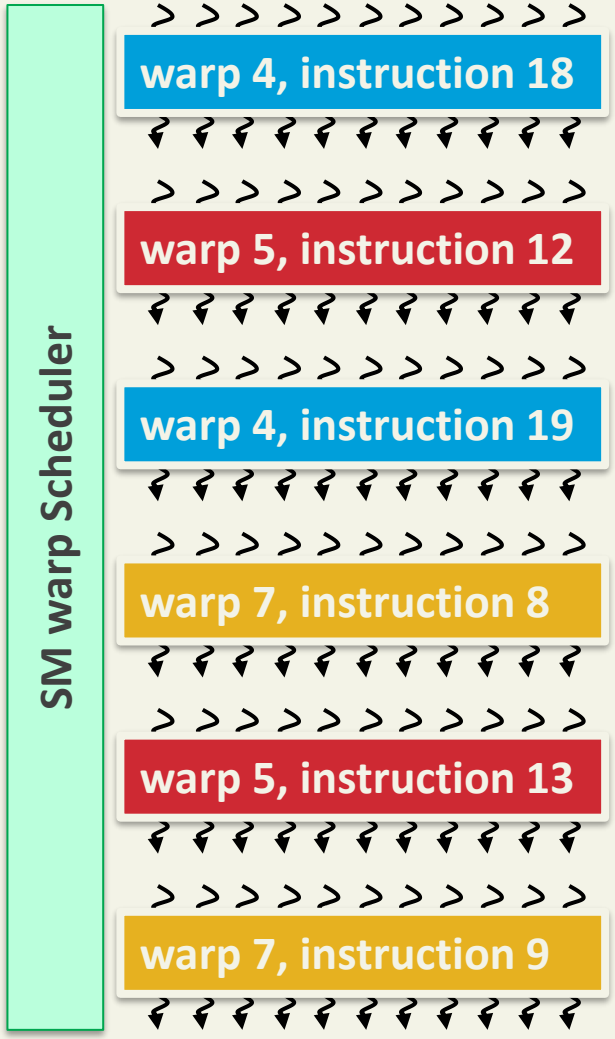
The complete story

1. The ***Grid*** is Launched
2. Blocks are assigned to SMs in arbitrary order
   - Each block is assigned to one SM.
     Each SM is assigned zero or more blocks.
   - There are limits on the number of blocks/threads the SM can track simultaneously. This is taken care of by the GPU.
3. Each block is divided into Warps whose execution is interleaved.
4. Warps are executed by the SM (each SP executes one thread).
   - Threads in a warp run simultaneously.
   - All threads in a warp execute the same instruction when selected

Zero-Overhead and Latency Tolerance

With many warps. those which are ready for consumption are eligible for execution (scheduling priority).

- **Latency hiding**:
  - While a warp is waiting for result from a **long-latency operation** (e.g. global memory access ~500 cylces, floating-point arithmetic, etc), the SM will pick **another warp** that's ready to execute to:
    - **avoid idle time**
    - make full use of the hardware despite long latency operations.
- **Zero-overhead thread scheduling**
  - *Having zero idle time* is referred to as **zero-overhead thread scheduling** in processor designs.



GPU Limits

- CUDA (the software) has limits (as discussed before).
- **GPU** also has limits on how many blocks and threads it can ***simultaneously track*** *(and schedule)*.
  - Hardware resources are required for SMs to maintain the thread, block IDs, and track their execution status.
- For example:
  - **G80** (16 SMs)
    - Each **SM** can track up to **8 blocks** or **768 threads** at a time
      - 3 blocks x 256 threads, or
      - 6 blocks x 128 threads, or …. etc
    - Max number threads at a time = 16 SMs x 768 threads = 12,288 threads
  - **G200** (30 SMs)
    - Each **SM** can process up to **8 blocks** or **1024 threads** at a time
    - Max threads: 30 SMs x 1024 threads = 30,720 threads
- If we assign to the SM more than its max amount of blocks (as per CUDA limits), they will be **scheduled for later execution**.

Benefits?

- Why is it good to know this stuff? (i.e. warps, GPU limits, etc.)
  - One benefit is to allow for **full utilization of each SM** on the GPU
  - Will discuss more on how these concepts are used when **improving performance** in the "CUDA Best Practices"

Granularity

An Example

Consider G80

- **CUDA Limits**: 512 threads per block, 2^{16} x 2^{16} blocks per grid.
  - These are the limits for CUDA 1.0 supported by G80
- **GPU Limits**: 8 blocks or 768 threads per SM

- Assume we have thousands of threads to run. To fully utilize each SM on G80, should we use 8X8, 16X16 or 32X32 threads per block?
  - For **32X32**, we have 1024 threads per Block. Not even one can fit into an SM!
  - For **8X8**, we have 64 threads per Block. Since each SM can take work with only 8 blocks at a time, this means 64x8=512 threads will go into each SM. But since SM needs 768 threads for full utilization, → **66% full - *underutilized (fewer warps to schedule)***
  - For **16X16**, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and **achieve full capacity and a lot of warps to schedule**.

Utilizing the Hardware

Key things that need to be considered:

- Have a number of blocks >= the number of SMs
  - Want to utilize all SMs
- Have a reasonable number of threads per block
  - Fully utilize each SM

- **Occupancy**
  - Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM
  - Occupancy varies over time as warps begin and end, and can be different for each SM
  - Low occupancy results in poor instruction issue efficiency; not enough eligible warps to hide latency between dependent instructions

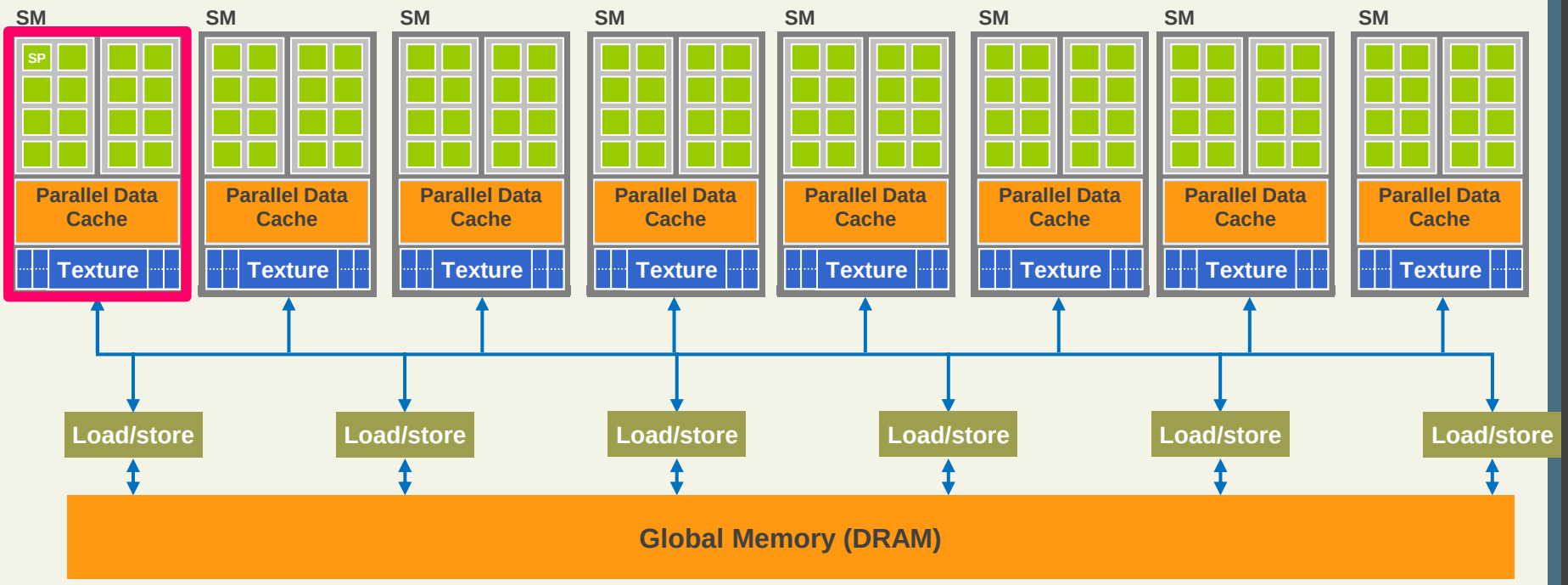
https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

Device Memories

Two types

1. ***Programmable***
   - Can control which data to put in that memory
   - Includes
     - Registers
     - Shared memory
     - Local memory
     - Constant memory
     - Global memory
2. ***Non-programmable***
   - *Cannot* control which data is put in that memory
   - Includes
     - L1 Cache memory
     - L2 Cache memory

Remember: GPU Design



Remember:

- A block is assigned to exactly one SM. An SM may run many blocks concurrently.
- All SMs can access the global memory
- We have different memories (as discussed before)

Device Memories

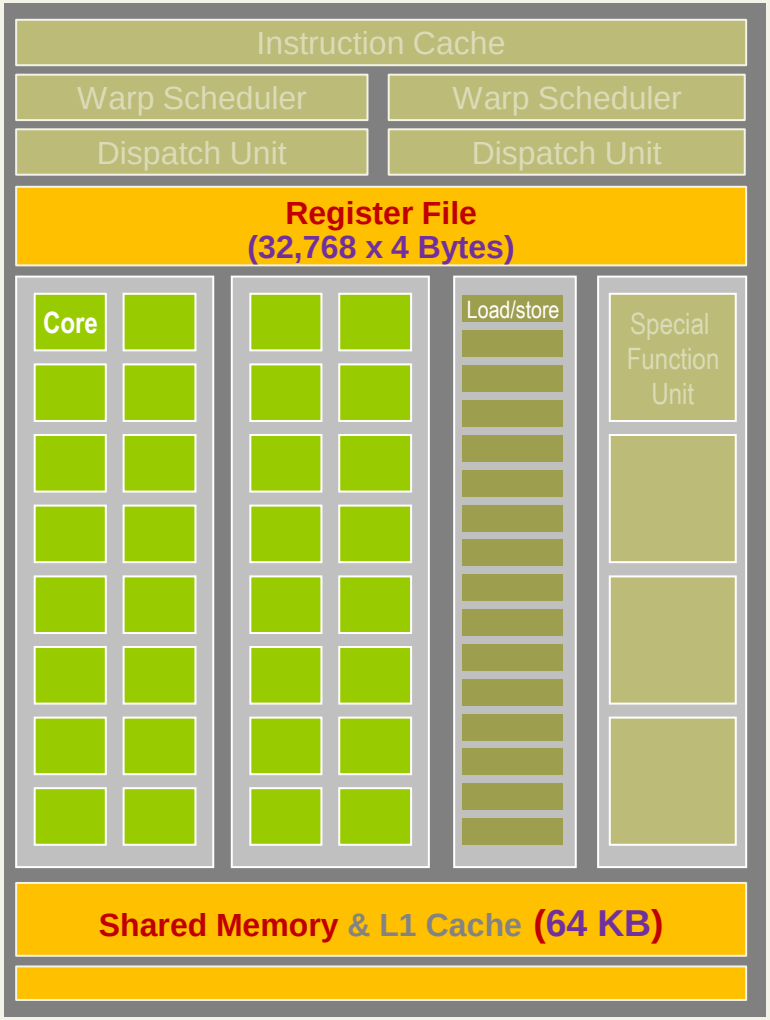
Registers

- **Private to each thread**
- ***Partitioned among threads* in a block**
- More threads means less registers per thread.
  - In diagram: if # of active threads = 1024, then each thread gets 32,768/1024 = **32 registers**

Shared memory

- **Shared by threads in the same block**
- ***Partitioned among blocks***
  - Remember that several blocks may be assigned to the same SM at the same time.
  - More blocks means less shared memory per block.
- There is **64KB** on-chip configurable memory, which is **partitioned between shared memory and L1 cache**.

?



SINGLE SM on Fermi Architecture

Device Memories

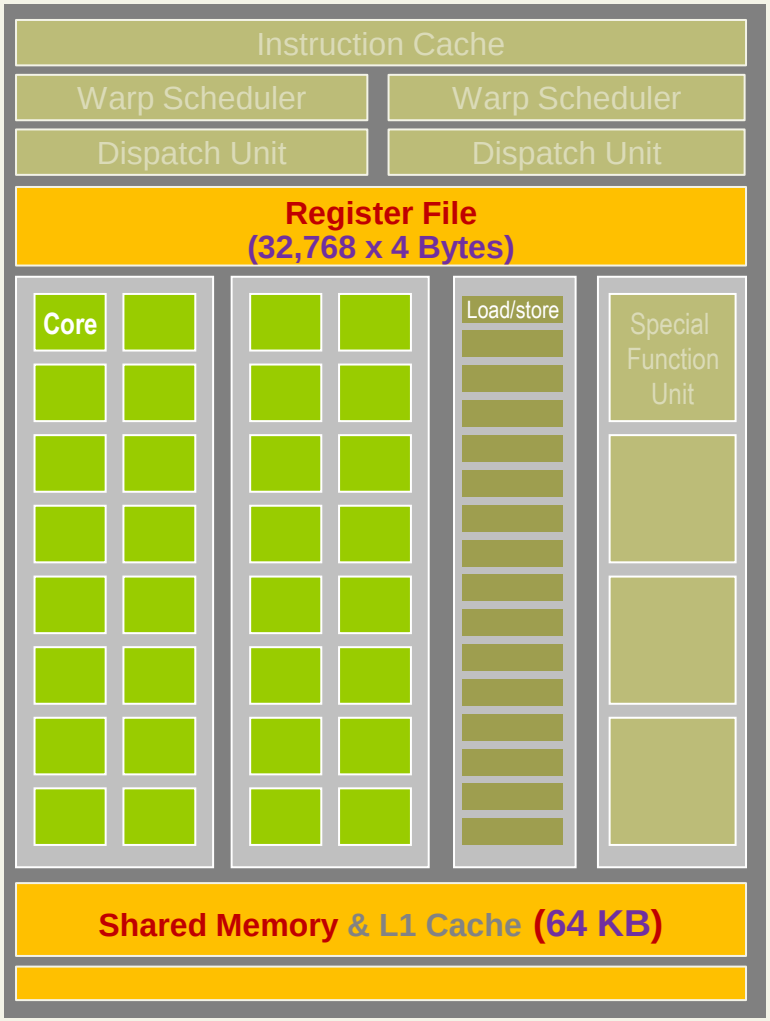
Global Memory (*not shown in figure*)

- **Shared by all threads in the grid**

Question: How this structure helps implementing zero-overhead thread scheduling?

- Zero-overhead (negligible context switching because each thread has its own registers and each block has its own shared memory

- GPU implements ***latency-hiding***
  - Many warps to run, and long latencies from reading global memory are hidden by warp-scheduling
  - The long waiting time of warp instructions is hidden by executing instructions from other warps

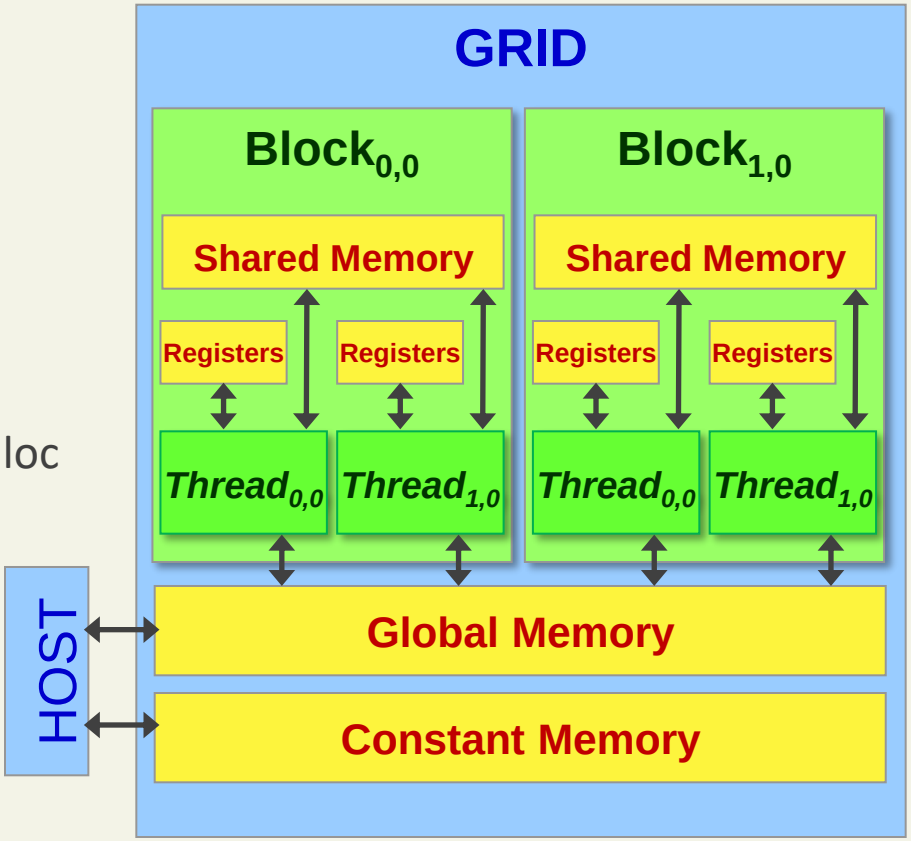


SINGLE SM on Fermi Architecture

Device Memory Model

Device can:

- Read/Write registers
  - ~1 cycle.
- Read/Write shared memory
  - ~5 cycles
- Read/Write global memory
  - ~500 cycles
  - Allocated by *host* using cudaMalloc
- Read only constant memory
  - ~5 cycles with caching
  - a **static global memory** area which is **cache**d.



Host can transfer data per-grid to/from global/constant memory.

Constant Memory

- Like global memory (DRAM) but has a ***dedicated on-chip cache*** for improved performance.
- **Initialized** in **host** code
  - ***Host*** can ***read/write***
  - ***Kernel*** can ***read-only***
- Has limited size (64 KB)
- **Which data is stored in constant memory?**
  - variables declared as `__constant__`
  - `__global__` **function parameters** are passed to the device **via** constant memory (limit is 4 KB)

Parallel Memory Sharing

Registers / Local Memory:

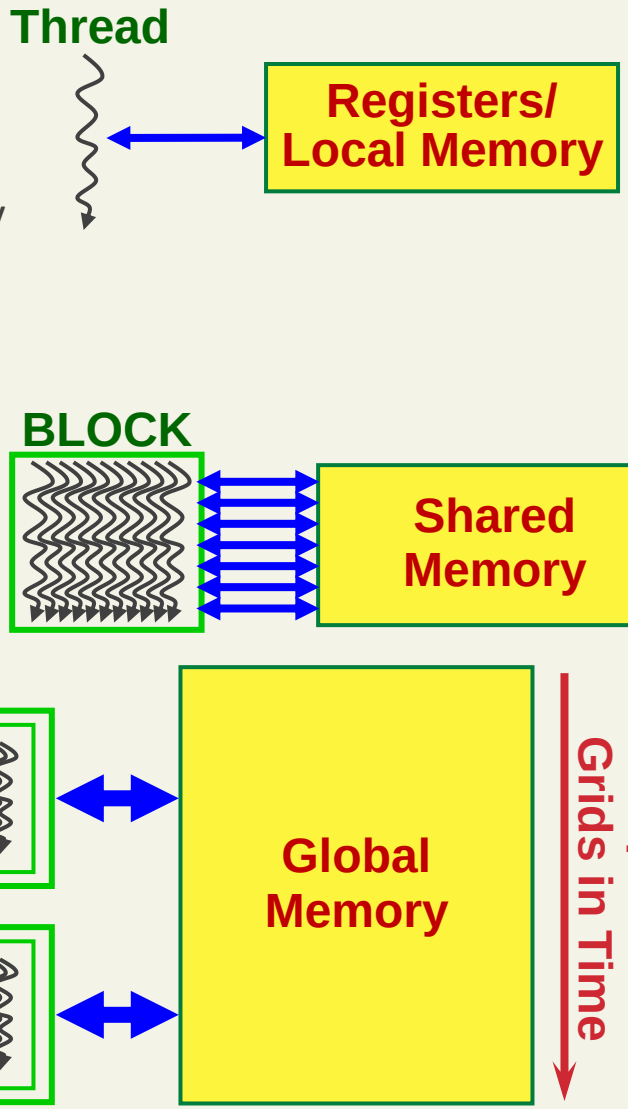
- Private per thread
- **Auto variables**
- **Register spill**
  - When out of registers, use local memory (see next slide)

Shared Memory:

- Shared by threads of the same block
- **Inter-thread communication**

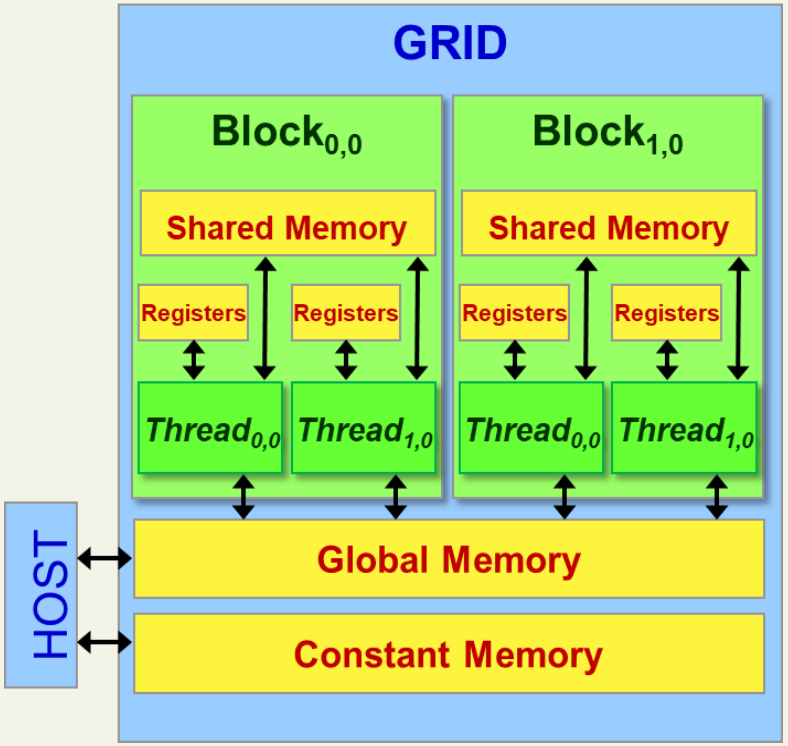
Global Memory:

- Shared by all threads
- **Inter-Grid communication**



Where is the Local Memory?

- Does ***not*** refer to a new physical memory
  - It is on the **global memory**
    - Data is put by the compiler
  - ***Local*** because **each thread** has its own **private area.**
  - Unlike global memory, it is **cached** (L1)



When is the Local Memory Used?

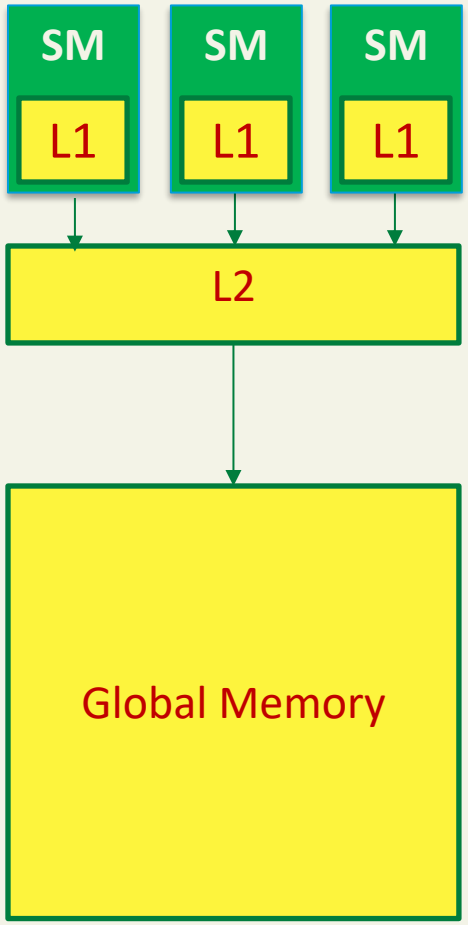
- When we run out of registers (Called *register spilling*)
  - Remember: there is a limit on # of registers per thread
- When declaring arrays inside kernels
  - Some arrays are still stored in registers if small and the compiler can resolve indexing (***registers aren't indexable***)

L1 and L2 Cache

- Cache is ***non-programmable*** small memory
- Cache memories (L1 and L2) help multiple threads that access the ***same memory segment*** so that they do not need to all go to the DRAM

Aside: L2 is coherent. L1 is not coherent.

  - "*Not coherent*" means that if two SMs are working on the same global memory location, it is no guaranteed that one SM will immediately see the changes made by the other SM



CUDA Variables

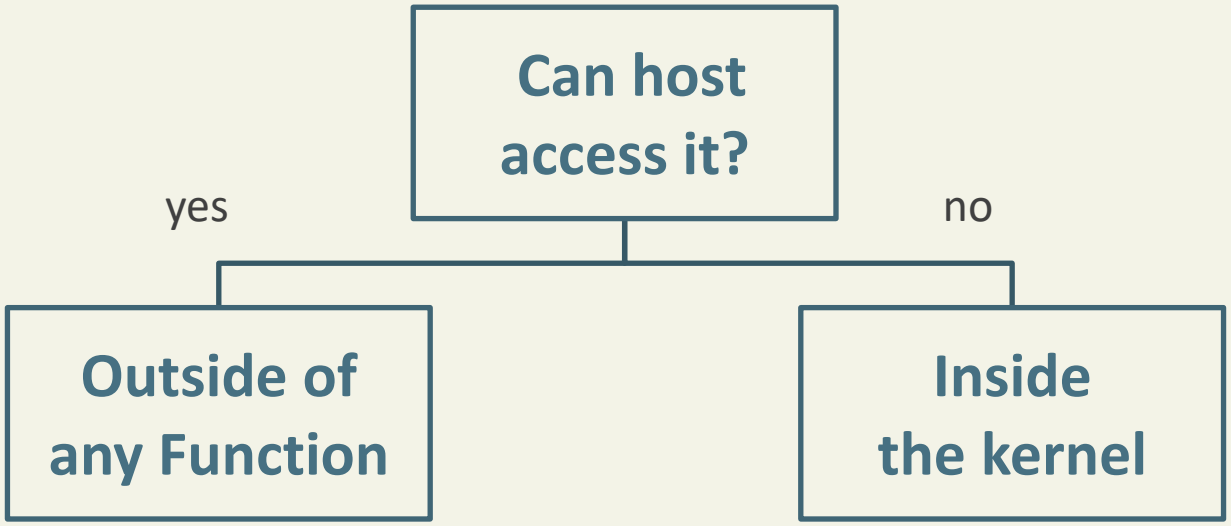
Var. Declaration	Memory			Scope	Lifetime	speed
`int x;`	Register	on-chip		thread	Thread	very fast
`int array[10]`	Local	off-chip	uses L1 & L2 Cache	thread	Thread	slow
`_shared_ int x;`	Shared	on-chip	on configurable Mem	block	Block	fast
`_device_ int x;`	Global	off-chip	on DRAM,small cache	grid+host	application	slow
`_constant_ int x;`	Constant	off-chip	has dedicated Cache	grid+host	application	fast

- ***Scope:*** identifies which threads can access the variable
  - i.e. a single thread, all threads of a block, or all threads of all grids)
- **Lifetime** specifies the portion of the program's execution duration when the variable is available for use:
  - **Thread:** variables don't exist after thread finishes its work (even if kernel is still running).
  - **Block:** shared variable don't exist once block finishes (even if kernel is still running)
  - **Application:** variables' contents are maintained throughout the execution of the application and are available to all kernels.

Automatic Variables

- **Automatic variables** are variables in a kernel functions with ***no modifier*** in their declaration
  - Can also (optionally) declare them as `__local__`.
- **Where** do automatic variables reside?
  - Automatic ***scalar variables*** reside in ***a register***
    - Example: `int x;` in previous table
    - Automatic scaler variables may also reside in ***Local Memory***
      - If we are out of registers, compiler will spill to thread local memory
      - If the variable's address is taken with the & operator and the compiler is not able to optimize the code.
  - Automatic ***arrays*** reside in *Local Memory* by default
    - there are some exceptions: small arrays may be saved in registers if there is enough registers and compiler can optimize their access.

Where to Declare Variables?




```
_constant_ int x;  //x in constant memory
_device_   int y;  //y in global memory
```

//all kernels share x and y

//host can access x and y using CUDA APIs:

e.g., cudaMemcpyToSymbol(…);

(also, use cudaMalloc to allocate space on global memory

```
int z;              //in register
int arr[10];        //in local
_shared_ int sh;    //in shared
```

Example

```
__device__ float glb; // glb is in global memory.
                      // glb accessible by all grids(pass data between kernels)
__constant__ float C = 1.2; // C is in constant memory.

//All foo1(), foo2(), and helper() have access to same glb and C variables
__global__ void foo1(float *array, int len){ ... }
__device__ void helper(...){ ... }

__global__ void foo2(float *array, int len){
  int i = threadIdx.x; // i is in a register. Every thread gets a private copy
  float a1[100]; // a1 is (often) in off-chip memory. threads get private copies

  __shared__ float sh; // sh is in shared on-chip memory. block threads share sh
  __shared__ float a2[BLOCK_SIZE]; //a2 is in shared on-chip memory.

  float *p_glb = &glb;  // pointer to glb which is in global memory
  const float *p_C = &C;// pointer to C which is in constant memory
  //...
}

int main(){
  float* arr;
  cudaMalloc(&arr, N * sizeof(float));// arr points to global device memory
  foo1<<<4, N / 4 >>>(arr, N);
  foo2<<<4, N / 4 >>>(arr, N);
  return 0;
}
```

Notes

- Why declare a variable as `_device_` (on global memory) if it is slow?? e.g. `__device__ int x;`
  - Allows for **collaboration across grids** (lifetime = application)
    - i.e. to pass information from one kernel invocation to another kernel invocation.
- **Pointers** can only point to global memory.
  - Two ways :
    - Allocated in host (cudaMalloc) and passed to kernel as parameter
      (e.g., M in **`__global__ void MatrixMul(float* M, ...)`**
    - Obtained as the address of a global variable:

      `float* ptr = &M;` //where M is global variable

Poor Performance!

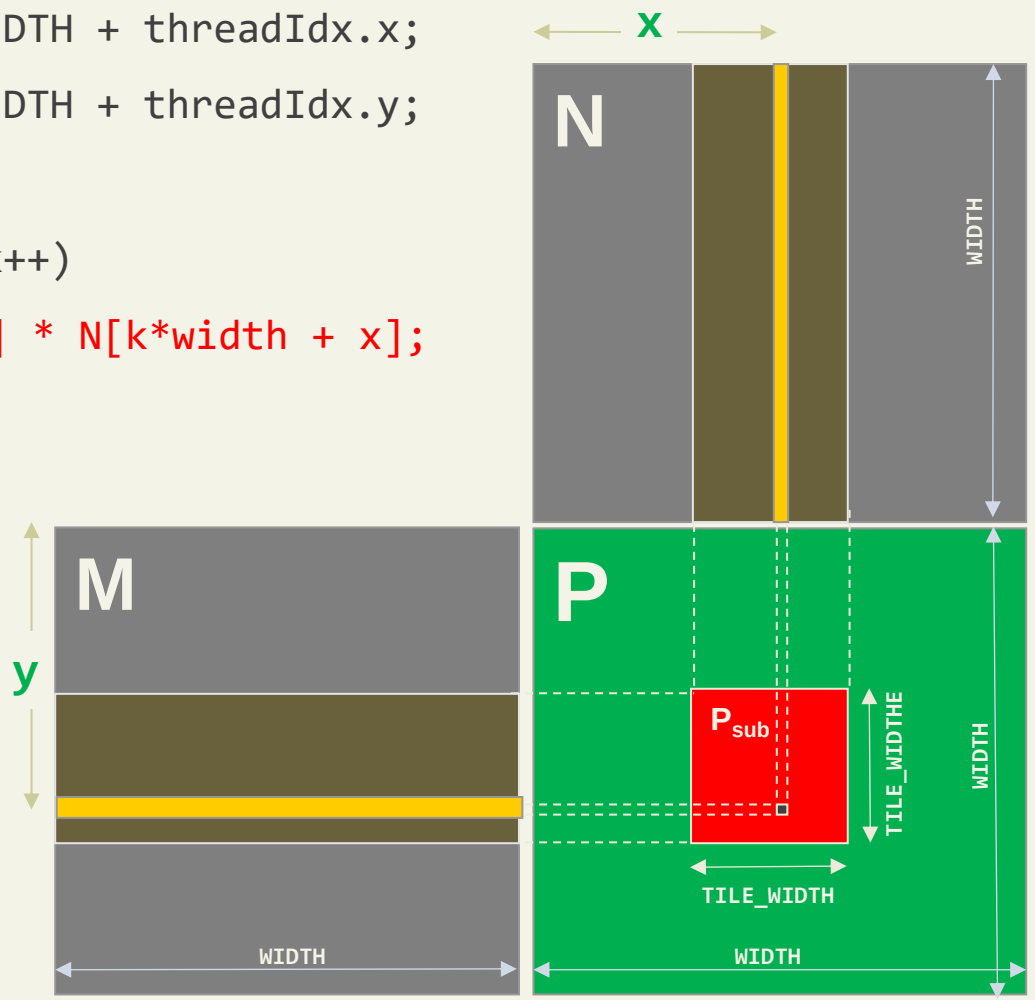
```
__global__ void MatrixMul(float* M, float* N, float* P, int width) {
    int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
    int y = blockIdx.y * TILE_WIDTH + threadIdx.y;
    float value = 0;
    for (int k = 0; k < width; k++)
        value += M[y*width + k] * N[k*width + x];
    P[y*width+x] = value;
}
```

Why poor performance again?

CGMA = 1.0 which limits GFLOPS.

- Actual ~22 GFLOPS
- Card can do > 370 GFLOPS

Need to drastically cut down global memory access to improve.



Summary

Today:

- Tiling
- CUDA Scalability
- Thread Scheduling on the H/W: Thread Lifecycle
- zero-overhead and latency tolerance
- GPU limits
- CUDA Memories Types (and Performance)

Next:

- CUDA Memories Types (and Performance)
- Memory Access Challenges
- Thread Performance
- More Example: Improving Performance of Matrix Multiplication