King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 13: GPU Compute APIs, Pt. 3

Markus Hadwiger, KAUST

Reading Assignment #7 (until Oct 20)

Read (required):

- Programming Massively Parallel Processors book (4th edition), **Chapter 5** (*Memory architecture and data locality*)

Next Lectures

Lecture 14: Thu, Oct 16

Lecture 15: Mon, Oct 20

no lecture on Oct 23 ! (fall break)

Lecture 16: Mon, Oct 27

Lecture 17: Tue, Oct 28 (make-up lecture; please choose times on discord!)

no lectures on Oct 30, Nov 3, Nov 6 ! (IEEE VIS conference)

Lecture 18: Mon, Nov 10

Lecture 19: Tue, Nov 11 (make-up lecture; please choose times on discord!)

Lecture 20: Thu, Nov 13

Semester Project (proposal until Oct 20!)

- Choosing your own topic encouraged!
  (see next slides for some suggestions)
  - Pick something that you think is really cool!
  - Can be completely graphics or completely computation, or both combined
  - Can be built on CS 380 frameworks, Vulkan SDK, CUDA SDK, ...
- Write short (1-2 pages) project proposal by Oct 20 at the latest
  - Talk to us before you start writing!
    (content and complexity should fit the lecture)
- Submit semester project with report (deadline: Dec 14)
- Present semester project, event in final exams week: Dec 15 (tbd!)

Semester Project Ideas (1)

Some ideas for topics

- Procedural shading with noise + marble etc. (GPU Gems 2, chapter 26)
- Procedural shading with noise + bump mapping (GPU Gems 2, chapter 26)
- Subdivision surfaces (GPU Gems 2, chapter 7)
- Ambient occlusion, screen space ambient occlusion
- Shadow mapping, hard shadows, soft shadows
- Deferred shading
- Particle system rendering + CUDA particle sort
- Advanced image filters: fast bilateral filtering, Gaussian kD trees
- Advanced image de-convolution (e.g., convex L1 optimization)
- PDE solvers (e.g., anisotropic diffusion filtering, 2D level set segmentation, 2D fluid flow)

Semester Project Ideas (2)

Some ideas for topics

- Distance field computation (GPU Gems 3, chapter 34)
- Livewire (“intelligent scissors“) segmentation in CUDA
- Linear systems solvers, matrix factorization (Cholesky, ...); with/without CUBLAS
- CUDA + matlab
- Fractals (Sierpinski, Koch, ...)
- Image compression
- Bilateral grid filtering for multichannel images
- Discrete wavelet transforms
- Fast histogram computations
- Terrain rendering from height map images; clipmaps or adaptive tesselation

COOPERATIVE GROUPS

Kyrylo Perelygin, Yuan Lin

GTC 2017

LEVELS OF COOPERATION: CUDA 9.0 or newer

For current coalesced set of threads:

```
auto g = coalesced_threads();
```

For warp-sized group of threads:

```
auto block = this_thread_block();
auto g = tiled_partition<32>(block)
```

For CUDA thread blocks:

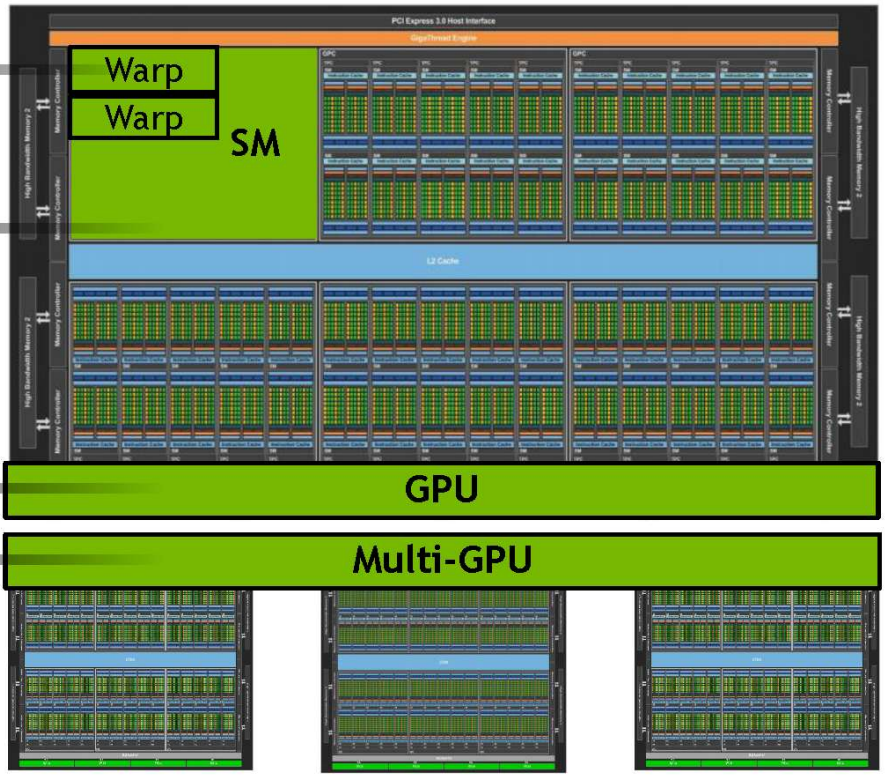
```
auto g = this_thread_block();
```

For device-spanning grid:

```
auto g = this_grid();
```

For multiple grids spanning GPUs:

```
auto g = this_multi_grid();
```



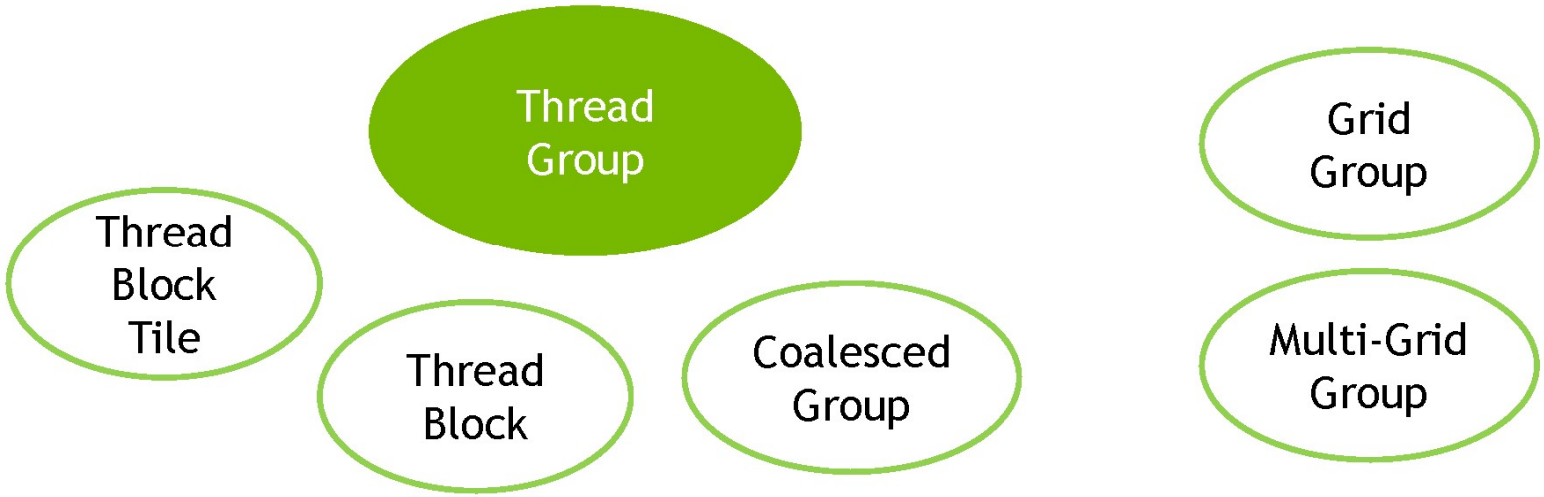
All Cooperative Groups functionality is within a **cooperative_groups::** namespace

THREAD GROUP

Base type, the implementation depends on its construction.

Unifies the various group types into one general, collective, thread group.

We need to extend the CUDA programming model with handles that can represent the groups of threads that can communicate/synchronize



THREAD BLOCK

Implicit group of all the threads in the launched thread block

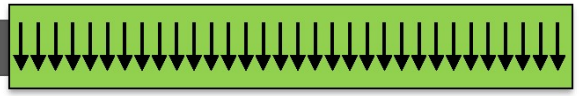
Implements the same interface as thread_group:

```
void sync();                // Synchronize the threads in the group
unsigned size();            // Total number of threads in the group
unsigned thread_rank();     // Rank of the calling thread within [0, size]
bool is_valid();            // Whether the group violated any API constraints
```

And additional thread_block specific functions:

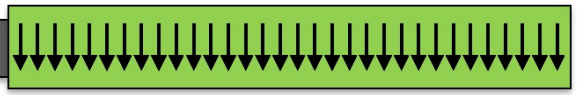
```
dim3 group_index();         // 3-dimensional block index within the grid
dim3 thread_index();        // 3-dimensional thread index within the block
```

PROGRAM DEFINED DECOMPOSITION

CUDA KERNEL 

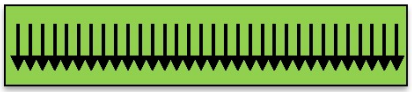
All threads launched

```
thread_block g = this_thread_block();
```

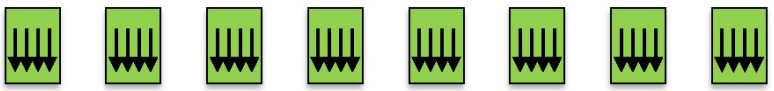
foobar(thread_block g) 

All threads in thread block

```
thread_group tile32 = tiled_partition(g, 32);
```



```
thread_group tile4 = tiled_partition(tile32, 4);
```



Restricted to powers of two, and <= 32 in initial release

GENERIC PARALLEL ALGORITHMS

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

Per-Warp

```
g = tiled_partition(this_thread_block(), 32);
reduce(g, ptr, myVal);
```

```
__device__ int reduce(thread_group g, int *x, int val) {
  int lane = g.thread_rank();
  for (int i = g.size()/2; i > 0; i /= 2) {
    x[lane] = val;        g.sync();
    val += x[lane + i];   g.sync();
  }
  return val;
}
```

COOPERATIVE GROUPS VS BUILT-IN FUNCTIONS

Example: warp aggregated atomic

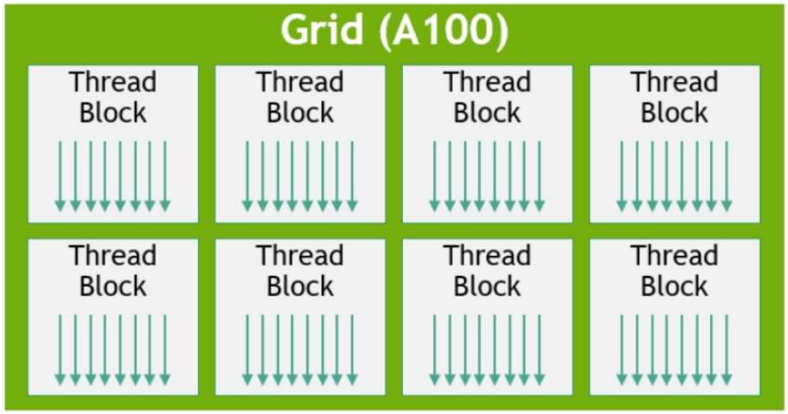
```
// increment the value at ptr by 1 and return the old value
__device__ int atomicAggInc(int *p);
```

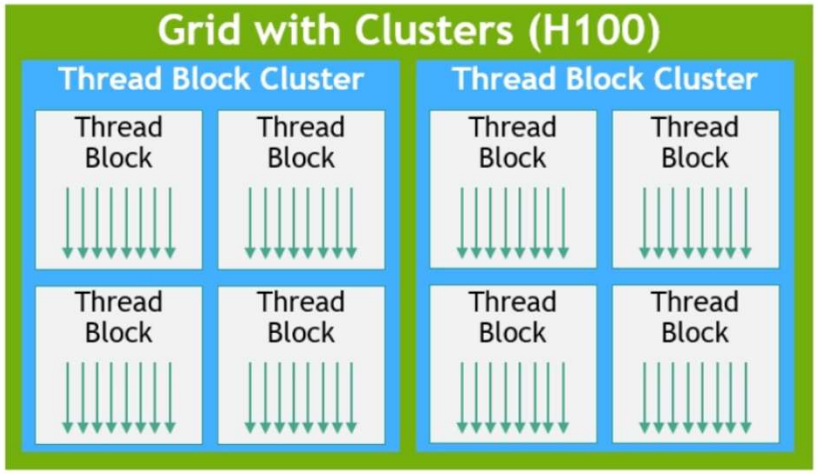
```
coalesced_group g = coalesced_threads();



int res;
if (g.thread_rank() == 0)
   res = atomicAdd(p, g.size());
res = g.shfl(res, 0);
return g.thread_rank() + res;
```

```
int mask = __activemask();
int rank = __popc(mask & __lanemask_lt());
int leader_lane = __ffs(mask) - 1;
int res;
if (rank == 0)
   res = atomicAdd(p, __popc(mask));
res = __shfl_sync(mask, res, leader_lane);
return rank + res;
```

New in CC 9.0: Thread Block Clusters

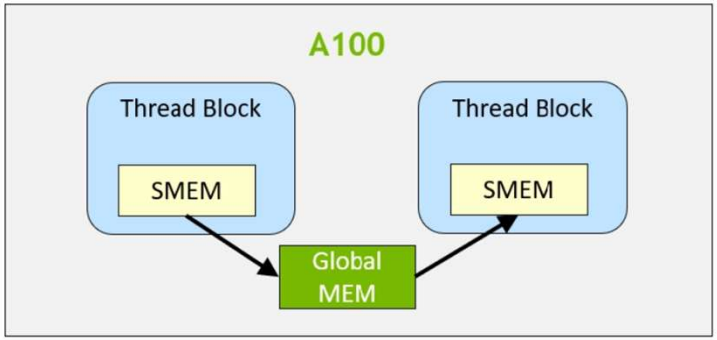
New thread hierarchy level!

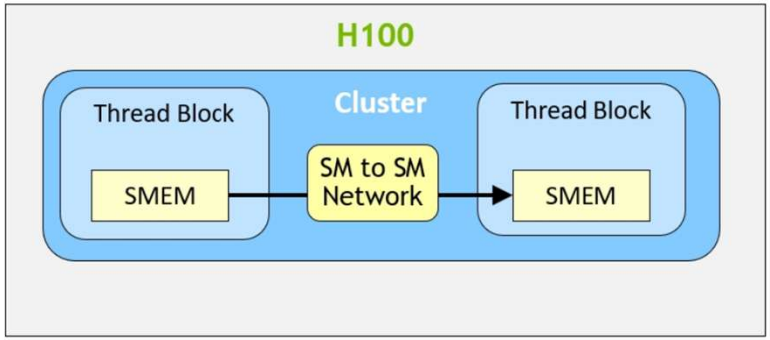




*all threads of a block are on the same **SM !***

*all blocks of a cluster are on the same **GPC !***





CUDA Software Development

CUDA Optimized Libraries:
math.h, FFT, BLAS, …

Integrated CPU + GPU
C Source Code

NVIDIA C Compiler

NVIDIA Assembly
for Computing (PTX)

CPU Host Code

CUDA
Driver

Profiler

Standard C Compiler

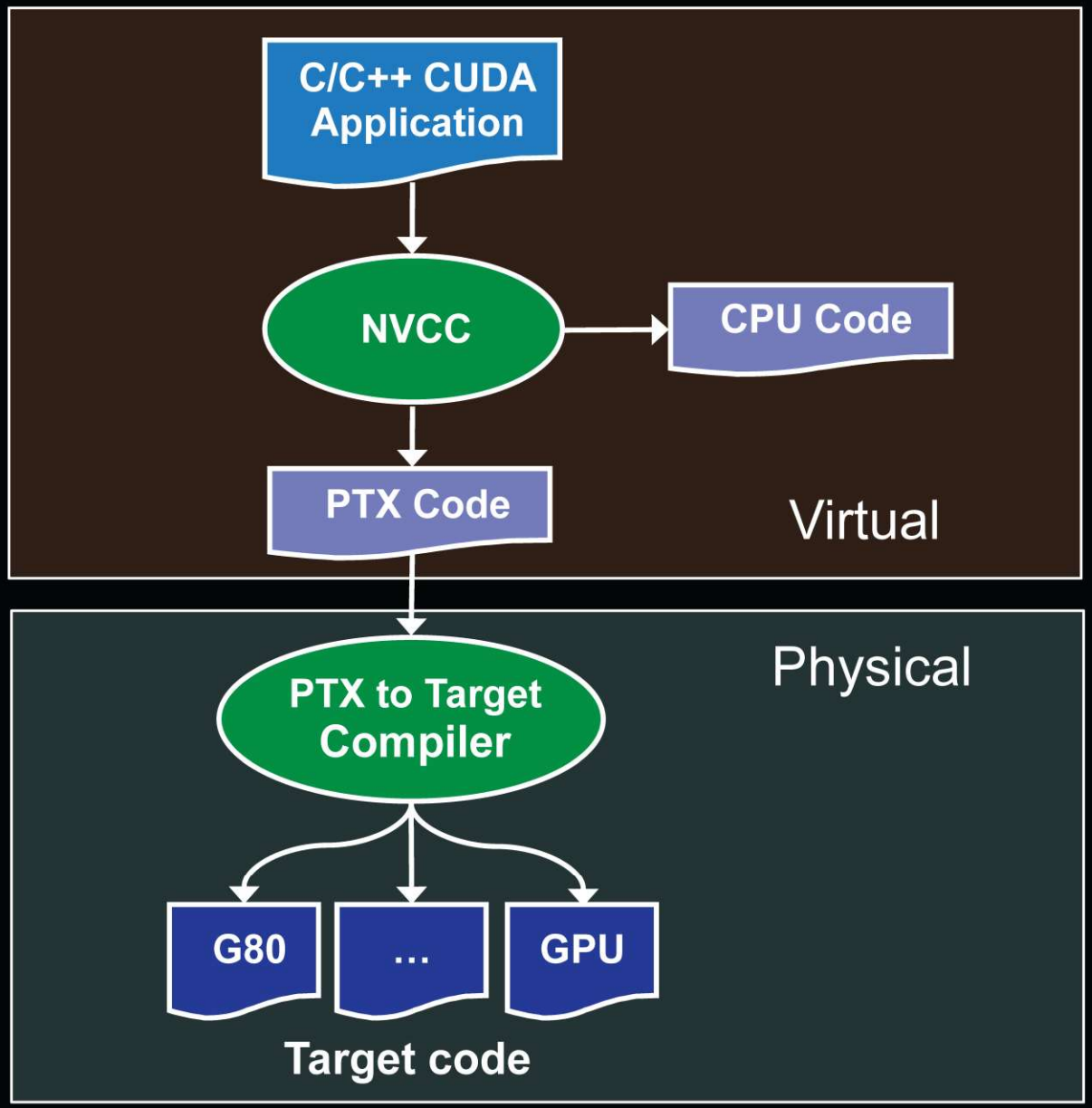
GPU

CPU

NVISION 08
THE WORLD OF VISUAL COMPUTING

NVIDIA

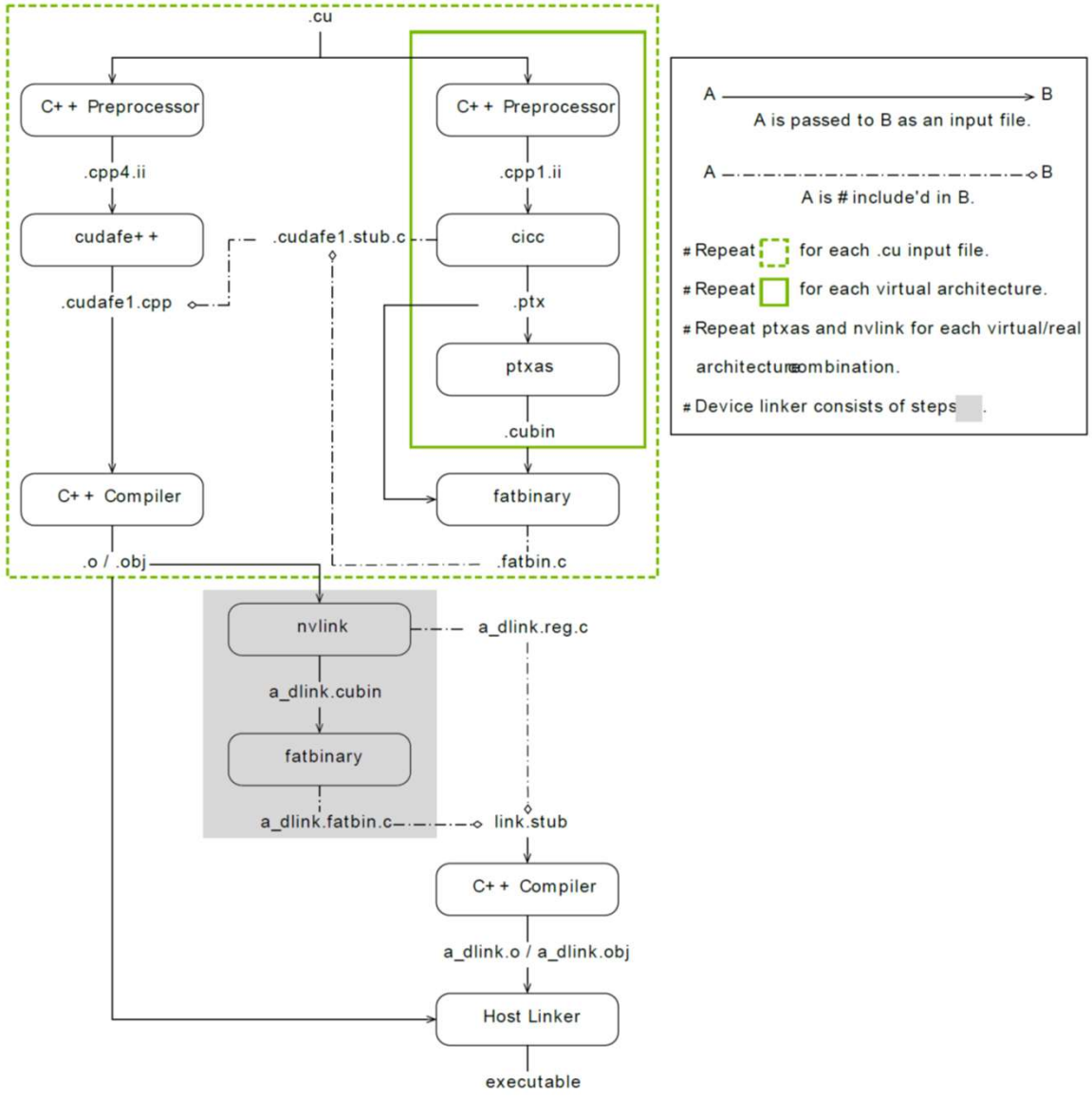
Compiling CUDA Code



CUDA Compilation Trajectory

CUDA Compiler Driver (NVCC) docs:

`CUDA_Compiler_Driver_NVCC.pdf`



CUDA Compilation Trajectory / Code Gen

4.2.7. Options for Steering GPU Code Generation

4.2.7.1. `--gpu-architecture` *arch* (`-arch`)

Specify the name of the class of NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

With the exception as described for the shorthand below, the architecture specified with this option must be a *virtual* architecture (such as compute_50). Normally, this option alone does not trigger assembly of the generated PTX for a *real* architecture (that is the role of `nvcc` option `--gpu-code`, see below); rather, its purpose is to control preprocessing and compilation of the input to PTX.

For convenience, in case of simple `nvcc` compilations, the following shorthand is supported. If no value for option `--gpu-code` is specified, then the value of this option defaults to the value of `--gpu-architecture`. In this situation, as only exception to the description above, the value specified for `--gpu-architecture` may be a *real* architecture (such as a sm_50), in which case `nvcc` uses the specified *real* architecture and its closest *virtual* architecture as effective architecture values. For example, `nvcc --gpu-architecture=sm_50` is equivalent to `nvcc --gpu-architecture=compute_50 --gpu-code=sm_50,compute_50`.

See Virtual Architecture Feature List for the list of supported *virtual* architectures and GPU Feature List for the list of supported *real* architectures.

from **`https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf`**

CUDA Compilation Trajectory / Code Gen

4.2.7.2. `--gpu-code code,...` (`-code`)

Specify the name of the NVIDIA GPU to assemble and optimize PTX for.

`nvcc` embeds a compiled code image in the resulting executable for each specified *code* architecture, which is a true binary load image for each *real* architecture (such as sm_50), and PTX code for the *virtual* architecture (such as compute_50).

During runtime, such embedded PTX code is dynamically compiled by the CUDA runtime system if no binary load image is found for the *current* GPU.

Architectures specified for options `--gpu-architecture` and `--gpu-code` may be *virtual* as well as *real*, but the *code* architectures must be compatible with the *arch* architecture. When the `--gpu-code` option is used, the value for the `--gpu-architecture` option must be a *virtual* PTX architecture.

For instance, `--gpu-architecture=compute_60` is not compatible with `--gpu-code=sm_52`, because the earlier compilation stages will assume the availability of `compute_60` features that are not present on `sm_52`.

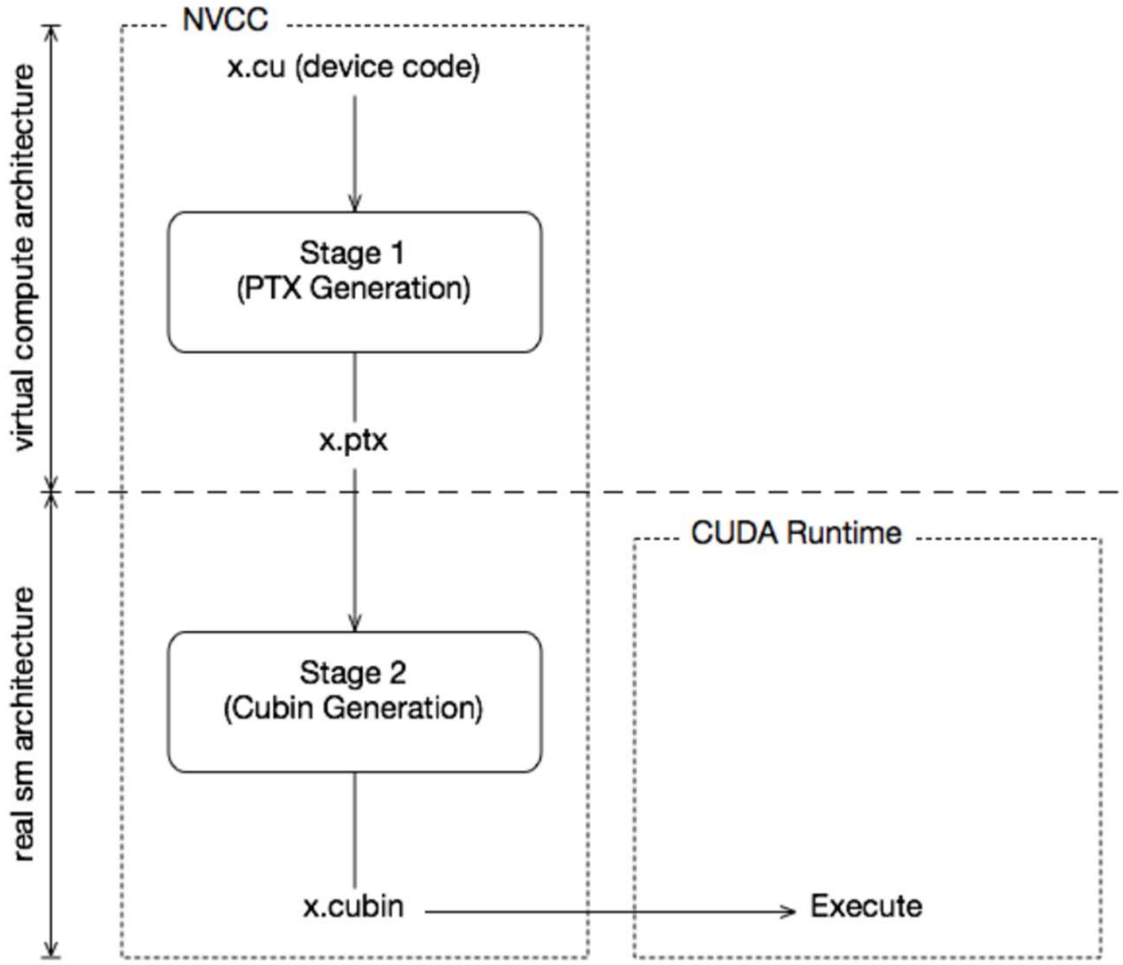
See Virtual Architecture Feature List for the list of supported *virtual* architectures and GPU Feature List for the list of supported *real* architectures.

Also look at compatibility guides:

https://docs.nvidia.com/cuda/pdf/NVIDIA_Ampere_GPU_Architecture_Compatibility_Guide.pdf

https://docs.nvidia.com/cuda/pdf/Hopper_Compatibility_Guide.pdf

Arch = ptx, Code = cubin

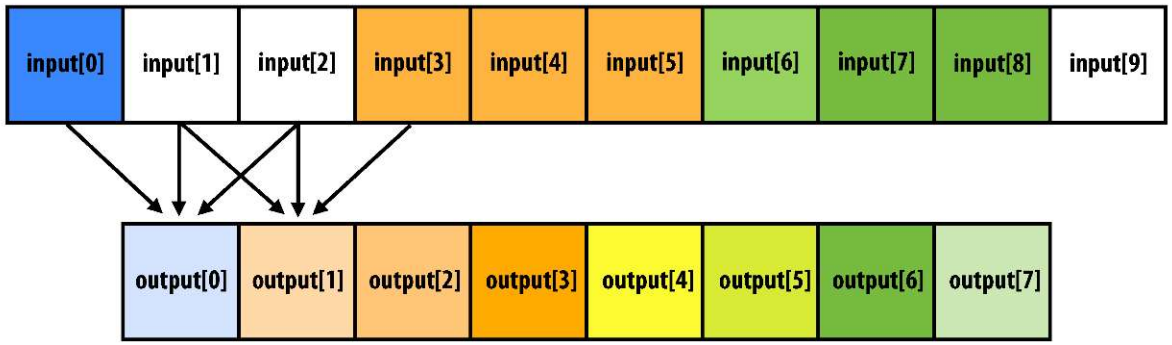


Two-Staged Compilation with Virtual and Real Architectures

Code Example #1: 1D Convolution

Example #1: 1D Convolution

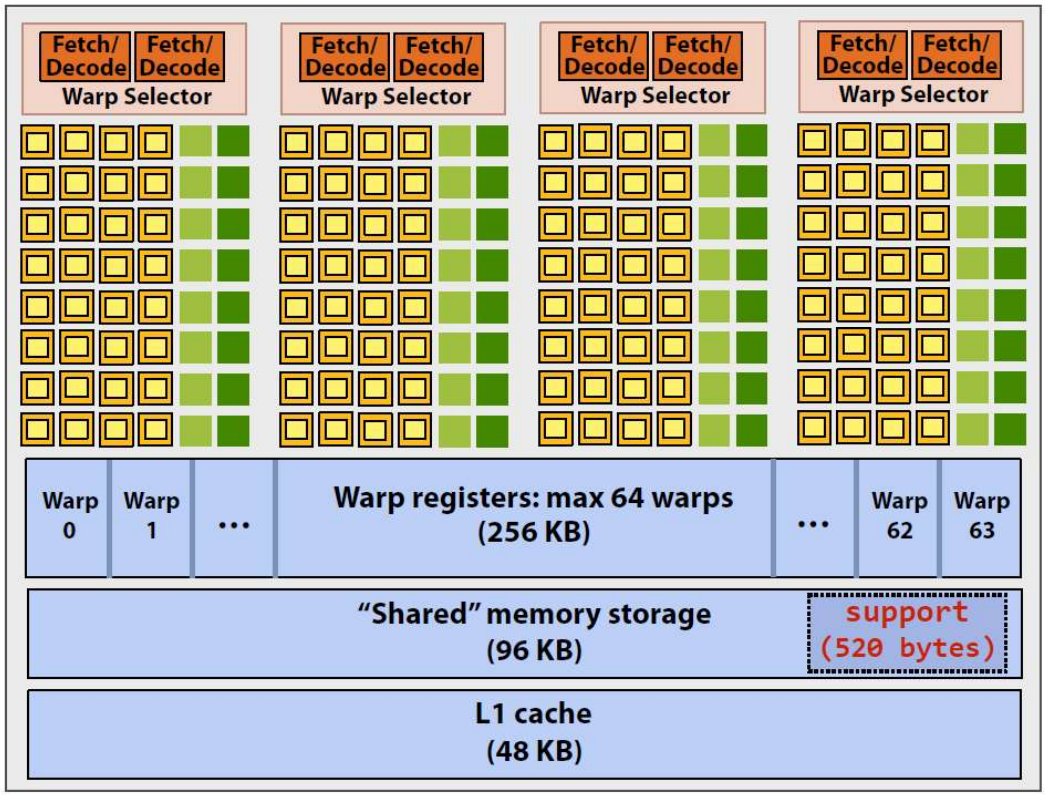
1D Convolution with 3-tap averaging kernel (every thread is averaging three inputs)




```
output[i] = (input[i] + input[i+1] + input[i+2]) / 3.f;
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

Running on a GP104 (Pascal) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

Recall, CUDA kernels execute as SPMD programs

On NVIDIA GPUs groups of 32 CUDA threads share an instruction stream. These groups called "warps".

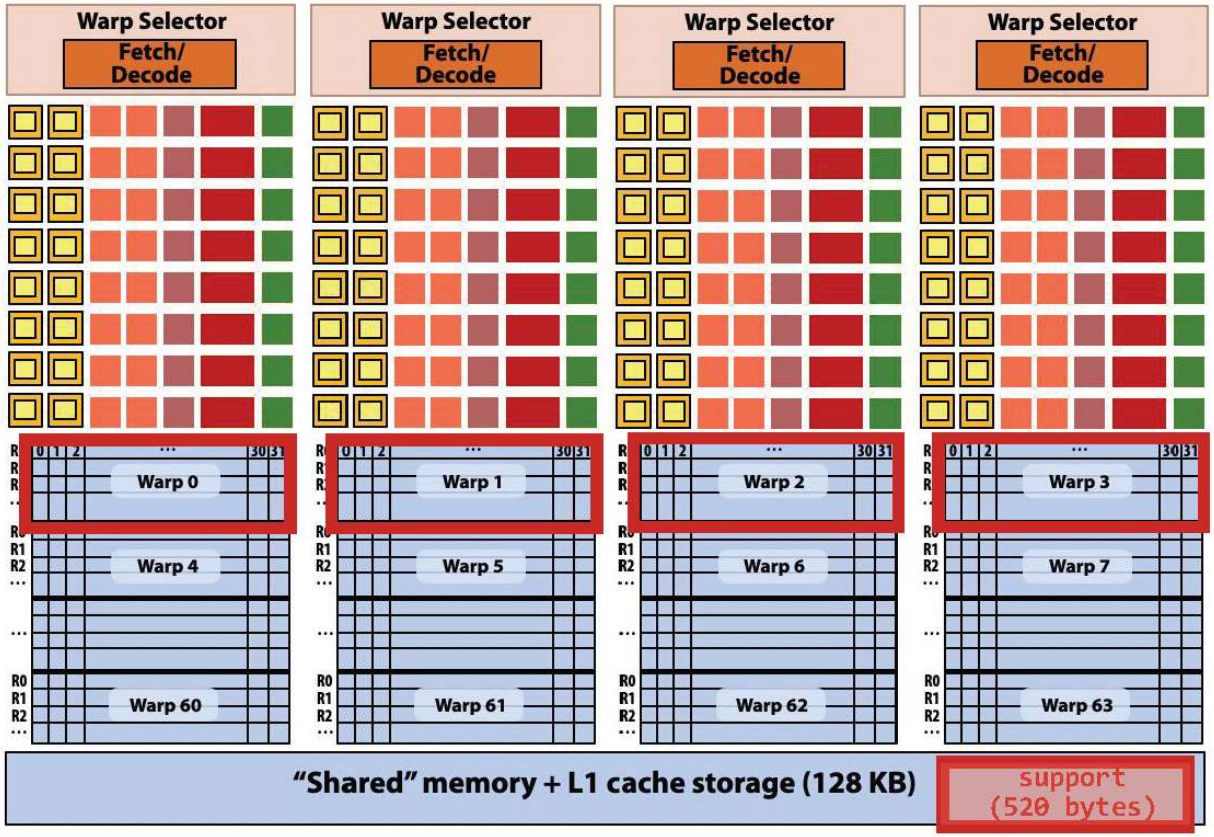
A `convolve` thread block is executed by 4 warps (4 warps x 32 threads/warp = 128 CUDA threads per block)

(Warps are an important GPU implementation detail, but not a CUDA abstraction!)

SM core operation each clock:

- **Select up to four runnable warps from 64 resident on SM core (thread-level parallelism)**
- **Select up to two runnable instructions per warp (instruction-level parallelism) *** *(but no ALU dual-issue!)*

Running on a V100 (Volta) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

A `convolve` thread block is executed by 4 warps
(4 warps x 32 threads/warp = 128 CUDA threads per block)

(sub-core == SM partition)

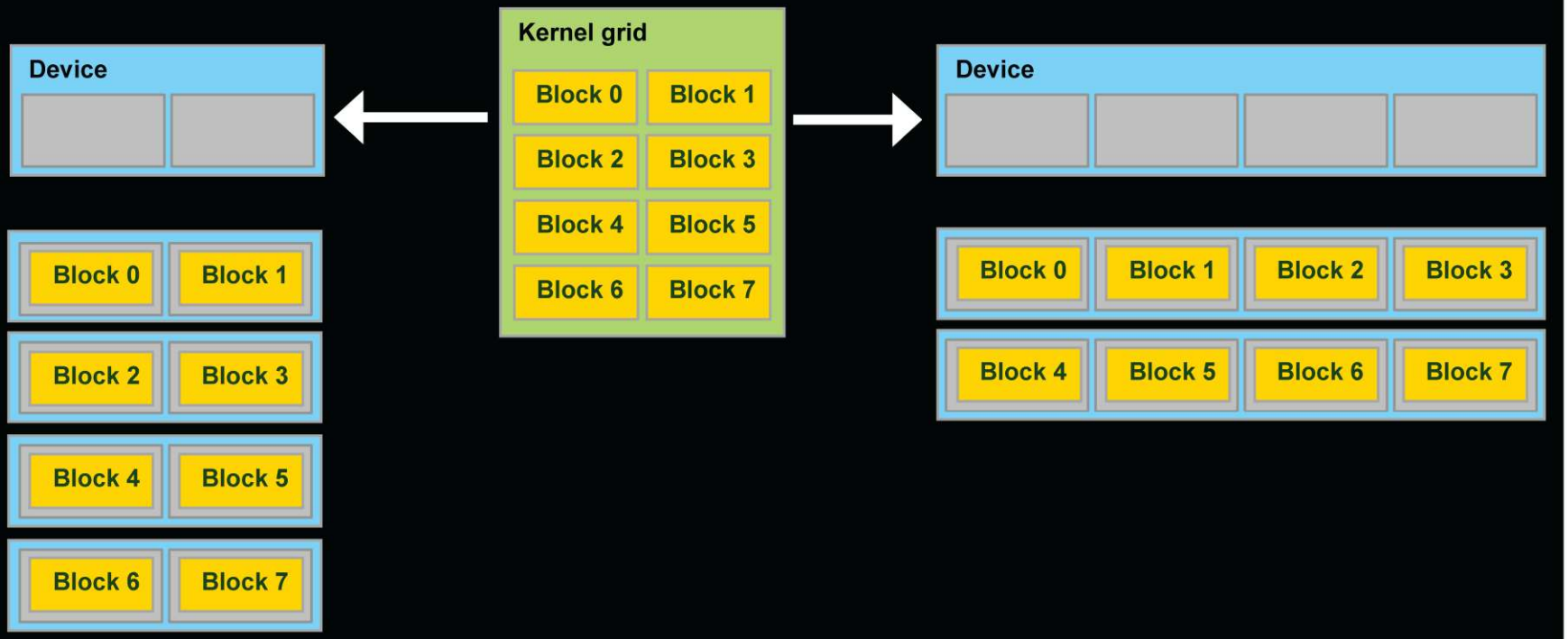
SM core operation each clock:

- **Each sub-core selects one runnable warp (from the 16 warps in its partition)**
- **Each sub-core runs next instruction for the CUDA threads in the warp (this instruction may apply to all or a subset of the CUDA threads in a warp depending on divergence)**

courtesy Kayvon Fatahalian

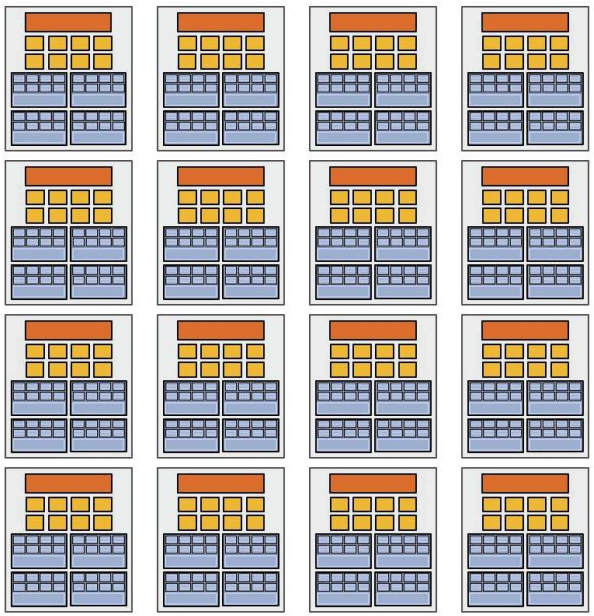
Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
  - A kernel scales across parallel multiprocessors

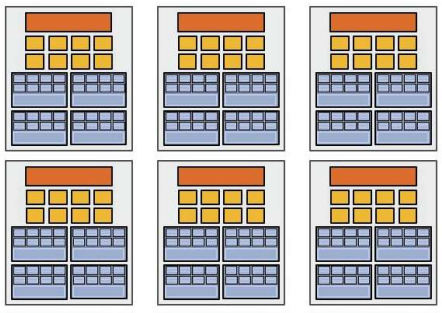


Code on Same SM Arch. But Different #SMs

Assigning work



High-end GPU
(16 cores)



Mid-range GPU
(6 cores)

Desirable for CUDA program to run on all of these GPUs without modification

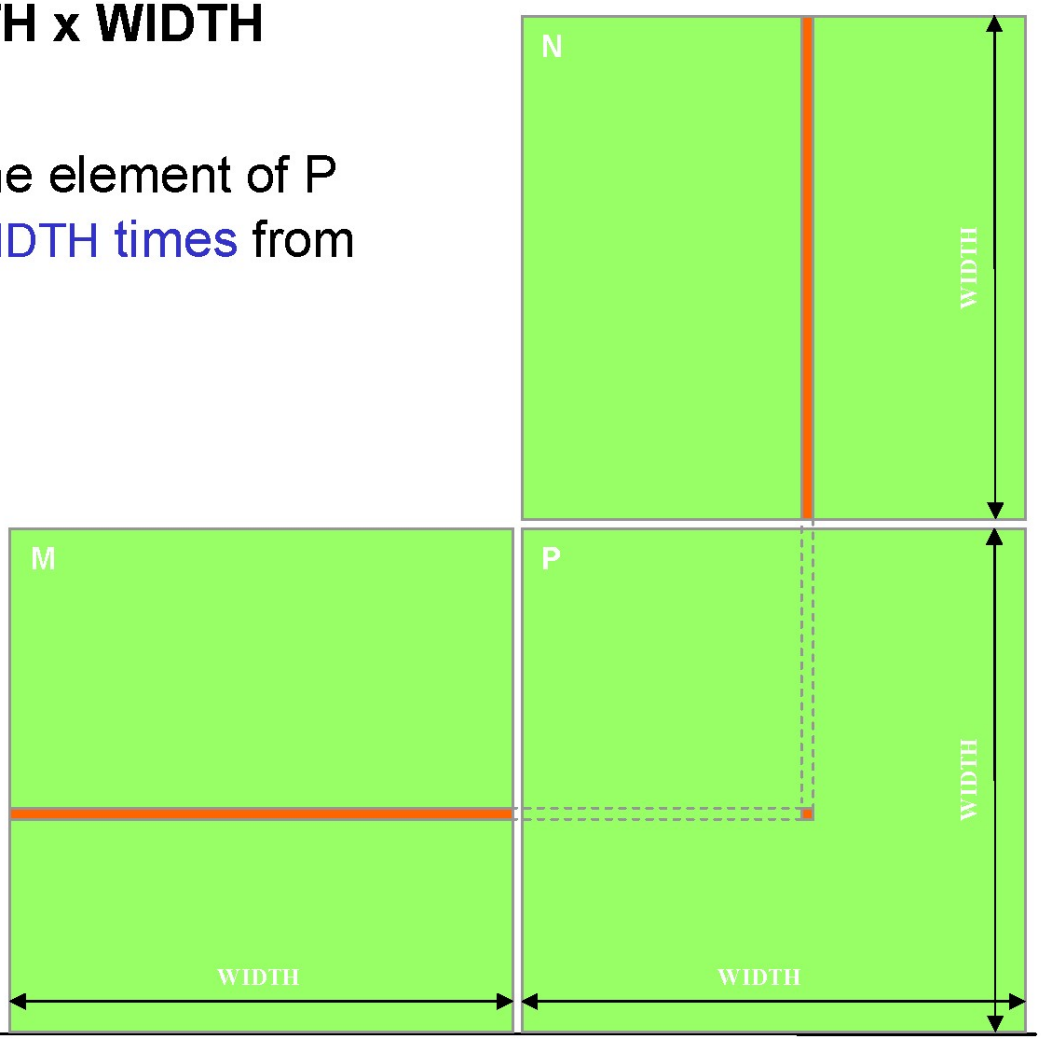
Note: there is no concept of `num_cores` in the CUDA programs I have shown you. (CUDA thread launch is similar in spirit to a forall loop in data parallel model examples)

(could now be up to 192 SMs, etc., …)

Code Example #2: Matrix Multiply

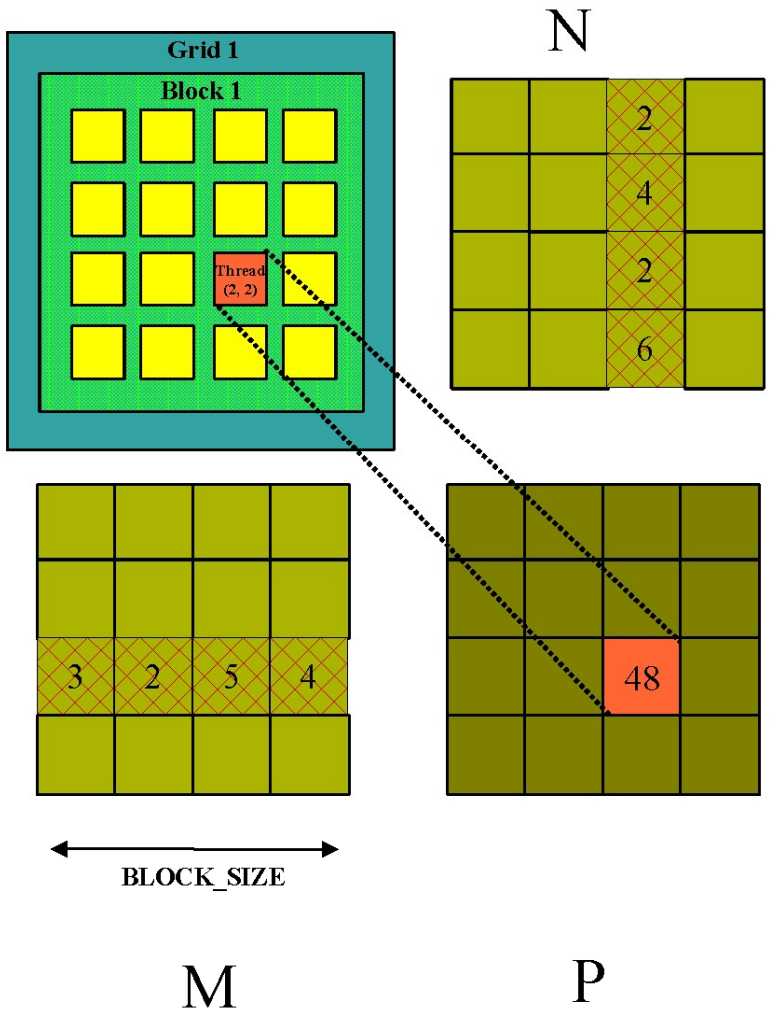
Programming Model: Square Matrix Multiplication

- **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



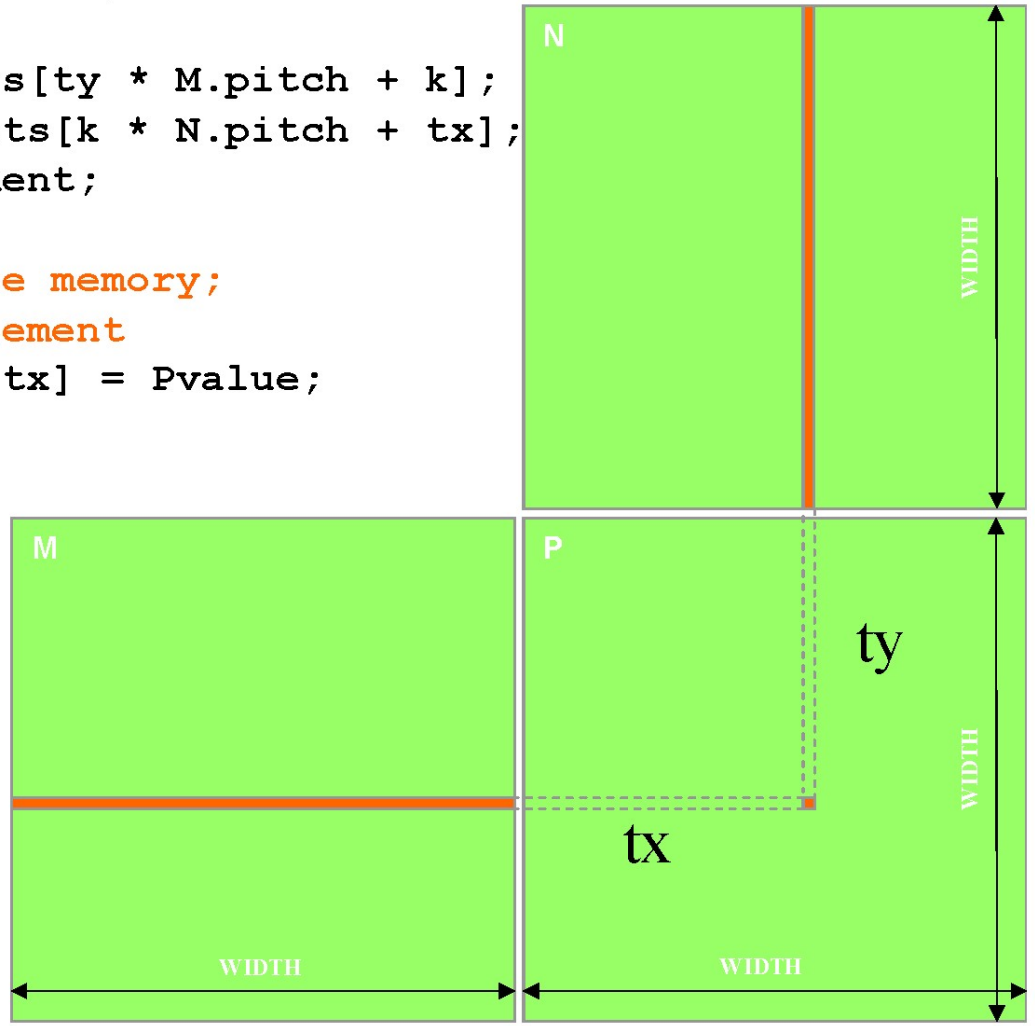
Multiply Using One Thread Block

- **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.)

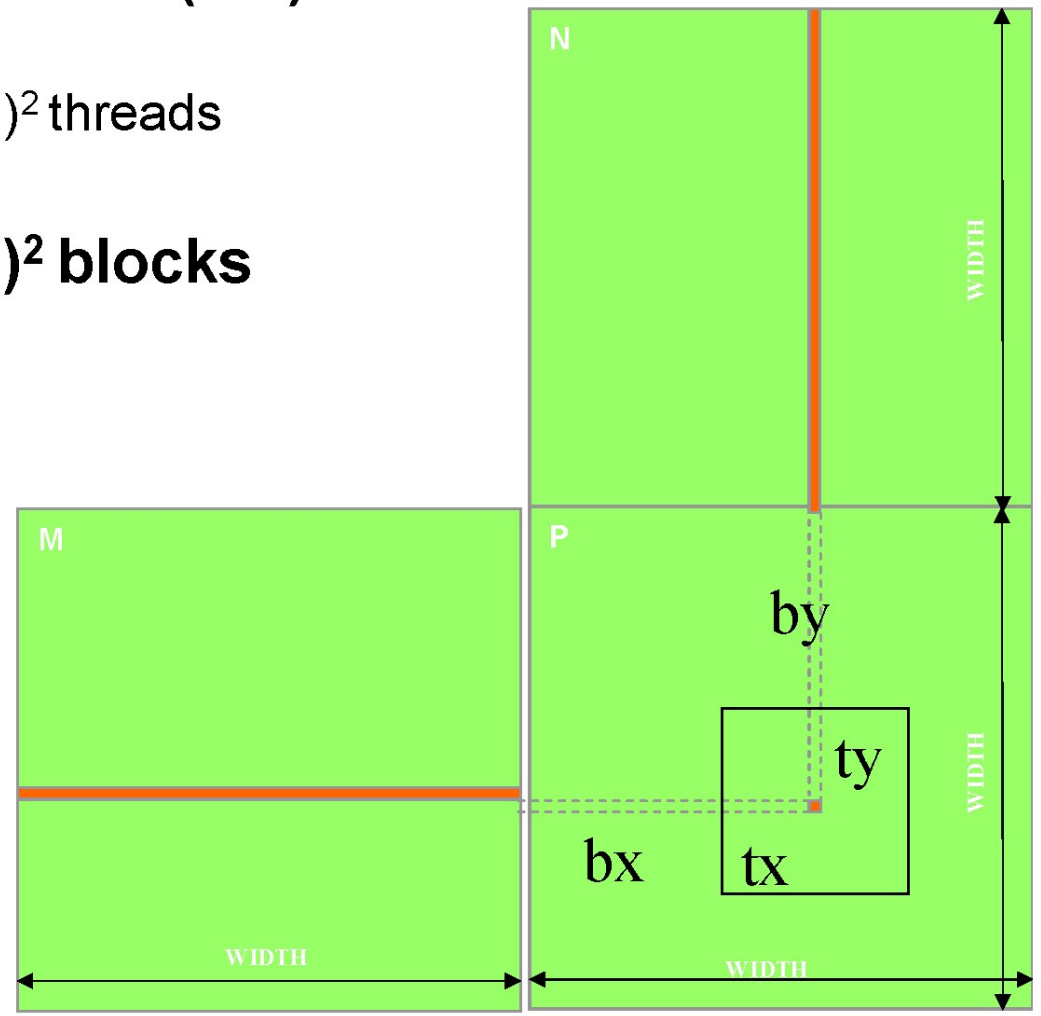
```
...
    for (int k = 0; k < M.width; ++k)
    {
      float Melement = M.elements[ty * M.pitch + k];
      float Nelement = Nd.elements[k * N.pitch + tx];
      Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty *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



Handling Arbitrary Sized Square Matrices

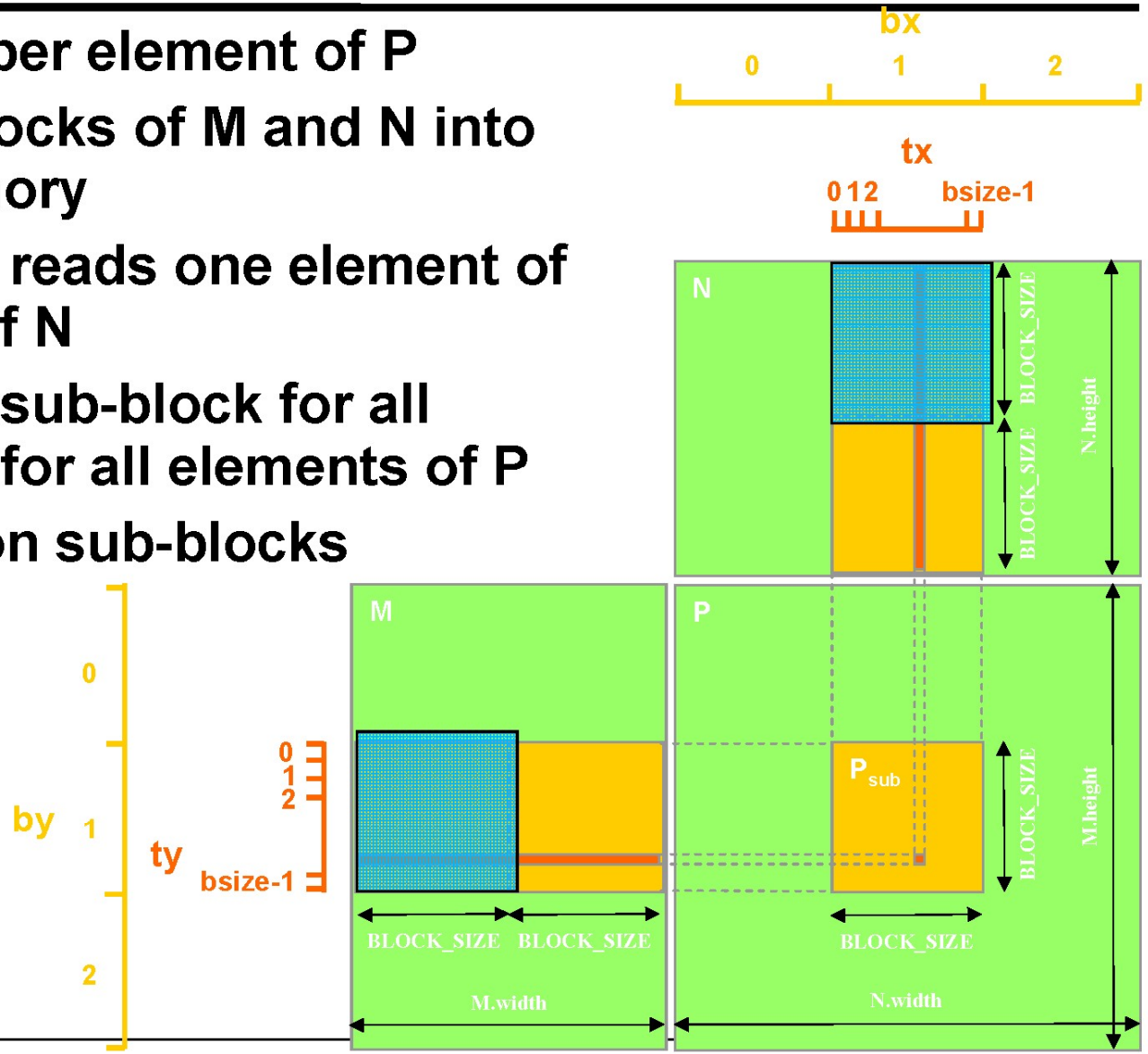
- **Have each 2D thread block to compute a $(BLOCK\_WIDTH)^2$ sub-matrix (tile) of the result matrix**
  - Each has $(BLOCK\_WIDTH)^2$ threads
- **Generate a 2D Grid of $(WIDTH/BLOCK\_WIDTH)^2$ blocks**

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



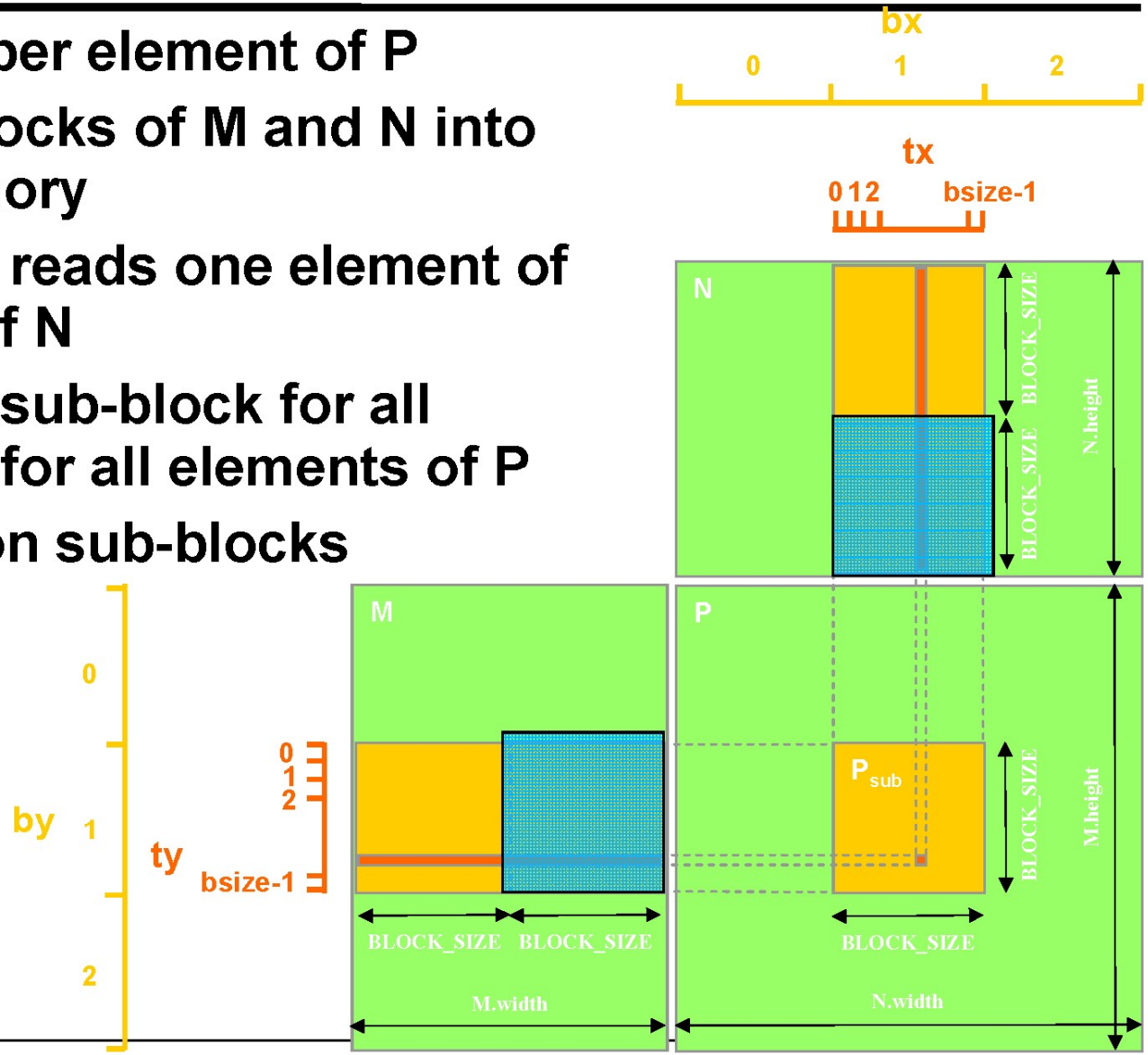
Multiply Using Several Blocks - Idea

- 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);
```

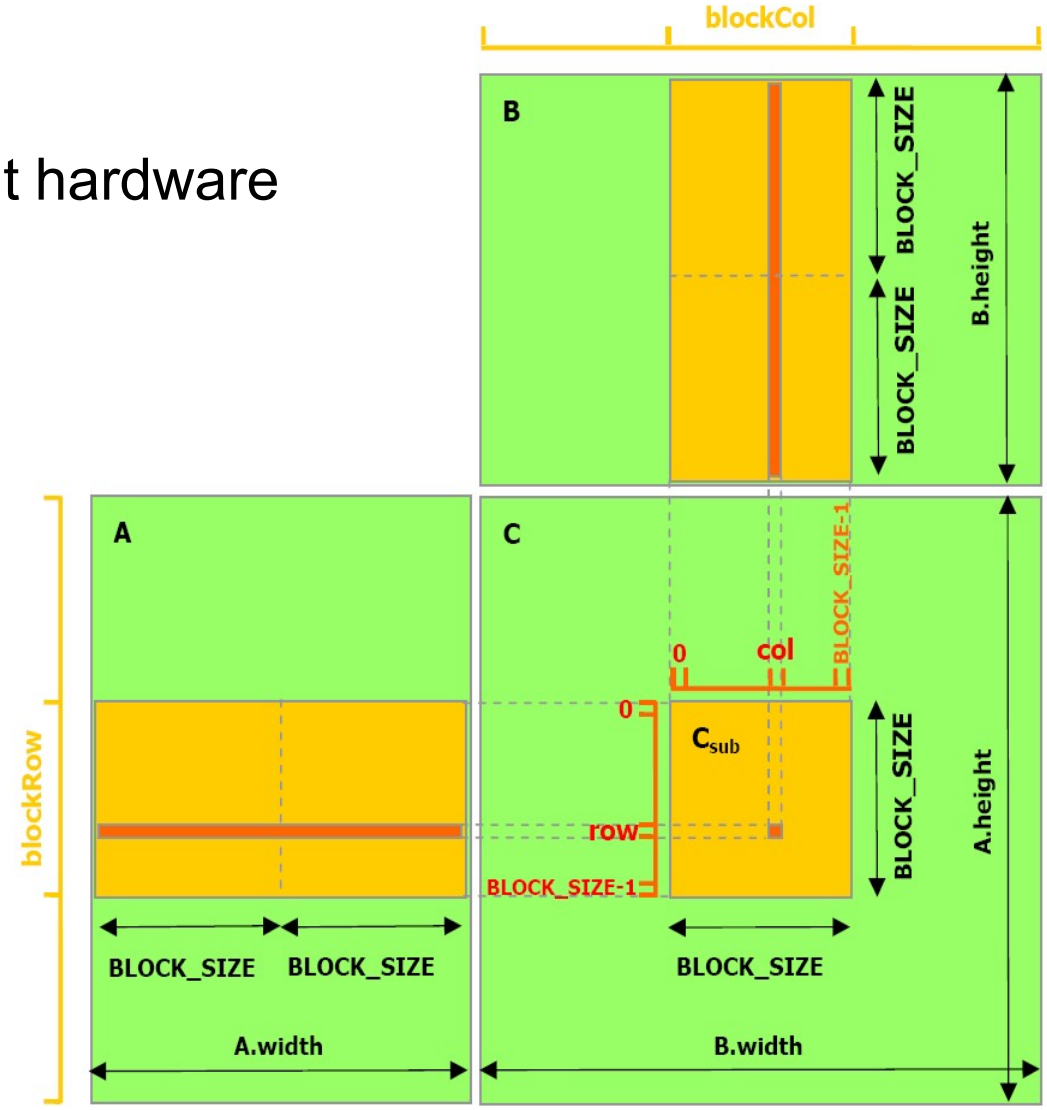
Example: Matrix Multiplication (2)

```
    // 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 per block
  - Perform row * column dot products in parallel

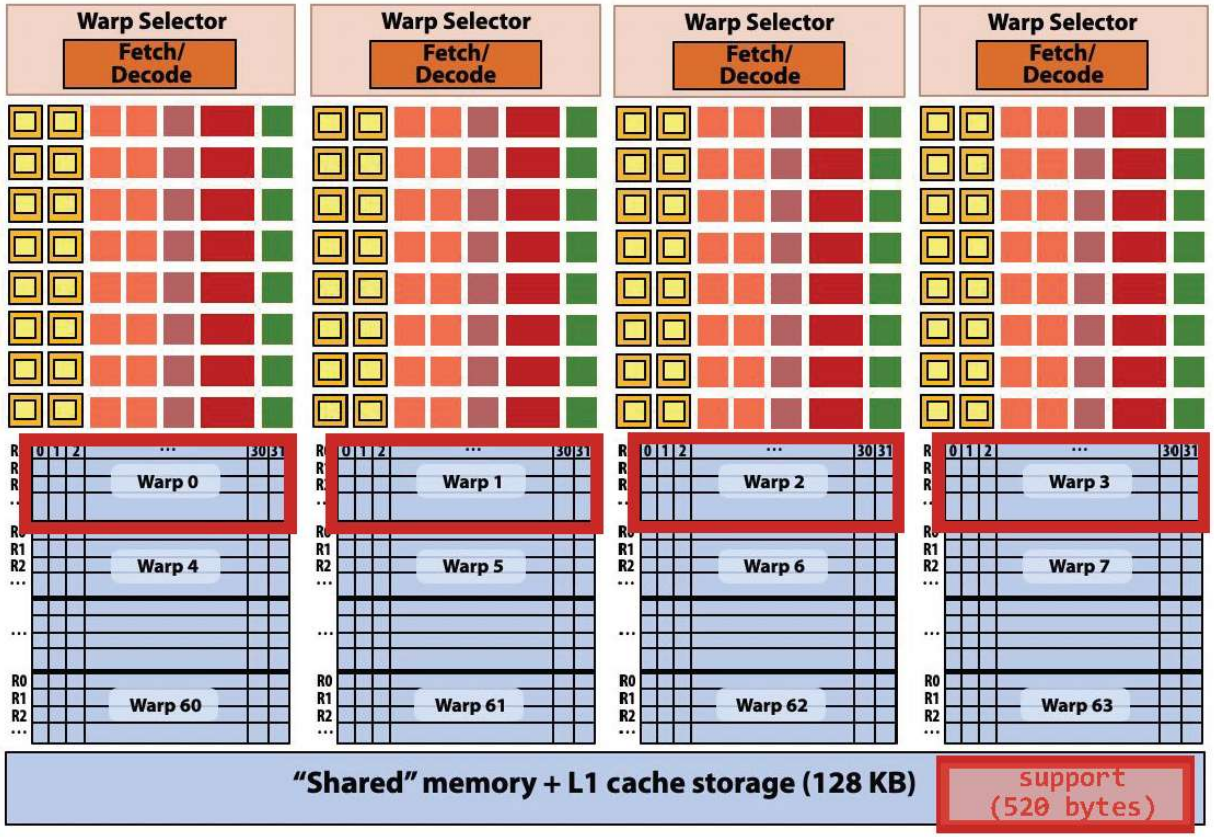


Example: Matrix Multiplication (4)

```
__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)

Running on a V100 (Volta) SM




```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input,
                         float* output)
{
   __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x +
               threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

A `convolve` thread block is executed by 4 warps
(4 warps x 32 threads/warp = 128 CUDA threads per block)

(sub-core == SM partition)

SM core operation each clock:

- **Each sub-core selects one runnable warp (from the 16 warps in its partition)**
- **Each sub-core runs next instruction for the CUDA threads in the warp (this instruction may apply to all or a subset of the CUDA threads in a warp depending on divergence)**

courtesy Kayvon Fatahalian

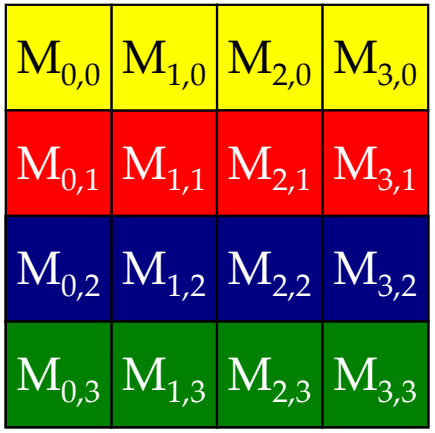
Limits in CUDA Programming Guide

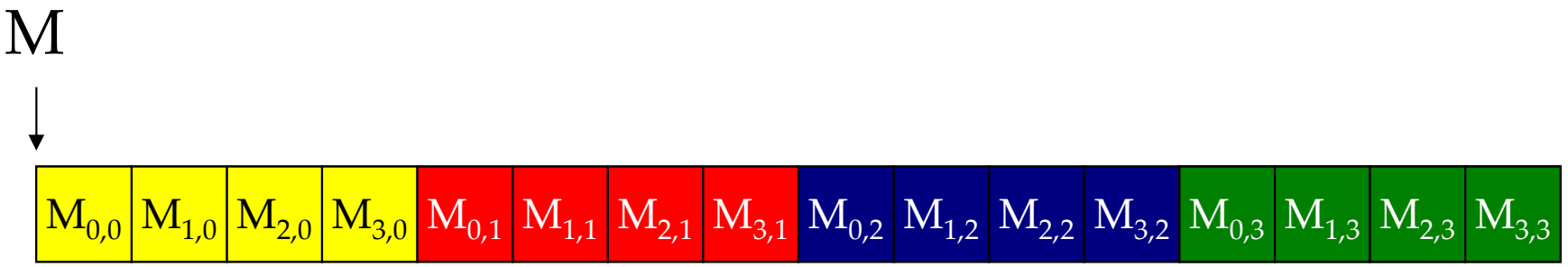
Chapter 20.2
(CUDA 13,
Sept 2, 2025

	Compute Capability								
Technical Specifications	7.5	8.0	8.6	8.7	8.9	9.0	10.0	11.0	12.0
Maximum number of resident grids per device (Concurrent Kernel Execution)	128								
Maximum dimensionality of grid of thread blocks	3								
Maximum x -dimension of a grid of thread blocks	$2^{31}-1$								
Maximum y- or z-dimension of a grid of thread blocks	65535								
Maximum dimensionality of thread block	3								
Maximum x- or y-dimensionality of a block	1024								
Maximum z-dimension of a block	64								
Maximum number of threads per block	1024								
Warp size	32								
Maximum number of resident blocks per SM	16	32	16		24	32		24	
Maximum number of resident warps per SM	32	64	48			64		48	
Maximum number of resident threads per SM	1024	2048	1536			2048		1536	
Number of 32-bit registers per SM	64 K								
Maximum number of 32-bit registers per thread block	64 K								
Maximum number of 32-bit registers per thread	255								
Maximum amount of shared memory per SM	64 KB	164 KB	100 KB	164 KB	100 KB	228 KB			100 KB
Maximum amount of shared memory per thread block[27]	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB			99 KB
Number of shared memory banks	32								
Maximum amount of local memory per thread	512 KB								
Constant memory size	64 KB								

What About Memory Performance? (more to come later…)

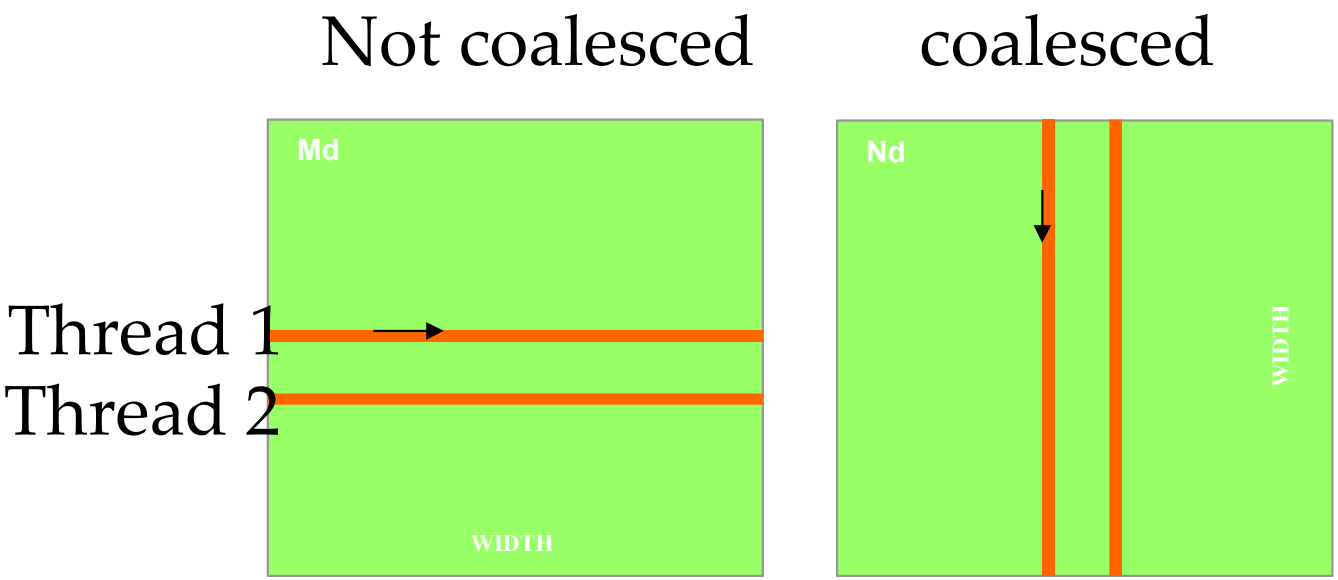
Memory Layout of a Matrix in C





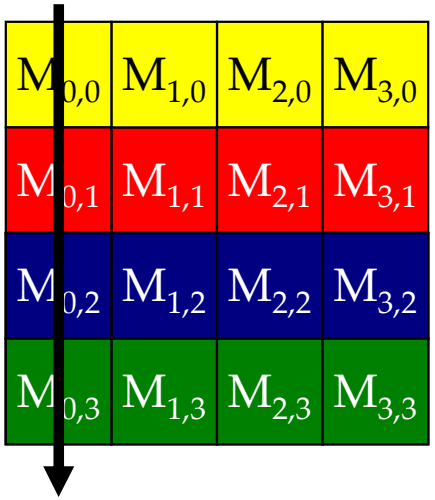
Memory Coalescing

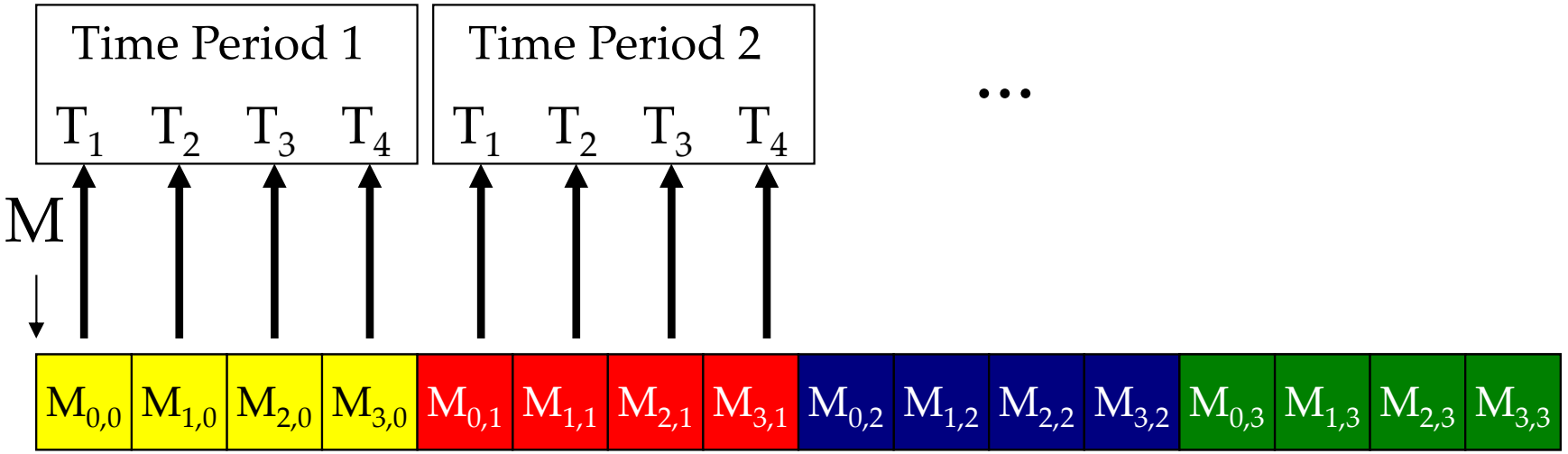
- When accessing global memory, peak performance utilization occurs when all threads in a half warp (full warp on Fermi+) access continuous memory locations.
- Requirements relaxed on >=1.2 devices; L1 cache on Fermi!



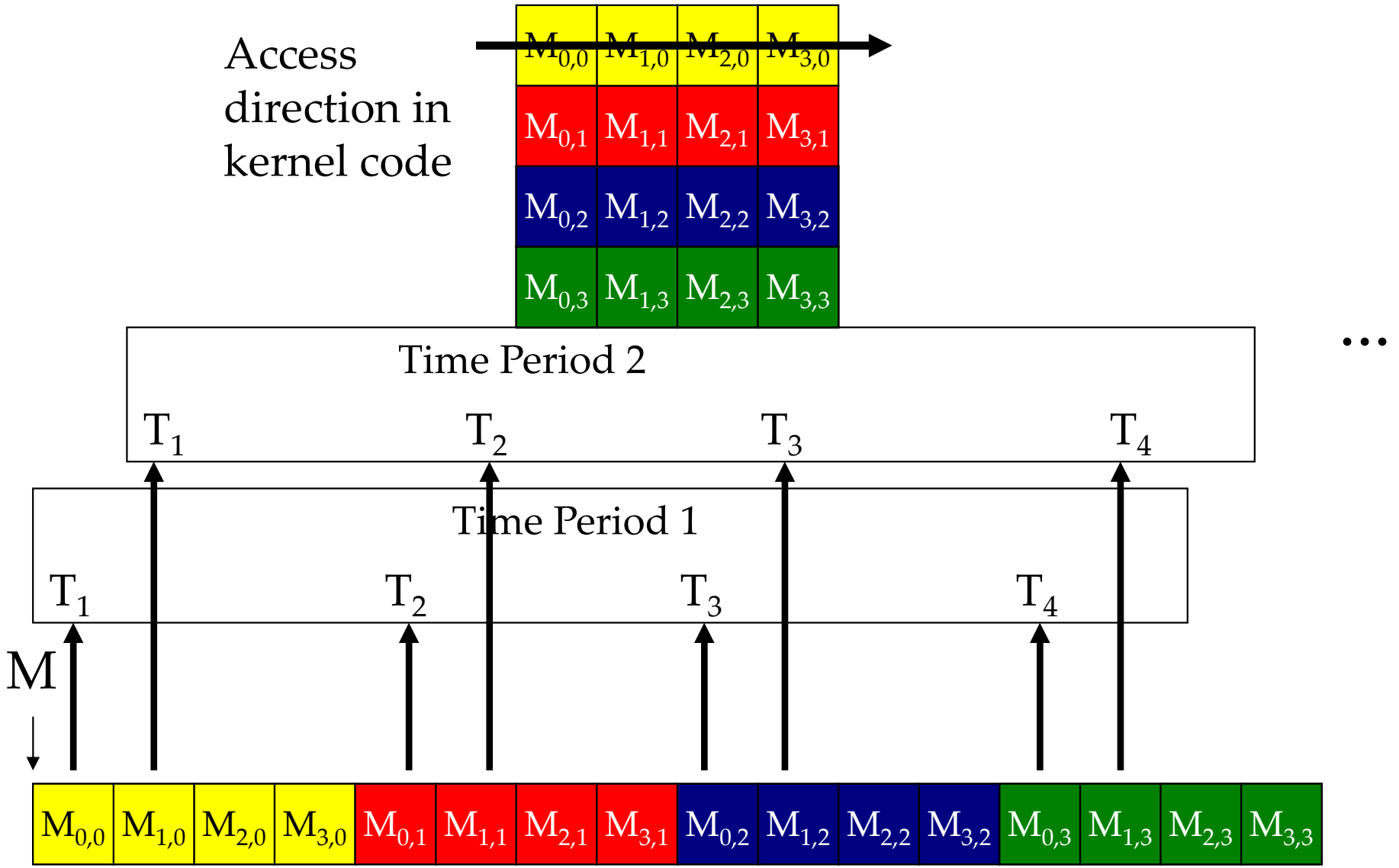
Memory Layout of a Matrix in C

Access direction in kernel code





Memory Layout of a Matrix in C



Thank you.