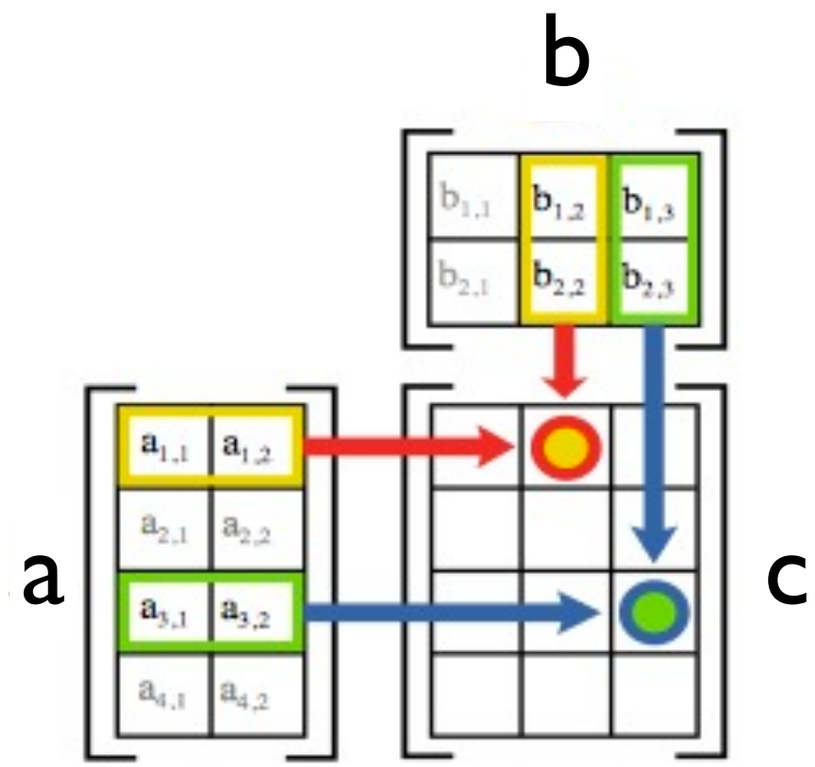
CUDA Advanced Threads

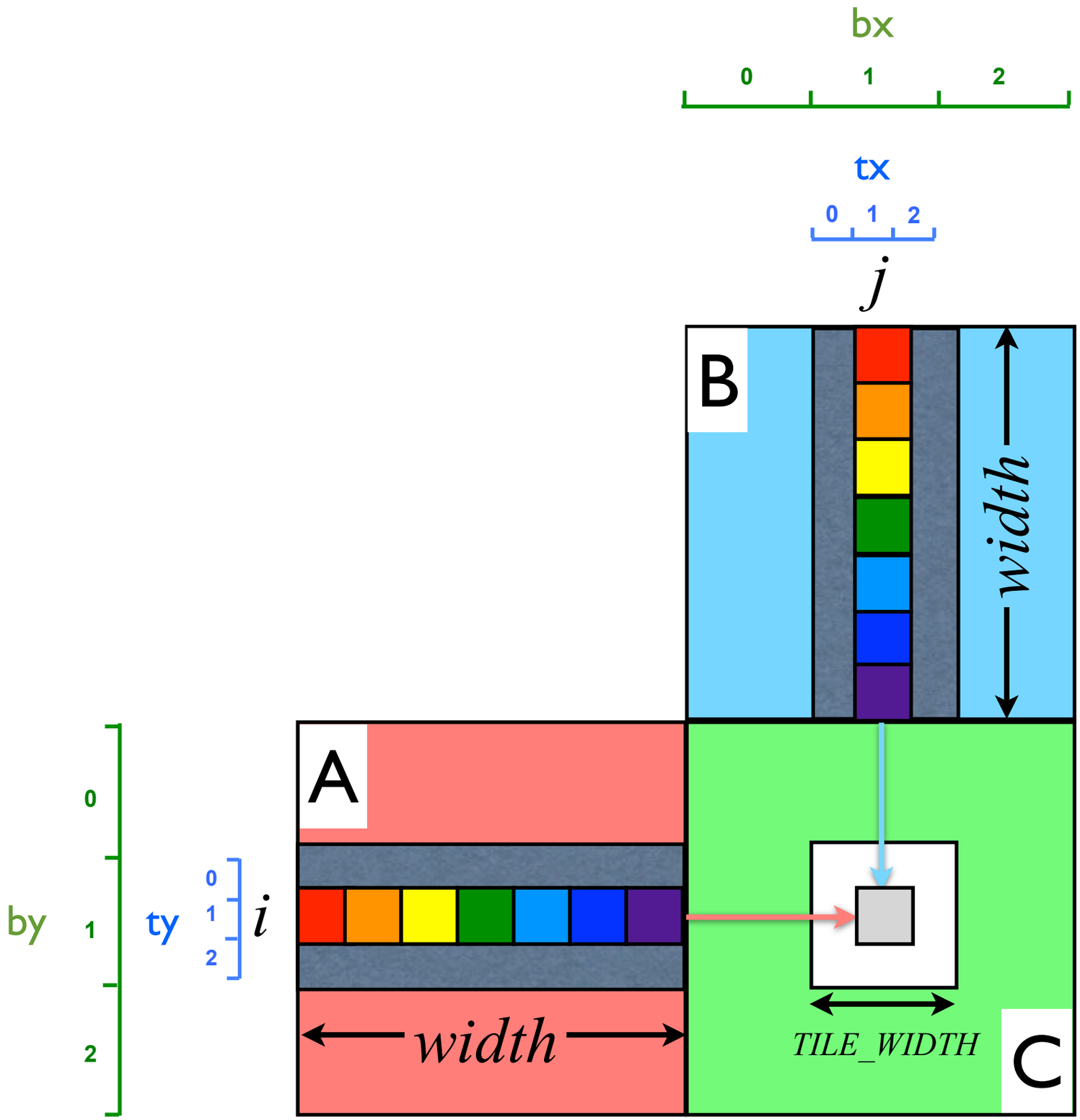
Matrix Multiply: Single Tile Approach

```
#define N 512

#include <stdio.h>

__global__ void matrixMult (int *a, int *b, int *c, int width);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 int size = N * N * sizeof(int);

 // initialize matrices a and b

 cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
 cudaMalloc((void **) &dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimGrid(1, 1);
 dim3 dimBlock(N, N);

 matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}

__global__ void matrixMult (int *A, int *B, int *C, int width)
{
 int k, sum = 0;
 int col = blockDim.x * blockIdx.x + threadIdx.x;
 int row = blockDim.y * blockIdx.y + threadIdx.y;

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += A[row * width + k] * B[k * width + col];
  C[row * width + col] = sum;
 }
}
```



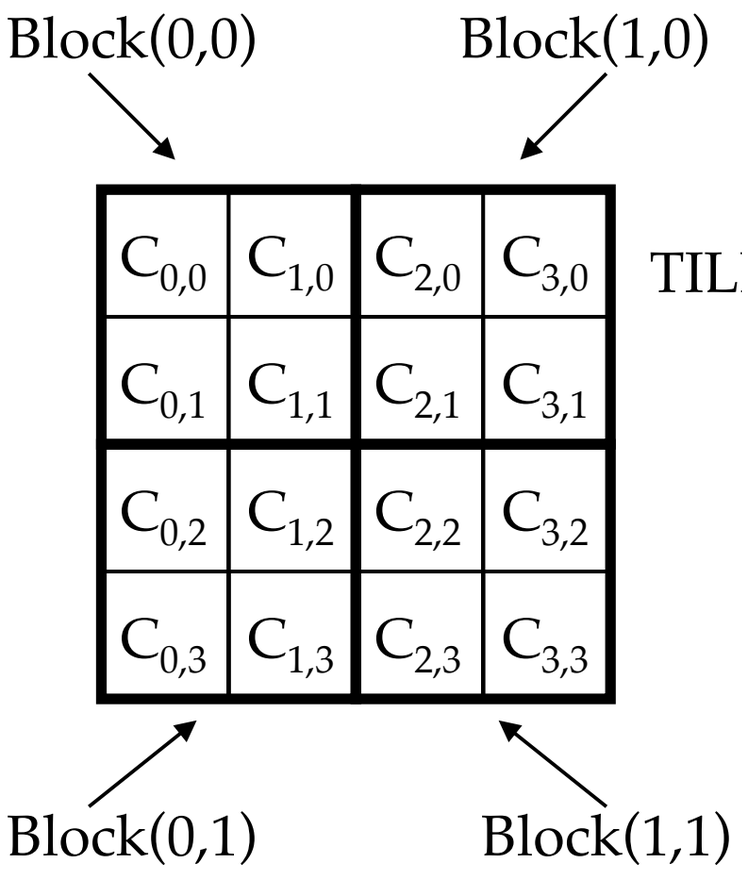
- One block of threads to compute matrix c.
  - Each thread computes the value of $c_{i,j}$.
- Each thread:
  - loads a row of matrix a and a column of matrix b.
  - performs one multiply and one add for each pair of $a_{i,j}$ and $b_{i,j}$ elements.
  - stores the result in $c_{i,j}$.

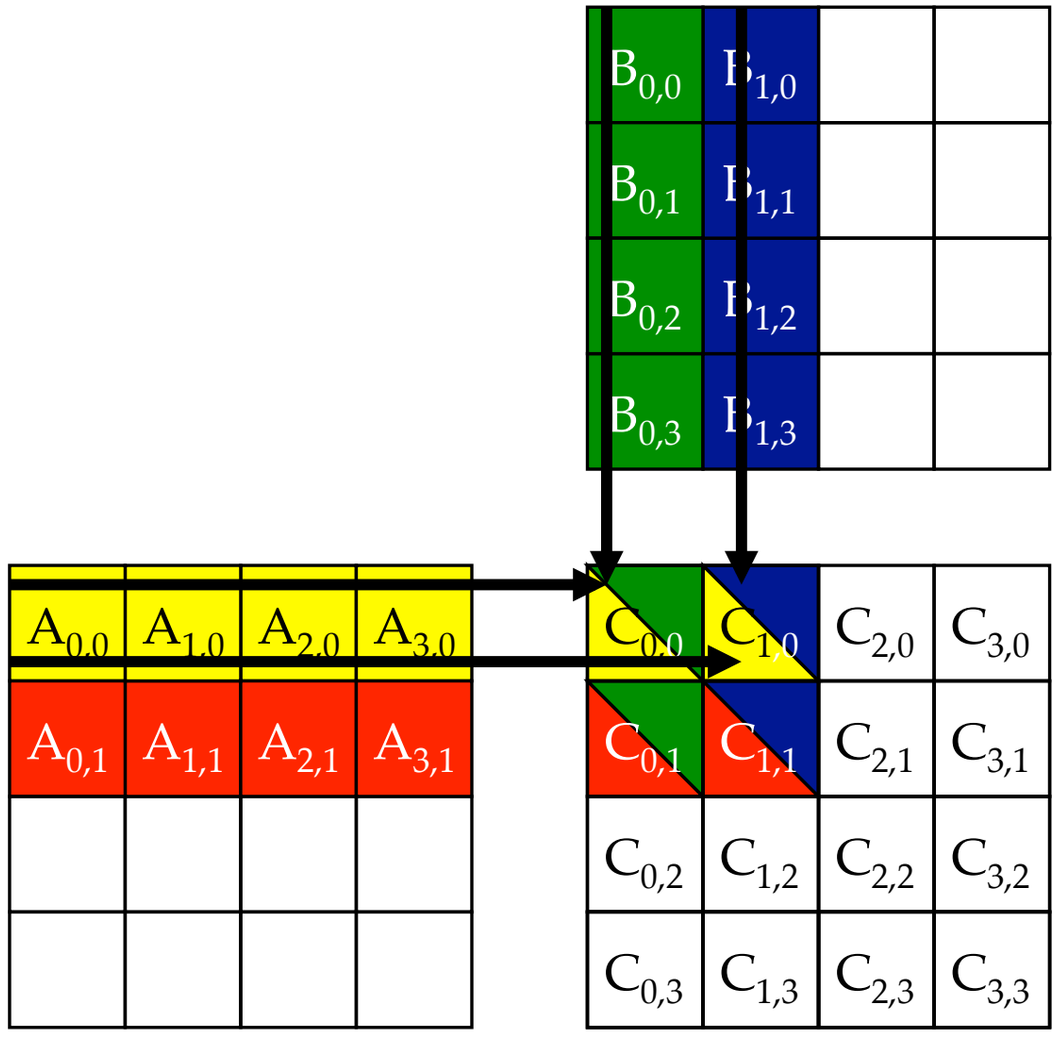
Matrix Multiplication Using Tiles



- Assume that the dimensions (width) of the square matrix is a multiple of the tile width.
- Break of matrix C into blocks.
- Each block calculates one submatrix (tile).
  - Each thread calculates one element of the tile.
  - Block size equals one tile width.

Small Example





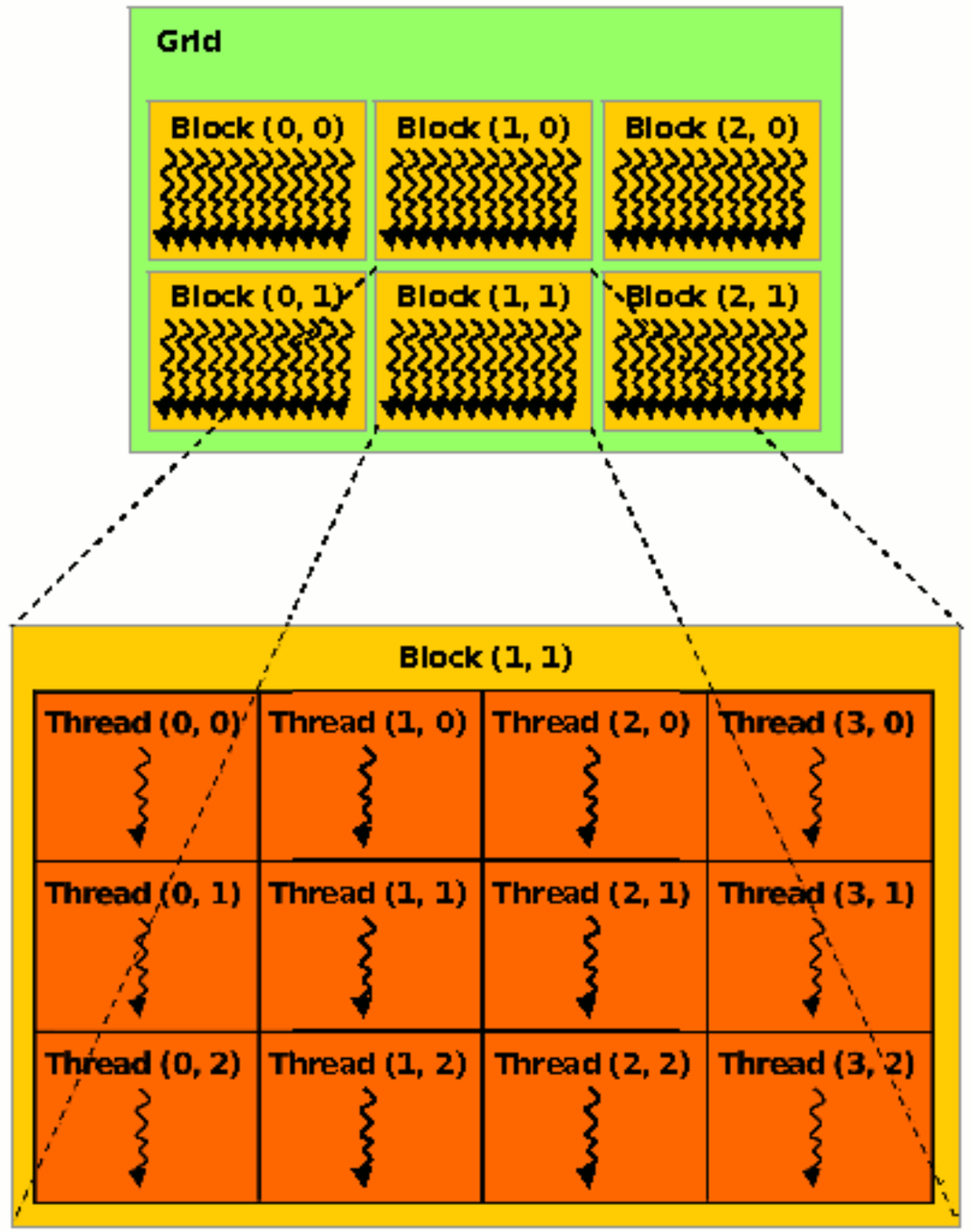
Matrix Multiply: Tile Approach

```
#define N 512
#define TILE_WIDTH 16
#include <stdio.h>

__global__ void matrixMult (int *a, int *b, int *c, int width);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 int size = N * N * sizeof(int);

 // initialize a and b matrices here

 cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
 cudaMalloc((void **) &dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
 dim3 dimGrid((int)ceil(N/dimBlock.x), (int)ceil(N/dimBlock.y));

 matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}

__global__ void matrixMult(int* A, int* B, int* C, int width)
{
 int k, sum = 0;
 int col = blockIdx.x*TILE_WIDTH + threadIdx.x;
 int row = blockIdx.y*TILE_WIDTH + threadIdx.y;

 if(col < width && row < width) {
  for (int k = 0; k < width; k++)
   sum += A[row * width + k] * B[k * width + col];
  C[row * width + col] = sum;
 }
}
```

```
#define N 512

#include <stdio.h>

__global__ void matrixMult (int *a, int *b, int *c, int width);

int main() {
 int a[N][N], b[N][N], c[N][N];
 int *dev_a, *dev_b, *dev_c;

 int size = N * N * sizeof(int);

 // initialize matrices a and b

 cudaMalloc((void **) &dev_a, size);
 cudaMalloc((void **) &dev_b, size);
 cudaMalloc((void **) &dev_c, size);

 cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);

 dim3 dimGrid(1, 1);
 dim3 dimBlock(N, N);

 matrixMult<<<dimGrid, dimBlock>>>(dev_a, dev_b, dev_c, N);

 cudaMemcpy(c, dev_c, size, cudaMemcpyDeviceToHost);

 cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}

__global__ void matrixMult (int *A, int *B, int *C, int width)
{
 int k, sum = 0;
 int col = blockDim.x * blockIdx.x + threadIdx.x;
 int row = blockDim.y * blockIdx.y + threadIdx.y;

 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += A[row * width + k] * B[k * width + col];
  C[row * width + col] = sum;
 }
}
```

NVIDIA GPU Memory Hierarchy

- Grids map to GPUs
- Blocks map to the MultiProcessors (MP)
- Threads map to Stream Processors (SP)
- Warps are groups of (32) threads that execute simultaneously



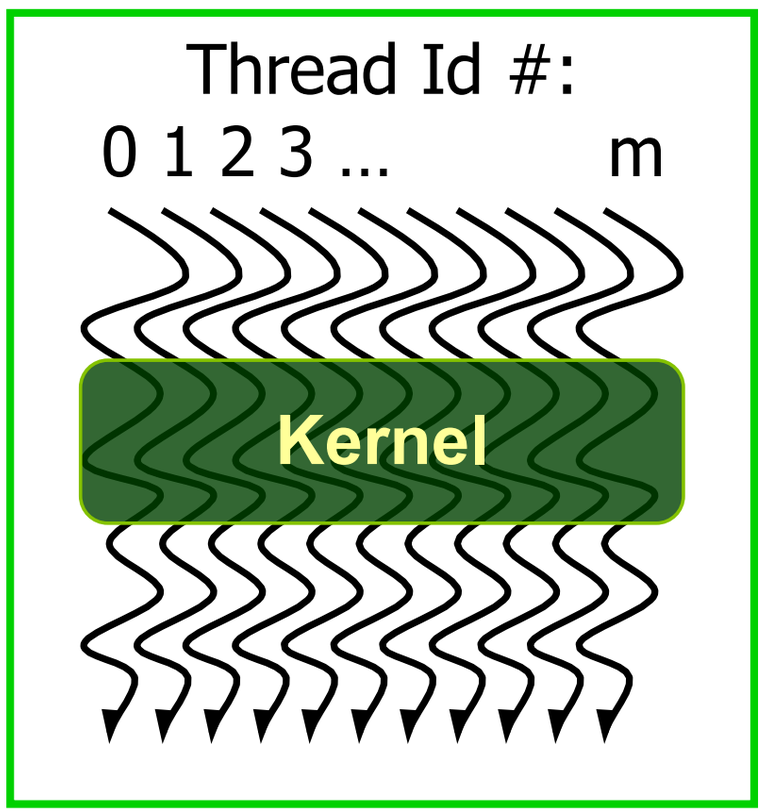
NVIDIA GPU Blocks and Grids

- In CUDA, a block is a group of threads.
  - They can execute concurrently or independently, and in no particular order.
  - Threads can be coordinated somewhat, using the _syncthreads() function as a barrier, making all threads stop at a certain point in the kernel before moving on en mass.

- In CUDA, a grid is a group of (thread) blocks, with no synchronization at all among the blocks.

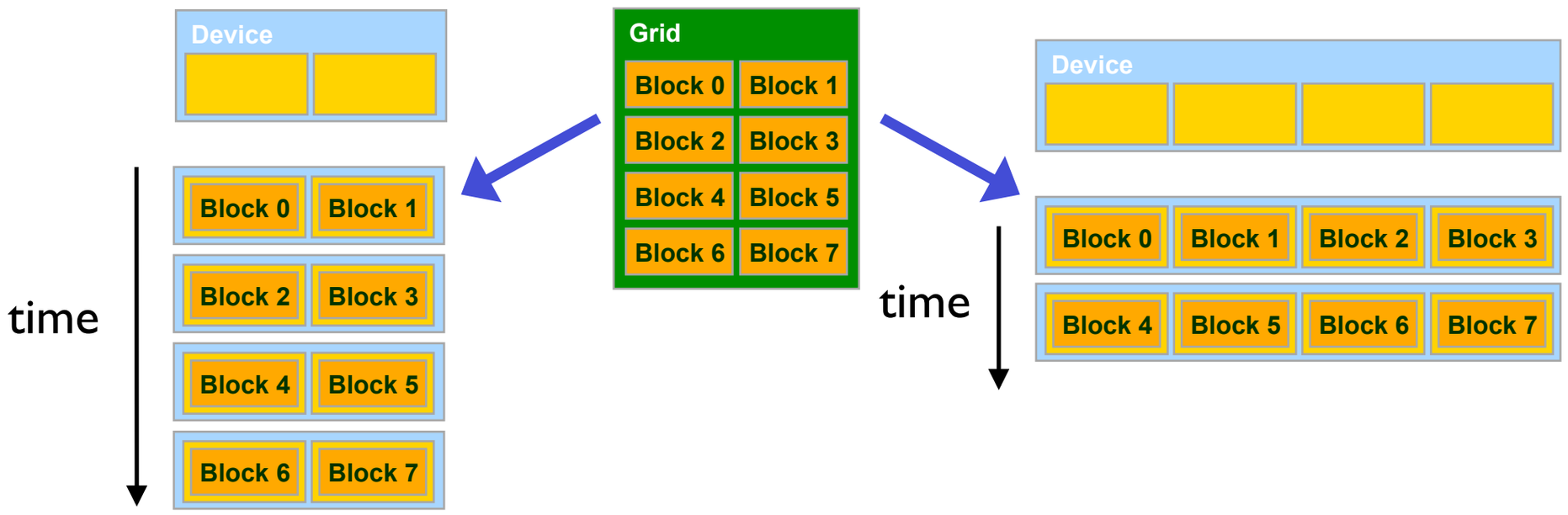
CUDA Thread Block

- All threads in a block execute the same kernel program.
  - Programmer declares block:
  - Block size 1 to 512 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data that can be synchronized while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!



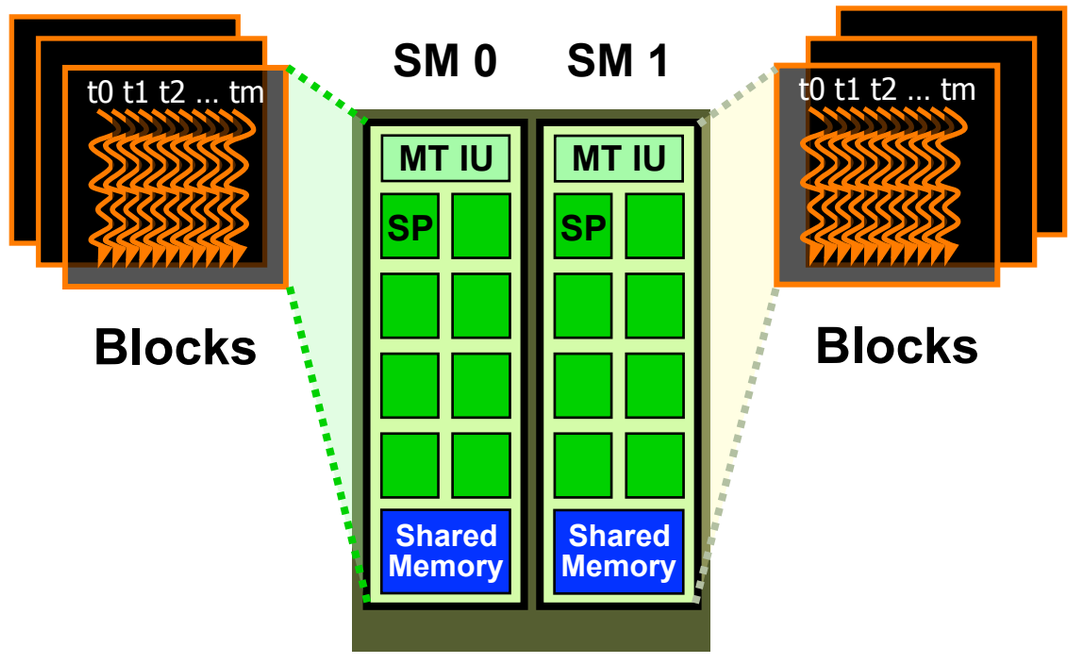
CUDA Thread Block

- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors
  - Each block can execute in any order relative to other blocks.



Executing Thread Blocks

- Threads are assigned to streaming multiprocessors (SMs) in block granularity
  - Up to 8 blocks to each SM as resource allows
  - Each SM can take up to 768 threads
    - Could be 256 (threads/block) × 3 blocks
    - Or 128 (threads/block) × 6 blocks, etc.
- Threads run concurrently
  - Each SM maintains thread/block id numbers
  - Each SM manages/schedules thread execution

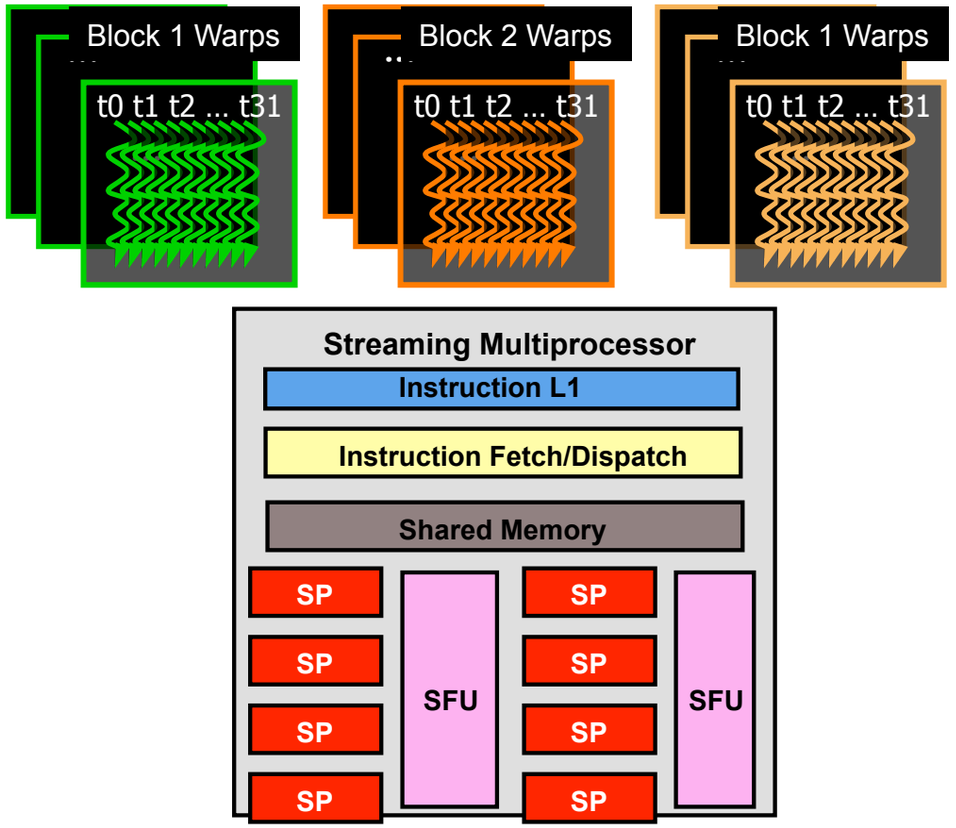


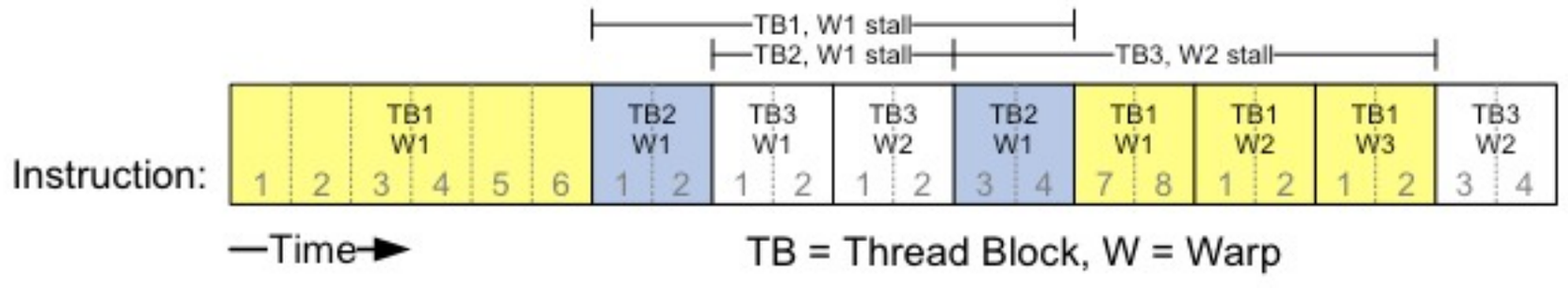
Thread Scheduling

- Each block is executed as 32-thread warps
  - An implementation decision, not part of the CUDA programming model.
  - Warps are scheduling units in an SM.

- 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

- Each SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by an SM
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy.
- All threads in a warp execute the same instruction when selected.





Block Granularity Issues

- For matrix multiplication using multiple blocks, should I use 8 × 8, 16 × 16 or 32 × 32 blocks?

- For 8 × 8, we have 64 threads per block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!

- For 16 × 16, 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 unless other resource considerations overrule.

- For 32 × 32, we have 1024 threads per Block. Not even one can fit into an SM!