# Lecture 11: Programming on GPUs (Part 2)

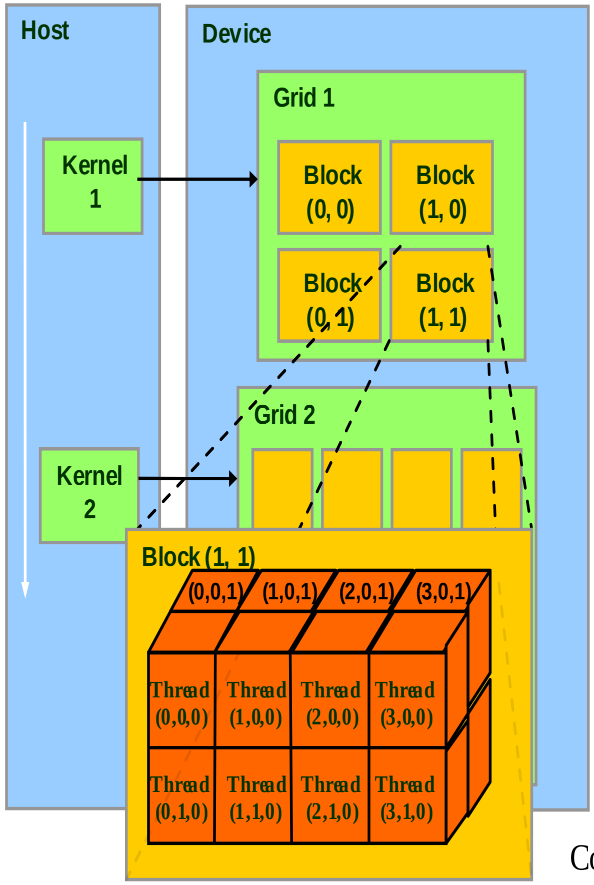
# GPU Vector Sums – Another Thread Allocation Method

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>

#define N 50
__global__ void add(int *a, int *b, int *c){
   int tid = threadIdx.x;  // handle the data at this index

   if(tid < N)    c[tid] = a[tid] + b[tid];
}

int main()
{
   int a[N], b[N], c[N], i;
   int *dev_a, *dev_b, *dev_c;
   …
   add <<<1, N>>>(dev_a, dev_b, dev_c);
   cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
   for(i=0; i < N; i++)
      printf("%d + %d = %d\n", a[i], b[i], c[i]);
   …
   return 0;
}
```

## kernel_routine<<<gridDim, blockDim>>>(args);

- A collection of blocks from a **grid** (1D, or 2D)
  - Built-in variable **gridDim** specifies the size (or dimension) of the grid.
  - Each copy of the kernel can determine which block it is executing with the built-in variable **blockIdx**.

- Threads in a block are arranged in 1D, 2D, or 3D arrays.
  - Built-in variable **blockDim** specifies the size (or dimensions) of block.
  - **threadIdx** index (or 2D/3D indices) thread within a block
  - maxThreadsPerBlock: The limit is 512 threads per block



Courtesy: NDVIA

# Language Extensions: Built-in Variables

- **`dim3 gridDim;`**
  - Dimensions of the grid in blocks (**`gridDim.z`** unused)
- **`dim3 blockDim;`**
  - Dimensions of the block in threads
- **`dim3 blockIdx;`**
  - Block index within the grid
- **`dim3 threadIdx;`**
  - Thread index within the block

# Specifying 1D Grid and 1D Block

```
/// host code
int main(int argc, char **argv) {
   float *h_x, *d_x; // h=host, d=device
   int nblocks=3, nthreads=4, nsize=3*4;

   h_x = (float *)malloc(nsize*sizeof(float));
   cudaMalloc((void **)&d_x,nsize*sizeof(float));
   my_first_kernel<<<nblocks,nthreads>>>(d_x);
   cudaMemcpy(h_x,d_x,nsize*sizeof(float),
   cudaMemcpyDeviceToHost);
   for (int n=0; n<nsize; n++)
      printf(" n, x = %d %f \n",n,h_x[n]);
   cudaFree(d_x); free(h_x);
}
```

```
/// Kernel code
__global__ void my_first_kernel(float *x)
{
int tid = threadIdx.x + blockDim.x*blockIdx.x;
x[tid] = (float) threadIdx.x;
}
```

| Block 0 | Thread 0 | Thread 1 | Thread 2 | Thread 3 |
|---|---|---|---|---|
| Block 1 | Thread 0 | Thread 1 | Thread 2 | Thread 3 |
| Block 2 | Thread 0 | Thread 1 | Thread 2 | Thread 3 |

Within each block of threads, **threadIdx.x** ranges from 0 to **blockDim.x-1**, so each thread has a unique value for tid

# GPU SUMs of a Long Vector

- Assume 65,535*512 >> N > 512, so we need to launch threads across multiple blocks.
- Let's use 128 threads per block. We need N/128 blocks.
  - N/128 is integer division. If N were < 128, N/128 would be **0**.
  - Actually compute (N+127)/128 blocks.
- **add <<<(N+127)/128, 128>>>(dev_a, dev_b, dev_c);**

```
#define N 4000
__global__ void add(int *a, int *b, int *c){
   int tid = threadIdx.x + blockDim.x*blockIdx.x;  // handle the data at this index

   if(tid < N)    c[tid] = a[tid] + b[tid]; // launch too many treads when N is not exact
}                                           // multiple of 128
```

# Specifying 1D Grid and 2D Block

If we want to use a 1D grid of blocks and 2D set of threads, then **blockDim.x, blockDim.y** give the block dimensions, and **threadIdx.x, threadIdx.y** give the thread indices.

```
Main()
{
   int nblocks = 2;
   dim3  nthreads(16, 4);
   my_second_kernel<<<nblocks, nthreads>>>(d_x);
}
```

**dim3** is a special CUDA datatype with 3 components **.x, .y, .z** each initialized to 1.

```
/// Kernel code
__global__ void my_second_kernel(float *x)
{
int tid = threadIdx.x + blockDim.x* threadIdx.y +blockDim.x*blockDim.y*blockIdx.x;
x[tid] = (float) threadIdx.x;
}
```

- In 3D blocks of threads, thread ID is computed by:

threadIdx.x +threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockDim.y

```
__global__ void KernelFunc(...);

main()
{
  dim3   DimGrid(100, 50);   // 5000 thread blocks
  dim3   DimBlock(4, 8, 8);  // 256 threads per block
  KernelFunc<<< DimGrid, DimBlock>>>(...);
}
```

## GPU Sums of Arbitrarily Long Vectors

- Neither dimension of a grid of blocks may exceed 65,535.
- Let’s use 1D grid and 1D block.

```
__global__ void add(int *a, int *b, int *c){
   int tid = threadIdx.x + blockIdx.x*blockDim.x;  // handle the data at this index

   while(tid < N){
       c[tid] = a[tid] + b[tid];
       tid += blockDim.x*gridDim.x;
    }
}
```

Principle behind this implementation:

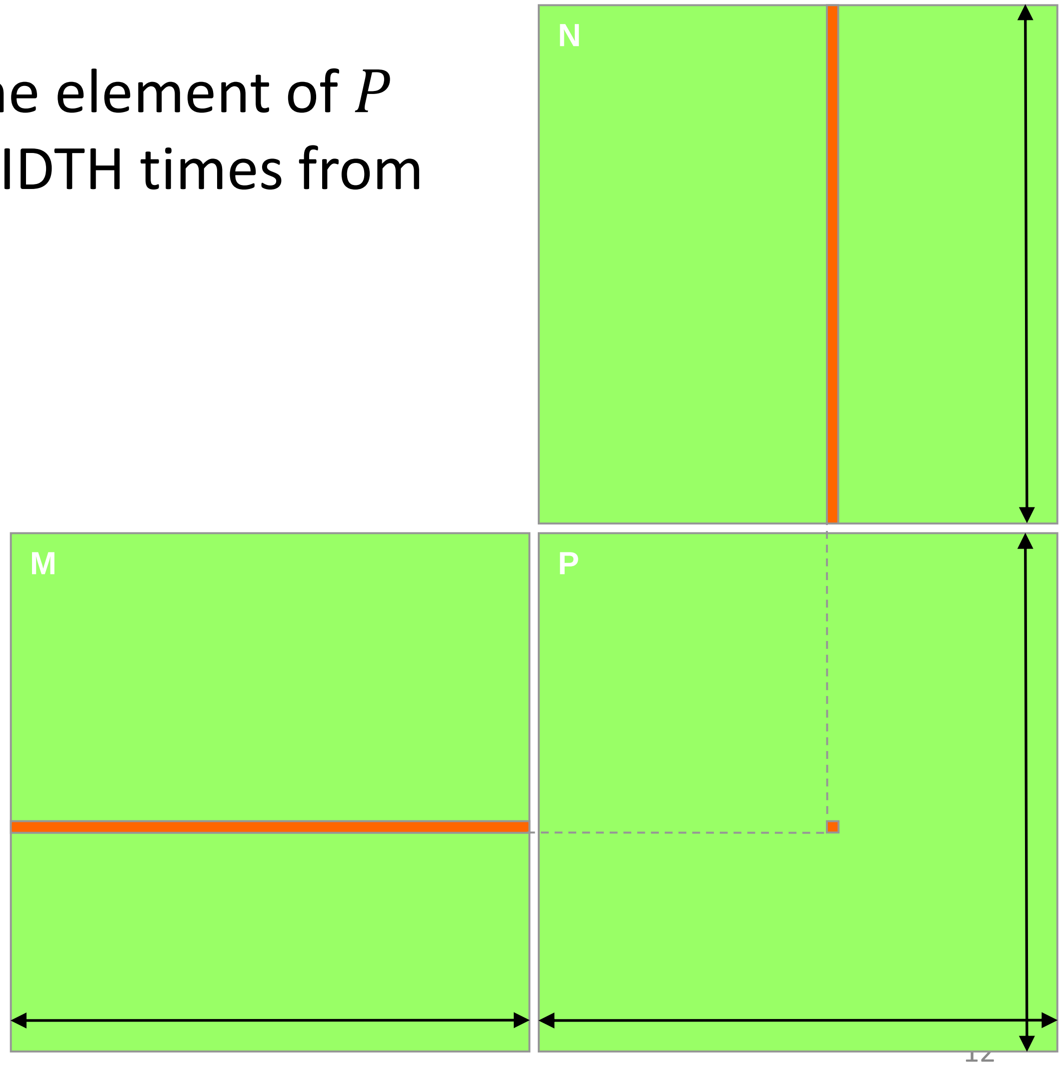
- Initial index value for each parallel thread is:
  **int tid = threadIdx.x + blockIdx.x*blockDim.x;**
- After each thread finishes its work at current index, increment each of them by the total number of threads running in the grid, which is **blockDim.x*gridDim.x**

```
#define N (55*1024)
__global__ void add(int *a, int *b, int *c){
   int tid = threadIdx.x + blockIdx.x*blockDim.x;  // handle the data at this index

   while(tid < N){
       c[tid] = a[tid] + b[tid];
       tid += blockDim.x*gridDim.x;
    }
}
int main()
{
…
   add <<<128, 128>>>(dev_a, dev_b, dev_c);
…
}
```

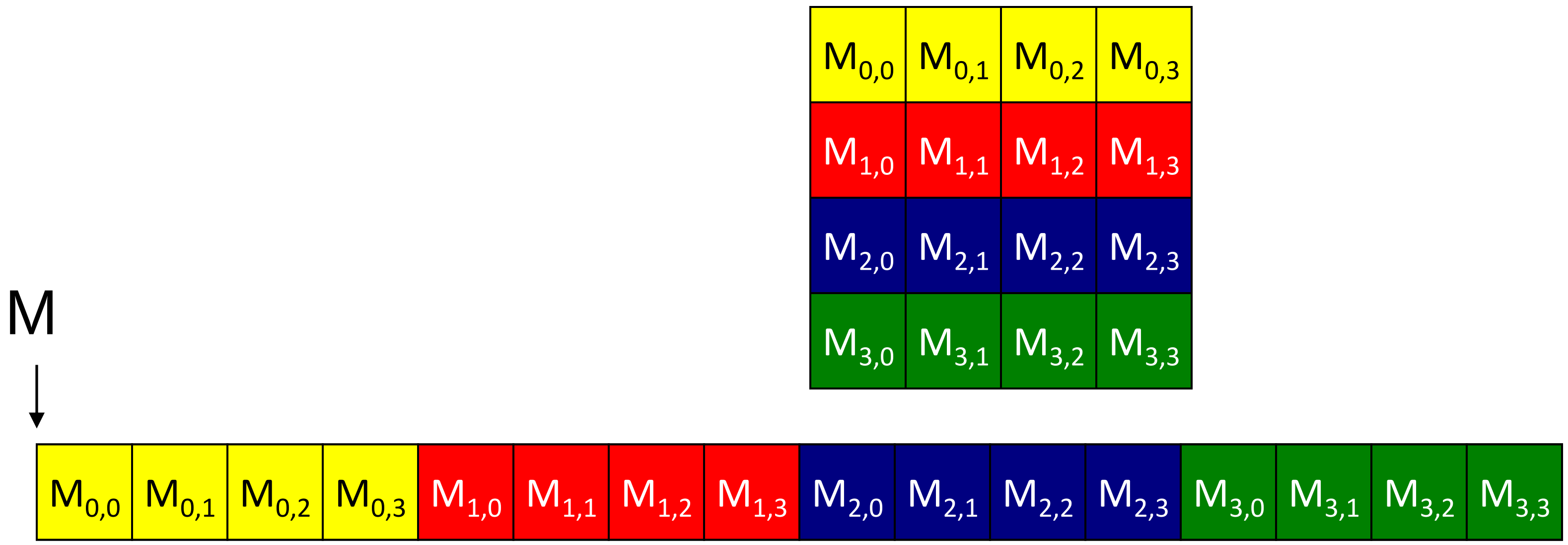
## Matrix Multiplication

- Demonstrate 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
  - Assume square matrix for simplicity

- $P = M \times N$ of size WIDTH×WIDTH
- **Without blocking**:
  - One thread handles one element of $P$
  - M and N are loaded WIDTH times from global memory



# C Language Implementation


M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

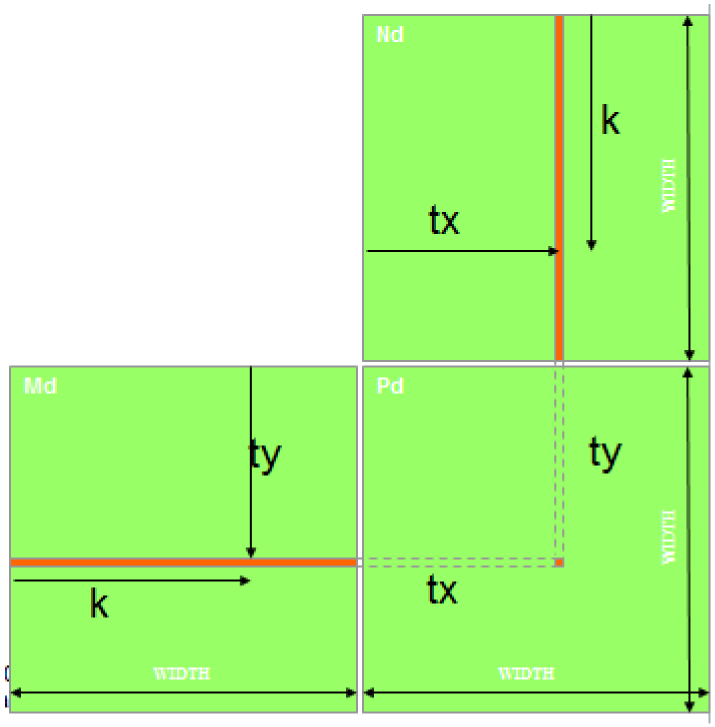
```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

## Data Transfer (Host/Device)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
    float *Md, *Nd, *Pd;
   …
    //1. Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

     cudaMalloc(&Nd, size);
     cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

     // Allocate P on the device
    cudaMalloc(&Pd, size);
    //2.  Kernel invocation code –
    …

     // 3. Read P from the device
     cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

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

# Kernel Function

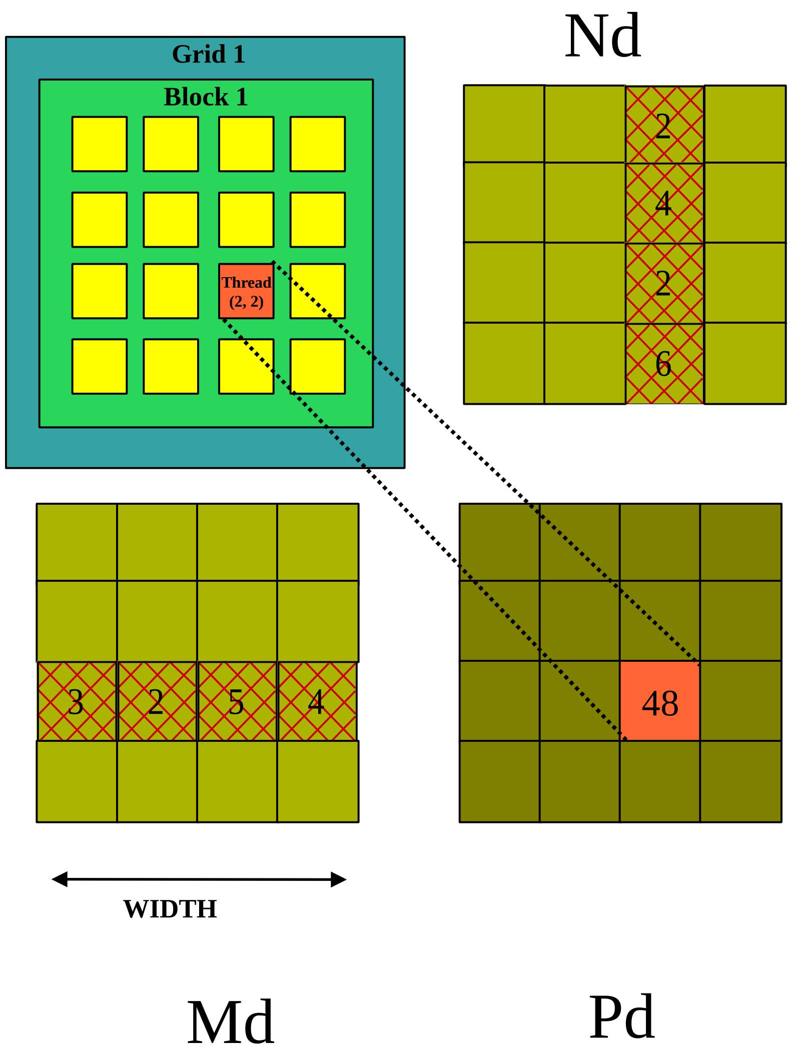
```
// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
       float Melement = Md[threadIdx.y*Width+k];
       float Nelement = Nd[k*Width+threadIdx.x];
       Pvalue += Melement * Nelement;
   }

  Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



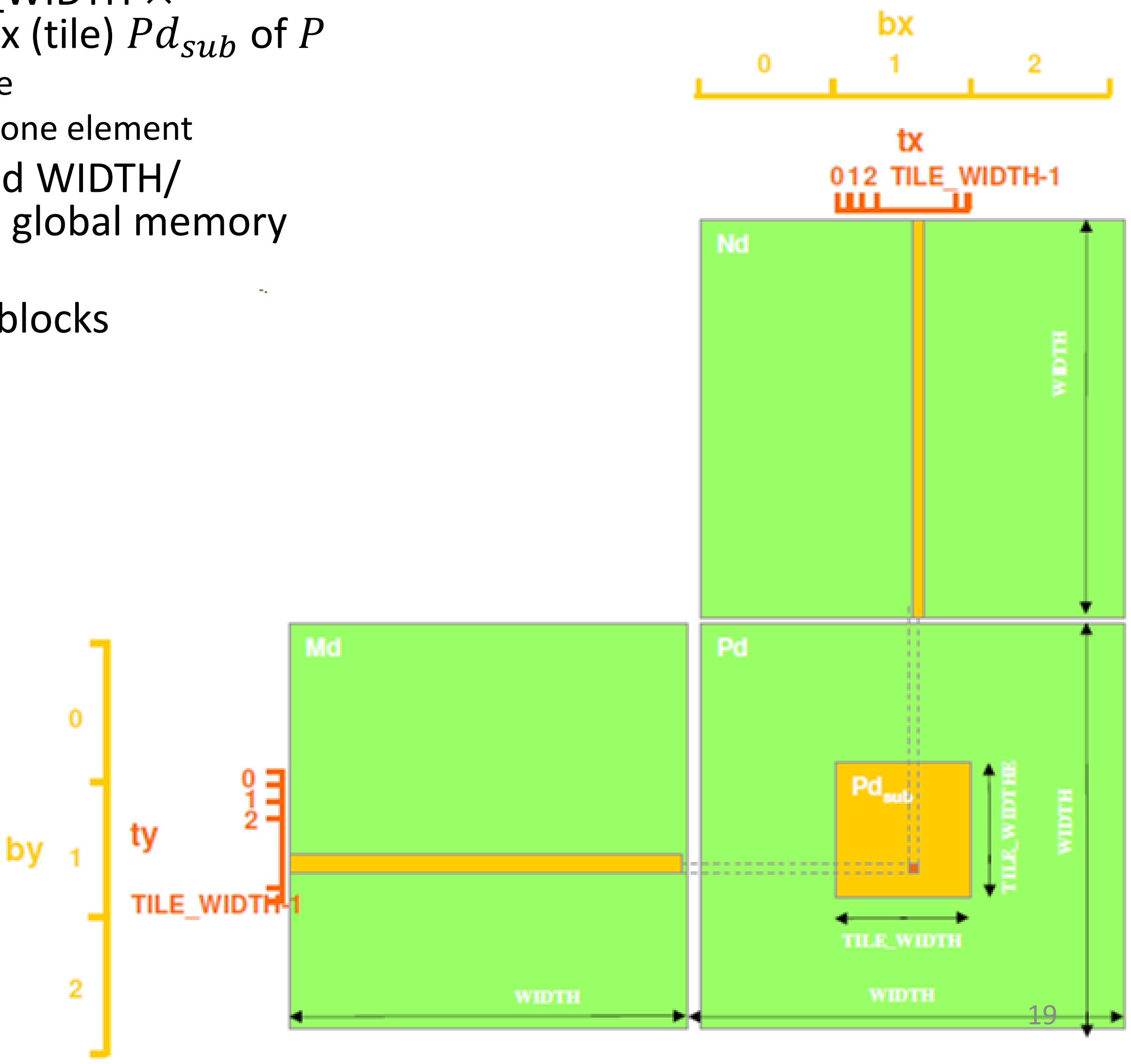
## Kernel Invocation

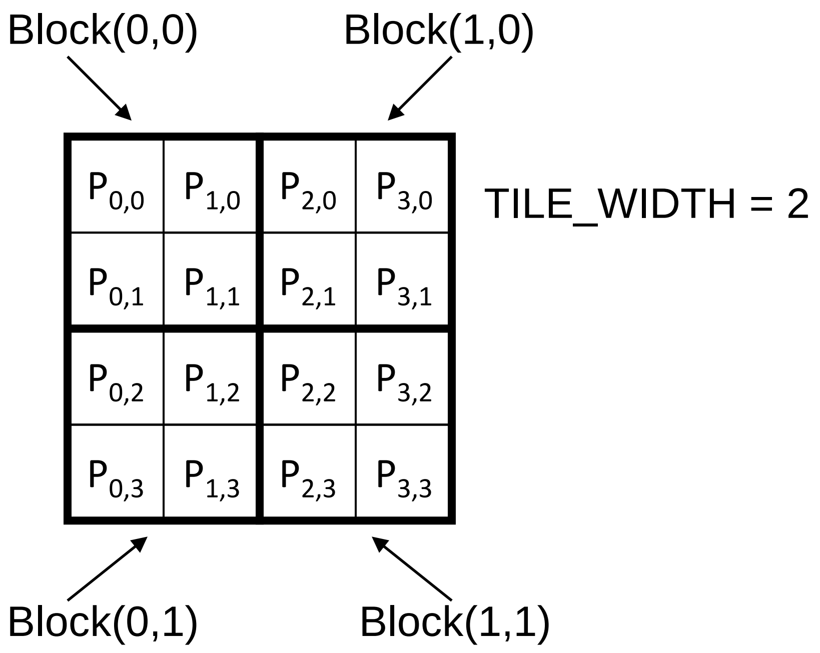
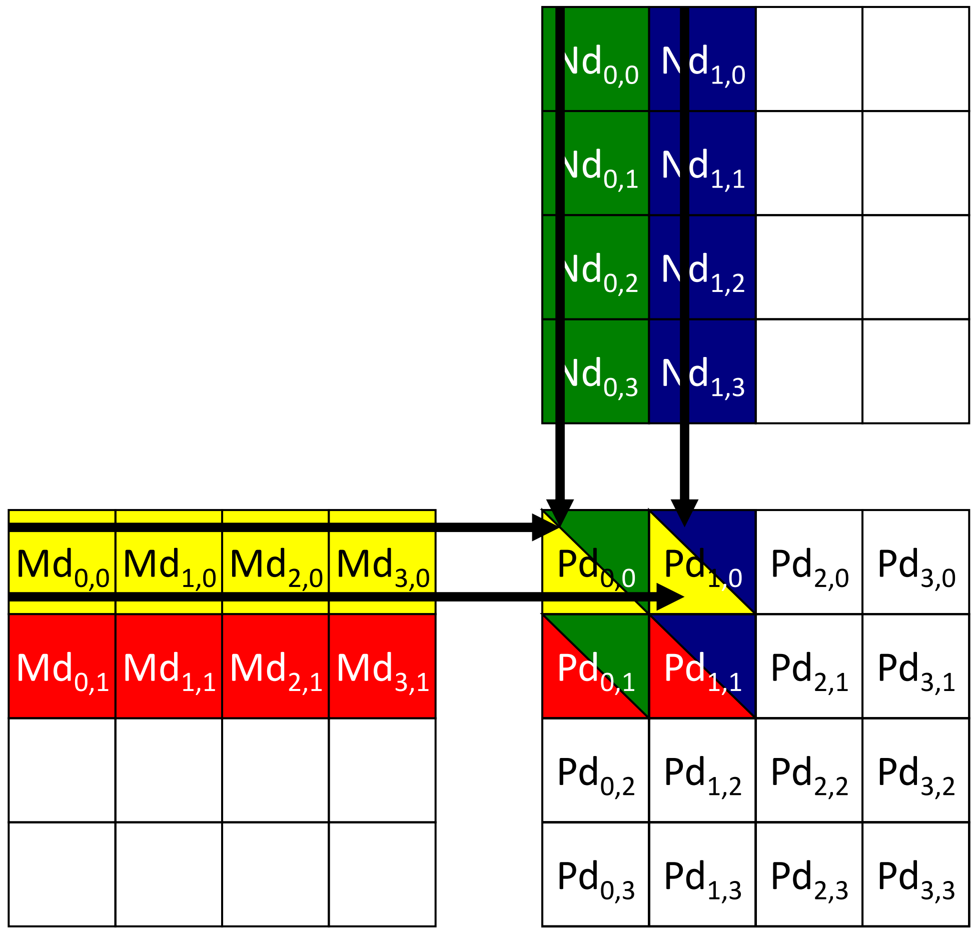
```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    …
   //2. Kernel invocation code – to be shown later
    // Setup the execution configuration
       dim3 dimGrid(1, 1);
      dim3 dimBlock(Width, Width);

    // Launch the device computation threads
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd,
    Width);
    …
}
```

- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd 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



- $P = M \times N$ of size WIDTH×WIDTH
- **With blocking**:
  - One **thread block** handles one BLOCK_SIZE × BLOCK_SIZE (or TILE_WIDTH × TILE_WIDTH) sub-matrix (tile) $Pd_{sub}$ of $P$
    - Block size equal tile size
    - Each thread calculates one element
  - M and N are only loaded WIDTH/ BLOCK_SIZE times from global memory
  - Genrate a 2D grid of (WIDTH/TILE_WIDTH)$^2$ blocks




Block(0,0)
Block(1,0)
Block(0,1)
Block(1,1)
P0,0 P1,0 P2,0 P3,0
P0,1 P1,1 P2,1 P3,1
P0,2 P1,2 P2,2 P3,2
P0,3 P1,3 P2,3 P3,3
TILE_WIDTH = 2

Nd0,0 Nd1,0
Nd0,1 Nd1,1
Nd0,2 Nd1,2
Nd0,3 Nd1,3
Md0,0 Md1,0 Md2,0 Md3,0
Md0,1 Md1,1 Md2,1 Md3,1
Pd0,0 Pd1,0 Pd2,0 Pd3,0
Pd0,1 Pd1,1 Pd2,1 Pd3,1
Pd0,2 Pd1,2 Pd2,2 Pd3,2
Pd0,3 Pd1,3 Pd2,3 Pd3,3

# Revised Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int
     Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

## Multithreading

- Cores in a streaming multiprocessor (SM) are Single Instruction Multiple Threads (SIMT) cores:
  - all cores execute the same instructions simultaneously, but with different data.
  - minimum of 32 threads all doing the same thing at (almost) the same time.
  - no “context switching”; each thread has its own registers, which limits the number of active threads
  - threads on each SM execute in groups of 32 called “warps” – execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data

- Suppose we have 1000 blocks, and each one has 128 threads – how does it get executed?
- On current Fermi hardware, would probably get 8 blocks running at the same time on each SM, and each block has 4 warps =) 32 warps running on each SM
- Each clock tick, SM warp scheduler decides which warp to execute next, choosing from those not waiting for
  - data coming from device memory (memory latency)
  - completion of earlier instructions (pipeline delay)
- Programmer doesn't have to worry about this level of detail (can always do profiling later), just make sure there are lots of threads / warps

## Spatial Locality

```
__global__ void good_kernel(float *x)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   x[tid] = threadIdx.x;
}
```

- 32 threads in a warp address neighboring elements of array x.
- If the data is correctly "aligned" so that x[0] is at the beginning of a cache line, then x[0]-x[31] will be in the same cache line.
  - Cache line is the basic unit of data transfer, 128 bytes cache line (32 floats or 16 doubles).
- Good spatial locality.

```
__global__ void bad_kernel(float *x)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    x[1000*tid] = threadIdx.x;
}
```

- Different threads within a warp access widely spaced elements of array x.
- Each access involves a different cache line, so performance is poor.

## Software View

At the top level, we have a master process which runs on the CPU and performs the following steps:

1. initializes card
2. allocates memory in host and on device
   – cudaMalloc(),…
3. copies data from host to device memory
   – cudaMemcpy(…, cudaMemcpyHostToDevice);
4. launches multiple instances of execution "kernel" on device
   – kernel_routine<<<gridDim, blockDim>>>(args);
5. copies data from device memory to host
   – cudaMemcpy(…, cudaMemcpyDeviceToHost);
6. repeats 3-5 as needed
7. de-allocates all memory and terminates
   – cudaFree()

## Software View

At a lower level, within the GPU:

1. each instance of the execution kernel executes on an SM

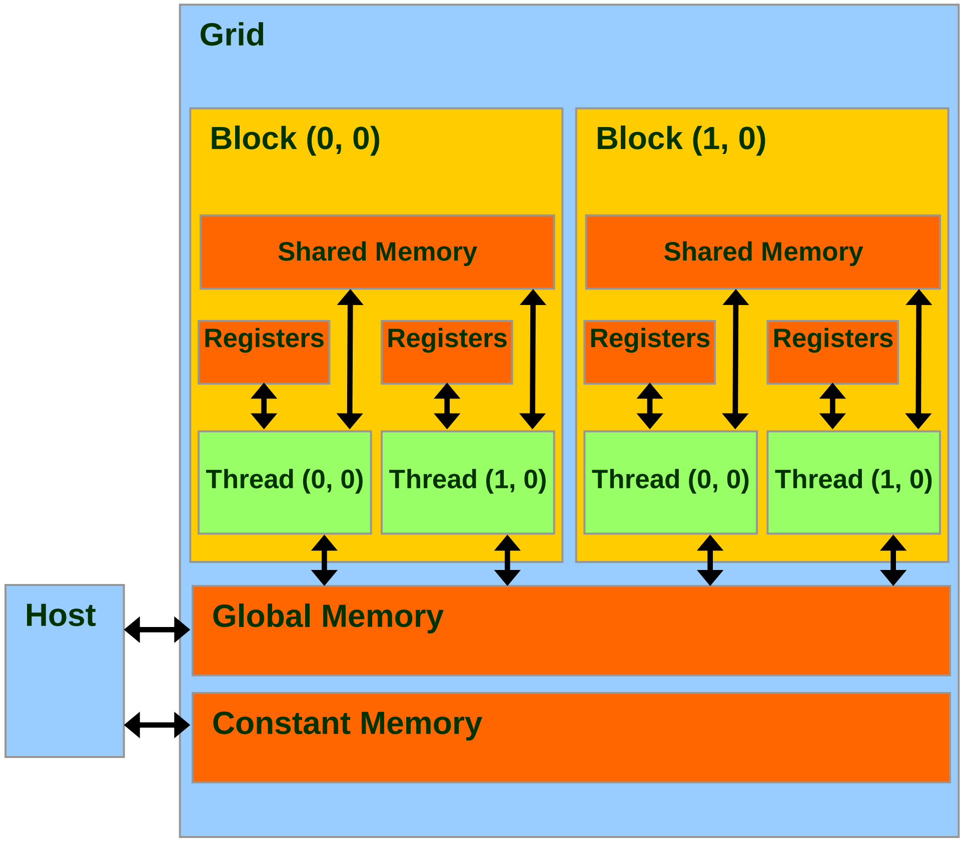
2. if the number of instances exceeds the number of SMs, then more than one will run at a time on each SM if there are enough registers and shared memory, and the others will wait in a queue and execute later

3. all threads within one instance can access local shared memory but can’t see what the other instances are doing (even if they are on the same SM)

4. there are no guarantees on the order in which the instances execute

# CUDA Memories

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



## 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

# Variable Types

| Variable declaration | Memory | Scope | Lifetime |
|---|---|---|---|
| `__device__ __local__ int LocalVar;` | local | thread | thread |
| `__device__ __shared__ int SharedVar;` | shared | block | block |
| `__device__ int GlobalVar;` | global | grid | application |
| `__device__ __constant__ int ConstantVar;` | constant | grid | application |

- the __device__ indicates this is a global variable in the GPU
  - the variable can be read and modified by any kernel
  - its lifetime is the lifetime of the whole application
  - can also declare arrays of fixed size
  - can read/write by host code using standard cudaMemcpy
- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`

- Constant variables
  - Very similar to global variables, except that they can't be modified by kernels
  - defined with global scope within the kernel file using the prefix __constant__
  - initialized by the host code using cudaMemcpyToSymbol, cudaMemcpyFromSymbol or cudaMemcpy in combination with cudaGetSymbolAddress
  - Only 64KB of constant memory, but big benefit is that each SM has a 8KB cache
- Pointers can only point to memory allocated or declared in global 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;
    ```
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

```
__global__ void lap(int I, int J,float *u1, float *u2) {
int i = threadIdx.x + blockIdx.x*blockDim.x;
int j = threadIdx.y + blockIdx.y*blockDim.y;
int id = i + j*I;
if (i==0 || i==I-1 || j==0 || j==J-1) {
   u2[id] = u1[id]; // Dirichlet b.c.'s }
else {
u2[id] = 0.25f * ( u1[id-1] + u1[id+1]
+ u1[id-I] + u1[id+I] );}
}
```

## Shared Memory

```
__shared__ int x_dim;
__shared__ float x[128];
```

- declares data to be shared between all of the threads in the thread block – any thread can set its value, or read it.
- Advantages of using shared memory
  - essential for operations requiring communication between threads
  - useful for data re-use
  - alternative to local arrays in device memory
  - reduces use of registers when a variable has same value for all threads

- If a thread block has more than one warp, it’s not pre-determined when each warp will execute its instructions – warp 1 could be many instructions ahead of warp 2, or well behind.

- Consequently, almost always need thread synchronization to ensure correct use of shared memory.

- Instruction
  - __syncthreads();

- inserts a “barrier”; no thread/warp is allowed to proceed beyond this point until the rest have reached it
- Total size of shared memory is specified by an optional third arguments when launching the kernel:
  - kernel<<<blocks,threads,shared_bytes>>>(...)

## Active Blocks per SM

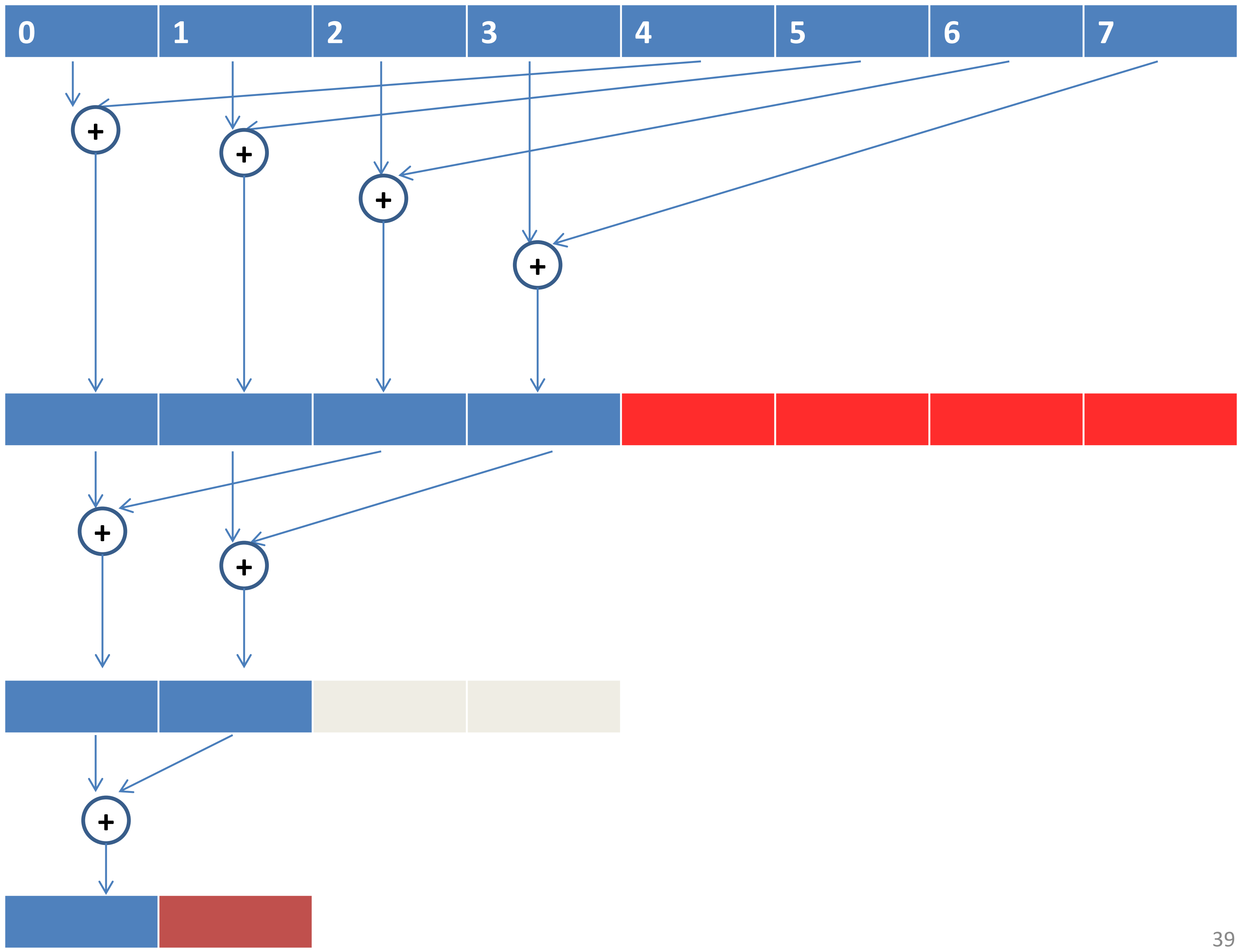
- Each block require certain resources:
  - threads
  - registers (registers per thread × number of threads)
  - shared memory (static + dynamic)
- Together these decide how many blocks can be run simultaneously on each SM – up to a maximum of 8 blocks
- General advice:
  - number of active threads depends on number of registers each needs
  - good to have at least 2-4 active blocks, each with at least 128 threads
  - smaller number of blocks when each needs lots of shared memory
  - larger number of blocks when they don't need shared memory
  - On Fermi card:
    - maybe 2 big blocks (512 threads) if each needs a lot of shared memory
    - maybe 6 smaller blocks (256 threads) if no shared memory needed
    - or 4 small blocks (128 threads) if each thread needs lots of registers

- Global memory resides in device memory (DRAM) - much slower access than shared memory
- So, a profitable way of performing computation on the device is to tile 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 by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing 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
- Constant memory also resides in device memory (DRAM) - much slower access than shared memory
  - But… cached!
  - Highly efficient access for read-only data
- Carefully divide data according to access patterns
  - R/Only → constant memory (very fast if in cache)
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very fast)
  - R/W inputs/results → global memory (very slow)

# Shared Memory and Synchronization for Dot Product

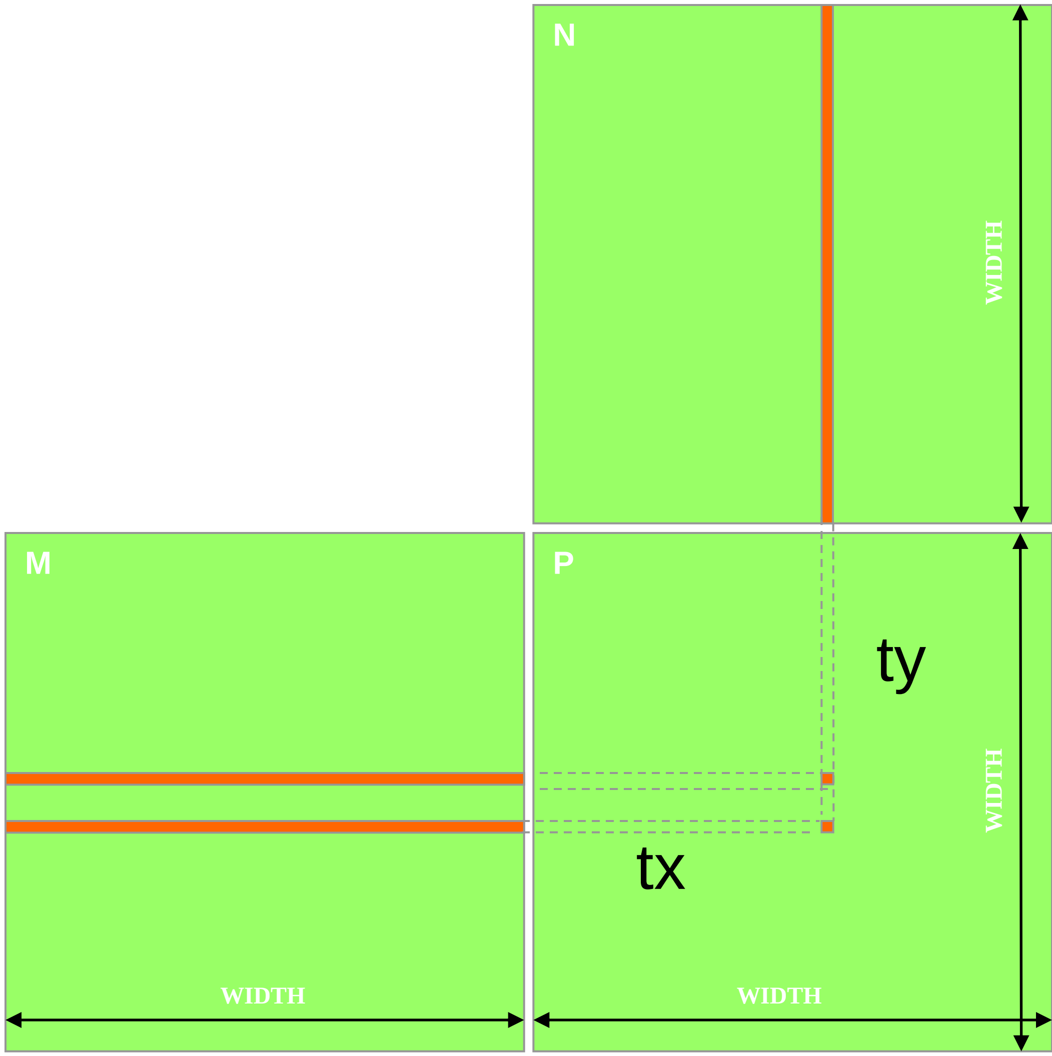
```
#define imin(a,b) ((a)<(b)?(a):(b))
const int N = 33*1024;
const int threadsPerBlock = 256;
const int blocksPerGrid = imin(32, (N+threadsPerBlock-1)/threadsPerBlock);
int main(){
   float *a, *b, c, *partial_c;
   float *dev_a, *dev_b, *dev_partial_c;
   a = (float*)malloc(N*sizeof(float));  b = (float*)malloc(N*sizeof(float));
   partial_c = (float*)malloc(blocksPerGrid*sizeof(float));
   cudaMalloc((void**)&dev_a,N*sizeof(float));
   cudaMalloc((void**)&dev_b,N*sizeof(float));
   cudaMalloc((void**)&dev_partial_c,blocksPerGrid*sizeof(float));
   // initialize a[] and b[] …
   cudaMemcpy(dev_a,a,N*sizeof(float),cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b,b,N*sizeof(float),cudaMemcpyHostToDevice);
   dot<<< blocksPerGrid, threadsPerBlock>>>(dev_a,dev_b,dev_partial_c);
   cudaMemcpy(partial_c,dev_partialc,blocksPerGrid*sizeof(float),cudaMemcpyDeviceToHost);
   c = 0;
   for(int i=0; i<blocksPerGrid;i++)    c += partial_c[i];
   // cuda memory free, etc.
}
```

```
__global__ void dot(float *a, float*b, float *c){
    __shared__ float cache[threadsPerBlock];
    //this buffer will be used to store each thread's running sum
    // the compiler will allocate a copy of shared variables for each block
    int tid = threadIdx.x + BlockIdx.x*blockDim.x;
    int cacheIndex = threadIdx.x;
    float temp = 0.0;
    while(tid < N){
        temp += a[tid]*b[tid];
        tid += blockDim.x*gridDim.x;
    }
    // set the cache values
    cache[cacheIndex]=temp;

    // we need to sum all the temporary values in the cache.
    // need to guarantee that all of these writes to the shared array
    // complete before anyone to read from this array.

    // synchronize threads in this block
    __syncthreads();    // This call guarantees that every thread in the block has
                        // completed instructions prior to __syncthreads() before the
                        // hardware will execute the next instruction on any thread.
// each thread will add two of values in cache[] and
// store the result back to cache[].
// We continue in this fashion for log_2(threadsPerBlock)
//steps till we have the sum of every entry in cache[].
// For reductions, threadsPerBlock must be a power of 2
    int i=blockDim.x/2;
    while(i!=0){
        if(cacheIndex <i)
            cache[cacheIndex] += cache[cacheIndex+i];
         __syncthreads();
         i/=2;
     }
     if(cacheIndex==0)
         c[blockIdx.x]=cache[0];
}
```


0
1
2
3
4
5
6
7
+
+
+
+
+
+
+

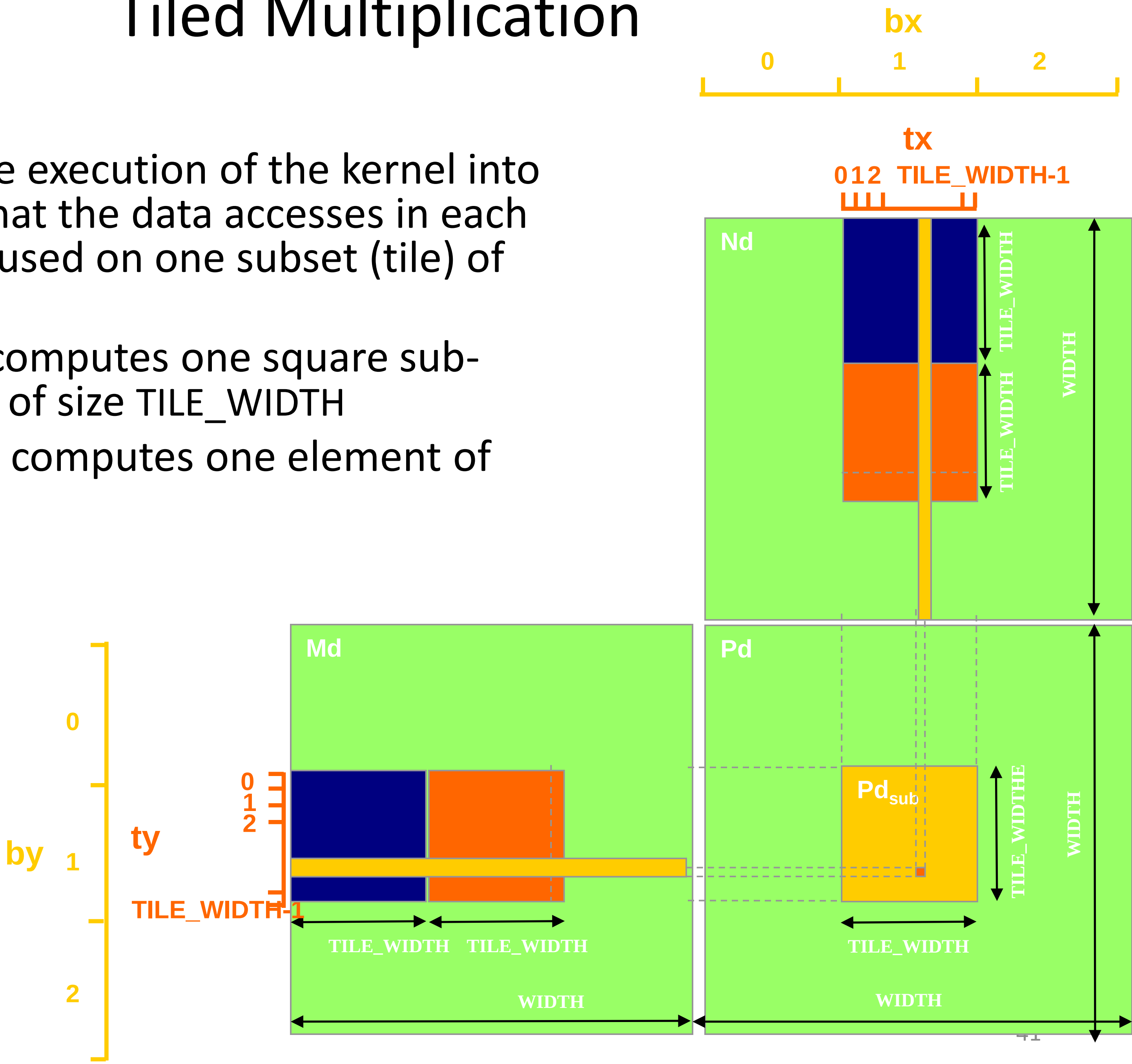
# Shared Memory to Reuse Global Memory Data

- Each input element is read by Width threads.
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth
  - **Tiled algorithms**



# Tiled Multiplication

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd
- Each block computes one square sub-matrix $Pd_{sub}$ of size TILE_WIDTH
- Each thread computes one element of $Pd_{sub}$

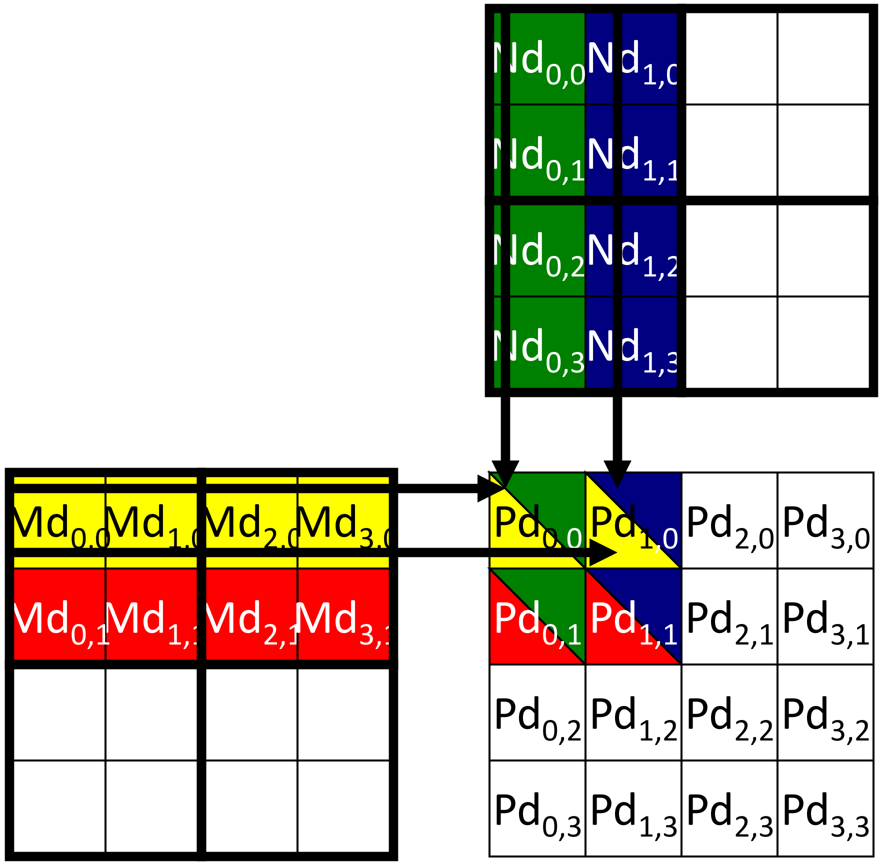


# Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P

Access order

| $P_{0,0}$ $\text{thread}_{0,0}$ | $P_{1,0}$ $\text{thread}_{1,0}$ | $P_{0,1}$ $\text{thread}_{0,1}$ | $P_{1,1}$ $\text{thread}_{1,1}$ |
|---|---|---|---|
| $M_{0,0} * N_{0,0}$ | $M_{0,0} * N_{1,0}$ | $M_{0,1} * N_{0,0}$ | $M_{0,1} * N_{1,0}$ |
| $M_{1,0} * N_{0,1}$ | $M_{1,0} * N_{1,1}$ | $M_{1,1} * N_{0,1}$ | $M_{1,1} * N_{1,1}$ |
| $M_{2,0} * N_{0,2}$ | $M_{2,0} * N_{1,2}$ | $M_{2,1} * N_{0,2}$ | $M_{2,1} * N_{1,2}$ |
| $M_{3,0} * N_{0,3}$ | $M_{3,0} * N_{1,3}$ | $M_{3,1} * N_{0,3}$ | $M_{3,1} * N_{1,3}$ |

# Breaking Md and Nd into Tiles



# Each Phase of a Thread Block Uses One Tile from Md and One from Nd

| | Phase 1 | | | Phase 2 | | |
|---|---|---|---|---|---|---|
| $T_{0,0}$ | $\mathbf{Md_{0,0}}$ ↓ $Mds_{0,0}$ | $\mathbf{Nd_{0,0}}$ ↓ $Nds_{0,0}$ | $PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$ | $\mathbf{Md_{2,0}}$ ↓ $Mds_{0,0}$ | $\mathbf{Nd_{0,2}}$ ↓ $Nds_{0,0}$ | $PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$ |
| $T_{1,0}$ | $\mathbf{Md_{1,0}}$ ↓ $Mds_{1,0}$ | $\mathbf{Nd_{1,0}}$ ↓ $Nds_{1,0}$ | $PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$ | $\mathbf{Md_{3,0}}$ ↓ $Mds_{1,0}$ | $\mathbf{Nd_{1,2}}$ ↓ $Nds_{1,0}$ | $PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$ |
| $T_{0,1}$ | $\mathbf{Md_{0,1}}$ ↓ $Mds_{0,1}$ | $\mathbf{Nd_{0,1}}$ ↓ $Nds_{0,1}$ | $PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$ | $\mathbf{Md_{2,1}}$ ↓ $Mds_{0,1}$ | $\mathbf{Nd_{0,3}}$ ↓ $Nds_{0,1}$ | $PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$ |
| $T_{1,1}$ | $\mathbf{Md_{1,1}}$ ↓ $Mds_{1,1}$ | $\mathbf{Nd_{1,1}}$ ↓ $Nds_{1,1}$ | $PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$ | $\mathbf{Md_{3,1}}$ ↓ $Mds_{1,1}$ | $\mathbf{Nd_{1,3}}$ ↓ $Nds_{1,1}$ | $PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$ |

time $\longrightarrow$

- Each thread block should have many threads
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
- There should be many thread blocks
  - A 1024*1024 Pd gives 64*64 = 4096 Thread Blocks
- Each thread block perform 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  - Memory bandwidth no longer a limiting factor

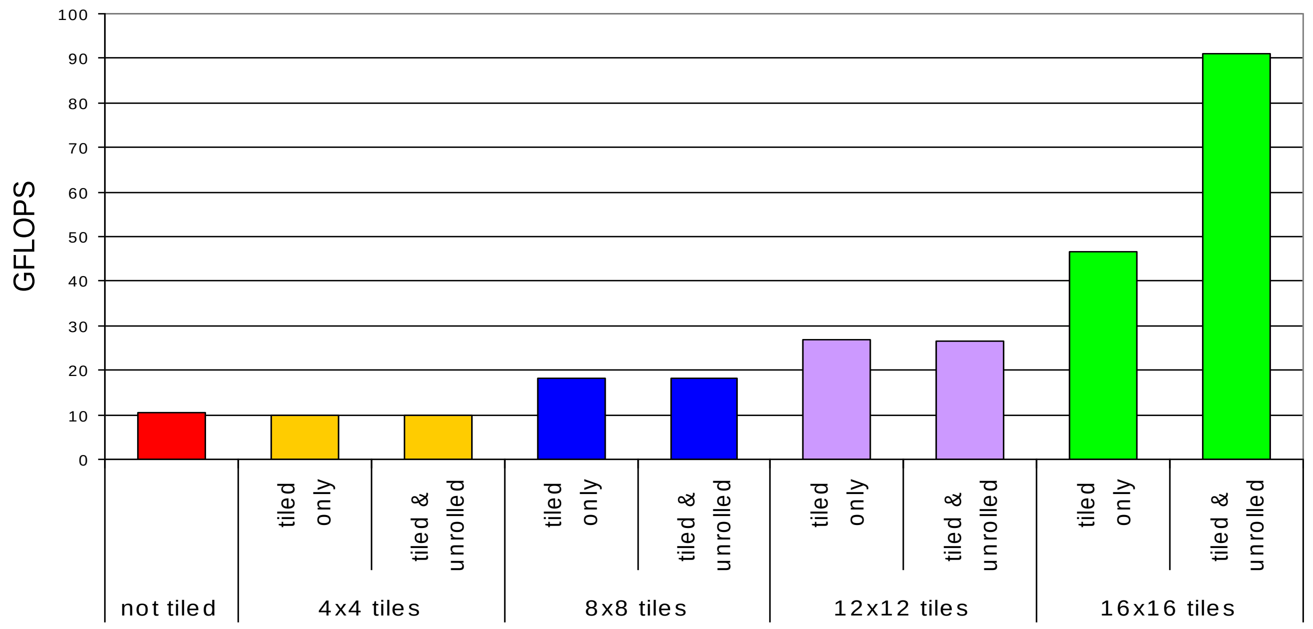
## Kernel Execution Configuration

```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width/TILE_WIDTH,
             Width/TILE_WIDTH);
```

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

   float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Coolaborative loading of Md and Nd tiles into shared memory
       Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
       Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
       __syncthreads();

       for (int k = 0; k < TILE_WIDTH; ++k)
          Pvalue += Mds[ty][k] * Nds[k][tx];
       Synchthreads();
   }
   Pd[Row*Width+Col] = Pvalue;
}
```

# Performance on G80

- Each SM in G80 has 16KB shared memory
  - SM size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B= 8KB shared memory usage per thread block, allowing only up to two thread blocks active at the same time
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 86.4B/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS



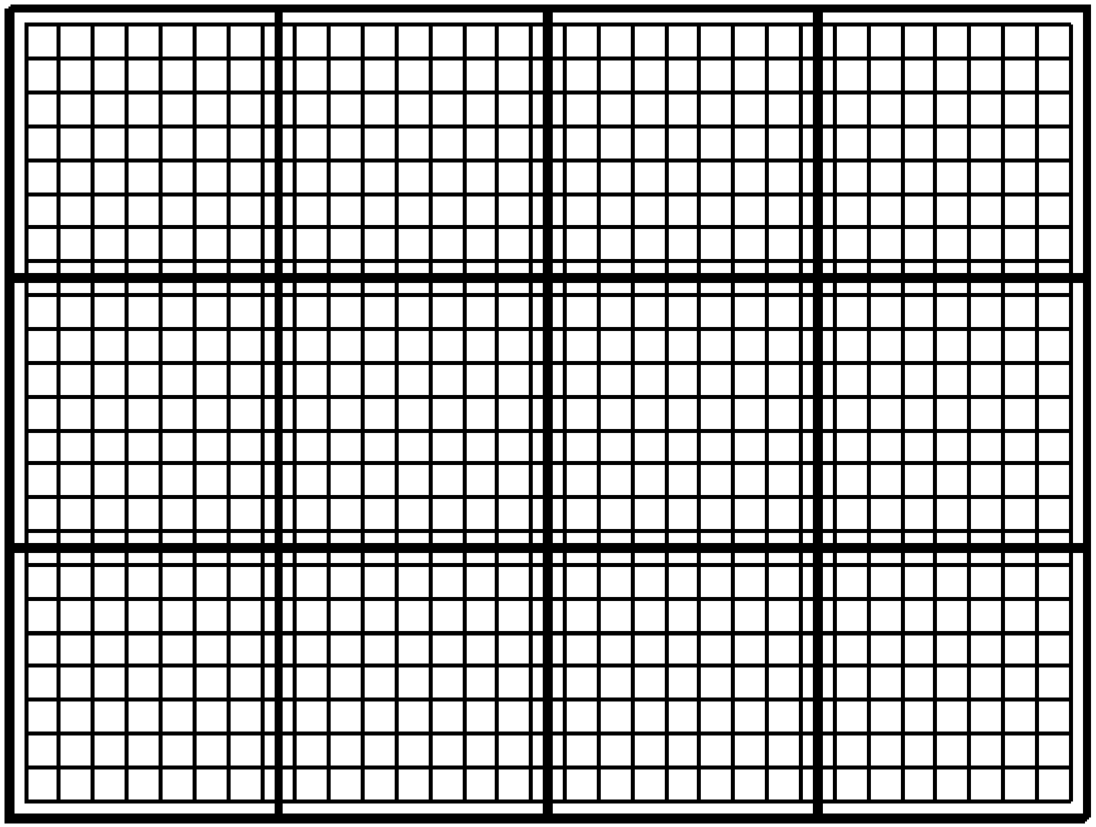
## 2D Laplace Solver

- Jacobi iteration to solve discrete Laplace equation on a uniform grid

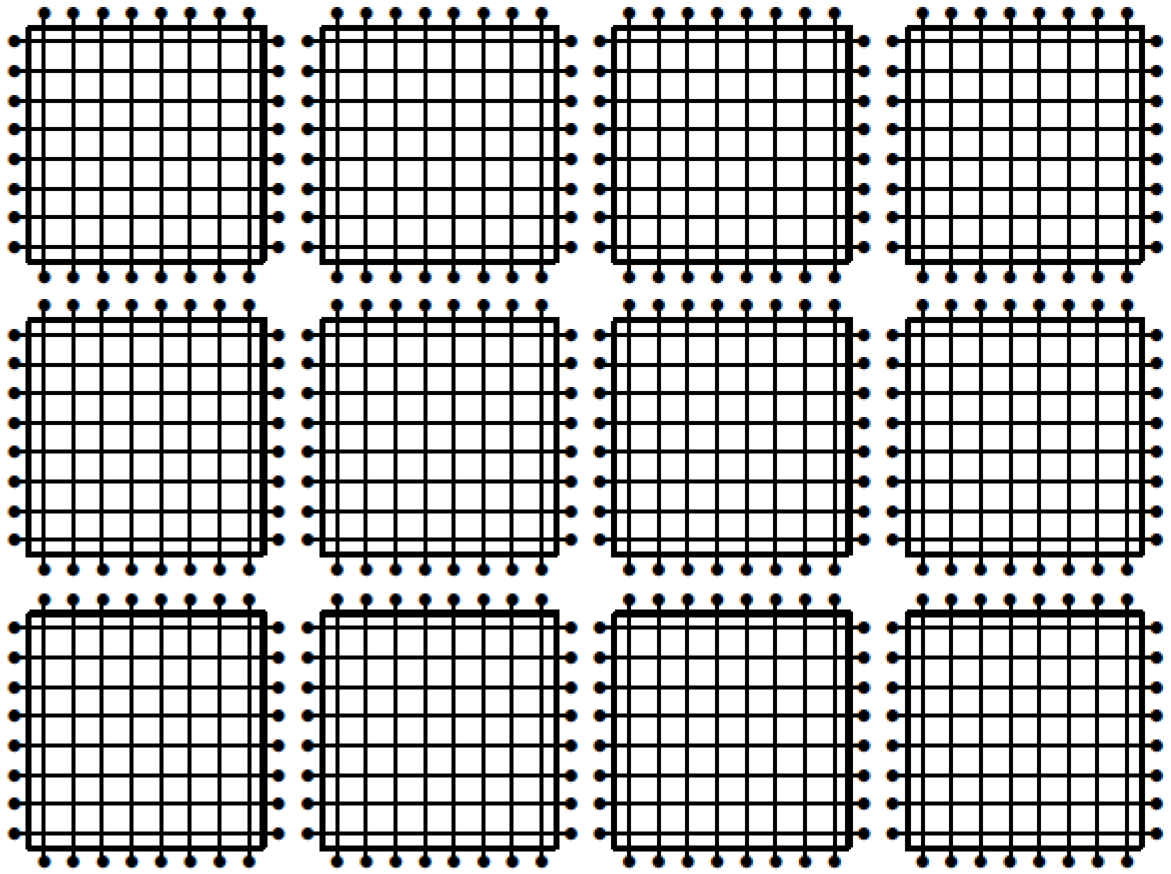
```
for (int j=0; j<J; j++) {
   for (int i=0; i<I; i++) {
      id = i + j*I; // 1D memory location
      if (i==0 || i==I-1 || j==0 || j==J-1)
          u2[id] = u1[id];
      else
      u2[id] = 0.25*( u1[id-1] + u1[id+1]
                    + u1[id-I] + u1[id+I] );
    }
}
```

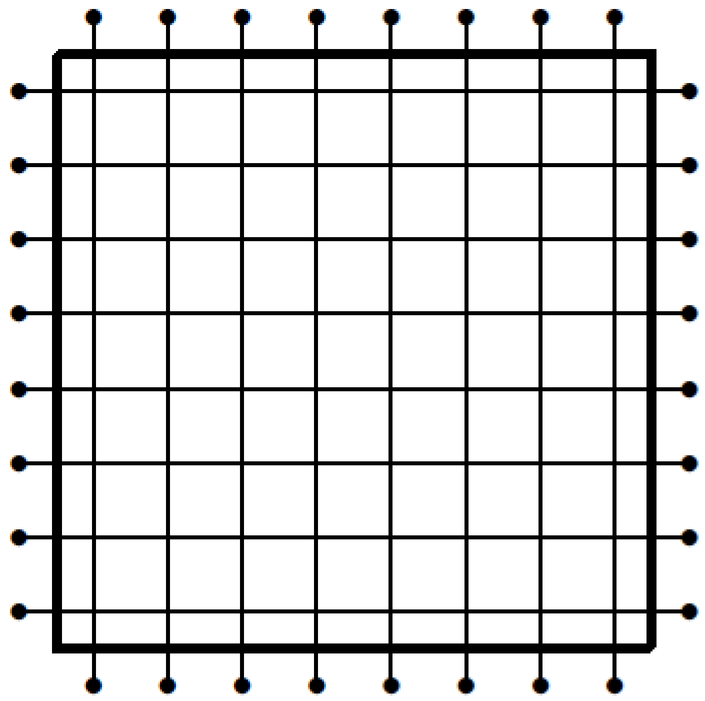
## 2D Laplace Solver Using CUDA

- each thread responsible for one grid point
- each block of threads responsible for a block of the grid
- conceptually very similar to data partitioning in MPI distributed-memory implementations, but much simpler



$\rightarrow$





- Each block of threads processes one of these grid blocks, reading in old values and computing new values.

```
__global__ void lap(int I, int J, float *u1, float *u2) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;
    int id = i + j*I;
    if (i==0 || i==I-1 || j==0 || j==J-1) {
        u2[id] = u1[id]; // Dirichlet b.c.'s
    }
    else {
        u2[id] = 0.25 * ( u1[id-1] + u1[id+1]
                        + u1[id-I] + u1[id+I] );
    }
}
```

Assumptions:

- I is a multiple of blockDim.x
- J is a multiple of blockDim.y

  grid breaks up perfectly into blocks

- I is a multiple of 32

Can remove these assumptions by

- testing if i, j are within grid
- padding the array in x to make it a multiple of 32, so each row starts at the beginning of a cache line – this uses a special routine cudaMallocPitch

## References

- http://developer.nvidia.com/nvidia-gpu-computing-documentation
- J. Sanders and E. Kandrot, CUDA by Example, An Introduction to General-Purpose GPU Programming