CUDA Thread Organization

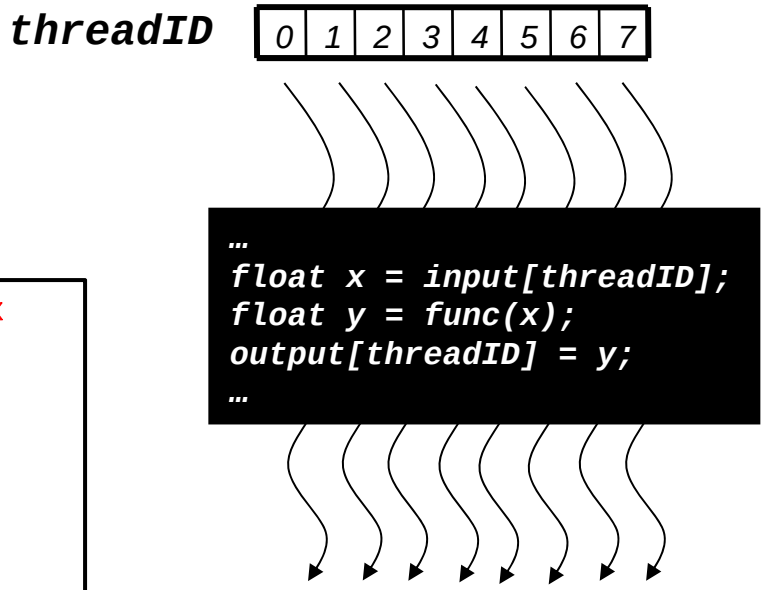
Outline

1. CUDA -- BLOCKs of THREADs
2. MMM (again)
3. Granularity considerations
4. API-centric view
5. Interacting BLOCKs and THREADs

Arrays of Parallel Threads

A CUDA kernel is executed by an ***array of threads***

- All threads run the same code (SIMT)
- Each thread has an ID that it uses to compute memory addresses and make control decisions




```
// Kernel definition -- specified with __global__ syntax

__global__ void VecAdd (float* A, float* B, float* C)
{
  int i = threadIdx.x;     // get unique (1D) thread ID
  C[i] = A[i] + B[i];      // do my bit of work
}



// Single block, but scales immediately to multiple blocks
int main()
{
   …
  // Kernel invocation
  VecAdd<<<1,N>>>(A,B,C); // # of CUDA threads is specified
                          // with the <<<…>>> syntax
```

Thread Blocks

Divide monolithic thread array into multiple blocks

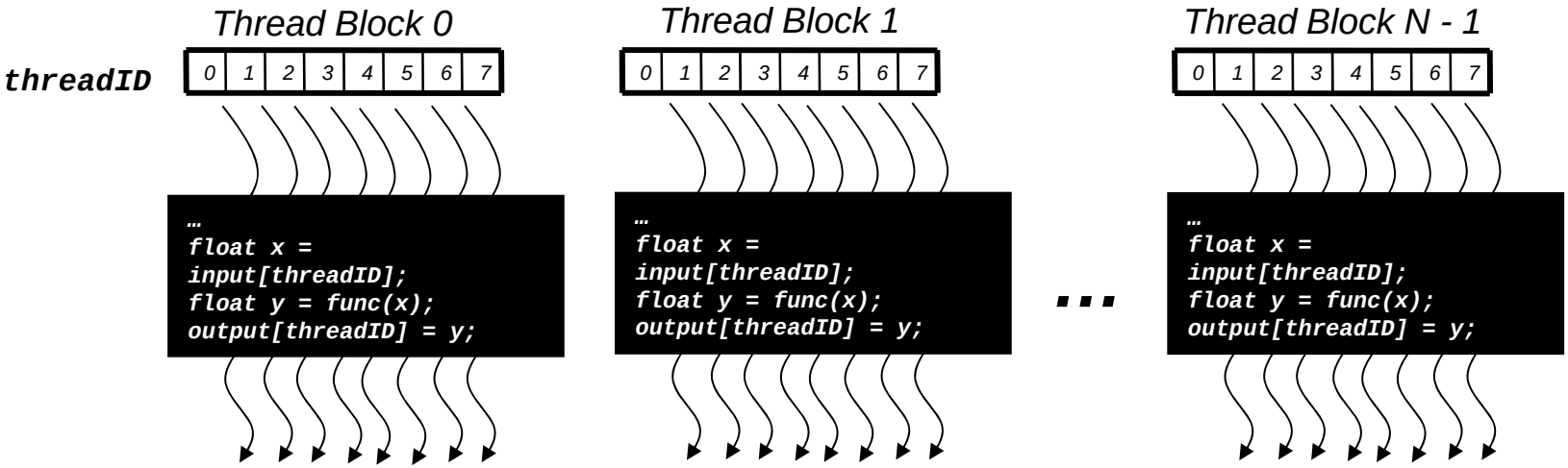
Block ⇔ Instantiation of a kernel

- **Block would be called a thread in a CPU!**

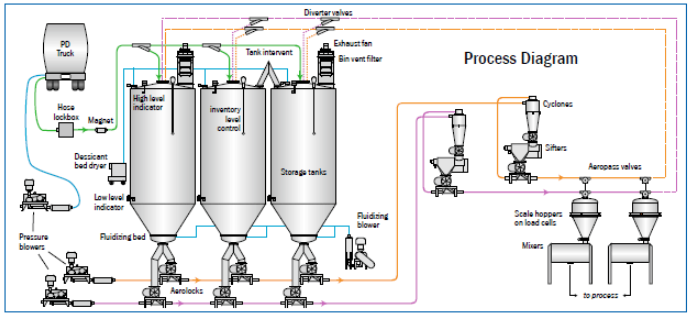
Blocks contain some number of threads executing in SIMD

- **Block instructions (executed by threads) are analogous to vector instructions**

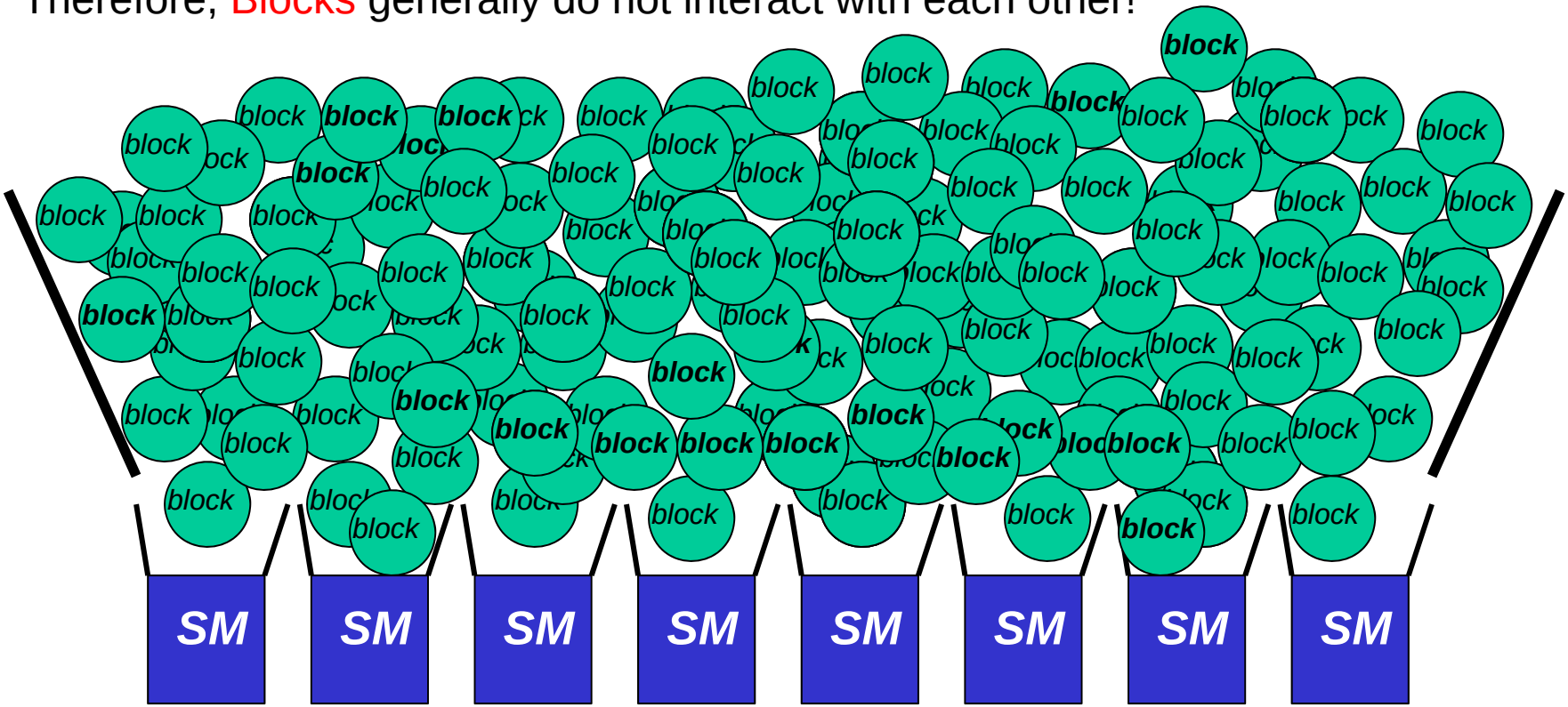
A Block is mapped to a single SM



Block Scheduling

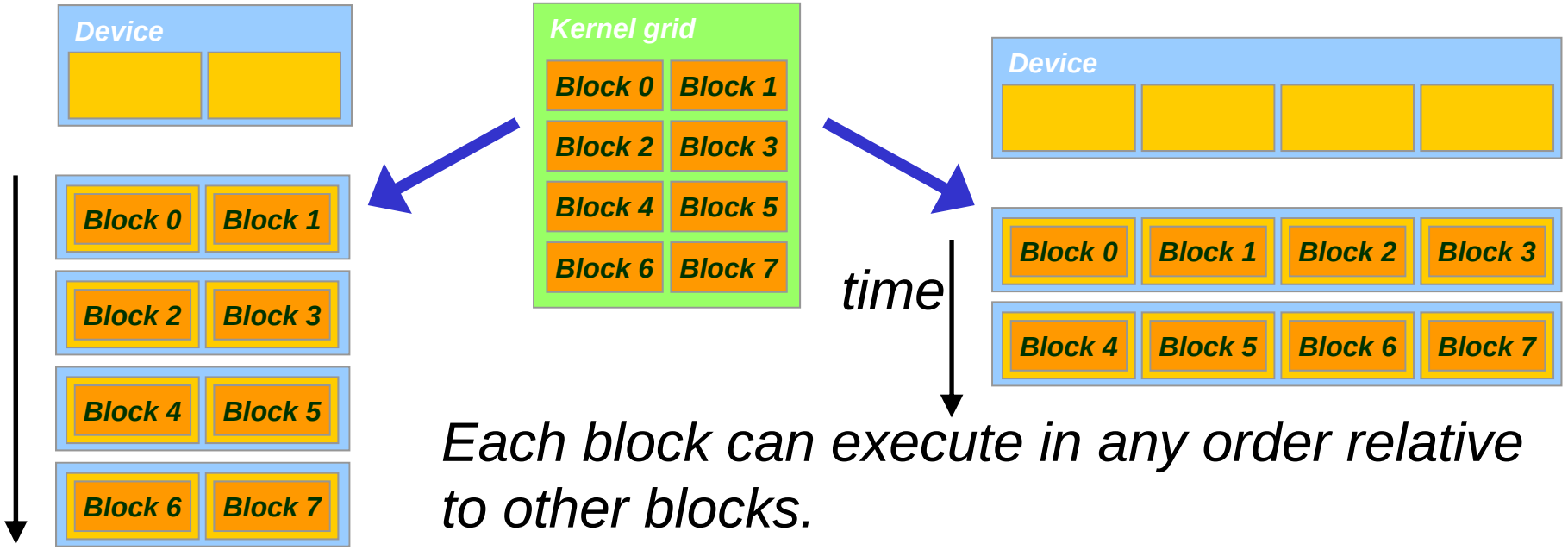


- Each **block** is an instantiation of a kernel and is analogous to a thread in Pthreads
- In CUDA programs, generally lots of blocks are generated
- Blocks run on exactly one SM
- Some number of Blocks can run at the same time on an SM (up to 8 on G200)
- There is no constraint on Block scheduling. Blocks can go in any order on any SM.
- Therefore, Blocks generally do not interact with each other!



Transparent Scalability

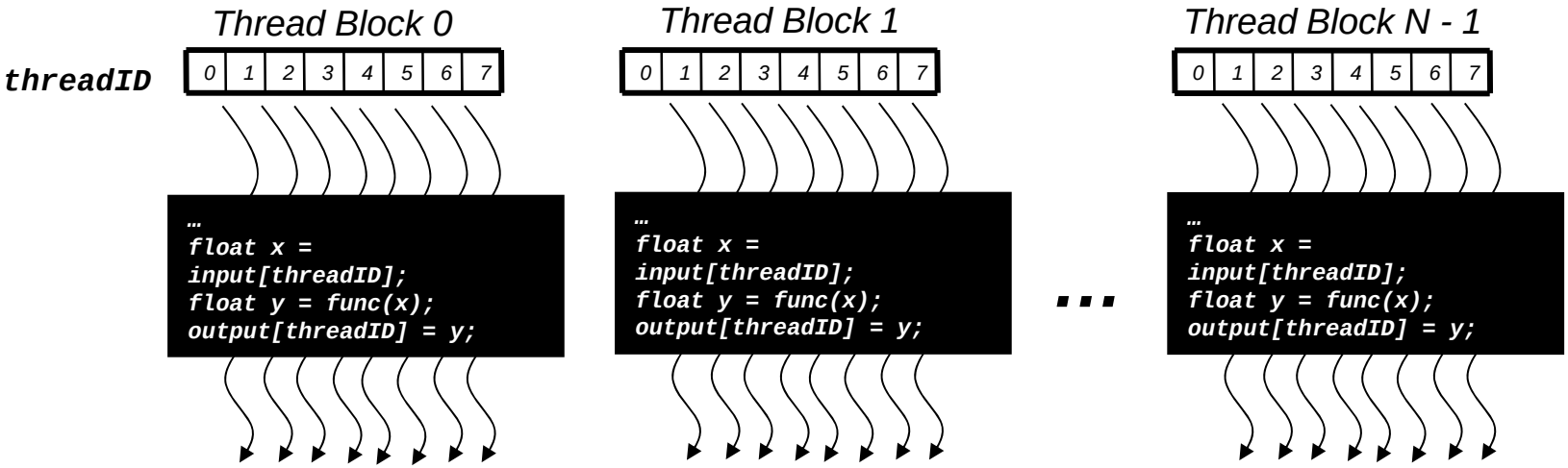
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors



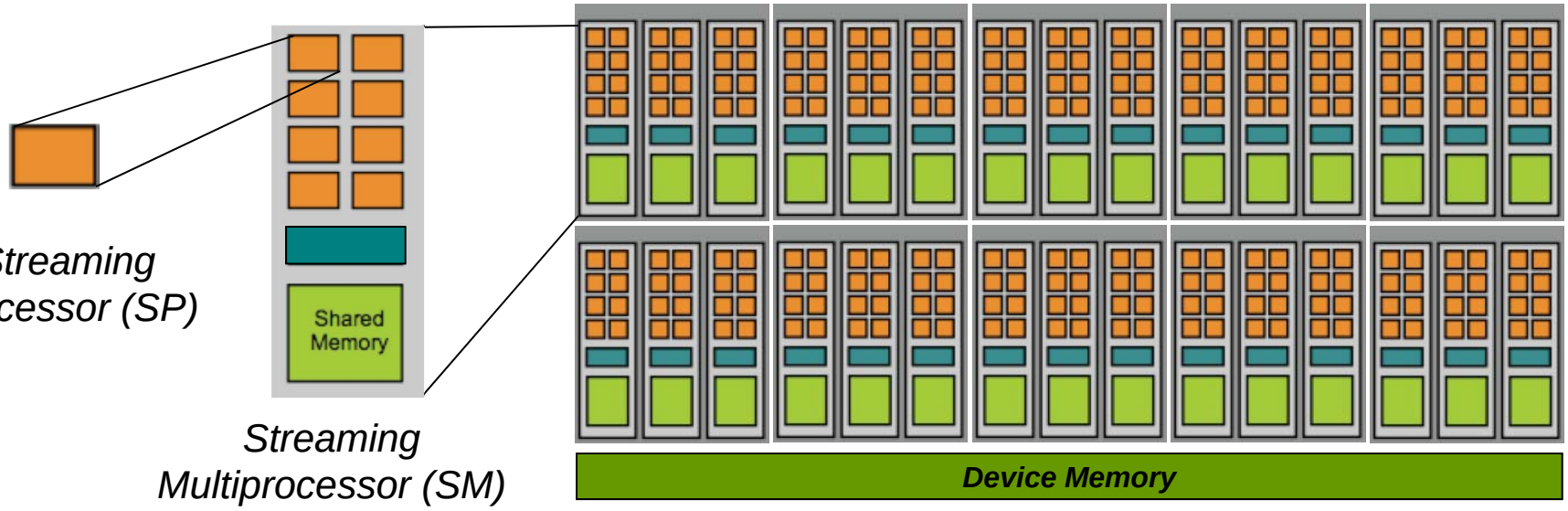
Each block can execute in any order relative to other blocks.

Thread Blocks: Scalable Cooperation

- **Divide monolithic thread array into multiple blocks**
  - *Why? This is purely hardware driven: it turns out that blocks are a good granularity of work (for DECOMPOSITION and ASSIGNMENT). A block is big enough to do real work on a **single** SM, but small enough to be worth creating in large numbers. The latter is important for scalability -- a large enough number of independent blocks can be mapped well onto 8 or 30 or 50 SMs!*
  - Threads ***within*** a block cooperate via **shared memory, atomic operations,** and **barrier synchronization**
  - Threads in ***different*** blocks **cannot** cooperate. ***That's right,*** *THREADS IN DIFFERENT BLOCKS CANNOT COOPERATE!*



Recall → NVIDIA GPU Architecture



8 SPs per SM

Tesla Architecture

- *30 Multiprocessors • 240 Processor cores*
- *4 GB Device memory • 1.3 GHz Clock*

RULES FOR THREAD ORGANIZATION

- THE GAME CAN BE PLAYED WITH A STANDARD EARTH DECK OF CARDS, DESPITE THE SLIGHTLY DIFFERING DECK ON BETA ANTARES IV.
- EACH PLAYER GETS SIX CARDS, EXCEPT FOR THE PLAYER ON THE DEALER'S RIGHT, WHO GETS SEVEN.
- THE SECOND CARD IS TURNED UP, EXCEPT ON TUESDAYS.
- TWO JACKS ARE A "HALF-FIZZBIN".
- IF YOU HAVE A HALF-FIZZBIN:
  - A THIRD JACK IS A "**SHRALK**" AND RESULTS IN DISQUALIFICATION;
  - ONE WANTS A KING AND A DEUCE, EXCEPT AT NIGHT, WHEN ONE WANTS A QUEEN AND A FOUR;
  - IF A KING HAD BEEN DEALT, THE PLAYER WOULD GET ANOTHER CARD, EXCEPT WHEN IT IS DARK, IN WHICH CASE HE'D HAVE TO GIVE IT BACK.
- THE TOP HAND IS A "ROYAL FIZZBIN", BUT THE ODDS AGAINST GETTING ONE ARE SAID TO BE "ASTRONOMICAL".

but seriously …

Rules for Thread Organization

THREADs are organized into BLOCKs:

- **Min(|THREADs|) / BLOCK = 1**
- **Max(|THREADs|) / BLOCK = 512**

BLOCKs always run on a single SM

Multiple BLOCKs can run at the "same time" on a single SM:

- **Max(|BLOCKs|) / 1 SM = 8**
- **Max(|BLOCKs|) / 30 SMs = 240**

But the number of threads that can run on a SM is less than the product of the maxima:

- **Max(|THREADs|) / 1 SM = 1,024** *>> not 8 x 512 = 4,096*
- **Max(|THREADs|) / 30 SMs = 30,720** *>> not 30 x 8 x 512 = 122,880*

So 1024 THREADs (on a single SM) can be organized into:

- **2 BLOCKs of 512** *>> limit on # of threads/BLOCK*
- **4 BLOCKs of 256**
- **8 BLOCKs of 128** *>> limit on # of BLOCKs*

Rules for Thread Organization, cont.

For multiple BLOCKs of THREADs, all BLOCKs must have the same size and shape.

- *What's "shape"? -- see on next slides*

Why choose one organization over another? What should be …

- **… the Total Number of THREADs per problem?**
  - **More THREADs means less work per THREAD**
  - *Choosing this is another issue -- see below*
- **… the Total Number of BLOCKs?**
  - **More BLOCKs means less work per BLOCK**
- **… the Number of THREADs per BLOCK?**
  - *follows from other two*

Total number of THREADs per SM depends on resources needed by each THREAD

- **must have sufficient available or else the THREAD launch will fail.**
  - *note: this happens without error message, so be careful!*

How many BLOCKs should those THREADs be organized into?

BLOCK *Shape*

- **BLOCKs of THREADs can vary in SHAPE**
  - **but not within a kernel launch**
- **BLOCKs can have up to 3 dimensions**
  - **Max(|THREADs|) / X Dimension = 512**
  - **Max(|THREADs|) / Y Dimension = 512**
  - **Max(|THREADs|) / Z Dimension = 64**

Recall → Max(|THREADs|) / BLOCK = 512

<u>Syntax:</u> Declare BLOCK shape as follows:

```
 dim3 dimBlock(16,8,2);   // dim3 declares a dimension specifier

// invoke kernel MatAdd with 1 BLOCK where that BLOCK has the shape
// specified by dimBLOCK
 MatAdd<<<1, dimBlock>>>(A, B, C);
```

More Rules for Thread Organization

- **BLOCKs are organized into a GRID**
- **A GRID of BLOCKs can vary in SHAPE**
- **A GRID can have up to 2 dimensions**
  - **Max(|BLOCKs|) / X Dimension = 65535**
  - **Max(|BLOCKs|) / Y Dimension = 65535**
  - **Max(|BLOCKs|) / Z Dimension = 1 (implied)**

So we can be running a GRID with 4,294,836,225 BLOCKs *(!!!)*

Q: But I thought we could only run 30720 THREADs per 30 SMs?

A: Correct -- these BLOCKs are run sequentially. This is why THREADs in different BLOCKs can't interact.

A: But there are also (obvious) system limitations that prevent programs with more than a small fraction of that number of BLOCKs from working. Example: size of system memory.

Specifying GRIDs, etc.

Syntax: **Declare GRID shape as follows:**

```
 dim3 dimGrid(4,4);        // dim3 declares a dimension specifier,
                           //   same as for THREADs in a BLOCK
 dim3 dimBlock(16,16,1);   // This time have 256 THREADs per BLOCK

// invoke kernel MatAdd with number of THREADs and BLOCK shape
//  and GRID shape specified by dimGrid and dimBLOCK
 MatAdd<<<dimGrid, dimBlock>>>(A, B, C);

// This declaration might make sense for processing arrays
//  that are 64 x 64
```

More on THREADs, BLOCKs, & GRIDs

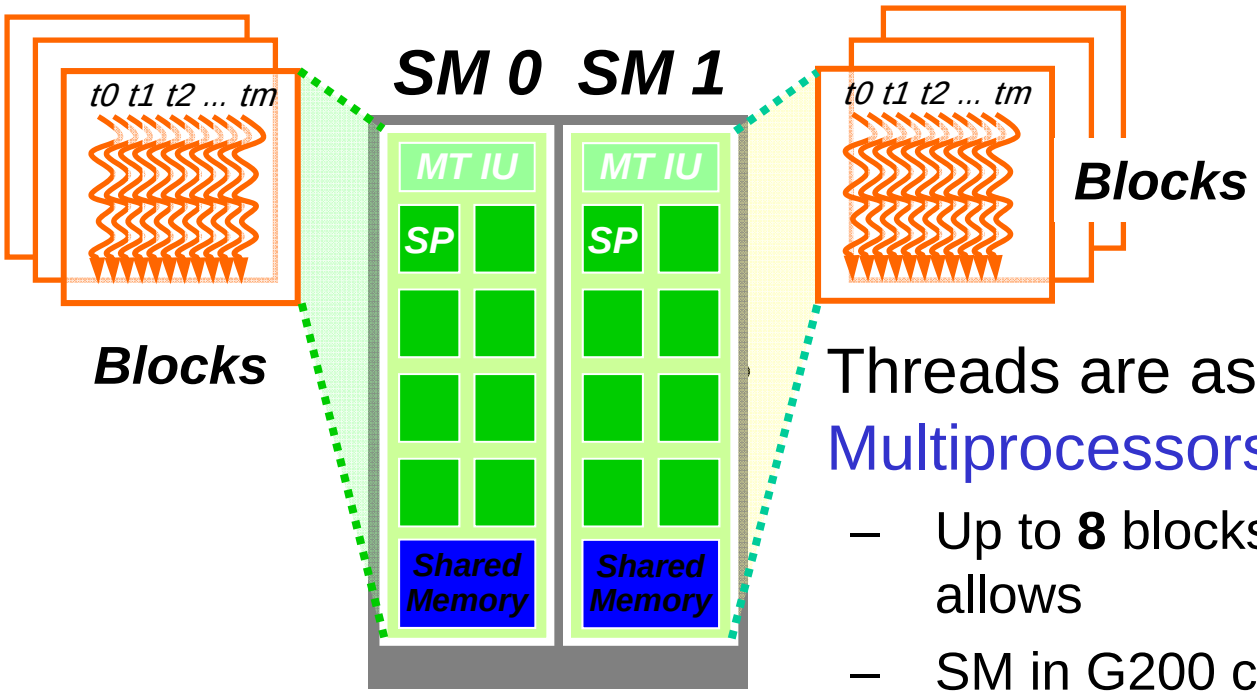
Syntax: **Find which THREAD or BLOCK we are as follows. These are all BUILT-IN VARIABLEs:**

```
// Which THREAD in the X dimension (or Y or Z) of the BLOCK?
threadIdx.x, threadIdx.y, threadIdx.z   // return the value

// Which BLOCK in the X dimension (or Y) of the GRID?
blockIdx.x, blockIdx.y   // return the value
```

Syntax: **Find the number of THREADs in a BLOCK dimension:**

```
// How many THREADs in the X dimension (or Y or Z) of the BLOCK?
blockDim.x, blockDim.y, BlockDim.z   // return the value

// Which BLOCK in the X dimension (or Y) of the GRID?
gridDim.x, gridDim.y   // return the value
```

G200 Example: Executing Thread Blocks



Threads are assigned to Streaming Multiprocessors in block granularity

- Up to **8** blocks to each SM as resource allows
- SM in G200 can take up to **1024** threads
  - Could be 256 (threads/block) * 4 blocks
  - Or 128 (threads/block) * 8 blocks, etc.

- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

Example: THREADs, BLOCKs, GRIDs

```
__global__ void MatAdd(float A[64][64], float B[64][64],
 float C[64][64])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

int main()
{
 …
 // Kernel invocation
 dim3 dimGrid(4,4);          // 4x4 GRID of 16 BLOCKs
 dim3 dimBLOCK(16,16,1);     // 16x16 BLOCK of 256 THREADs per BLOCK
 MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

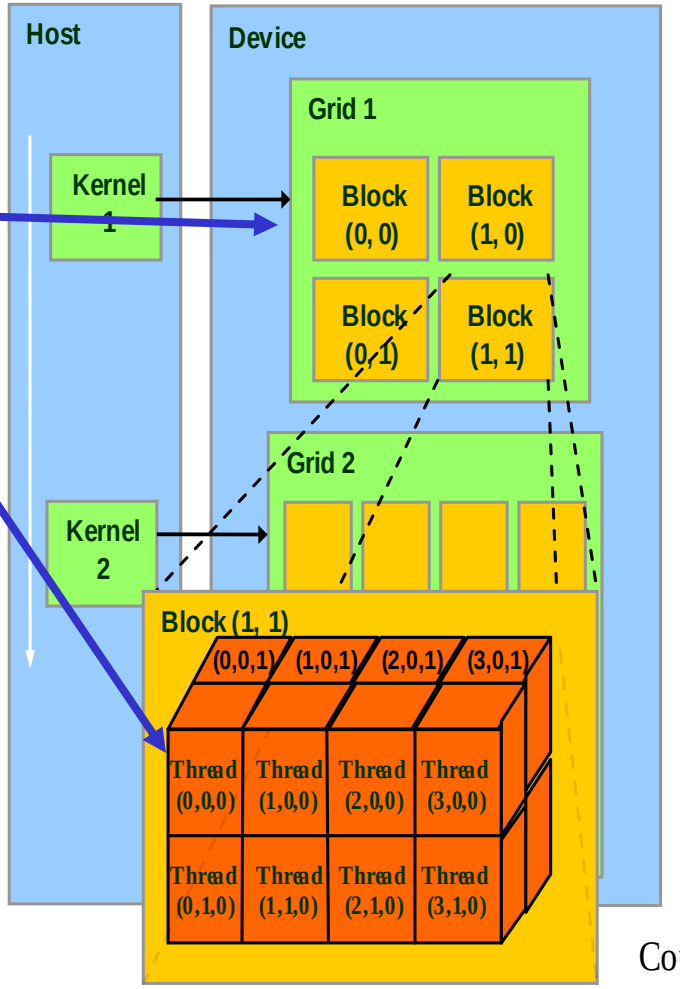
```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);   // 5000 thread blocks
dim3   DimBlock(4, 8, 8);  // 256 threads per block
size_t SharedMemBytes = 64;// 64 bytes shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit CUDA synch needed for blocking

Block IDs and Thread IDs, review

- Group **BLOCKs** into **GRIDs**
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Why? Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …

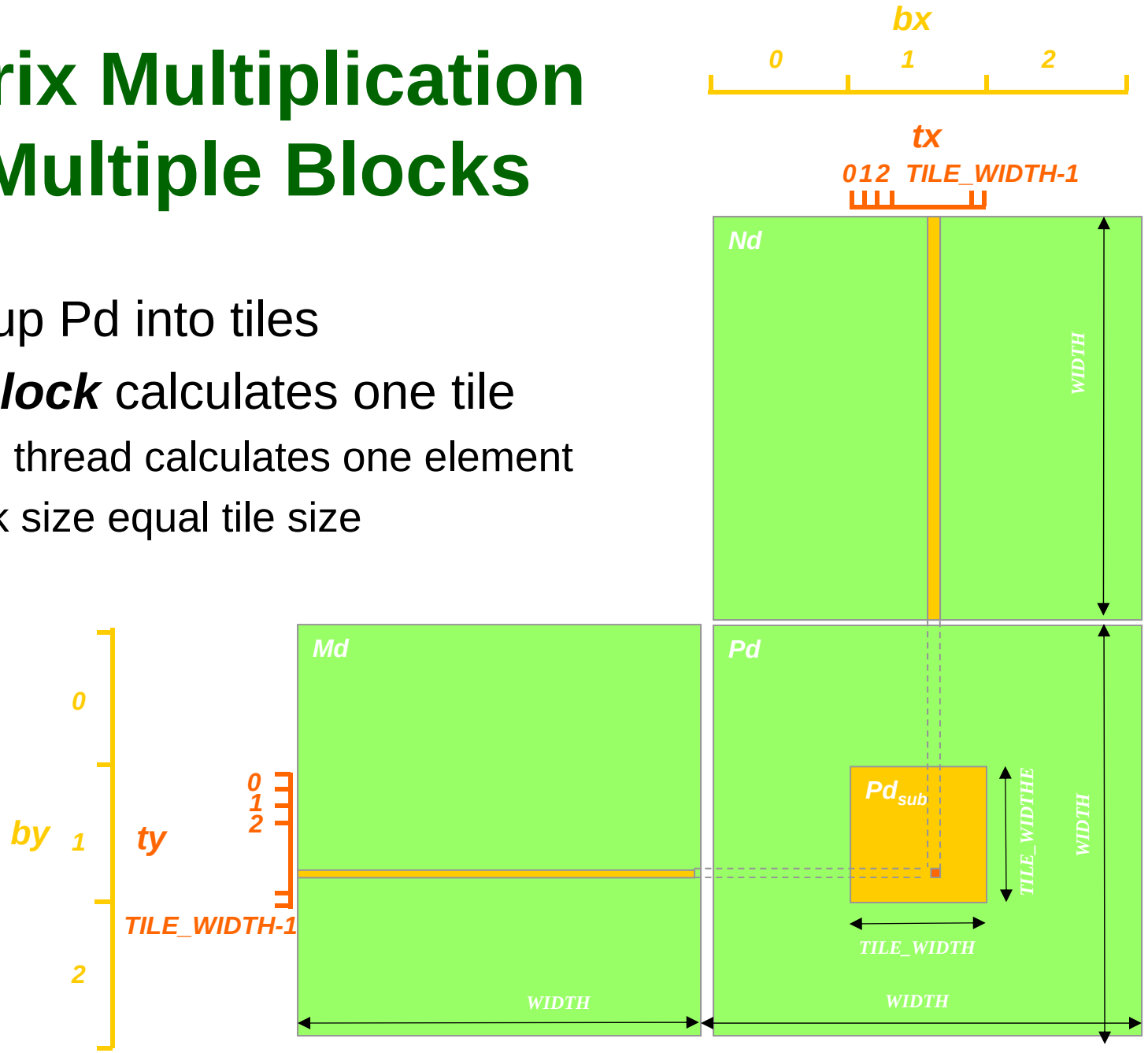


Courtesy: NDVIA

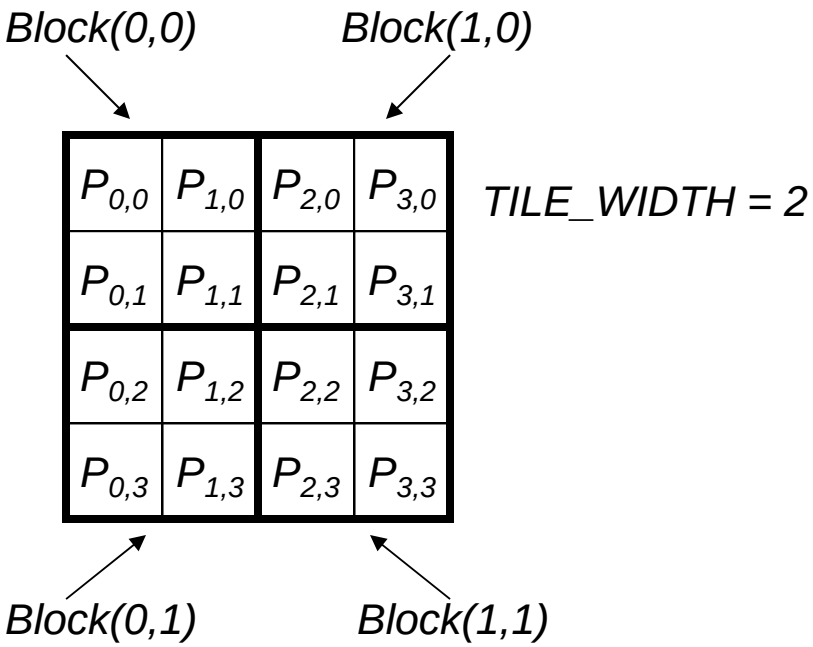
2. MMM Example with multiple blocks

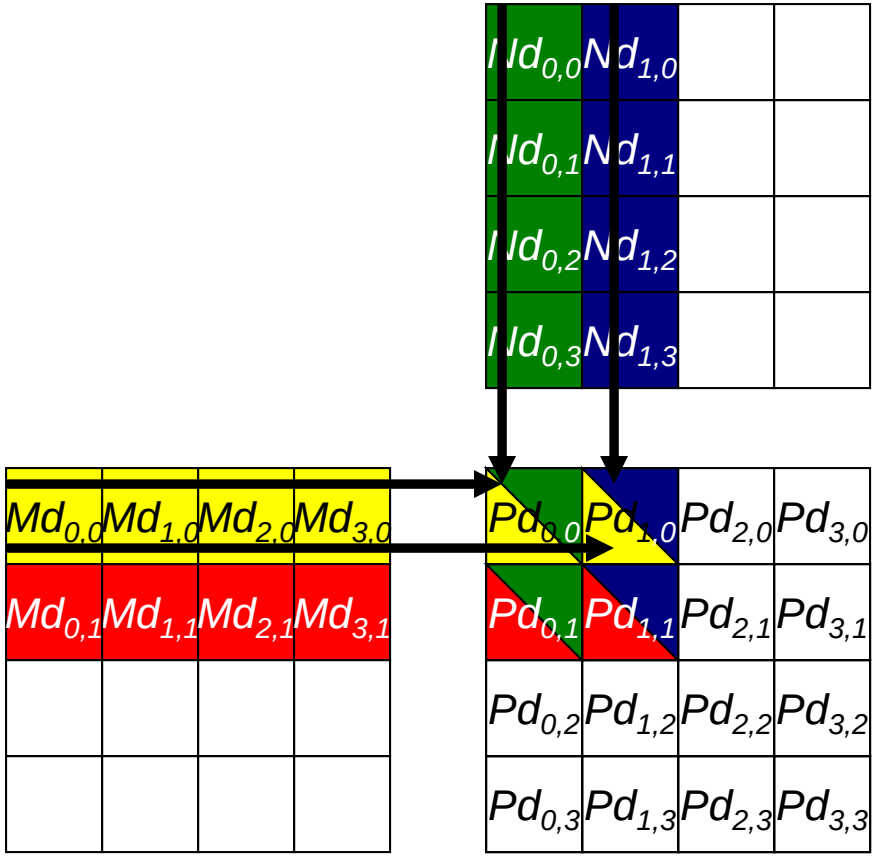
2: Matrix Multiplication Using Multiple Blocks

- Break-up Pd into tiles
- Each ***block*** calculates one tile
  - Each thread calculates one element
  - Block size equal tile size



A Small Example: Multiplication





Revised Matrix Multiplication Kernel using Multiple Blocks

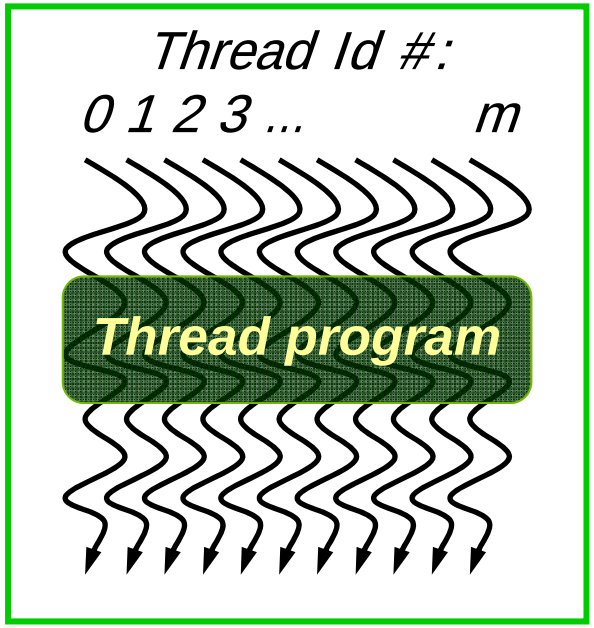
```
// Single thread still does a single dot product, but within a submatrix
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // Calculate the row index of the Pd element and M
  // blockIdx.x gets ID within a block
   int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
  // 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;
}
```

3. Granularity Considerations

3: CUDA Thread Block -- Review

- All threads in a block execute the same kernel program (SIMT)
- Programmer declares block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- All blocks in a grid (kernel invocation) have the same dimensions (shape)
- 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 and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocs!

CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

G200 Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1024 threads, there are 16 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM.
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1024 threads, it can take up to 4 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Although this is allowed on the SM, too big for a BLOCK.

```
// MMM generalized so that threads potentially compute a number of output pixels rather
//   than one as is assumed in the previous versions.
// This kernel assumes threads arranged as 2D blocks and blocks arranged as a 2D grid
__global__ void matrixMul(float* C, float* A, float* B, int row_a, int col_a, int col_b)
{
    int i, j, k, m;

    // Block index
    const int bId_x = blockIdx.x;
    const int bId_y = blockIdx.y;
    // Local thread index
    const int local_tid_x = threadIdx.x;
    const int local_tid_y = threadIdx.y;

    // Number of rows and columns of the result matrix to be evaluated by each block
    const int rows_per_block = (row_a / gridDim.x);
    const int cols_per_block = (col_b / gridDim.y);

    const int rows_per_thread = rows_per_block / blockDim.x;
    const int cols_per_thread = (cols_per_block / blockDim.y);

    // Row and column indices of the result matrix that the current block has to compute
    const int blockStartId_row = bId_x * rows_per_block;
    const int blockEndId_row = (bId_x+1) * rows_per_block - 1;
    const int blockStartId_col = bId_y * cols_per_block;
    const int blockEndId_col = (bId_y+1) * cols_per_block - 1;

    const int threadStartId_row = blockStartId_row + local_tid_x * rows_per_thread;
    const int threadEndId_row = blockStartId_row + (local_tid_x+1)* rows_per_thread - 1;
    const int threadStartId_col = blockStartId_col + local_tid_y * cols_per_thread;
    const int threadEndId_col = blockStartId_col + (local_tid_y+1)* cols_per_thread - 1;

    int resultId;
    float sum = 0;

    for(i = threadStartId_row; i <= threadEndId_row; i++) {
        for(j = threadStartId_col; j <= threadEndId_col; j++) {
            sum = 0;
            resultId = i*col_b + j;
            for(k = 0; k < col_a; k++) {
                sum = sum + A[i*col_a + k] * B[k*col_b + j];
            }
            C[resultId] = sum;
        }
    }
}
```

```
// Launching the kernel
dim3 my_block(block_dim_x, block_dim_y);
dim3 my_grid(grid_dim_x, grid_dim_y);
matrixMul<<<my_grid, my_block>>>(d_c, d_a, d_b, 8, 4, 12);
```

MMM Examples

Can vary
-- number of BLOCKs
-- number of THREADs per BLOCK (1-512, bounded by local SM memory)
-- amount of WORK per THREAD (bounded by local SM memory)

Assume 30 SMs
Max 1024 THREADs per SM (at a time)
Max 8 BLOCKs per SM (at a time)

Try a matrix size that fits in global memory: 7680x7680 *(x8x3)* (~1.42GB total)
-- BLOCK size = 16x16 = 256
-- GRID size = 480x480 = 230400
-- 7680 BLOCKs per SM (by coincidence)

Real questions

- Is there any advantage to performing more work per THREAD?
- What's the best way to reuse data? From previous work, try to make inner ii,jj loops as big as possible.

4. Some Additional API Features (now that we know about blocks)

4: Application Programming Interface

- The API is an extension to the C programming language
- It consists of:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
    - A host component to control and access one or more devices from the host
    - A device component providing device-specific functions

Language Extensions: 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

Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

- Some mathematical functions (e.g. `sin(x)`) have a less accurate, but faster device-only version (e.g. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`

Host Runtime Component

- Provides functions to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Error handling

- Initializes the first time a runtime function is called

- A host thread can invoke device code on only one device
  - Multiple host threads required to run on multiple devices

Device Runtime Component: Synchronization Function

- **`void __syncthreads();`**
- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing ***shared or global memory***
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

5. Example:
Simple THREAD/BLOCK interaction

5: Example

The purpose of this example is to look at simple THREAD interaction under various scenarios. Assume →

- Some number of SMs
- Global memory only
- Possibly multiple BLOCKs

Averaging filter

- Input = 1D array with N elements
- Output = 1D array with N elements
- Operation – Each output array element is the average of its corresponding element in the input array and its neighbors.

Example Parameters

Averaging filter

- Input = 1D array with N elements
- Output = 1D array with N elements
- Operation – Each output array element is the average of its corresponding element in the input array and its neighbors.

0. Read and Write Different/Same arrays.

N → B[i] = (A[i-1]+A[i]+A[i+1])/3

Y → A[i] = (A[i-1]+A[i]+A[i+1])/3

1. Many iterations or One iteration?

N → One iteration

Y → Many iterations

2. Many Blocks or One Block?

N → One BLOCK

Y → Many BLOCKs

8 Cases

CASES 0-3 -- N(N/Y)(N/Y)

2. Single Block

1. One OR Many iterations

0. Same OR Different arrays

The first four cases are all Single Block.

The Single Block simplifies synchronization, but since it runs on a single SM, there is a restricted amount of local memory to hold intermediate values and so both a limited number of threads and problem size.

CASE 0 – NNN:
R/W different arrays
1 iteration
1 BLOCK

Easy: No sync needed.

```
__global__ void Average(float* A, float* B, int N)
{
  int i = threadIdx.x;
  if (i > 0 && i < N-1)
    B[i] = (A[i-1] + A[i] + A[i+1])/3;
}

int main()
{
 …
 // Kernel invocation
  dim3 dimBlock(256,1,1);
  Average<<<1, dimBlock>>>(A, B, 256);
}
```

CASE 1 – NNY:
R/W same array
1 iteration
1 BLOCK

```
__global__ void Average(float* A, int N)
{
  int i = threadIdx.x;
  if (i > 0 && i < N-1) {
    float X = (A[i-1] + A[i] + A[i+1])/3;
    __syncthreads();
    A[i] = X;
  }
}

int main()
{
 …
 // Kernel invocation
  dim3 dimBLOCK(256,1,1);
  Average<<<1, dimBLOCK>>>(A, 256);
}
```

Must SYNC between reads and writes. Otherwise there is a race condition: some threads may read data that has already been written.

CASE 2 – NYN:
R/W different arrays
Multiple iterations
1 BLOCK

```
__global__ void Average(float* A, float* B, int N)
{
  int i = threadIdx.x;
  for (int j = 0; j < 50; j++) {
    if (i > 0 && i < N-1) {
      B[i] = (A[i-1] + A[i] + A[i+1])/3;
      __syncthreads();
      A[i] = (B[i-1] + B[i] + B[i+1])/3;
      __syncthreads();
    }
  }
}

int main()
{
 …
 // Kernel invocation
  dim3 dimBlock(256,1,1);
  Average<<<1, dimBlock>>>(A,B,256);
}
```

Must SYNC between writes. Otherwise there is a race condition: threads may be on different iterations and so read wrong data.

CASE 3 – NYY:
R/W same array
Multiple iterations
1 BLOCK

```
__global__ void Average(float* A, int N)
{
  int i = threadIdx.x;
  for (int j = 0; j < 100; j++) {
    if (i > 0 && i < N-1) {
      float X = (A[i-1] + A[i] + A[i+1])/3;
      __syncthreads();
      A[i] = X;
      __syncthreads();
    }
  }
}

int main()
{
 …
 // Kernel invocation
  dim3 dimBlock(256,1,1);
  Average<<<1, dimBlock>>>(A,256);
}
```

Must SYNC between reads and writes. Otherwise there is a race condition: threads may be on different iterations and so read wrong data.

Comment – What's the drawback of SYNCTHREADS?

At the end of the SYNC all activity must be completed, e.g., all writes by all threads in the BLOCK must have finished. This means that, if necessary, there must be enough space to hold all of these written intermediate values.

Why is this potentially a big deal? Let's look at the CASE with a single array.

Code without SYNC looks like this:
A[i] = (A[i-1]+A[i]+A[i+1])/3

Code with SYNC looks like this:
x = (A[i-1]+A[i]+A[i+1])/3
SYNC
A[i] = x

The issue is not that there is an extra operation (involving **x**), this would be done anyway.
Rather it is that there must be a variable **x** available for each thread for the life of the kernel.

This also gives an idea about why there is a limit on the number of threads per BLOCK (although this is not the direct reason, that has to do with the number of contexts in the SMs).
If there are 1024 THREADs in a BLOCK and a BLOCK runs on a single SM, then there needs to be enough space on a single SM for all of the data being held by each THREAD.

Comment – What about solving larger problems?

That is, what about problems where the array is bigger than the number of threads?
For array size N and number of THREADs T, each thread handles a number N/T output elements and N/T + 2 input elements (assuming padding at the very ends, which is also needed in the other case).

It makes a small difference whether there is just one array or multiple arrays (which swap).

- With multiple arrays, this works fine as before: just SYNC after every write.
- With a single array, you have to save a couple of values, but this works as well. The reason is that each THREAD need not save all of the intermediate values – it only needs to save the "boundary" elements to prevent a race condition.

CASES 4-7 -- Y(N/Y)(N/Y)

2. Multiple Blocks

1. One OR Many iterations

0. Same OR Different arrays

With multiple BLOCKs, each BLOCK takes a part of the input/output arrays.

That means that for all cases--except single-iteration/different-arrays--there must be communication among BLOCKs.

This is because of the edge elements, that is, the elements on the left and right ends of the output array. For all cases except 5, these depend on the inputs of other BLOCKs.

The key factor is that BLOCKs are all completely independent: they can be scheduled to start and complete in any order. That means that you cannot SYNC among THREADs in different BLOCKs.

CASE 4 – YNN:
R/W different arrays
1 iteration
Multiple BLOCKs

Easy: No sync needed.

```
__global__ void Average(float A[N], float B[N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x
  if (i > 0 && i < N)
    B[i] = (A[i-1] + A[i] + A[i+1])/3;
}

int main()
{
 …
 // Kernel invocation
  dim3 dimGrid(16,1,1);
  dim3 dimBlock(256,1,1);
  Average<<<dimGrid, dimBLOCK>>>(A, B, 4096);
}
```

CASES 5-7 – Y (NY|YN|YY)

R/W same array

single iteration

OR

R/W same array

multiple iterations

OR

R/W different arrays

multiple iterations

AND

Multiple BLOCKs

For all of these cases, Multiple BLOCKs means Multiple KERNEL LAUNCHES

Again, you cannot SYNC among THREADs in different BLOCKs

```
// CASE 5 -- Different Arrays, Multiple iterations
__global__ void Average1(float* A, float* B, int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x
  if (i > 0 && i < N-1)
    B[i] = (A[i-1] + A[i] + A[i+1])/3;
}

__global__ void Average2(float A[N], float B[N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x
  if (i > 0 && i < N-1)
    A[i] = (B[i-1] + B[i] + B[i+1])/3;
}

int main()
{
// Kernel invocations -- Since these are asynchronous,
// must force to launch all at once.
  dim3 dimGrid(16,1,1);
  dim3 dimBlock(256,1,1);
  for (int i = 0; i < 50; i++) {
    Average1<<<dimGrid, dimBlock>>>(A, B, N);
    cudaThreadSynchronize();
    Average2<<<dimGrid, dimBlock>>>(A, B, N);
    cudaThreadSynchronize();
  }
}
// cudaThreadSynchronize() → Blocks until the device has completed all
//  preceding requested tasks
```