CUDA

More on Blocks/Threads

Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (**`nvcc -deviceemu`**) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread
- Running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
    - Compile with –g –G   debug with: cuda-gdb <program name>
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. **`printf`**) and vice-versa
  - Detect deadlock situations caused by improper usage of **`__syncthreads`**

2

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode

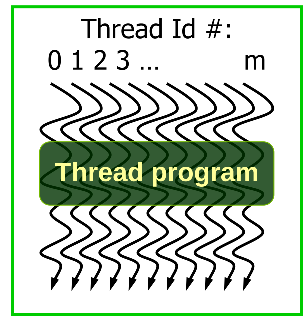
Floating Point

- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- 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 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!
  - End kernel and go back to host to enforce order

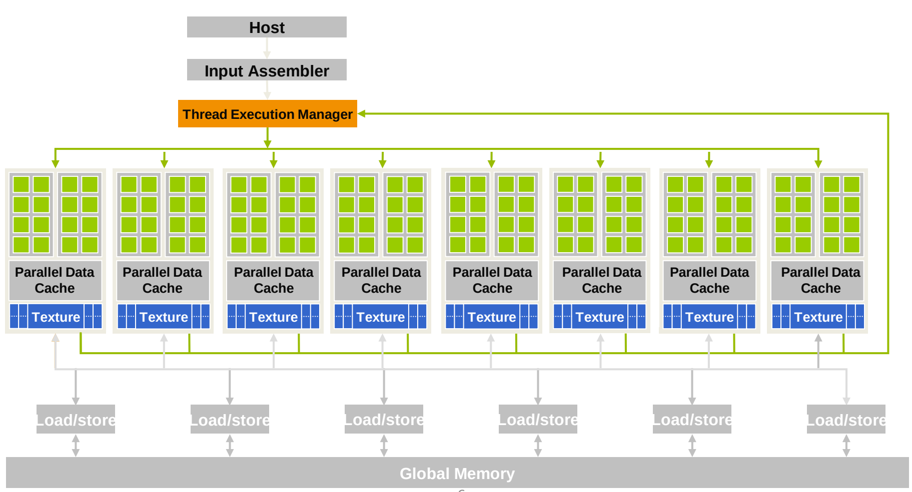
CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

G80 CUDA mode – A Review

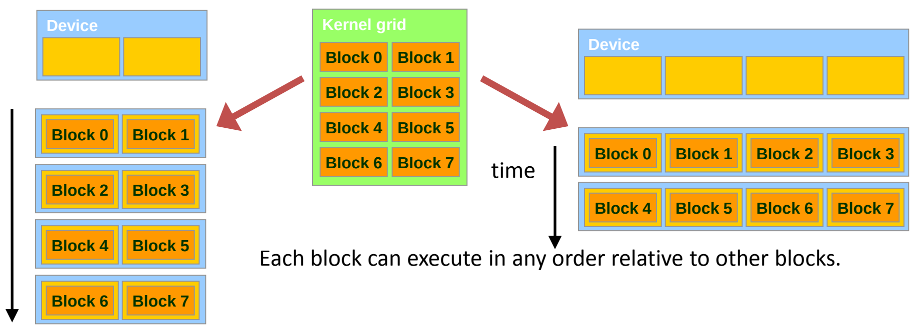
- Processors execute computing threads
- New operating mode/HW interface for computing



6

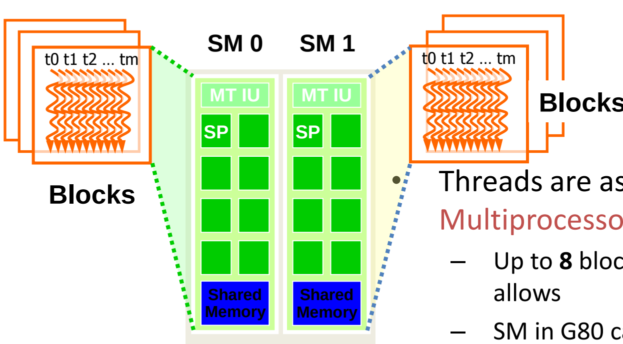
Transparent Scalability

- 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.

G80 Example: Executing Thread Blocks

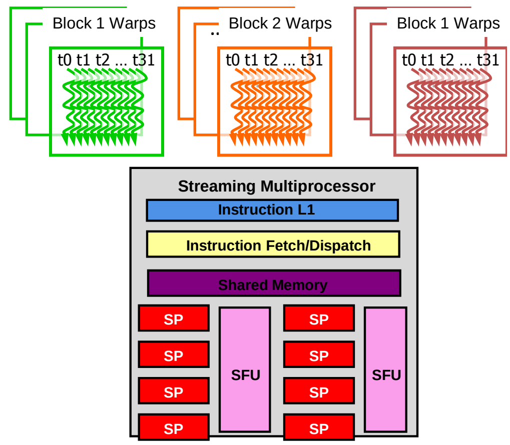


- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **8** blocks to each SM as resource allows
  - SM in G80 can take up to **768** threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

8

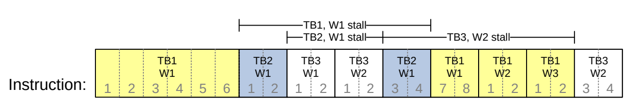
G80 Example: Thread Scheduling

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in 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



G80 Example: Thread Scheduling (Cont.)

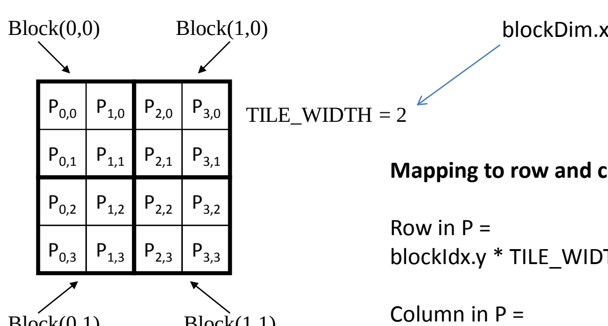
- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in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



G80 Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 threads per block?
  - For 8X8, 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 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!
- Our earlier Julia fractal implementation not as good as it could have been; why not?

Sub-Blocks and Threads



Mapping to row and column

Row in P =
blockIdx.y * TILE_WIDTH + threadIdx.y

Column in P =
blockIdx.x * TILE_WIDTH + threadIdx.x

Then map to 1D array

P[Row * WIDTH + Column] = Value

Example

- Matrix Mul program:

```
#define DIM 4

__global__ void MatrixGenerate(int* M, int* N, int* P, int width)
{
      int row = blockIdx.y * blockDim.x + threadIdx.y;
      int col = blockIdx.x * blockDim.x + threadIdx.x;
      P[row * width + col] = (row + col);
}

dim3 blocks(DIM/2, DIM/2);
dim3 threads(DIM/2, DIM/2);
MatrixGenerate<<<blocks,threads>>>(dev_m, dev_n, dev_p, DIM);
```

Improved Julia Fractal

- Change block/thread size to better utilize thread support per SM

```
#define DIM 3008  // 16*188

__global__ void kernel(char *ptr)
{
      int row = blockIdx.y * blockDim.x + threadIdx.y;
      int col = blockIdx.x * blockDim.x + threadIdx.x;
      int offset = col + row * DIM;
      ptr[offset] = julia(row,col);
}

dim3 blocks(188,188);
dim3 threads(16,16);
kernel<<<blocks,threads>>>(dev_charmap);
```

Long Vectors

- Using 1 block, limited to 512 threads
- Maximum of 65535 blocks
- If you want to operate on something longer than 65535 even if it's 1D then we have to combine blocks and threads

Block 0	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
Block 1	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
Block 2	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
Block 3	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
…					
Block 32000					

1D array index = (blockIdx.x * blockDim.x) + threadIdx.x = 0 to 32000*5+4 = 160,004

Arbitrarily Long Vectors

- The limit is 512 threads per block, so there is a failure if the vector is of size N and N/512 > 65535
  - N > 65535*512 = 33,553,920 elements
  - Pretty big but we could have the capacity for up to 4GB
- Solution
  - Have to assign range of data values to each thread instead of each thread only operating on one value

Some Additional API Features

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`
- Etc.
  - 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

Device Runtime Component: Mathematical Functions

- 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