King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 12: GPU Compute APIs, Pt. 2

Markus Hadwiger, KAUST

Reading Assignment #6 (until Oct 7)

Read (required):

- Programming Massively Parallel Processors book (4th edition), **Chapter 3** (*Multidimensional grids and data*)

Read (optional):

- Inline PTX Assembly in CUDA: `Inline_PTX_Assembly.pdf`
- Dissecting GPU Architectures through Microbenchmarking:

  Volta: `https://arxiv.org/abs/1804.06826`
  Turing: `https://arxiv.org/abs/1903.07486`

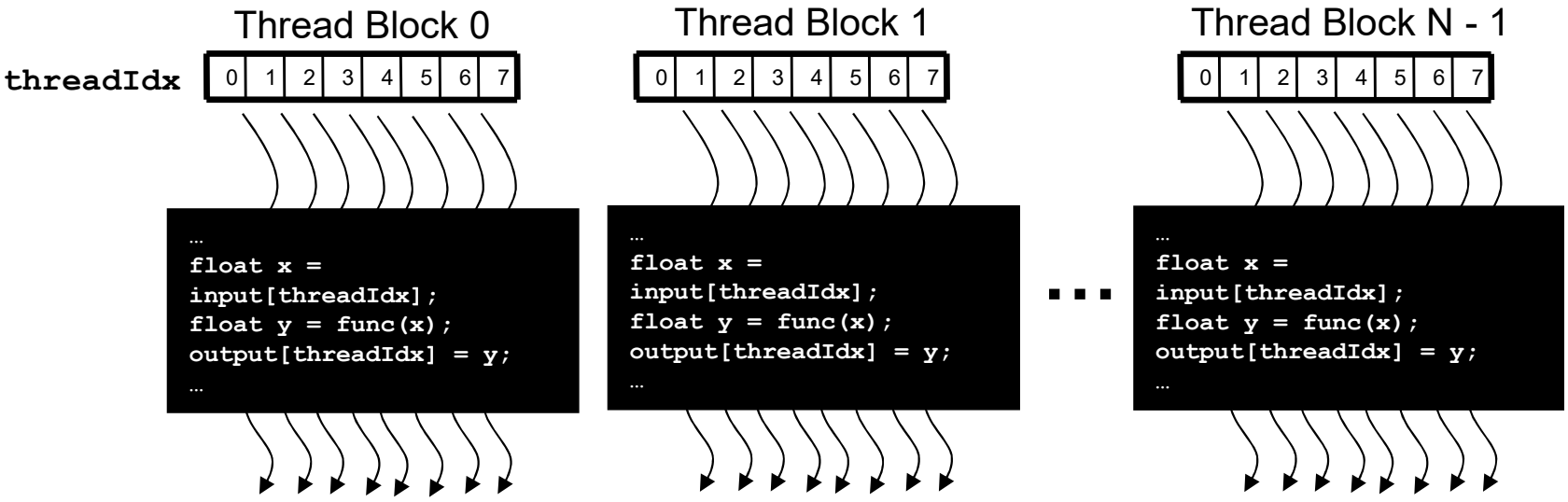
`https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9839-discovering-the-turing-t4-gpu-architecture-with-microbenchmarks.pdf`

Ampere: `https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s33322/`

GPU Compute APIs

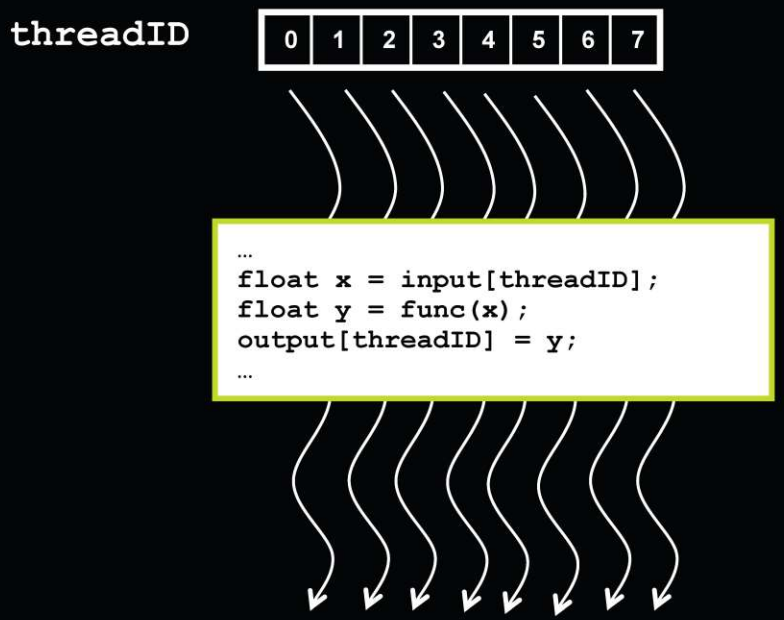
Threads in Block, Blocks in Grid

- Identify work of thread via
  - `threadIdx`
  - `blockIdx`



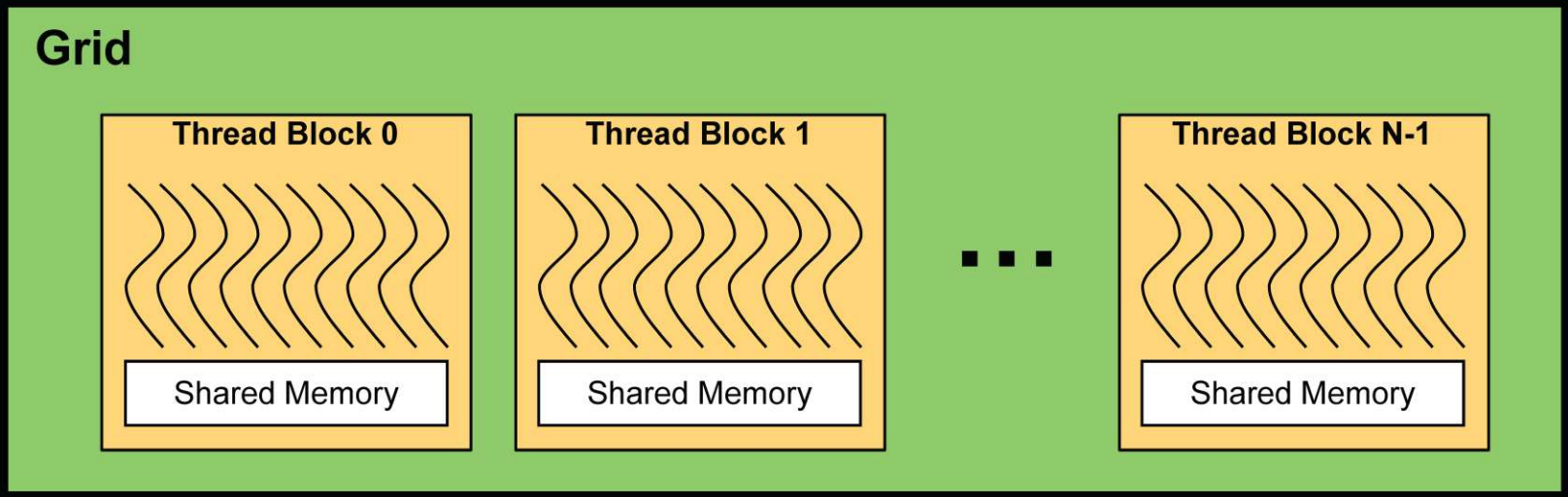
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



Thread Batching

- Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate*
- Allows programs to *transparently scale* to different GPUs



* brand new on Hopper: thread block clusters

Launching Kernels

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(…)
```

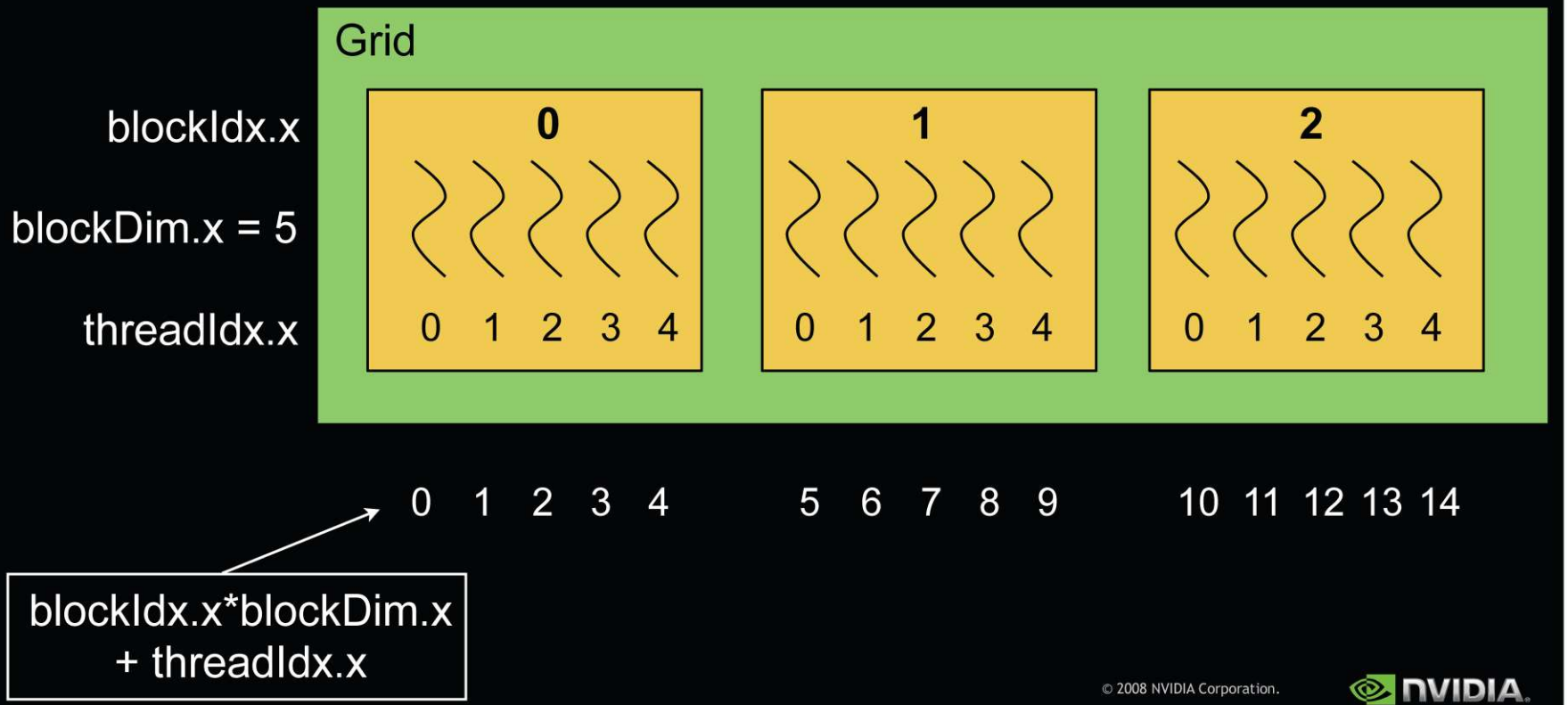
- Execution Configuration ("<<< >>>")
  - dG - dimension and size of grid in blocks
    - Two-dimensional: x and y
    - Blocks launched in the grid: dG.x * dG.y
  - dB - dimension and size of blocks in threads:
    - Three-dimensional: x, y, and z
    - Threads per block: dB.x * dB.y * dB.z
  - Unspecified dim3 fields initialize to 1

CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
  - Map from local thread ID (threadIdx) to a global ID which can be used as array indices



Increment Array Example

CPU program

```
void inc_cpu(int *a, int N)
{
  int idx;

  for (idx = 0; idx<N; idx++)
    a[idx] = a[idx] + 1;
}

int main()
{
  ...
  inc_cpu(a, N);
}
```

CUDA program

```
__global__ void inc_gpu(int *a, int N)
{
  int idx = blockIdx.x * blockDim.x
          + threadIdx.x;
  if (idx < N)
    a[idx] = a[idx] + 1;
}

int main()
{
  …
  dim3 dimBlock (blocksize);
  dim3 dimGrid( ceil( N / (float)blocksize) );
  inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```

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
- Allowed in conditional code only if the conditional is uniform across the entire thread block

Synchronization

- Threads in the same block can communicate using shared memory

- __syncthreads()
  - Barrier for threads only within the current block
- __threadfence()
  - Flushes global memory writes to make them visible to all threads

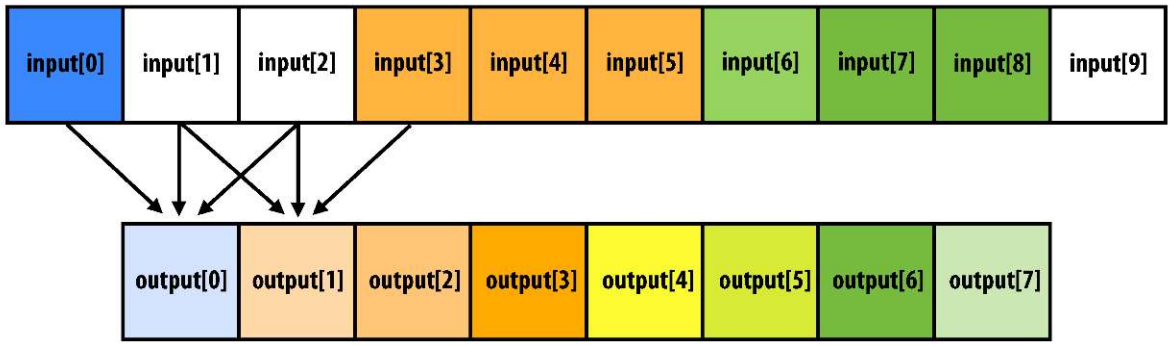
Plus newer sync functions, e.g., from compute capability 2.x on: __syncthreads_count(), __syncthreads_and/or(), __threadfence_block(), __threadfence_system(), …

Now: *Must* use versions with _sync suffix, because of Independent Thread Scheduling (compute capability 7.x and newer)!

Code Example #1: 1D Convolution

Example #1: 1D Convolution

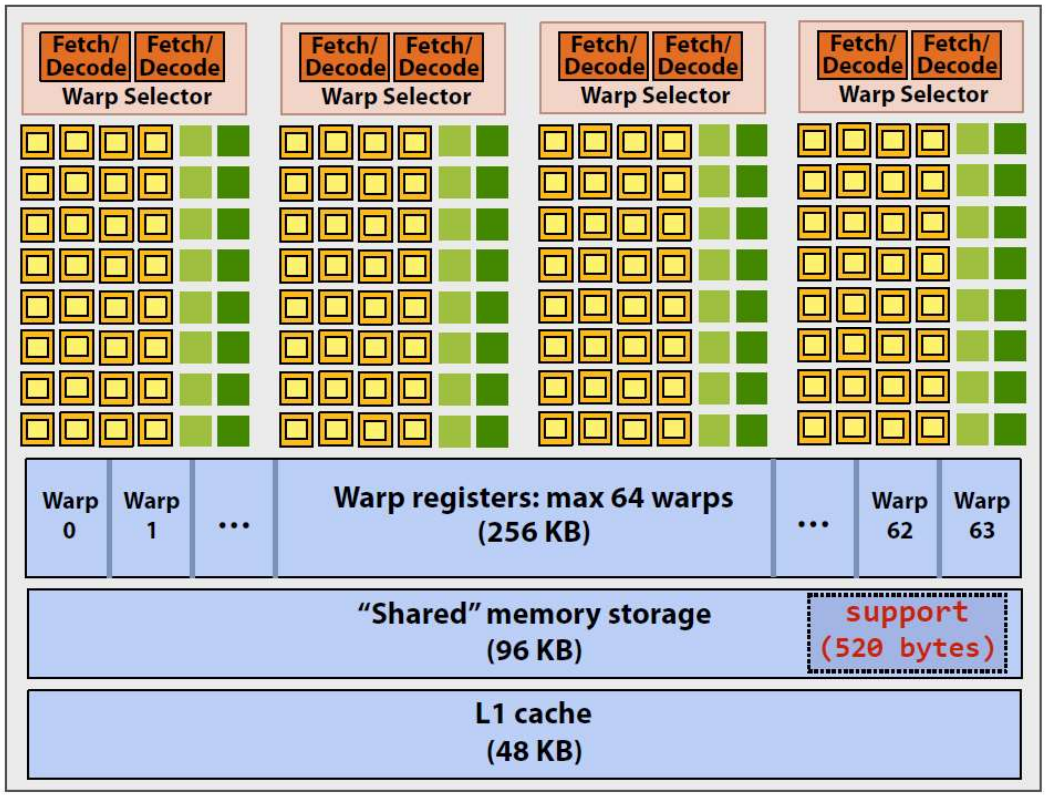
1D Convolution with 3-tap averaging kernel (every thread is averaging three inputs)




```
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".

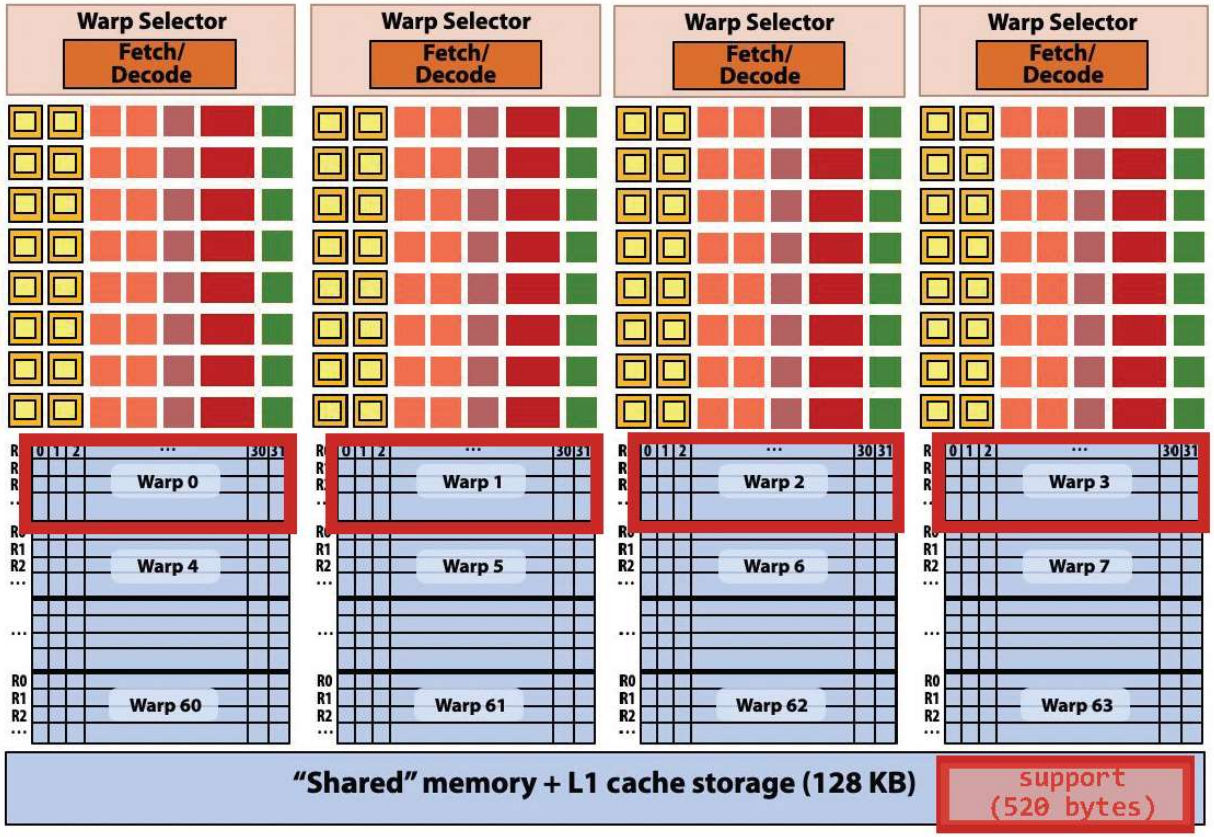
A `convolve` thread block is executed by 4 warps (4 warps x 32 threads/warp = 128 CUDA threads per block)

(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)

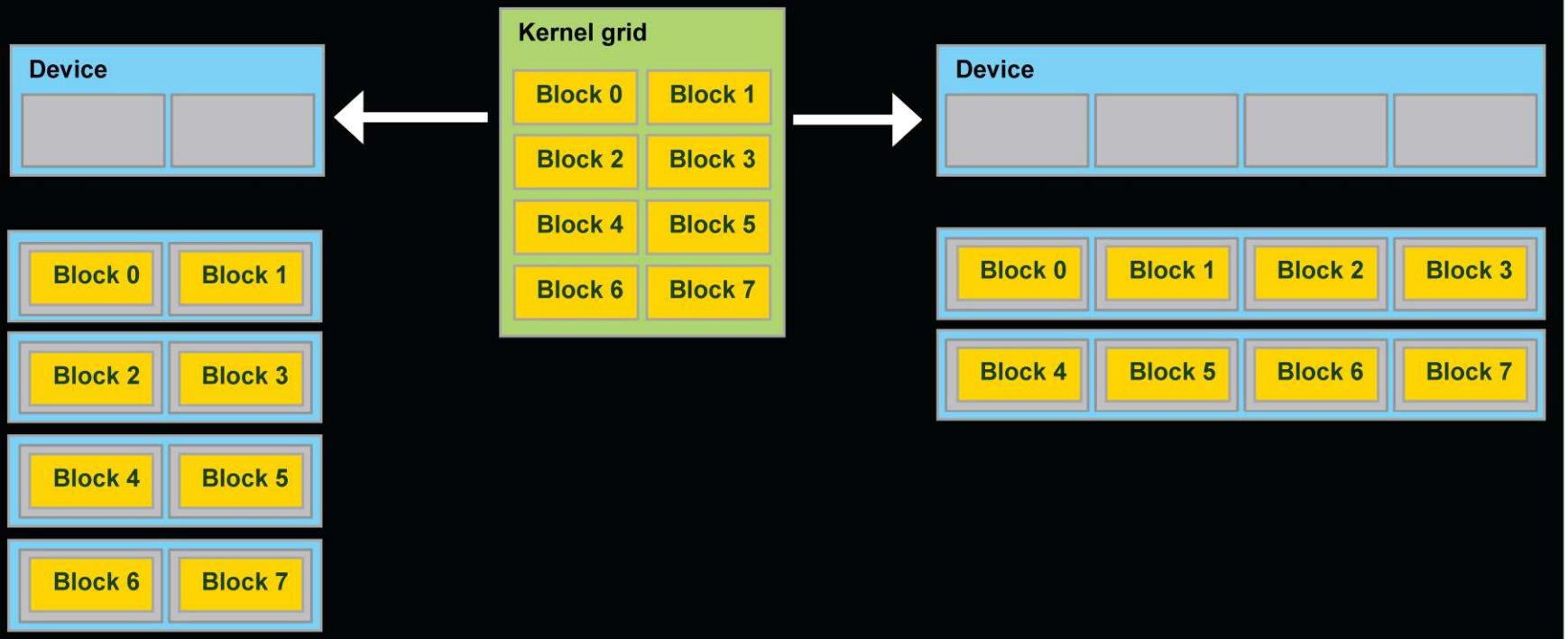
SM core operation each clock:

- **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

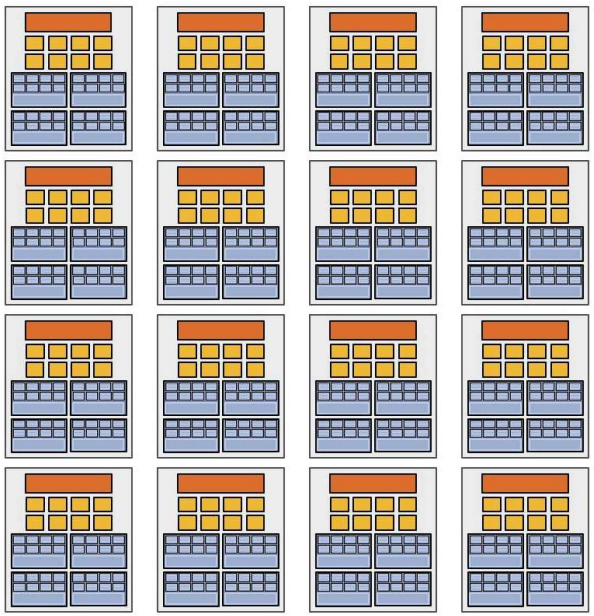
Transparent Scalability

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

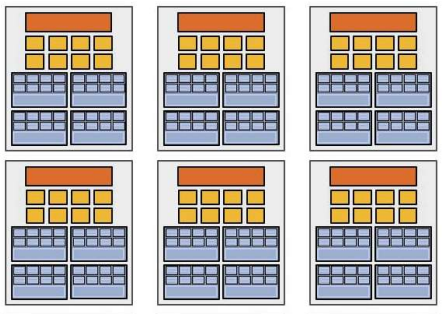


Code on Same SM Arch. But Different #SMs

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 144 SMs, etc., …)

Thank you.