# CSC266 Introduction to Parallel Computing using GPUs
# GPU Architecture I (Execution)

Sreepathi Pai

October 25, 2017

URCS

# Outline

Quick Recap

Streams and Command Queues

GPU Execution of Kernels

Warp Divergence

More on GPU Occupancy

# Outline

# Launch

- Determine a thread block size: say, 256 threads
- Divide work by thread block size
  - Round up
  - $\lceil N/256 \rceil$
- Configuration can be changed *every* call

```
int threads = 256;
int Nup = (N + threads - 1) / threads;
int blocks = Nup / threads;

vector_add<<<blocks, threads>>>(...)
```

# Kernel Launch Configuration

- GPU kernels are SPMD kernels
  - Single-program, multiple data
  - All threads execute the same code
- Number of threads to execute is specified at launch time
  - As a *grid* of $B$ *thread blocks* of $T$ threads each
  - Total threads: $B \times T$
- Reason: Only threads within the same thread block can communicate with each other (cheaply)
  - Other reasons too, but this is the only algorithm-specific reason

# Blocking and Non-blocking APIs

- Blocking API (or operation)
  - CPU waits for operation to finish
  - e.g. simple `cudaMemcpy`
- Non-blocking API (or operation)
  - CPU does not wait for operation to finish
  - e.g. kernel launches
  - You can wait explicitly using special CUDA APIs
- Operations queue up
  - Multiple kernels can be launched
  - They will execute by default in launch order

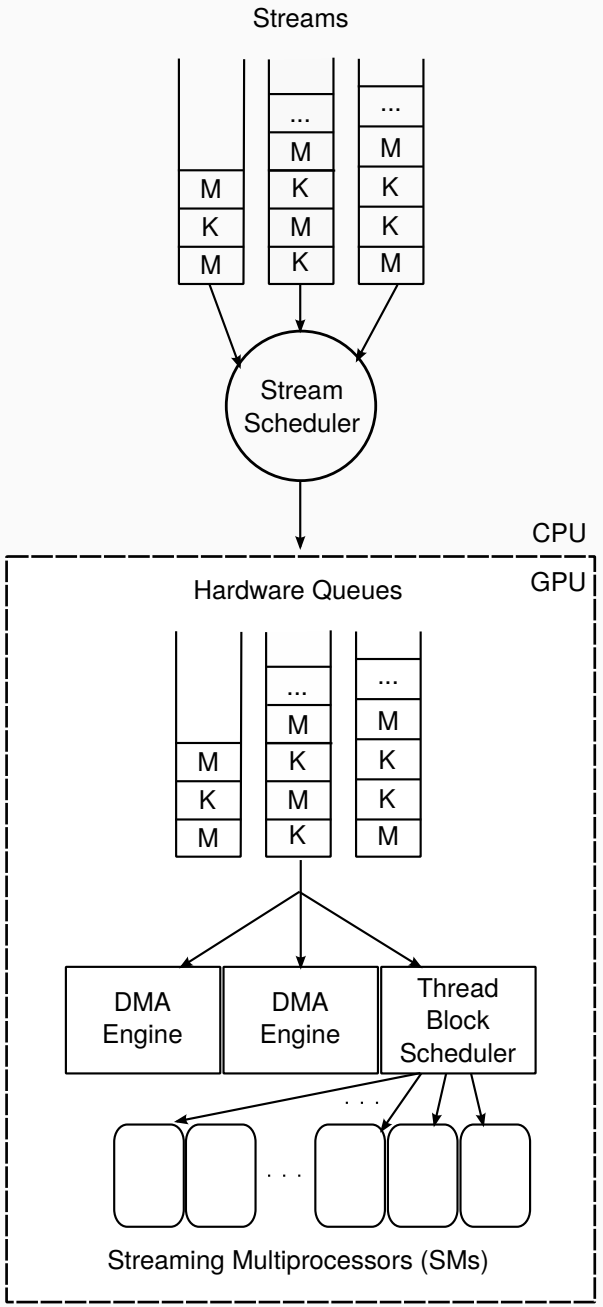
# Outline

# The Default Stream

- A GPU can do multiple things in parallel
  - Just like a CPU
  - Most common: overlapping memory copies and kernel executions
- Main programming construct: **Stream**
  - Purely software construct
- Stream is conceptually equivalent to a CPU thread
  - Operations in same stream happen in order
  - Operations in different streams can happen in *any* order
- Stream 0 is the *default stream*
  - All operations not on an explicit stream are on this stream

# Command Queue

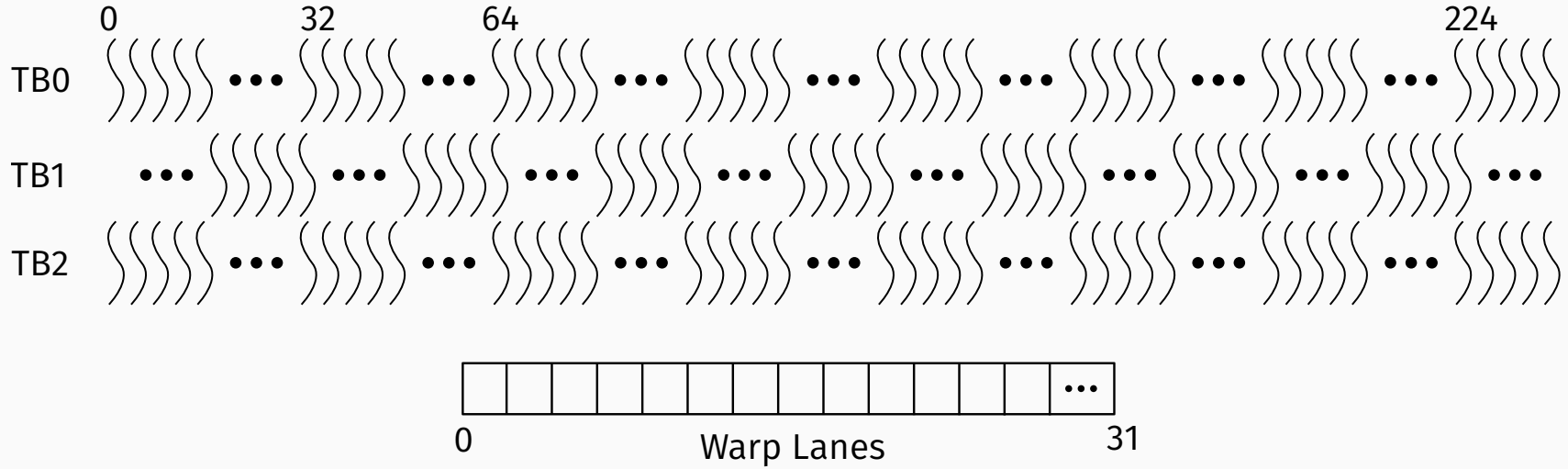
- Hardware construct
- Streams map to command queues
  - Many (streams)-to-one (hardware queue)
- About 32 hardware queues in Kepler (Hyper-Q)



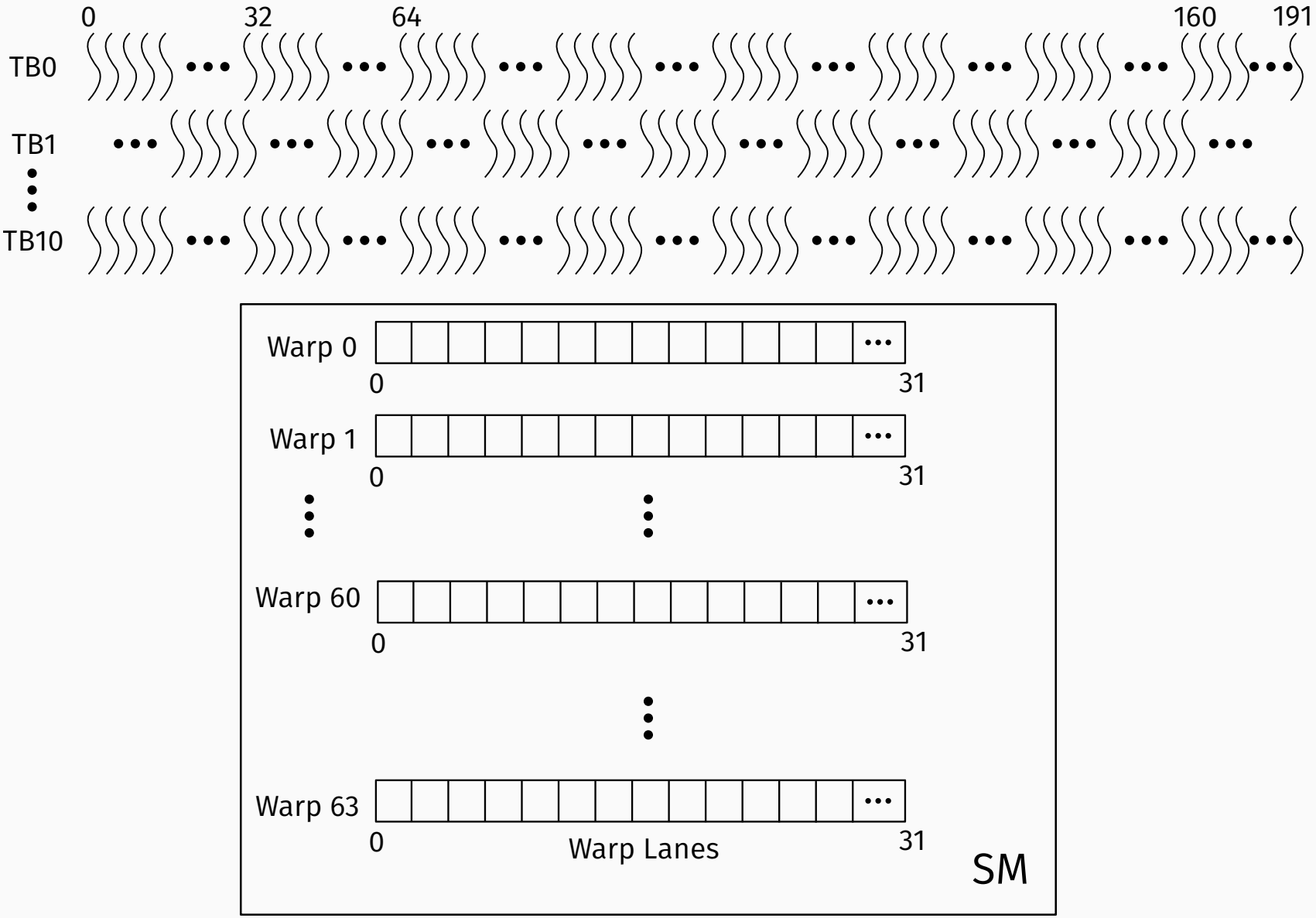
# Outline

# Why Grids? NVIDIA: Hardware Scalability

# Threads to Warps



# Thread blocks to an SM



# GPU Occupancy

- CPU threads share resources by time multiplexing
  - One thread owns all CPU resources (registers, etc.) for its time slice
  - Context-switches are performed by OS
- GPU threads *do not share* resources
  - Own fixed partition of resources for entire lifetime of thread
  - Context-switches are performed by hardware every few cycles
- Changing number of threads changes *utilization* of resources

# GPU Resources per SM (NVIDIA Kepler)

| Resource | Available | Maximum |
|---|---|---|
| Threads | 2048 | 1024/block |
| Shared Memory | 48K (max) | 48K/block |
| Registers | 65536 | 255/thread |
| Thread Blocks | 16 | 16/SM |

- Every block consumes:
  - $T$ threads
  - $T \times R$ registers where $R$ is registers per thread
  - 1 block
  - $SM$ shared memory per block (optional)
- The resource that gets exhausted first determines occupancy and residency
  - *Occupancy*: number of hardware threads utilized
  - *Residency*: number of hardware blocks utilized

# GPU Occupancy: Example 1

```
kernel<<<2048, 32>>>()
```

- $T = 32$
  - thread limit $2048/32 = 64$ thread blocks
- $R = 100$ ($100 \times 32 = 3200$ per thread block)
  - register limit $65536/3200 = 20$ thread blocks
- $SM = 1K$
  - SM limit $48K/1K = 48$ thread blocks

- Limiting resource: thread blocks (16)
- Residency: 16
- Occupancy: $(16 \times 32)/2048 = 25\%$

# GPU Occupancy: Example 2

```
kernel<<<2048, 64>>>()
```

- $T = 64$
  - thread limit $2048/64 = 32$ thread blocks
- $R = 100$ ($100 \times 64 = 6400$ per thread block)
  - register limit $65536/6400 =?$ thread blocks
- $SM = 1K$
  - SM limit $48K/1K = 48$ thread blocks

- Limiting resource: ?
- Residency: ?
- Occupancy: $(? \times 64)/2048 =?\%$

# How many threads?

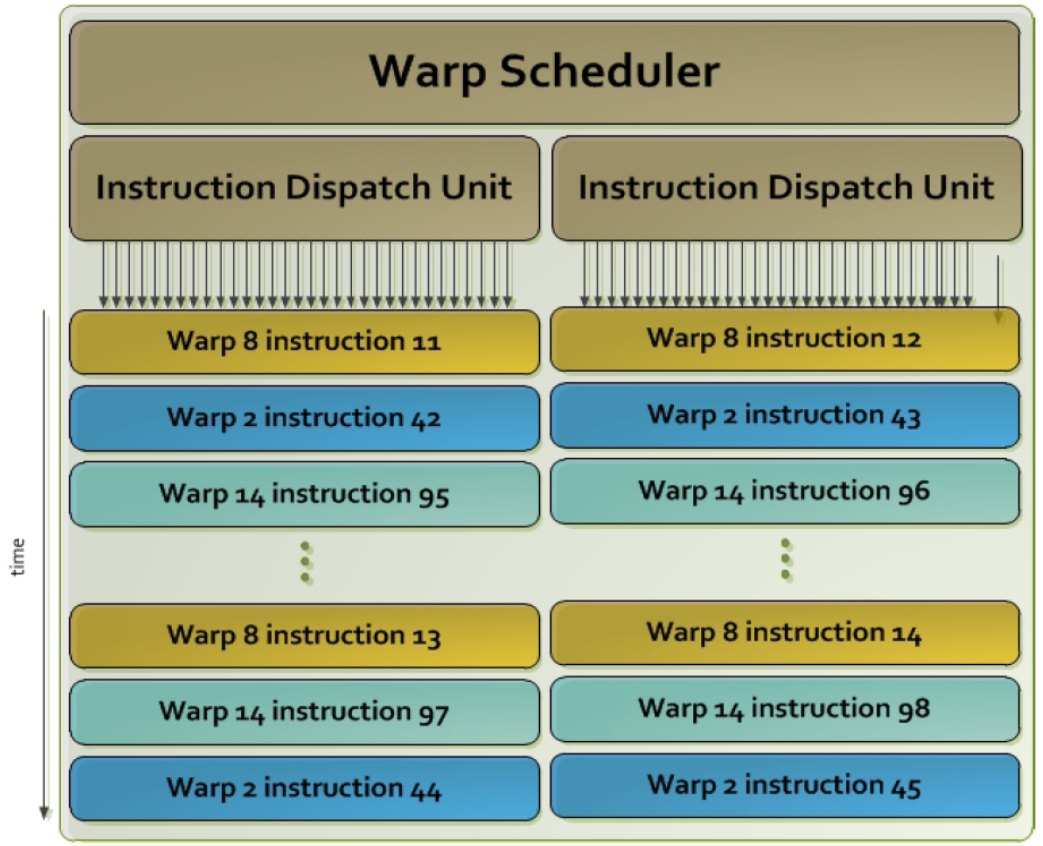
- Try to maximize utilization (NVIDIA Manual)
- Later today: Better strategy

# Summary

- Thread blocks are mapped to SMs "whole"
  - Atleast one thread block must fit
  - No partial thread blocks
- Upto *res* thread blocks per SM
  - *res* is residency
  - Different for different kernels
- Once all SMs are occupied, remaining blocks wait
  - Start running once currently running blocks finish

# Outline

# SIMT Issue




- All threads in a warp execute the same instruction (same PC)
- What happens when:
  - that instruction is a conditional branch?
  - is a load that misses for some threads but not others?

# Divergence

- If threads in a warp decide execute different PCs, the warp *splits*
- Two directions for a branch
  - Two splits
  - Each split is executed serially
  - Nested branches also split correctly
- Join back at a pre-determined "meet" point
  - Immediate post-dominator

# Example

```
if (cond) {
  x = 1;
} else {
  y = 1;
}
```

- Assume warps contains four threads each
- Assume only T0, T2 have `cond == true`.

```
Time        T0     T1     T2     T3
0         x = 1          x = 1
1                y = 1          y = 1
```

- If `cond` is true for all threads

```
Time        T0      T1      T2      T3
0         x = 1   x = 1   x = 1   x = 1
```

# Tackling Divergence

- Threads in the same warps should avoid divergent conditions
  - Easier said than done
- Threads in the same warp should try to access locations in same memory line
  - Memory divergence *repeats* requests until all threads have received data
- Compiler will predicate instructions
  - No divergence – both sides executed
  - Predicated instructions are executed but do not commit
  - Shown as `[]` below

```
Time        T0       T1       T2       T3
0         x = 1   [x = 1]   x = 1   [x = 1]
1        [y = 1]   y = 1   [y = 1]   y = 1
```

# Outline

# Occupancy Recap

- GPUs partition resources among running threads
- NVIDIA Manual says maximize occupancy
    - Why?

# Reasoning about occupancy

```
kernel <<<x, y>>>()
```

- Consider:
  - 1 Thread Block
  - $N$ thread blocks, $N$ equal to number of SMs/SMX
  - $N * \mathit{Residency}$ thread blocks
  - $> N * \mathit{Residency}$ thread blocks

# Less Occupancy?

- Is there a case to reduce occupancy/residency?
  - i.e. let threads consume more resources?
  - smaller thread blocks?

# Better Performance at Lower Occupancy

Multiplication of two large matrices, single precision (SGEMM):

| | CUBLAS 1.1 | CUBLAS 2.0 | |
|---|---|---|---|
| Threads per block | 512 | 64 | 8x smaller thread blocks |
| Occupancy (G80) | 67% | 33% | 2x lower occupancy |
| Performance (G80) | 128 Gflop/s | 204 Gflop/s | 1.6x higher performance |

Batch of 1024-point complex-to-complex FFTs, single precision:

| | CUFFT 2.2 | CUFFT 2.3 | |
|---|---|---|---|
| Threads per block | 256 | 64 | 4x smaller thread blocks |
| Occupancy (G80) | 33% | 17% | 2x lower occupancy |
| Performance (G80) | 45 Gflop/s | 93 Gflop/s | 2x higher performance |

Volkov, V., "Better Performance at Lower Occupancy", GTC 2010

# Volkov's Insights

- Do more parallel work per thread to hide latency with fewer threads (i.e. increase ILP)
  - Unroll
- Use more registers per thread to access slower shared memory less
  - Shared memory latency comparable to registers, but
  - Shared memory throughput is lower!
- Both may be accomplished by computing multiple outputs per thread
- Note that Volkov underutilizes threads, but maxes out registers!
  - Fermi had 63 registers/thread, Kepler has 255 registers/thread
  - Why have a register limit?