CUDA Programming Introduction II Architecture and Threading Primer

Nikos Hardavellas

Some slides/material from:
UToronto course by Andreas Moshovos
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas
Universitat Jena by Waqar Saleem
NVIDIA by Simon Green and many others
Real World Techonologies by David Kanter

1

Execution Configuration

- Must specify when calling a `__global__` function:

```
<<< Dg, Db [, Ns [, S]] >>>
```

- where:
  - `dim3 Dg`: grid dimensions in blocks
  - `dim3 Db`: block dimensions in threads
  - `size_t Ns`: per block additional number of shared memory bytes to allocate
    - optional, defaults to 0
    - more on this much later on
  - `cudaStream_t S`: request stream(queue)
    - optional, default to 0.
    - Compute capability >= 1.1

2

Built-in Variables

- **`dim3 gridDim`**
  - Number of blocks per grid, in 2D (.z always 1)
- **`uint3 blockIdx`**
  - Block ID, in 2D (blockIdx.z = 1 always)
- **`dim3 blockDim`**
  - Number of threads per block, in 3D
- **`uint3 threadIdx`**
  - Thread ID in block, in 3D

3

Execution Configuration Examples

- 1D grid / 1D blocks

```
dim3 gd(1024)
dim3 bd(64)
akernel<<<gd, bd>>>(...)
gridDim.x = 1024, gridDim.y = 1,
blockDim.x = 64, blockDim.y = 1,
blockDim.z = 1
```

- 2D grid / 3D blocks

```
dim3 gd(4, 128)
dim3 bd(64, 16, 4)
akernel<<<gd, bd>>>(...)
gridDim.x = 4, gridDim.y = 128,
blockDim.x = 64, blockDim.y = 16,
blockDim.z = 4
```

4

Synchronous vs. Asynchronous data transfers

- `cudaMemcpy()` is mostly synchronous, i.e., blocking
  - Synchronous when size >64KB
  - Transfers ≤64KB are asynchronous
  - Host can always modify data after call returns w/o worries
    - Transfers ≤64KB go into intermediate buffer
- `cudaMemcpyAsync()` is asynchronous, i.e., non-blocking
  - Requires pinned memory (`cudaMallocHost()`)
  - Requires stream ID
  - Pinned memory allows for faster data transfers
    - But not pageable, so it may stress the memory system
- No need to synchronize data copy H→D with following kernel within the same stream

5

Asynchronous data transfers

```
cudaMemcpyAsync(    a_d,
                    a_h,
                    size,
                    cudaMemcpyHostToDevice,
                    0 /* default stream ID */);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

- Kernel is also sent to default stream 0
- Kernel starts on device after Memcpy finishes
- cudaMemcpyAsync() and Kernel are asynchronous with respect to CPU
- cpuFunction can start executing (almost) immediately
- Overlap data transfer and execution on device with computation on the host CPU

6

deviceOverlap field

- Allows concurrent H→D data copy and kernel execute on device
- deviceQuery in SDK reports this field
- Requires pinned memory
- Data transfer and kernel invocation require different non-default streams

- From the CUDA C Best Practices Guide:
  "memory copy, memory set functions, and kernel calls that use the default stream begin only after all preceding calls on the device (in any stream) have completed, and no operation on the device (in any stream) commences until they are finished."

7

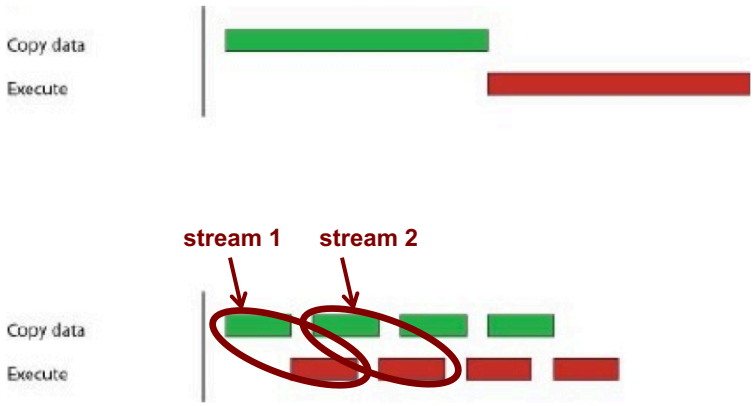
Example: Overlap Memcpy and Kernel Execution

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
... // malloc, init and copy memory to stream1

// now work on stream 2
cudaMalloc(&a_d, size);
cudaMallocHost(&a_h, size); /* pinned memory */
... // initialize a_h

cudaMemcpyAsync(a_d,
                a_h,
                size,
                cudaMemcpyHostToDevice,
                stream2);

kernel<<<grid, block, 0, stream1>>>(otherData_d);
```

8

Staged Concurrent Copy and Execute

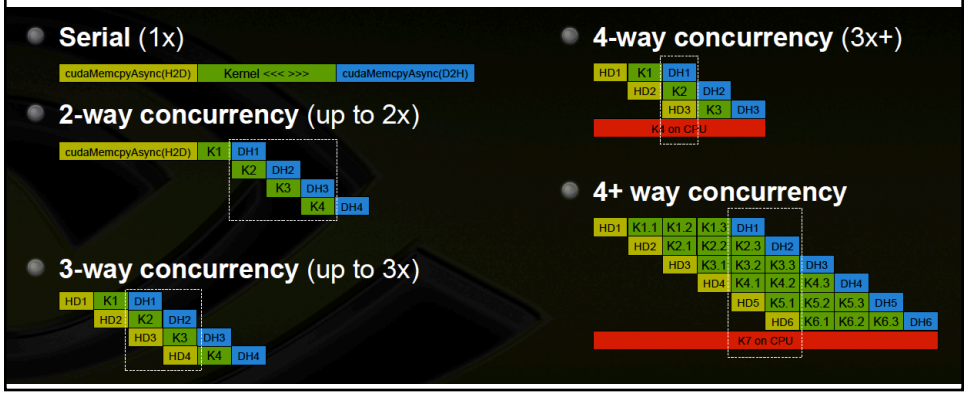
```
size = N*sizeof(float)/nStreams;
nBlocks = N/(nThreads*nStreams);
dir = cudaMemcpyHostToDevice;
for (i=0; i < nStreams; i++) {
  k = i*N/nStreams; /* offset */
  cudaMemcpyAsync(a_d+k, a_h+k, size, dir, stream[i]);
  kernel<<<nBlocks, nThreads,0, stream[i]>>> (a_d+k);
}
```



9

Streams and Concurrency

- Stream
  - Sequence of operations that execute in issue-order on GPU
- Programming model used affects concurrency
  - CUDA operations in different streams may run concurrently
  - CUDA operations from different streams may be interleaved



10

Default Stream (a.k.a. Stream 0)

- Stream used when no stream is specified
- Completely synchronous w.r.t. host and device
  - As if cudaDeviceSynchronize() inserted before and after every CUDA operation
- Exceptions – asynchronous w.r.t. host
  - Kernel launches in the default stream
  - cudaMemcpy*Async
  - cudaMemset*Async
  - cudaMemcpy within the same device
  - H2D cudaMemcpy of 64kB or less
    - Or during the last 64kB of a larger transfer

11

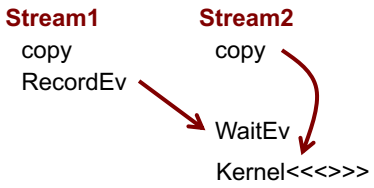
Explicit Synchronization

- Synchronize everything
  - **cudaDeviceSynchronize** ()
    - Deprecated version: **cudaThreadSynchronize**()
  - Blocks host until all issued CUDA calls are complete across all streams
- Synchronize w.r.t. a specific stream
  - **cudaStreamSynchronize** (streamid)
  - Blocks host until all CUDA calls in streamid are complete
- Synchronize using Events
  - Create specific 'Events', within streams
  - **cudaEventCreate** (&event)
  - **cudaEventRecord** (event, stream)
  - **cudaEventSynchronize** (event)
  - **cudaStreamWaitEvent** (stream, event)
  - **cudaEventQuery** (event)

12

Event Synchronization Example

```
cudaEvent_t event;
cudaEventCreate (&event);

// H2D copy of new input
cudaMemcpyAsync (d_in, in, size,
                 H2D, stream1);
cudaEventRecord (event, stream1);

// D2H copy of previous result
cudaMemcpyAsync (out, d_out, size,
                 D2H, stream2);

// wait for event in stream1
cudaStreamWaitEvent (stream1, event );
// must wait for 1 and 2
kernel <<< , , , stream2 >>> (d_in, d_out);
```



13

Implicit Synchronization

- Implicitly synchronization operations
  - cudaMallocHost (pinned memory)
  - cudaHostAlloc (like the above + more flags)
  - cudaMalloc (device memory)
  - cudaFree
  - cudaMemcopy* (no Async suffix, except last 64kB)
  - cudaMemset* (no Async suffix, except last 64kB)
  - cudaDeviceSetCacheConfig (changes to L1/ShM)

14

Variable Declarations – Will revisit later

- **__device__**
  - example: `__device__ int DeviceVar;`
  - stored in device global memory (large, high latency, no L1 cache)
  - Allocated with `cudaMalloc(…)` call; `__device__` qualifier implied
  - accessible by all threads
  - lifetime: application
- **__constant__**
  - example: `__constant__ int ConstantVar;`
  - same as `__device__`, but L1 cached and read-only by GPU
  - written by CPU via `cudaMemcpyToSymbol(...)` call
  - lifetime: application
- **__shared__**
  - example: `__shared__ int SharedVar;`
  - stored in on-chip shared memory (very low latency)
  - accessible by all threads in the same thread block
  - lifetime: kernel launch
- **Unqualified variables:**
  - scalars and built-in vector types are stored in registers (if don't fit: local memory)
  - arrays of more than 4 elements or run-time indices stored in device memory

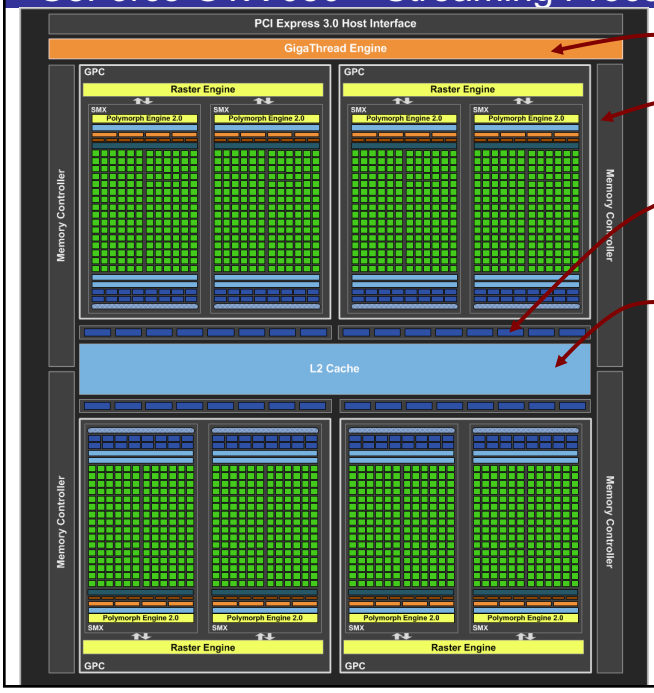
15

Example: Initialize a Constant Array

```
#define DSIZE 32
__constant__ int mydata[DSIZE];

int main() {
  ...
  int *h_mydata;
  h_mydata = new int[DSIZE];
  for (int i = 0; i < DSIZE; i++)
    h_mydata[i] = ....; // initialize
  cudaMemcpyToSymbol(mydata, h_mydata, DSIZE*sizeof(int));
  ...
}

__global__ void mykernel(...) {
  ...
  int myval = mydata[threadIdx.x];
  ...
}
```

16

GeForce GTX 680 – Streaming Processor Array (SPA)



Block scheduling

Memory access

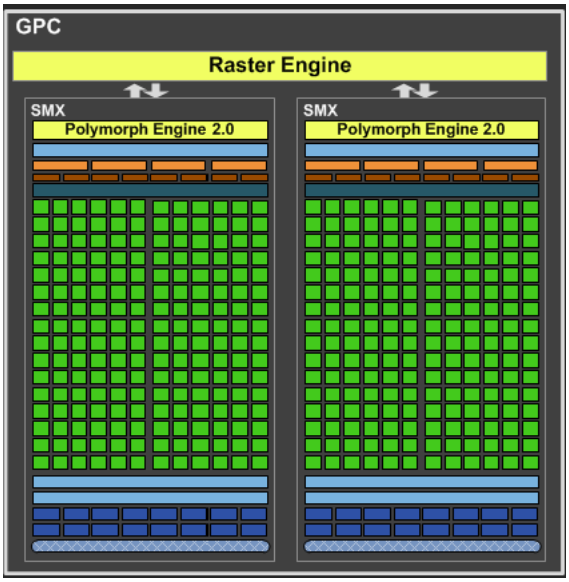
ROP units (blending, Z-buffering, antialiasing)

512KB L2 cache

- 3.54bn transistors
- 1536 CUDA cores
- Core: 1.006GHz
  - Up to 1.1GHz
- Mem: 6.008GHz (?)
  - Lab machines report 3.004GHz
- 3.09 TFLOPs
- 195W

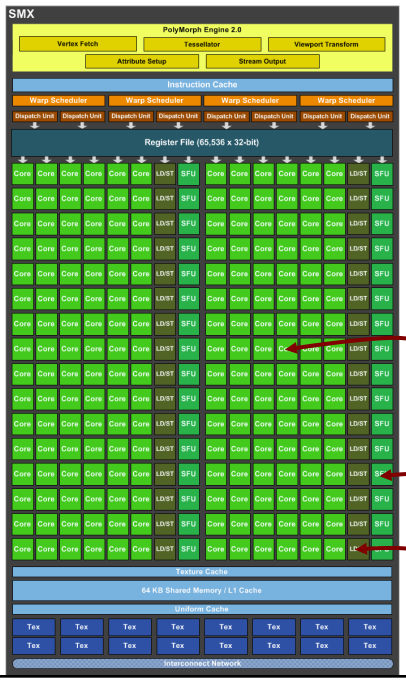
17

GeForce GTX 680 – Graphics Processing Cluster



18

GeForce GTX 680 – Streaming Multiprocessor



- **SM (a.k.a. SMX, SMP)**
  - Streaming Multiprocessor
  - Multi-threaded processor
    - 192 CUDA cores
    - 1 to 2048 threads active
  - Shared instruction fetch per 32 threads
  - Fundamental processing unit for CUDA thread block
- **SP (a.k.a. CUDA core)**
  - Streaming Processor
  - Scalar ALU for a single CUDA thread
- **SFU**
  - Special function unit
- **LDST**
  - Memory access unit

19

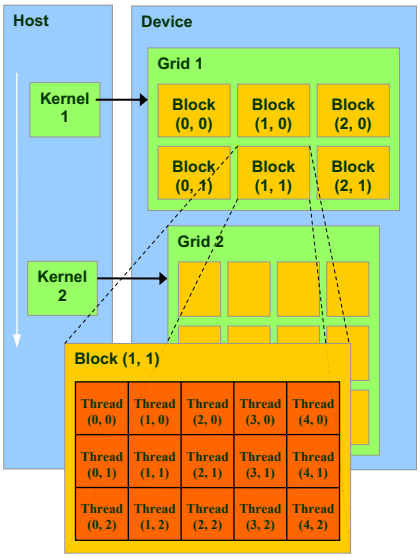
Scheduling Threads for Execution

- Break data into Blocks (grid)
- Break Blocks into Warps
  - 32 consecutive threads
- Allocate Resources
  - Registers, Shared Mem, Barriers
- Then allocate for execution

20

Thread Life

- Grid is launched on the SPA
  - Kepler allows up to 32-way grid concurrency (streams)
    - GTX680: up to 16 grids
- Thread Blocks are serially distributed to all the SMs
  - Potentially >1 Thread Block per SM
- Each SM launches **Warps** of 32 Threads
  - 3 levels of parallelism
- SM schedules and executes **Warps** that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks

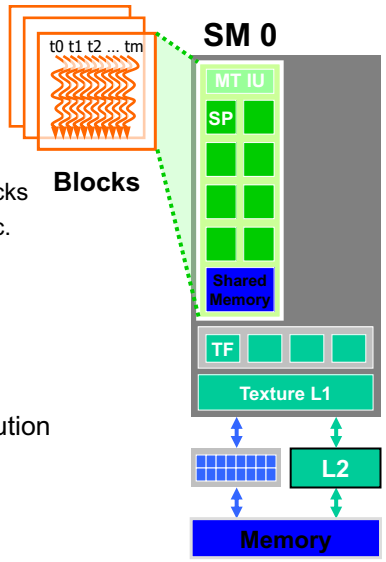


21

Stream Multiprocessors Execute Blocks

- Threads are assigned to SMs at Block granularity
  - Up to **16 Blocks** per SM
  - Up to **64 Warps** per SM
  - Up to **2K threads** per SM
    - Could be 512 (threads/block) * 4 blocks
    - Or 256 (threads/block) * 8 blocks, etc.
  - NOTE: actual # as resources allow
- Threads run concurrently
  - SM assigns/maintains thread id #s
  - SM manages/schedules thread execution

All numbers are for GTX680 (3.0 capability)

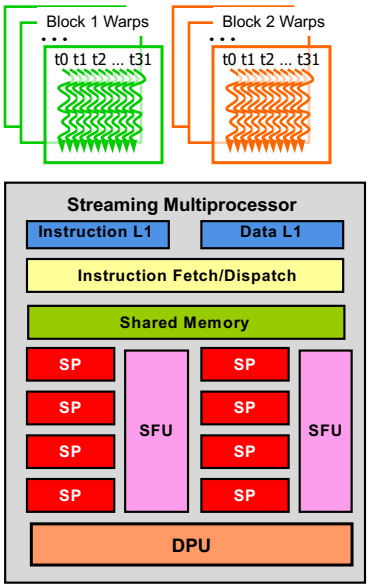


More info on limits at:
http://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications

22

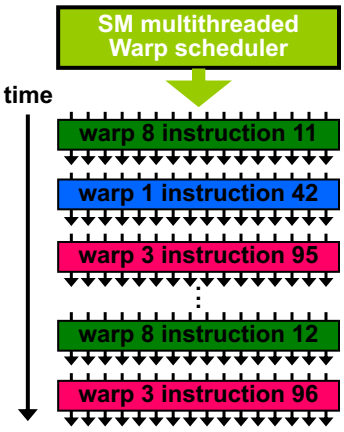
Thread Scheduling and Execution

- Each Thread Blocks is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warp: primitive scheduling unit
- All threads in warp:
  - same instruction
  - control flow causes some to become inactive
  - Up to **512M instructions** per kernel

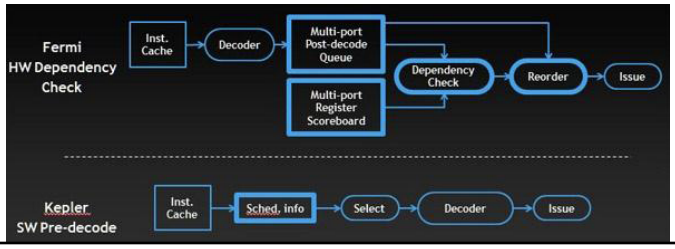


23

Warp Scheduling

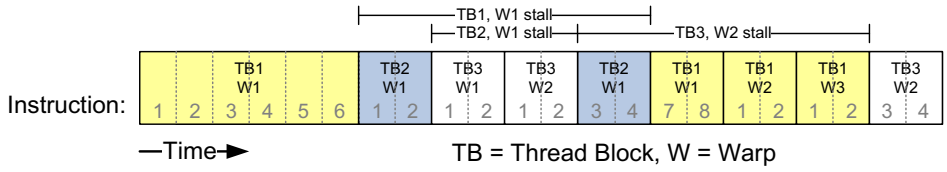


- SM hardware implements zero-overhead Warp scheduling
  - Instruction provides math pipeline latency (compiler knows)
  - Scheduler masks out ineligible warps
    - e.g., operands not ready
  - Select warp to schedule next based on a prioritized scheduling policy
  - Decode instruction
  - Issue instruction
  - All threads in a Warp execute the same instruction when selected



24

Warp Scheduling: Hiding Thread stalls



25

How many warps are there?

- If 3 blocks are assigned to an SM and each Block has 256 threads, how many Warps are there in an SM?

26

How many warps are there?

- 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
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution on an SM with 32 CUDA cores.
    - Each CUDA core, i.e., SP, has an integer arithmetic unit (ALU), a floating-point arithmetic unit (FPU), and an integer and FP multiplier/divider

27

Warp Scheduling Ramifications

- If one global memory access is needed for **every 4 instructions**, and SM has 8 SPs, then:
- **A minimal of 13 Warps are needed to fully tolerate a 200-cycle memory latency**
- Why?

28

Warp Scheduling Ramifications

- If one global memory access is needed for **every 4 instructions**, and SM has 8 SPs, then:
- **A minimal of 13 Warps are needed to fully tolerate a 200-cycle memory latency**
- Why?
  - Every 4 insts a thread stalls for memory
  - Every Warp occupies 4 cycles during which the same instruction executes on all its threads
    - Only 8 CUDA cores (SPs) for 32 threads in a Warp
  - Hence, every 16 cycles a thread stalls
    - i.e., 4 insts * 4 cycles/inst = 16 cycles
  - Must hide 200 cycles every 4 insts (every 16 cycles)
  - 200/16 =12.5 or at least 13 warps

29

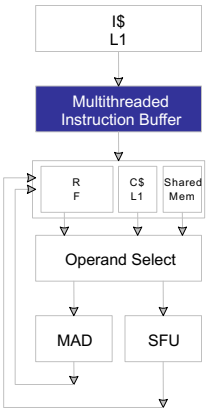
Granularity Considerations: Block & Thread limits per SM

- For Matrix Multiplication or any 2D-type of computation, should I use 8X8, 16X16 or 64X64 tiles?
  - For 8X8, we have 64 threads per Block.
    - Thread/SM limit = 2048→ up to 32 Blocks.
    - Blocks/SM limit = 16→ only 1024 threads will go into each SM
  - For 16X16, we have 256 threads per Block.
    - Thread/SM limit = 2048→ up to 8 Blocks.
    - Blocks/SM limit = 16 → full capacity unless other resource considerations overrule.
  - For 64x64, we have 4096 threads per Block.
    - Thread/block limit = 1024 → Not even one can fit into an SM.

30

SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction L1 cache
  - into any instruction buffer slot
- Issue one "ready-to-go" warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand ***scoreboarding*** used to prevent hazards
- Issue selection based on round-robin/age of warp: not public
- SM broadcasts the same instruction to 32 Threads of a Warp
- That's the theory → warp scheduling may use heuristics



31

Scoreboarding

- How to determine if a thread is ready to execute?
- A ***scoreboard*** is a table in hardware that tracks
  - instructions being fetched, issued, executed
  - resources they need (functional units and operands)
  - which instructions modify which registers
- Old concept from CDC 6600 (1960s) to separate memory and computation

32

Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes ready after the needed values are deposited
  - prevents hazards
  - cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
  - any thread can continue to issue instructions until scoreboarding prevents issue
  - allows Memory/Processor ops to proceed in shadow of Memory/Processor ops

33

Scoreboarding example

- Consider three separate instruction streams: warp1, warp3 and warp8

t=k

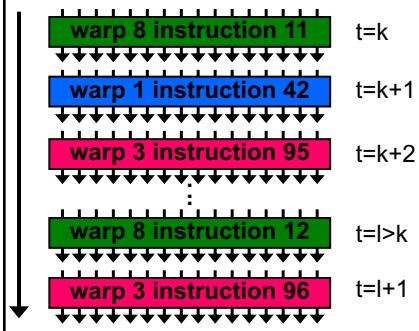
Warp	Current Instruction	Instruction State
Warp 1	42	Computing
Warp 3	95	Waiting
Warp 8	11	Operands ready to go
…		

Schedule at time k

34

Scoreboarding example

- Consider three separate instruction streams: warp1, warp3 and warp8



Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Waiting
Warp 8	11	Computing
…		

Schedule at time k+1

35