ME964
High Performance Computing for Engineering Applications

Execution Scheduling in CUDA

Revisiting Memory Issues in CUDA

February 17, 2011

"Computers are useless. They can only give you answers."
Pablo Picasso

Before We Get Started…

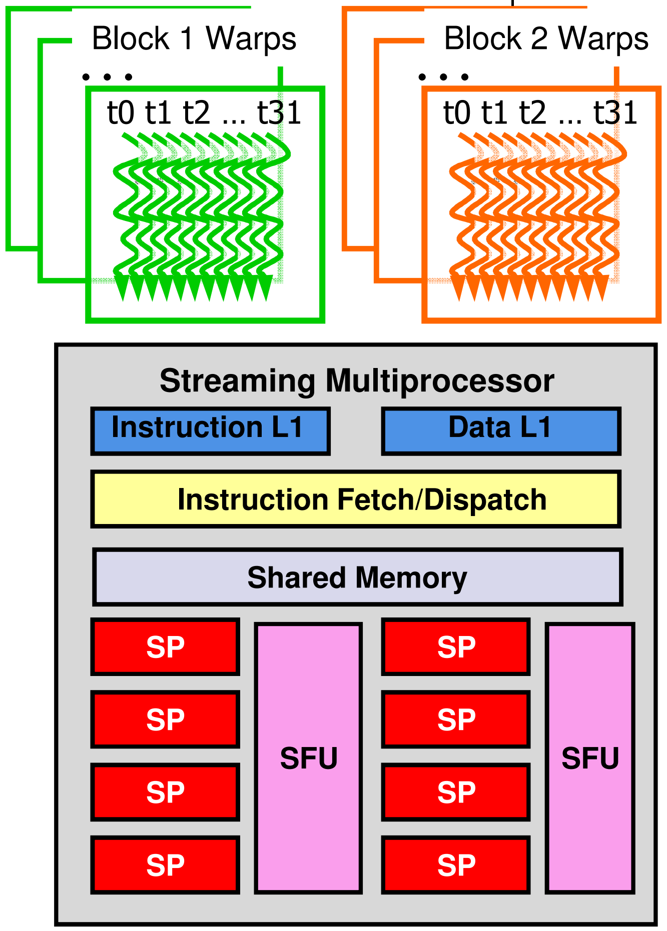
- Last time
  - Wrapped up tiled matrix-matrix multiplication using shared memory
    - Shared memory used to reduce some of the pain associated with global memory accesses
  - Discussed thread scheduling for execution on the GPU

- Today
  - Wrap up discussion about execution scheduling on the GPU
  - Discuss global memory access issues in CUDA

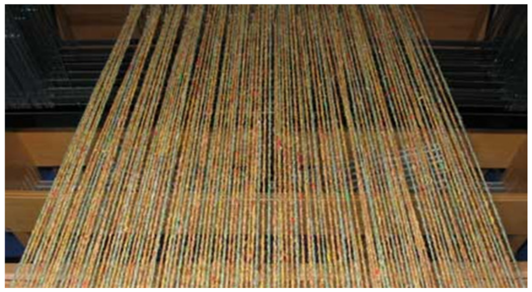
- Other issues
  - HW3 due tonight at 11:59 PM
    - Use Learn@UW drop-box to submit homework
    - Note that timing should be done as shown on slide 30 of the pdf posted for lecture 02/08
    - Building of CUDA code in Visual Studio: issues related to “include” of a file that hosts the kernel definition (either don’t include, or force linking)
  - HW4 was posted. Due date: 02/22, 11:59 PM
  - Please indicate your preference for midterm project on the forum

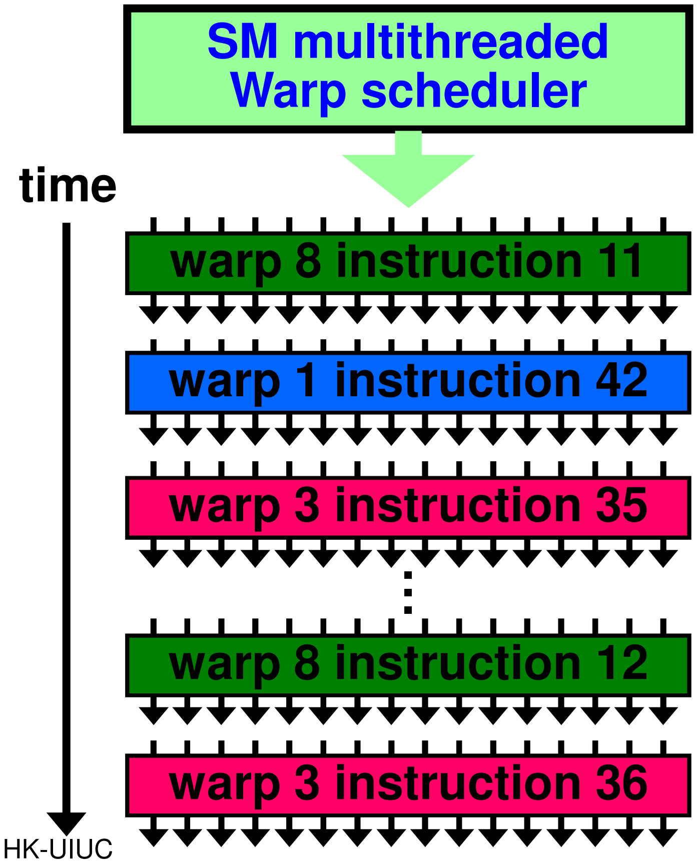
Thread Scheduling/Execution

- Each Thread Block is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model
- Warps are the basic scheduling units in SM
- Example (draws on figure at right):
- Assume 2 blocks are processed by an SM and each Block has 512 threads, how many Warps are managed by the SM?
  - Each Block is divided into 512/32 = 16 Warps
  - There are 16 * 2 = 32 Warps
  - At any point in time, only *one* of the 32 Warps will be selected for instruction fetch and execution.



SM Warp Scheduling



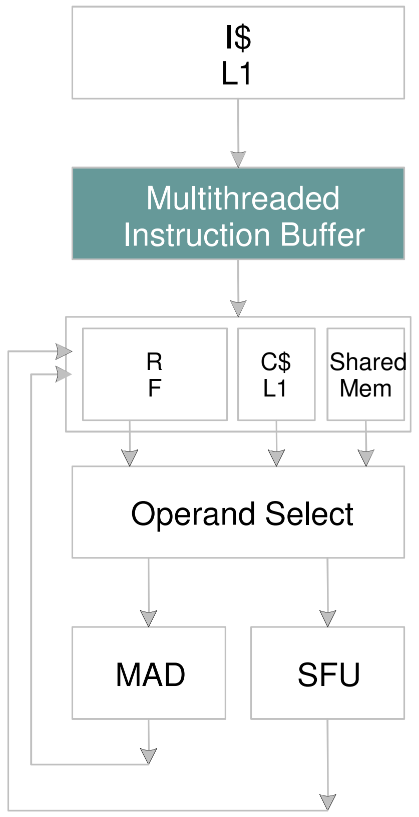


- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp on C1060
- How is this relevant?
  - Suppose your code has one global memory access every six simple instructions
  - Then, a minimum of 17 Warps are needed to fully tolerate 400-cycle memory latency:

$$400/(6 * 4)=16.6667 \Rightarrow 17 \text{ Warps}$$

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 per 4 cycles
  - From any warp - instruction buffer slot
  - Operand scoreboarding used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp



Scoreboarding

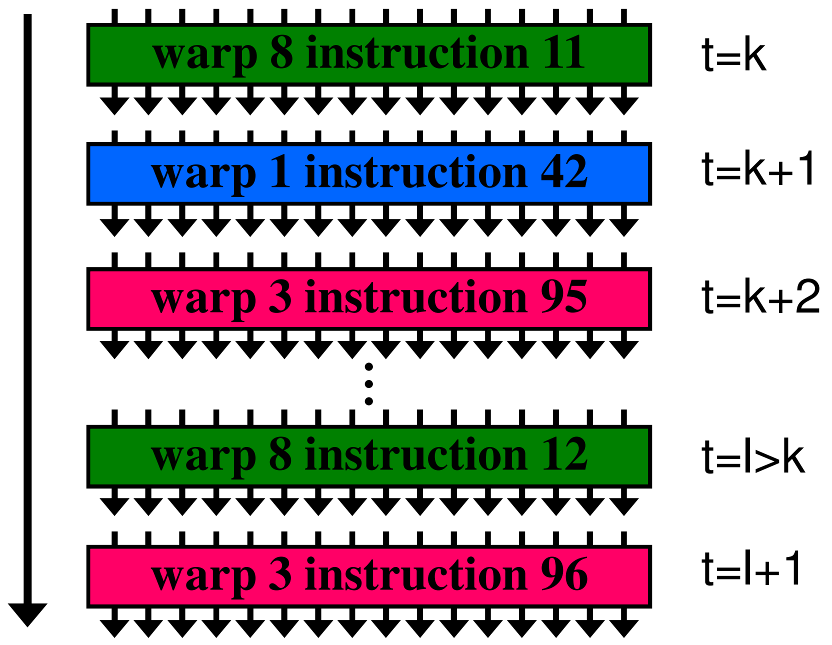
- Used to determine whether a thread is ready to execute

- A ***scoreboard*** is a table in hardware that tracks
  - Instructions being fetched, issued, executed
  - Resources (functional units and operands) needed by instructions
  - Which instructions modify which registers

- Old concept from CDC 6600 (1960s) to separate memory and computation

Scoreboarding from Example

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

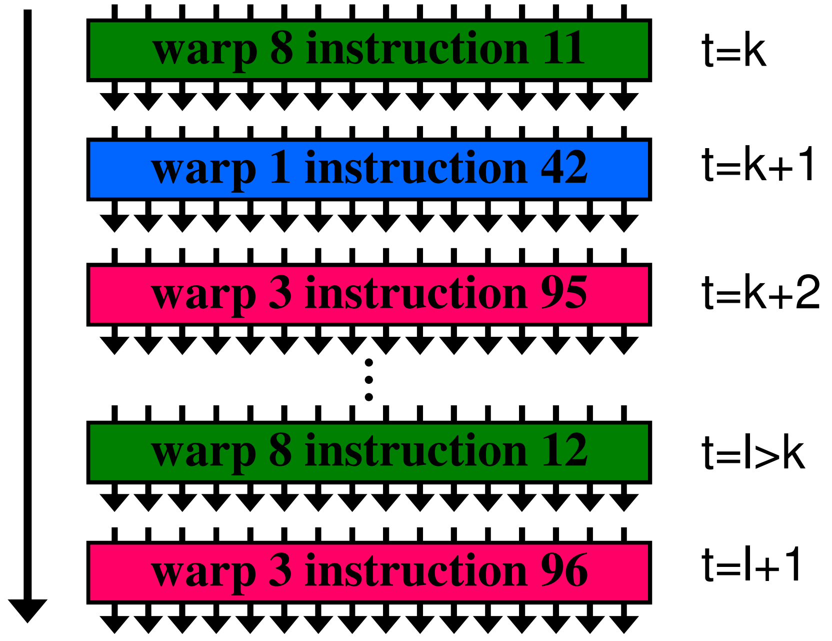


Warp	Current Instruction	Instruction State
Warp 1	42	Computing
Warp 3	95	Computing
Warp 8	11	Operands ready to go
…		

Schedule at time k ←

Scoreboarding from Example

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

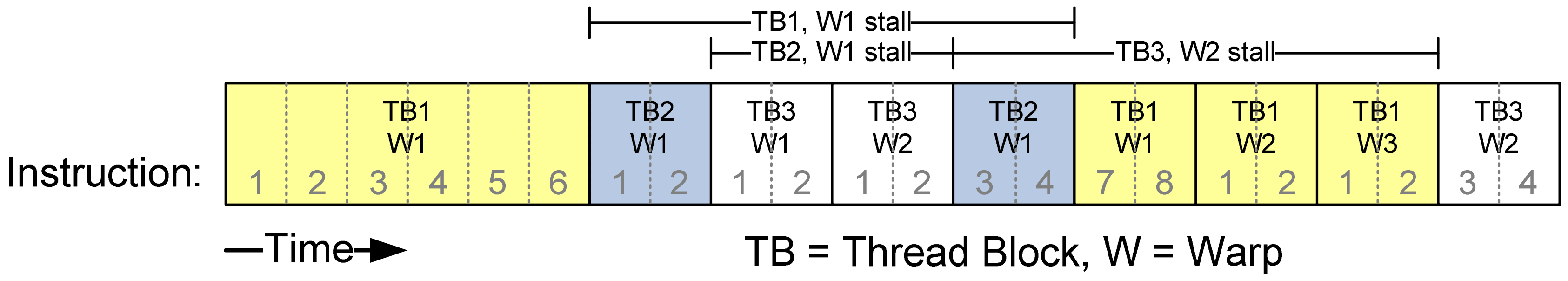


Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Computing
Warp 8	11	Computing
…		

Schedule at time k+1 ←

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



Granularity Considerations

[NOTE: Specific to Tesla C1060]

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?
  - For 8X8, we have 64 threads per Block. Since each Tesla C1060 SM can manage up to 1024 threads, it could take up to 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 unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. This is not an option anyway (we need less then 512 per block)
- NOTE: this type of thinking should be invoked for your target hardware (from where the need for auto-tuning software…)

ILP vs. TLP Example

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 20 registers
- Also, assume global loads have an associated overhead of 400 cycles
  - 3 Blocks can run on each SM

- If a Compiler can use two more registers to change the dependence pattern so that 8 independent instructions exist (instead of 4) for each global memory load
  - Only two blocks can now run on each SM
  - However, one only needs 400 cycles/(8 instructions *4 cycles/instruction) $\approx$ 13 Warps to tolerate the memory latency
  - Two Blocks have 16 Warps. The performance can be actually higher!

Summing It Up…

- When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity

- The threads of a block execute concurrently on one SM, and multiple blocks (up to 8) can execute concurrently on one SM

- When a thread block finishes, a new block is launched on the vacated SM

A Word on HTT

[Detour: slide 1/2]

- The traditional host processor (CPU) may stall due to a cache miss, branch misprediction, or data dependency
- Hyper-threading Technology (HTT): an Intel-proprietary technology used to improve parallelization of computations
- For each processor core that is physically present, the operating system addresses two virtual processors, and shares the workload between them when possible.
- HT works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources.
  - This allows a hyper-threading processor to appear as two "logical" processors to the host operating system, allowing the operating system to schedule two threads or processes simultaneously.
- Similar to the use of multiple warps on the GPU to hide latency
  - The GPU has an edge, since it can handle simultaneously up to 32 warps (on Tesla C1060)

SSE
[Detour: slide 2/2]

- Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD's 3DNow!
  - SSE contains 70 new instructions

- Example
  - Old school, adding two vectors. Corresponds to four x86 FADD instructions in the object code

```
vec_res.x = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```

  - SSE pseudocode: a single 128 bit 'packed-add' instruction can replace the four scalar addition instructions

```
movaps xmm0,address-of-v1 ;xmm0=v1.w | v1.z | v1.y | v1.x
addps xmm0,address-of-v2 ;xmm0=v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x movaps address-of-vec_res,xmm0
```

Finished Discussion Execution Scheduling

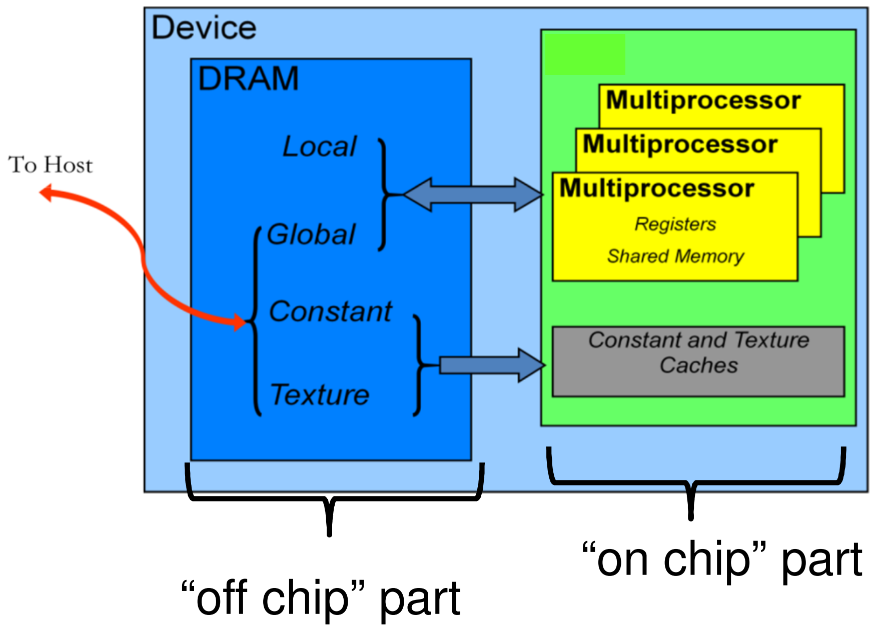
Revisit Memory Accessing Topic

Important Point

- In GPU computing, memory operations are perhaps most relevant in determining the overall efficiency (performance) of your code

The Device Memory Ecosystem

[Quick Review]



Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached only on devices of compute capability 2.x.

- The significant change from 1.x to 2.x device capability was the caching of the local and global memory accesses

Global Memory and Memory Bandwidth

- These change from card to card. On Tesla C1060:
  - 4 GB in GDDR3 RAM
  - Memory clock speed: 800 MHz
  - Memory interface: 512 bits
  - Peak Bandwidth: $800*10^6*(512/8)*2 = 102.4$ GB/s

- Effective bandwidth of your application: very important to gauge
  - Formula, effective bandwidth (B_r - bytes read, B_w - bytes written)

$$\text{Effective bandwidth} = ((B_r + B_w)/10^9)/\text{time}$$

  - Example: kernel copies a 2048×2048 matrix from global memory, then copies matrix back to global memory. Does it in a certain amount of "time" [seconds]

$$\text{Effective bandwidth} = ((2048^2 \cdot 4 \cdot 2)/10^9)/\text{time}$$

  - 4 above comes from four bytes per float, 2 from the fact that the matrix is both read from and written to the global memory. The 10^9 used to get an answer in GB/s.

Global Memory

- The global memory space is not cached on Tesla C1060 (1.3)
  - Very important to follow right access pattern to get maximum memory throughput

- Two aspects of global memory access are relevant when fetching data into shared memory and/or registers

  - The layout of the access to global memory (the pattern of the access)

  - The size/alignment of the data you try to fetch from global memory

The Memory Access Layout

- The important concept here is that of “coalesced memory access”
- The basic idea:
  - Suppose each thread in a half-warp accesses a global memory address for a load operation at some point in the execution of the kernel
  - These threads can access global memory data that is either (a) neatly grouped or (b) scattered all over the place
  - Case (a) is called a “coalesced memory access”; if you end up with (b) this will adversely impact the overall program performance
  - Analogy
    - Can send one semi truck on six different trips to bring back each time a bundle of wood
    - Alternatively, can send semi truck to one place and get it back fully loaded with wood

Global Memory Access Compute Capability 1.3

- A global memory request for a warp is split in two memory requests, one for each half-warp
- The following 5-stage protocol is used to determine the memory transactions necessary to service all threads in a half-warp

- Stage 1: Find the memory segment that contains the address requested by the lowest numbered active thread. The memory segment size depends on the size of the words accessed by the threads:
  - 32 bytes for 1-byte words,
  - 64 bytes for 2-byte words,
  - 128 bytes for 4-, 8- and 16-byte words.

- Stage 2: Find all other active threads whose requested address lies in the same segment

- Stage 3: Reduce the transaction size, if possible:
  - If the transaction size is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes;
  - If the transaction size is 64 bytes (originally or after reduction from 128 bytes) and only the lower or upper half is used, reduce the transaction size to 32 bytes.

- Stage 4: Carry out the transaction and mark the serviced threads as inactive.

- Stage 5: Repeat until all threads in the half-warp are serviced.