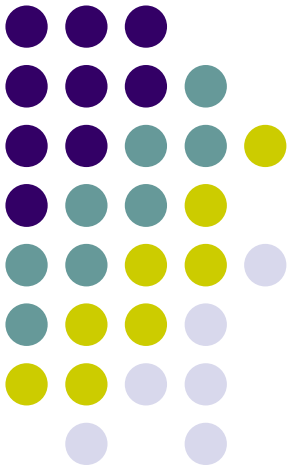
ME964
High Performance Computing for Engineering Applications

Memory Issues in CUDA

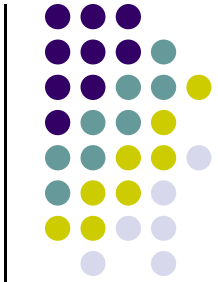
Execution Scheduling in CUDA

February 23, 2012



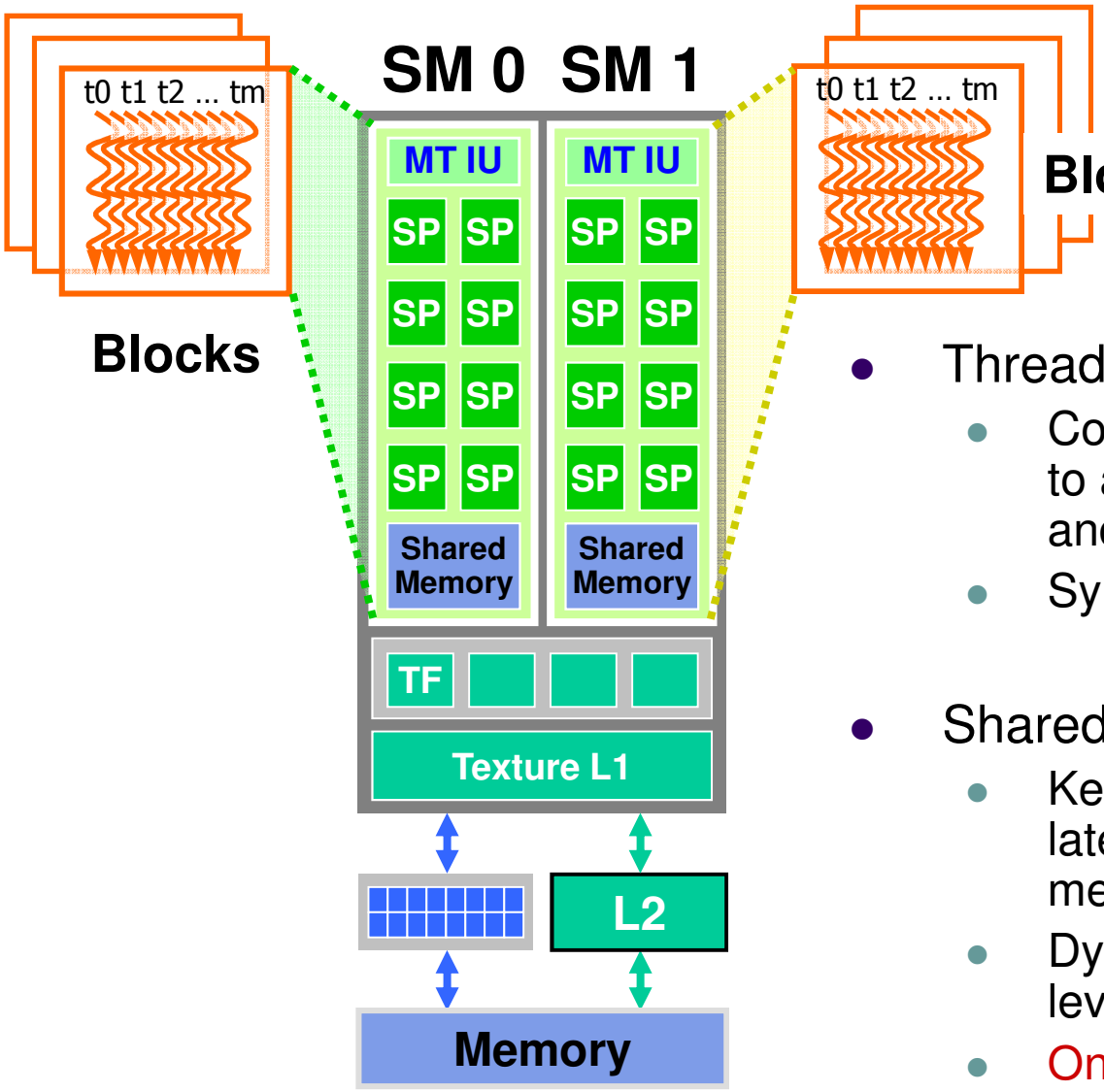
"Computers are useless. They can only give you answers."
Pablo Picasso

Before We Get Started…



- Last time
  - Discussed about memory hierarchy on NVIDIA GPUs
  - Discussed: device memory, registers, local memory, and shared memory
  - Wrapped up tiled matrix-matrix multiplication using shared memory
    - Shared memory used to reduce some of the pain associated with global memory accesses
- Today
  - Discuss execution scheduling on the GPU
  - Discuss global memory access issues in CUDA
- Other issues
  - HW4 due tonight at 11:59 PM
  - HW5 posted today or tomorrow
    - It is more challenging, take a look at it after Tu
    - We'll cover debugging CUDA code on Tu

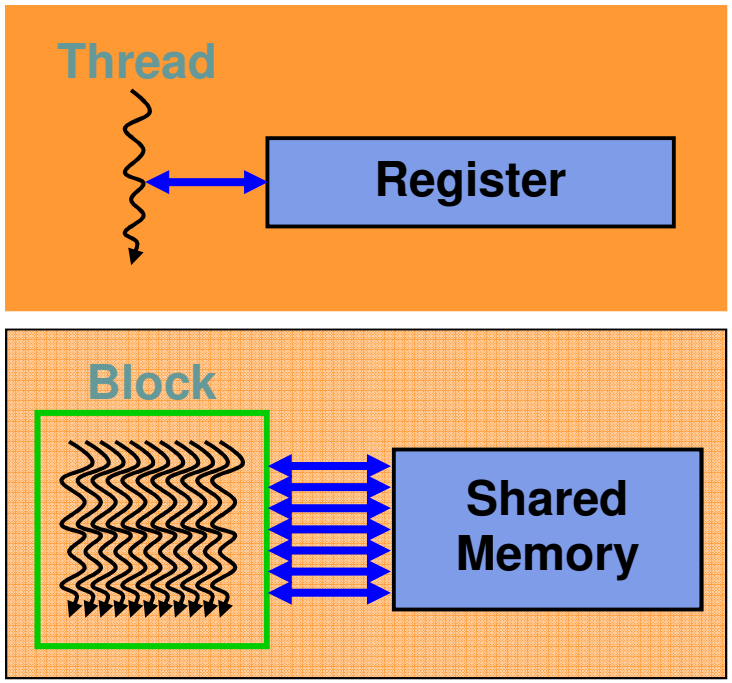
The Shared Memory in the Context of the TPC/SM Architecture [NVIDIA G80]



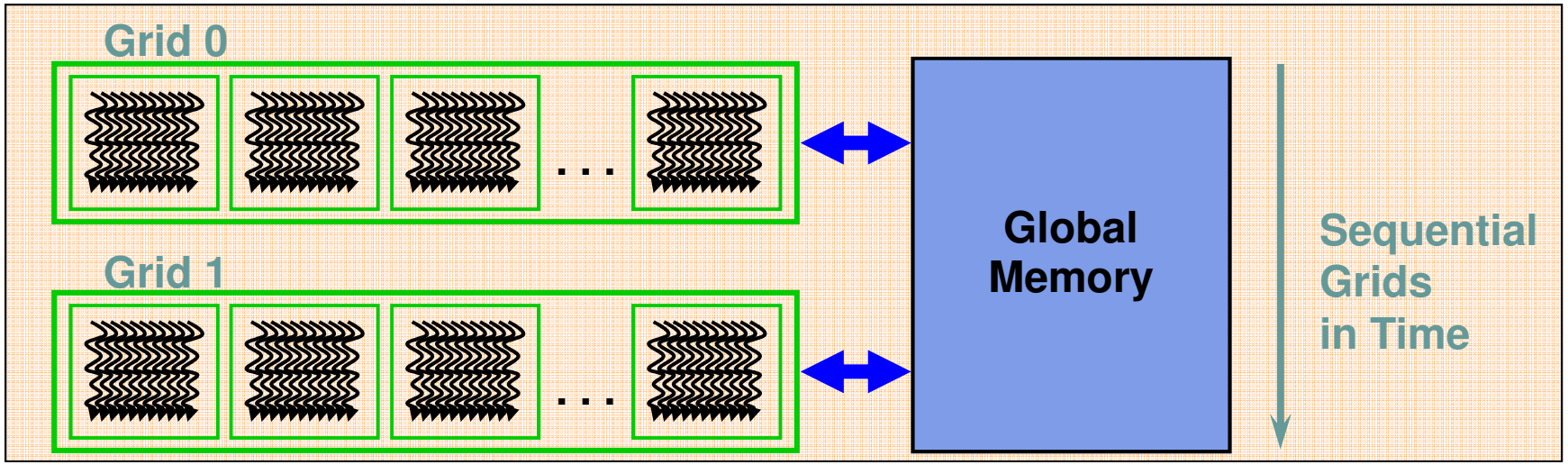
Picture shows one TPC on NVIDIA G80 card

- Threads in a Block:
  - Cooperate through data accessible to all of them both in Global Memory and Shared Memory
  - Synchronize at barrier instruction
- Shared Memory, traits
  - Keeps data close to processor (low latency), thus reduces trips to global memory
  - Dynamically allocated at the SM level to each Block
  - One of the limiting resources

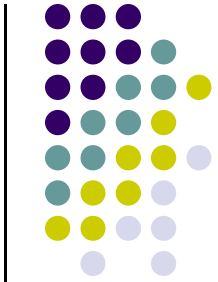
The Three Most Important Parallel Memory Spaces



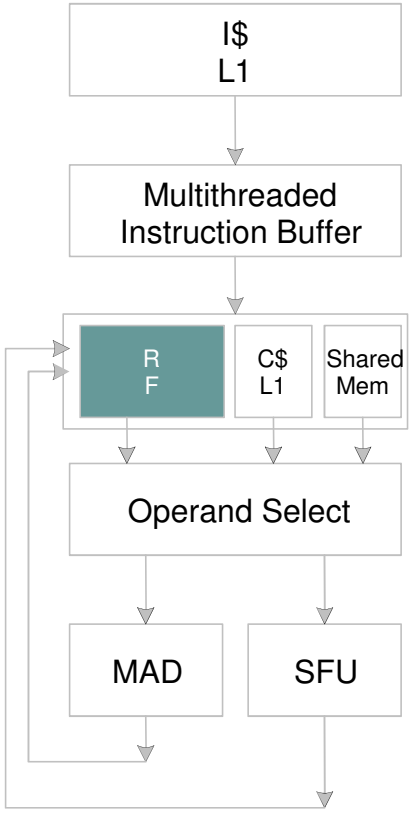
- Register: per-thread basis
  - Private per thread
  - Can spill into local memory (potential performance hit if not cached)
- Shared Memory: per-block basis
  - Shared by threads of the same block
  - Used for: Inter-thread communication
- Global Memory: per-application basis
  - Available for use to all threads
  - Used for: Inter-thread communication
  - Also used for inter-grid communication



SM Register File (RF) [Tesla C1060]



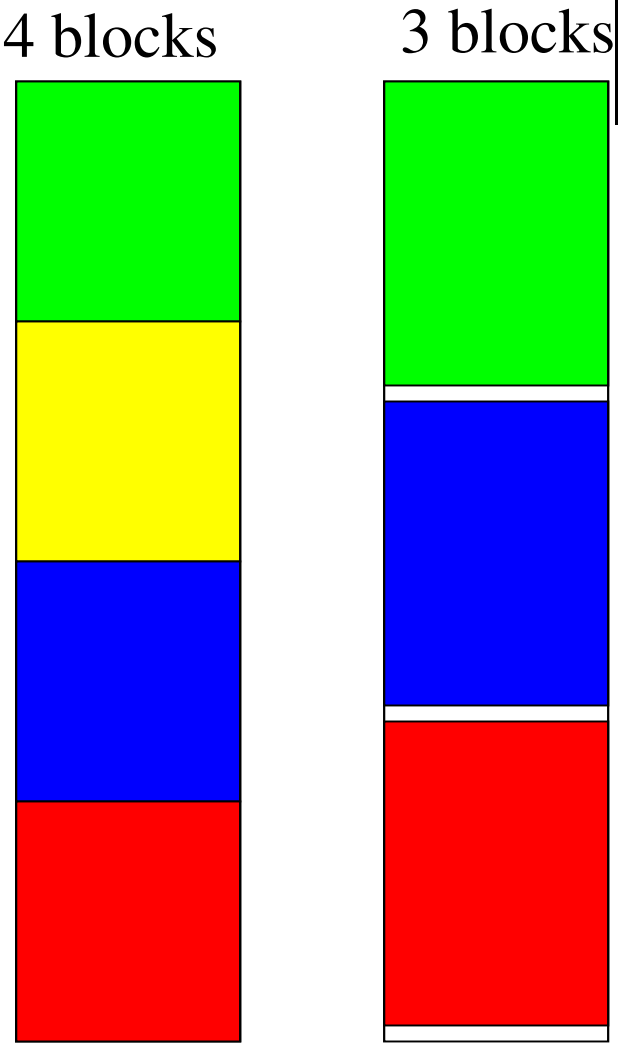
- Register File (RF)
  - 64 KB (16,384 four byte words)
  - Provides 4 operands/clock cycle
  - Note: typical CPU has less than 20 registers per core
- TEX pipe can also read/write RF
  - 3 SMs share 1 TEX
- Global Memory Load/Store pipe can also read/write RF



See Appendix F of the CUDA Programming Guide for amount of register memory available on different compute capabilities

Programmer View of Register File

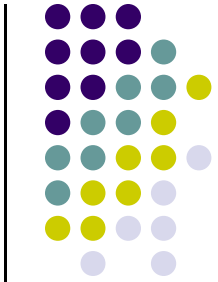
- Number of **32 bit** registers in one SM:
  - 8K registers in each SM in G80
  - 16K on Tesla C1060
  - 32K on Tesla C2050
- Size of Register File dependent on your compute capability, not part of CUDA
- Registers are dynamically partitioned across all Blocks assigned to the SM
- Once assigned to a Block, these registers are NOT accessible by threads in other Blocks
- A thread in a Block can only access registers assigned to itself (out of the allocation of the Block it belongs to)



Possible per-block partitioning scenarios of the RF available on the SM

Matrix Multiplication Example

[Tesla C1060]

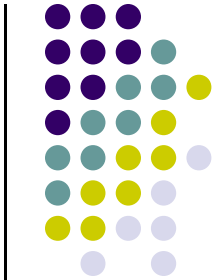


- If each Block has 16X16 threads and each thread uses 20 registers, how many threads can run on each SM?
  - Each Block requires 20*256 = 5120 registers
  - 16,384 = **3** * 5120 + change
  - So, three blocks can run on an SM as far as registers are concerned

- What if each thread increases the use of registers from 20 to 22?
  - Each Block now requires 22*256 = 5632 registers
  - 16,384 < 16896= 5632 *3
  - Only two Blocks can run on an SM, about 33% **reduction of parallelism**!!!

- Example shows why understanding the underlying hardware is essential if you want to squeeze performance out of parallelism
  - One way to find out how many registers you use per thread is to invoke the compile flag **`-ptax-options=-v`** when you compile with **`nvcc`**

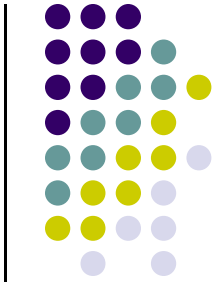
More on Dynamic Partitioning



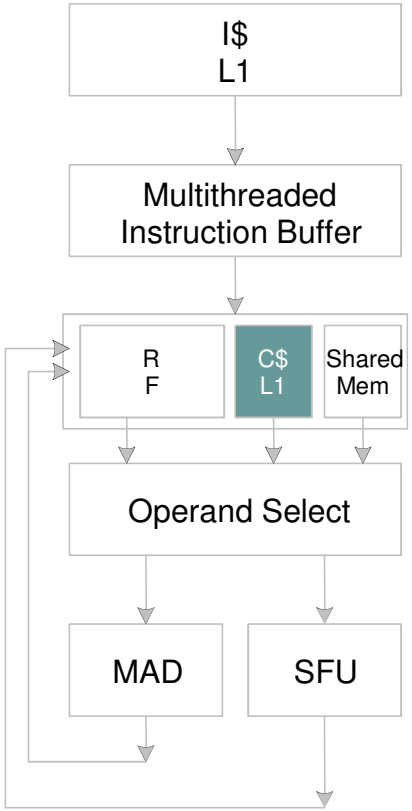
- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each, or run a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between instruction-level parallelism and thread level parallelism
    - TLP: many threads are run
    - ILP: few threads are run, but for each thread several instructions can be executed simultaneously

See Volkov's talk, "Better performance at lower occupancy": http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

Constant Memory



- This comes handy when all threads use the same *constant* value in their computation
  - Example: π, some spring force constant, e=2.7173, etc.
- Constants are stored in DRAM but cached on chip
  - There is a limited amount of L1 cache per SM
  - Might run into slow access if for example have a large number of constants used to compute some complicated formula (might overflow the cache…)
- A constant value can be broadcast to all threads in a warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block
  - When all threads in a warp read the same constant memory address this is as fast as a register



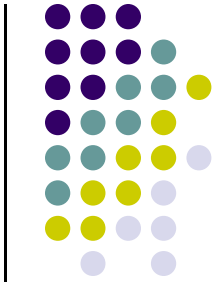
Example, Use of Constant Memory

[For compute capability 2.0 (GTX480, C2050) – due to use of "printf"]

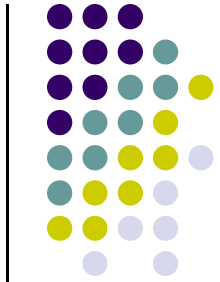
```
#include <stdio.h>

// Declare the constant device variable outside the body of any function
__device__ __constant__ float dansPI;


// Some dummy function that uses the constant variable
__global__ void myExample() {
        float circum = 2.f*dansPI*threadIdx.x;
        printf("Hello thread %d, Circ=%5.2f\n", threadIdx.x, circum) ;
}


int main(int argc, char **argv) {
        float somePI = 3.141579f;

        cudaMemcpyToSymbol(dansPI, &somePI, sizeof(float));
        myExample<<<1, 16>>>();
        cudaThreadSynchronize();

        return 0;
}
```

```
Hello thread 0, Circ= 0.00
Hello thread 1, Circ= 6.28
Hello thread 2, Circ=12.57
Hello thread 3, Circ=18.85
Hello thread 4, Circ=25.13
Hello thread 5, Circ=31.42
Hello thread 6, Circ=37.70
Hello thread 7, Circ=43.98
Hello thread 8, Circ=50.27
Hello thread 9, Circ=56.55
Hello thread 10, Circ=62.83
Hello thread 11, Circ=69.11
Hello thread 12, Circ=75.40
Hello thread 13, Circ=81.68
Hello thread 14, Circ=87.96
Hello thread 15, Circ=94.25
```

Memory Issues Not Addressed Yet…



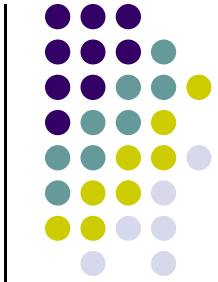
- Not all global memory accesses are equivalent
  - How can you optimize memory accesses?
  - Very relevant question
- Not all shared memory accesses are equivalent
  - How can optimize shared memory accesses?
  - Moderately relevant questions
- To do justice to these topics we'll need to talk first about scheduling threads for execution
  - Coming up next…



Execution Scheduling Issues

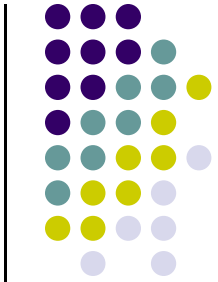
[NVIDIA cards specific]

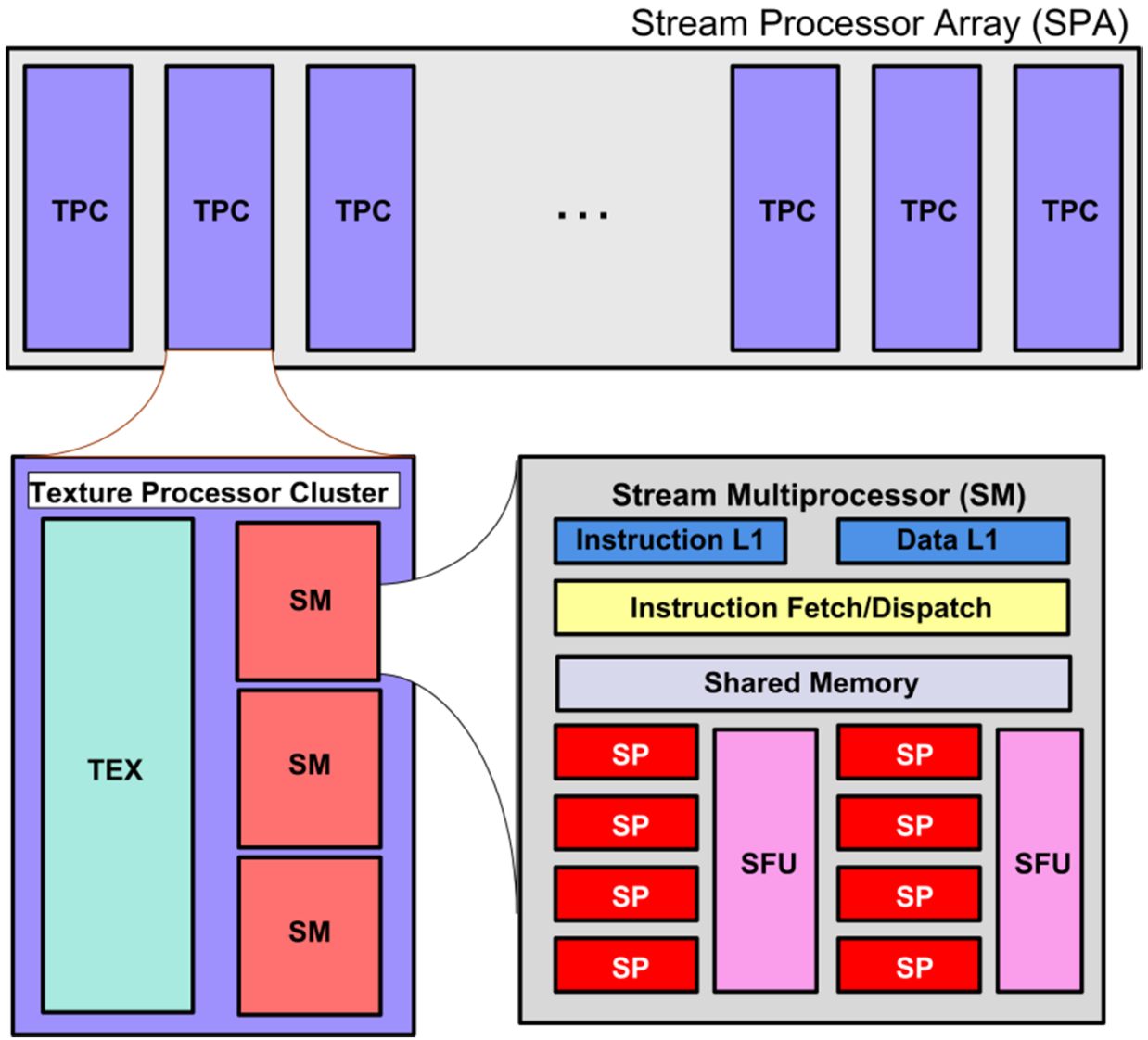
Thread Execution Scheduling



- Topic we are about to discuss:
  - You launch on the device many blocks, each containing many threads
  - Several blocks can get executed simultaneously on one SM (8 SPs). How is this possible?

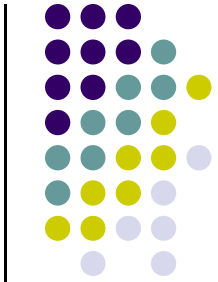
GeForce-8 Series HW Overview





CUDA Thread Block

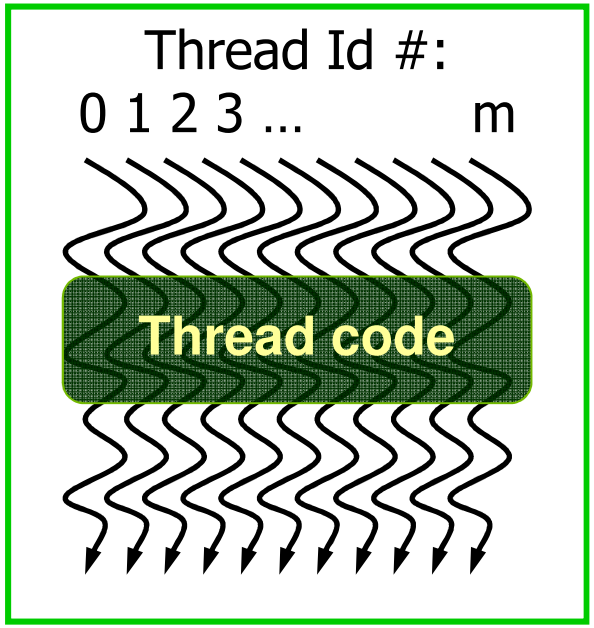
[We already know this…]



- In relation to a Block, the programmer decides:
  - Block size: from 1 to 1024 threads
  - Block dimension (shape): 1D, 2D, or 3D
  - Higher order configurations projected to 1D representation

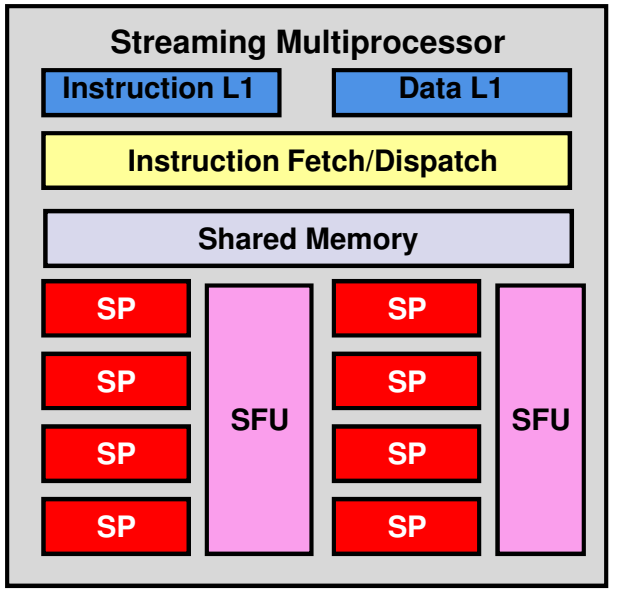
- Threads have thread idx numbers within Block
- Threads within Block share data and may synchronize while each is doing its work
- Thread program uses thread idx to select work and address shared data
- Beyond the concept of thread idx we brought into the picture the concept of thread id and how to compute a thread id based on the thread index (the 1D projection idea)

CUDA Thread Block



Thread Scheduling/Execution

- Each Thread Block divided in 32-thread "Warps"
  - This is an implementation decision, not part of the CUDA programming model

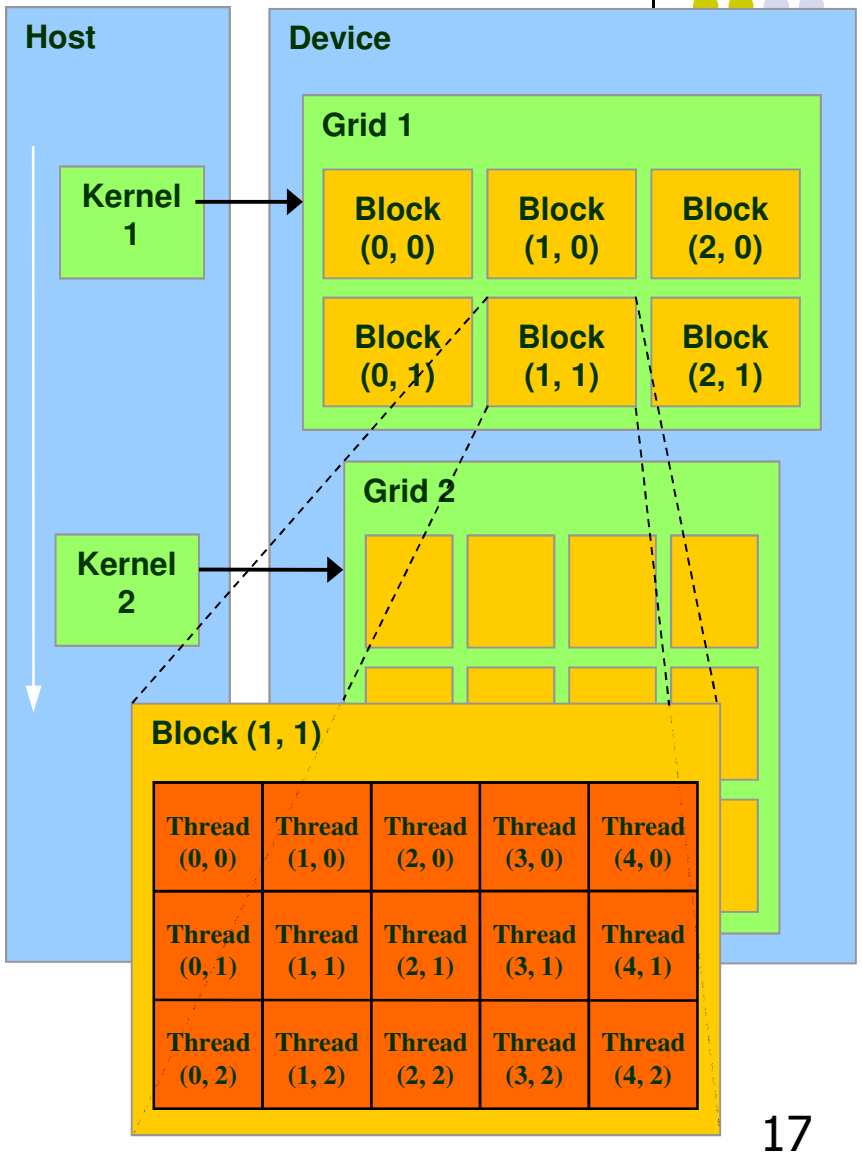


- Warps are the basic scheduling unit in SM

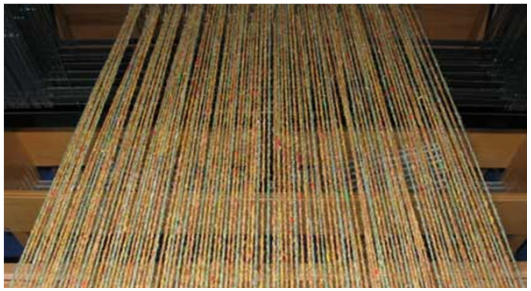
- If 3 blocks are processed by an SM and each Block has 256 threads, how many Warps are managed by the SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.

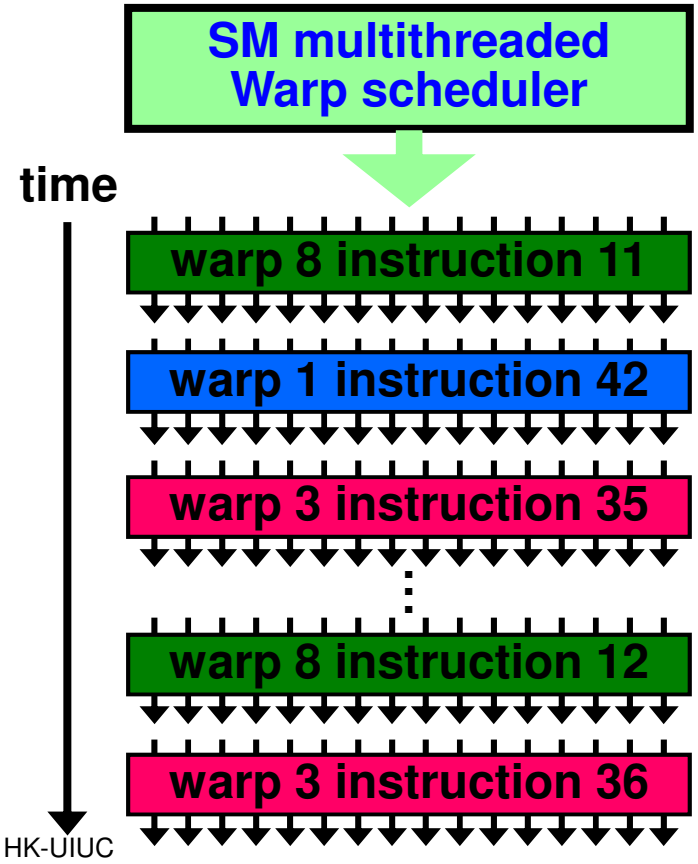
Scheduling on the Hardware

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SMs
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
- SM schedules and executes Warps that are ready to run
- As Thread Blocks complete kernel execution, resources are freed
  - SPA can launch next Block[s] in line
- NOTE: Two levels of scheduling:
  - For running [desirably] a large number of blocks on a small number of SMs (30/16/14/etc.)
  - For running up to 24 (or 32, on Tesla C1060) warps of threads on the 8 SPs available on each SM



SM Warp Scheduling

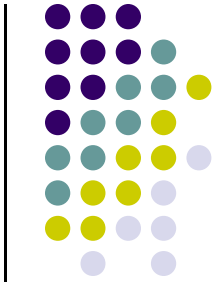




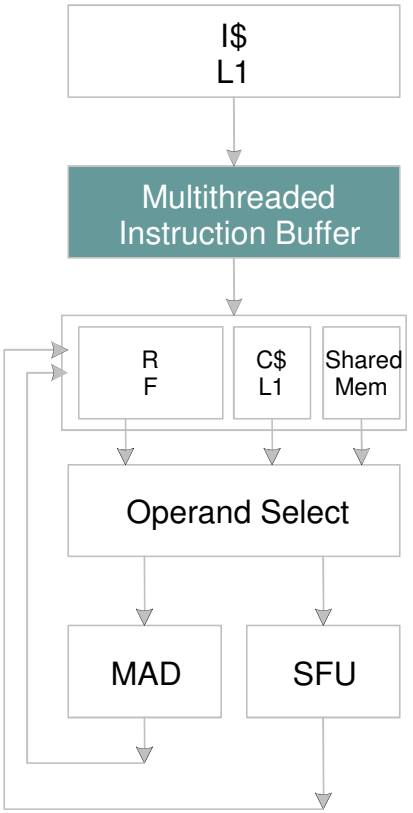
- SM hardware implements almost zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when 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:

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

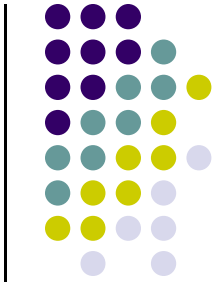
SM Instruction Buffer – Warp Scheduling



- 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

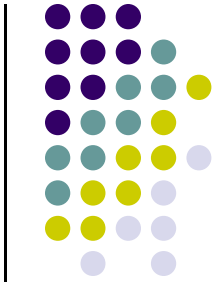


- Used to determine whether a thread is ready to execute

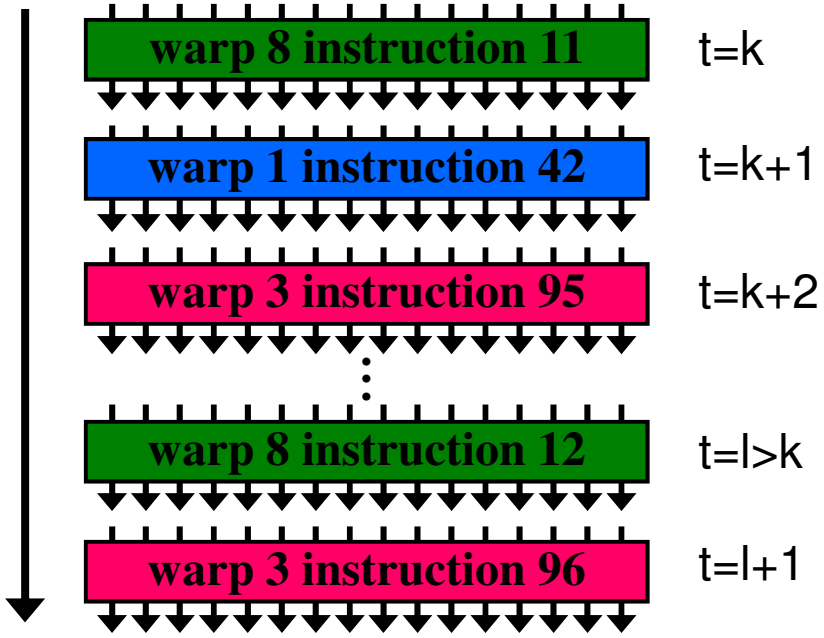
- A ***scoreboard*** is a table in hardware that tracks
  - Instructions being fetched, issued, executed
  - Resources (functional units and operands) needed by instructions
  - Which instructions modify which registers

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

Scoreboarding from Example



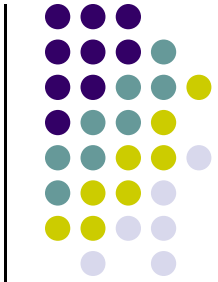
- Consider three separate instruction streams: warp1, warp3 and warp8



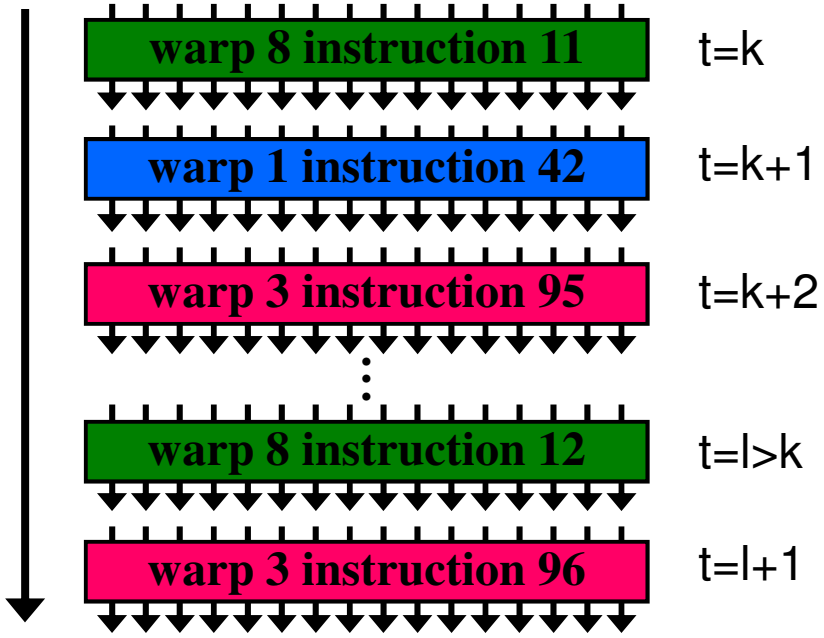
Warp	Current Instruction	Instruction State
Warp 1	42	Computing
Warp 3	95	Computing
Warp 8	11	Operands ready to go
…		

Schedule at time k ←

Scoreboarding from Example



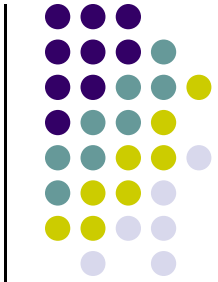
- Consider three separate instruction streams: warp1, warp3 and warp8



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

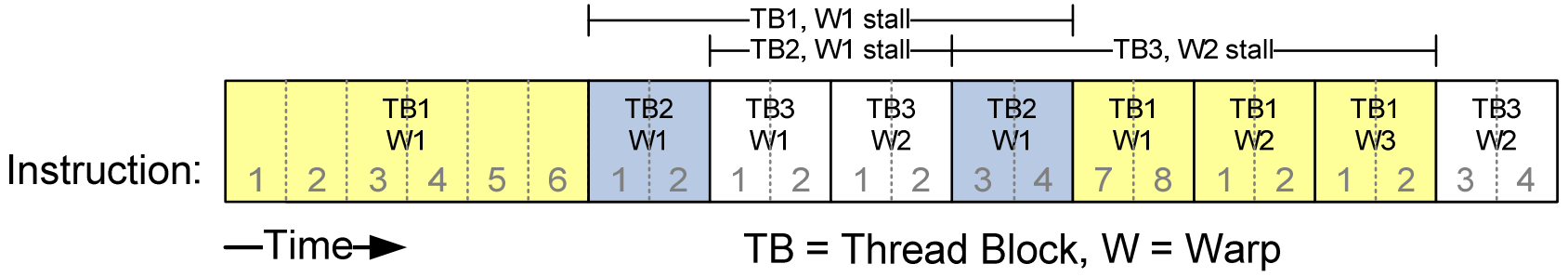
Schedule at time k+1 ←

Scoreboarding



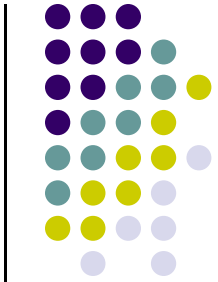
- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes “ready” after the needed values are deposited
  - Prevents hazards
  - Cleared instructions are eligible for issue

- Decoupled Memory/Processor pipelines
  - Any thread can continue to issue instructions until scoreboarding prevents issue



Granularity Considerations

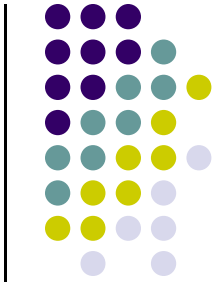
[NOTE: Specific to Tesla C1060]



- 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.
    - Next you need to see how much shared memory and how many registers get used in order to understand whether you can actually have four blocks per SM
- NOTE: this type of thinking should be invoked for your target hardware (from where the need for auto-tuning software…)

ILP vs. TLP Example

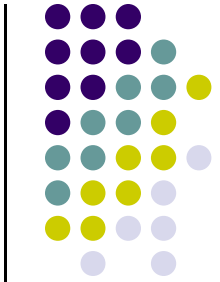
[C1060 specific]



- 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…



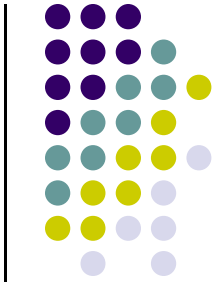
- When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity

- 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

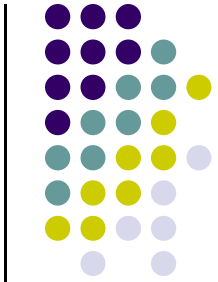
[Detour: slide 1/2]



- 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]



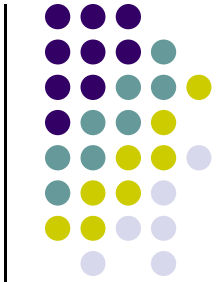
- Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors in response to AMD's 3DNow!
  - SSE contains 70 new instructions

- 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