Final Review

UNIVERSITY OF CALIFORNIA
UCRIVERSIDE

Logistics

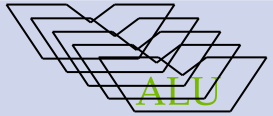
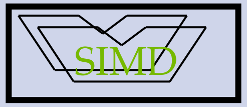
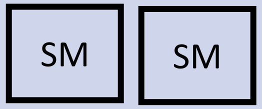
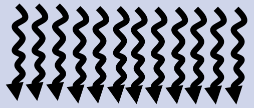
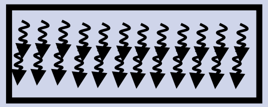
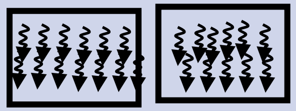
- Start Midterm next class!
- Same style as Midterm, 5 questions
- Will be cumulative, so I expect your answers to incorporate all that you have learned
- Due on Wednesday June 10th
- Lab 4 Due Friday June 12th
- Quiz 4 Due Monday June 5th
- I will have your quiz 3 graded by tomorrow evening
- Quiz 4 - take your 4 lowest scores among the past 3 quizzes. Write what your thought process was for your answer, describe the mistakes you make and correct them

Computer Architecture

- Threads and processes
  - What they contain and how they relate in hardware and software
- Cache hierarchy
  - Understand the memory gap
  - SW leads to HW design
- Principles of spacial and temporal locality
  - How to write code to apply them
  - HW leads to SW design
- Specialization towards parallel processing
- These are foundational concepts questions will not be explicitly mentioning them but will have implied understanding

GPU Architecture

- Warps contain 32 threads and execute on a SIMD unit
- SM Cores contain multiple SIMD Units run entire Thread Blocks
- GPU Contains multiple SMs

	Scalar	Vector	Core	Card
Hardware				
	ALU Unit	SIMD Unit	SM	GPU
Threads				
	Thread	Warp	Thread Block	Block Grid
Memory	Register File		L1 Cache	L2 / Memory
Address Space	Local per thread		Shared Memory	Global

Midterm Question 1

You are the head architect for a new open source GPU project. In your first design meeting, layout how YOU believe the architecture should be designed. As it is an open sourced project programmability and ease of use are important considerations. Explain why you designed it in that way. Defend your design with any reasoning you feel is valid along with a use case

- Goal is to understand the connection between why GPUs are designed the way they are and the motivation behind them
- People gave motivations such as it need to be; data parallel, throughput orientated, easy to program, thousands of threads, etc..
- But did not provide how the architecture satisfies those requirements
- Some gave hardware design; alu -> simd -> sm, reg files, memory system, etc…
- But did not provide any reasoning for why they decided to design in this way
- Answer needed to link the two together with solid reasoning

GPU Architecture

- Hardware constraints
- Limit to number of threads and thread block per SM

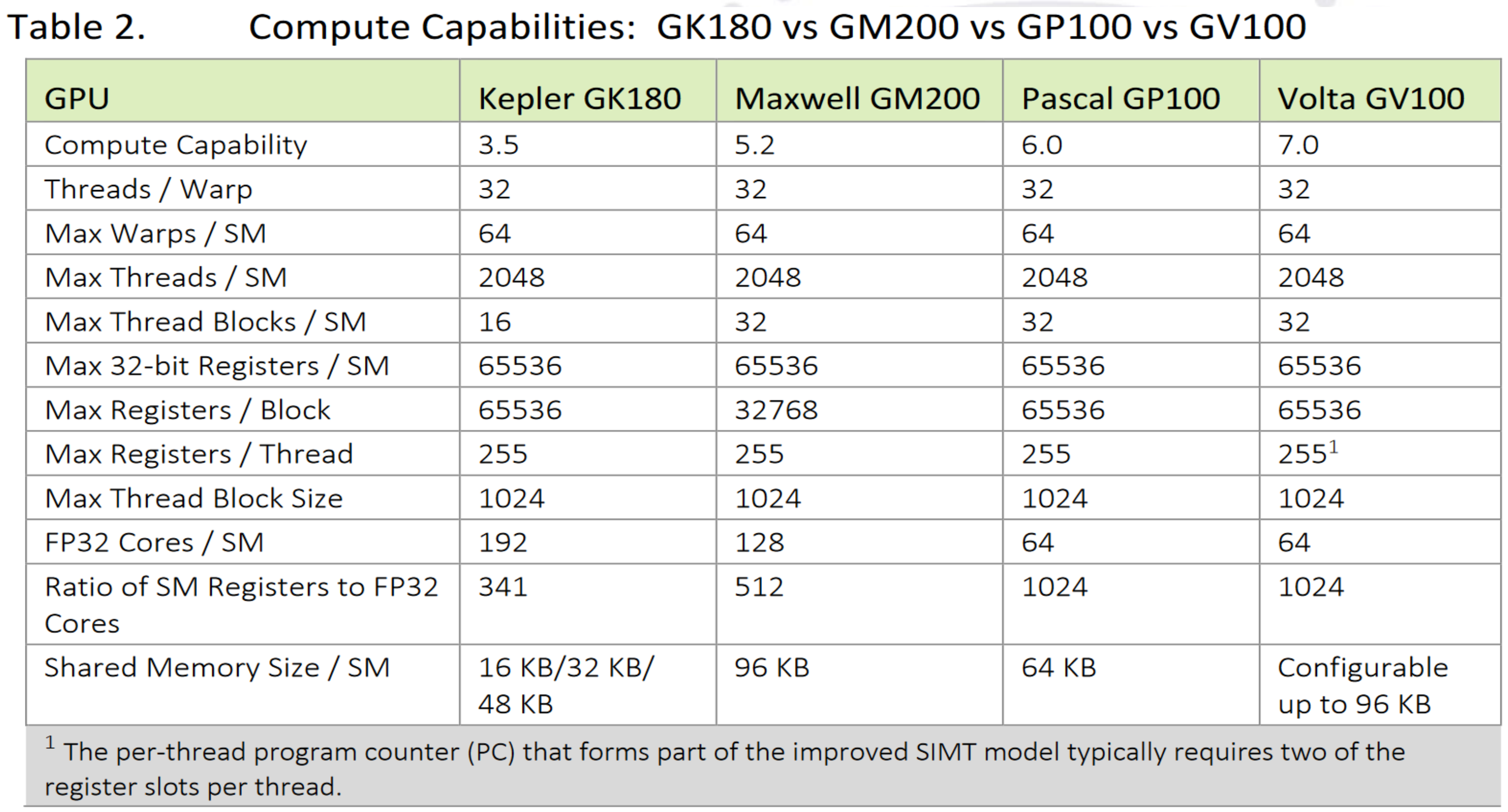


Table 2. Compute Capabilities: GK180 vs GM200 vs GP100 vs GV100

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255[1]
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

[1] The per-thread program counter (PC) that forms part of the improved SIMT model typically requires two of the register slots per thread.

GPU Architecture

- Hardware constraints examples
- An SM is fully occupied if it is running the maximum number of threads
- 2 blocks with 1024 threads – Fully occupied
- 32 blocks with 32 threads – not fully occupied
- Typically you want the number of threads per block to be divisible by 32 and have at least 64 threads
- Multidimensional blocks get linearlized
- Block size of (16,16,4) = 16*16*4 =1024 threads

Max warps / SM	64
Max Threads / SM	2048
Max Thread Blocks / SM	32
Max Thread Block Size	1024

Midterm Question 2

A member of your group suggests a Nvidia style GPU. Cost is a concern, so the total number of SIMD units is limited 32. You are presented with three options, 32 SMs with 1 SIMD unit each, 16 SMs with 2 SIMD units, or 8 SMs with 4 SIMD units. Evaluate each option, giving pros and cons for each. Justify your decision with any reasoning you feel is valid.

- All option have the same theoretical computation because, they all have a total of 32 SIMD units
- So one option is not necessarily faster than any other one
- The difference comes from how you program it
- Some gave arguments for more SIMD per SM to utilize shared memory more, better tiling perhaps
- Less SIMD units forces smaller thread block, so synchronizing within a thread block is less overhead
- It really depends on how you think the hardware will be use, some use cases fit better on other hardware

GPU System Architecture

- GPU within the computer system architecture
- Connected Over PCIe
- Memory copied from Host Memory to Device Memory
- Different ways of allocating and coping memory
- Paged, Pinned Memory, Unified Memory
    - cudaMalloc, cudaHostMalloc, cudaMallocManaged

Midterm Question 3

Later in the project, someone suggests integrating a couple of CPU cores within the same chip as your GPU, instead of the typical connection over PCIe. Do you think this is a good idea? How would this affect programmability? Or the design of the GPU cores? What are the drawbacks? Justify your decision with any reasoning you feel is valid

- A lot of confusion that an integrated chip would be programmed differently than a discrete system
- This is not the case
- Complexity of a system doesn’t necessarily mean more complexity to program
- Main difference is that there is no pcie to connect so CPU and GPU share memory systems; memory and caches.
- No need to copy any data
- The GPU cores can be the same, but some drawback are reduced space for GPU cores

CUDA Programming

- Allocate, Copy to Device, Launch, Copy to Host
  - Cudamemcopy(dest,src,size,direction)
  - globalFunction<<<gridDim,BlockDim>>>(args)
- Allocate and copy data only pointed to by pointers
- Block and Grid size are 3 Dimensional
- Threads are assigned a Thread id and Block id in each dimension
  - Determine proper block and grid size for any input size
  - How to assign data with thread and block ids e.g...
  - Row = blockIdx.y*blockDim.y + threadIdx.y;
  - Col = blockIdx.x*blockDim.x + threadIdx.x;

Midterm Question 4

You and your buddy have developed a GPU program for an imaginary GPU, the G1000. The G1000 has 16 SMs with a maximum of 1024 threads/SM. You developed your program to have a block size of 1024 and a grid size of 16 to fully utilize the G1000. The program is work efficient, but each thread does a significant amount of work. The day after you finish coding, a new GPU comes, the G2000, with 32 SM and a maximum of 2048 threads/SM. Your friend suggests buying the G2000 to speed up your new program but realizes that changes to your program will be needed. They suggest all you need to do is half the block size and double the grid size, then the G2000 would be fully utilized. Do you agree or disagree with this modification? Why or why not? Show by example. Whether or not the G2000 is fully utilized would you expect any speedup in you program? Give any reason you feel is valid.

- Almost everyone got this one
- G2000 would not be fully utilized since each thread block will have 512 threads which underutilize the 2048 threads/ SM
- So you should not expect much speedup do to the same number of threads being used
- Or maybe some speedup because we might have more parallelism with more SMs

Midterm Question 5

After some debate, your friend then asks how you would modify your program to fully utilize the G2000? Would those changes affect the amount of work done per thread? If so how? Justify your modification with any reasoning you feel is valid.

- Almost everyone got this one
- To fully utilize the hardware you need to have a grid size of 32 and a block size of 2048
- If you do this you would need to modify the program so threads do ¼ of the work

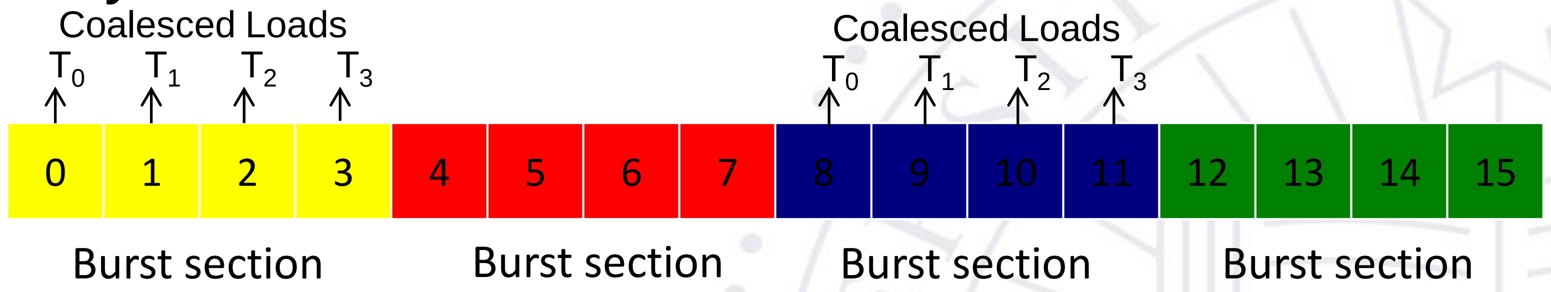
Midterm Question 6

The debate ends when you both realize you do not have any money to buy the new card. Using the G1000, your program was only designed o run with a fixed data size and breaks when using a larger dataset. Your friend proposes two options, scale the grid size to fit the dataset or tile the algorithm. Which do you choose? Give the pros and cons for both

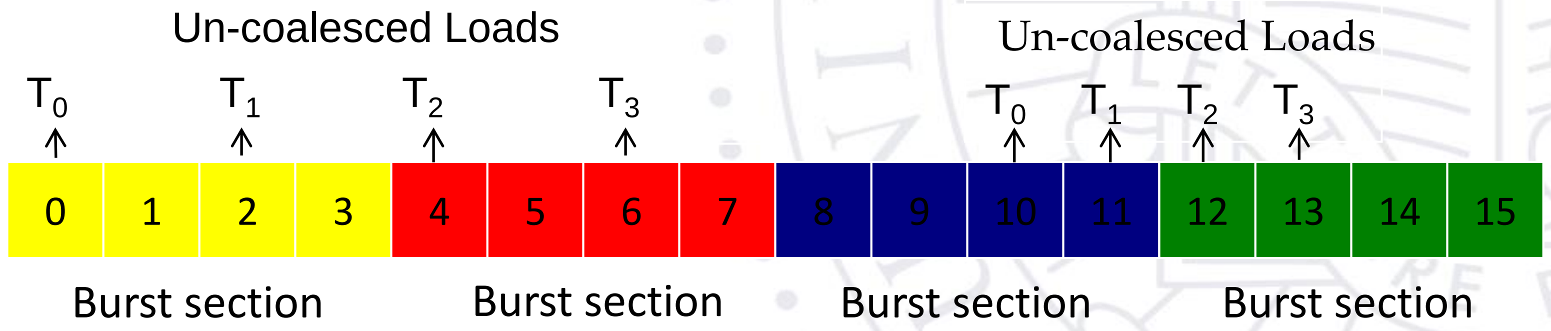
- Scaling grid size is easy enough to do and it works well, however performance won’t scale if we are already fully utilizing the gpu
- Tiling requires more changes, but it could potentially increase performance if shared memory is used or other localities are taken advantage of

Memory coalescing

- When all threads of a warp execute a load instruction, if all accessed locations are contiguous, only one DRAM request will be made and the access is fully coalesced.



- When the accessed locations spread across burst section boundaries Coalescing fails and Multiple DRAM requests are made

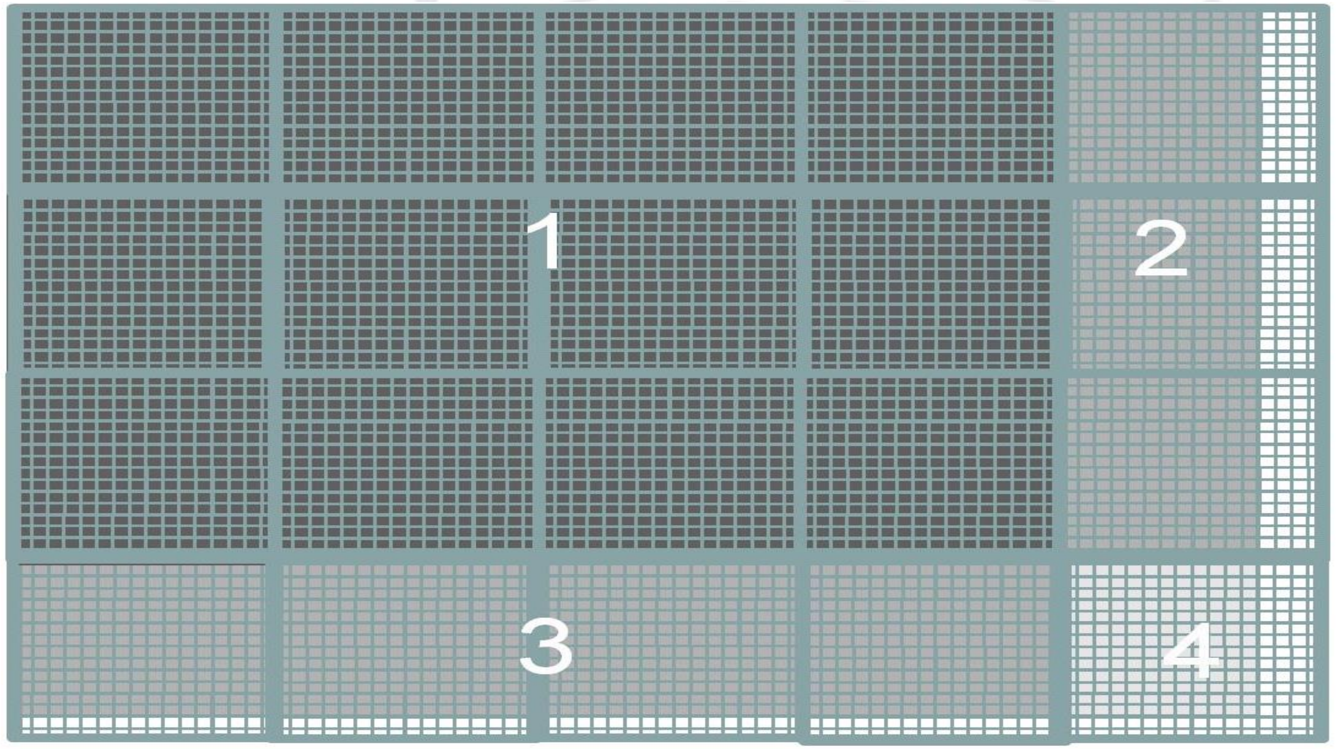


Memory coalescing

- Be able to spot and modify code to address memory coalescing concerns
- This affect thread access patterns
- Loads across threads access memory contiguously
- Threads read across a row and access down a column
- Or load into shared memory if your access pattern cannot be easily altered

Warp Divergence

- Divergence only occurs when threads within a warp go through different control paths
- 1) all threads are active
- 2) All warps have divergence
- 3) Some threads are inactive but no warp divergence
- 4) Some warps have divergence

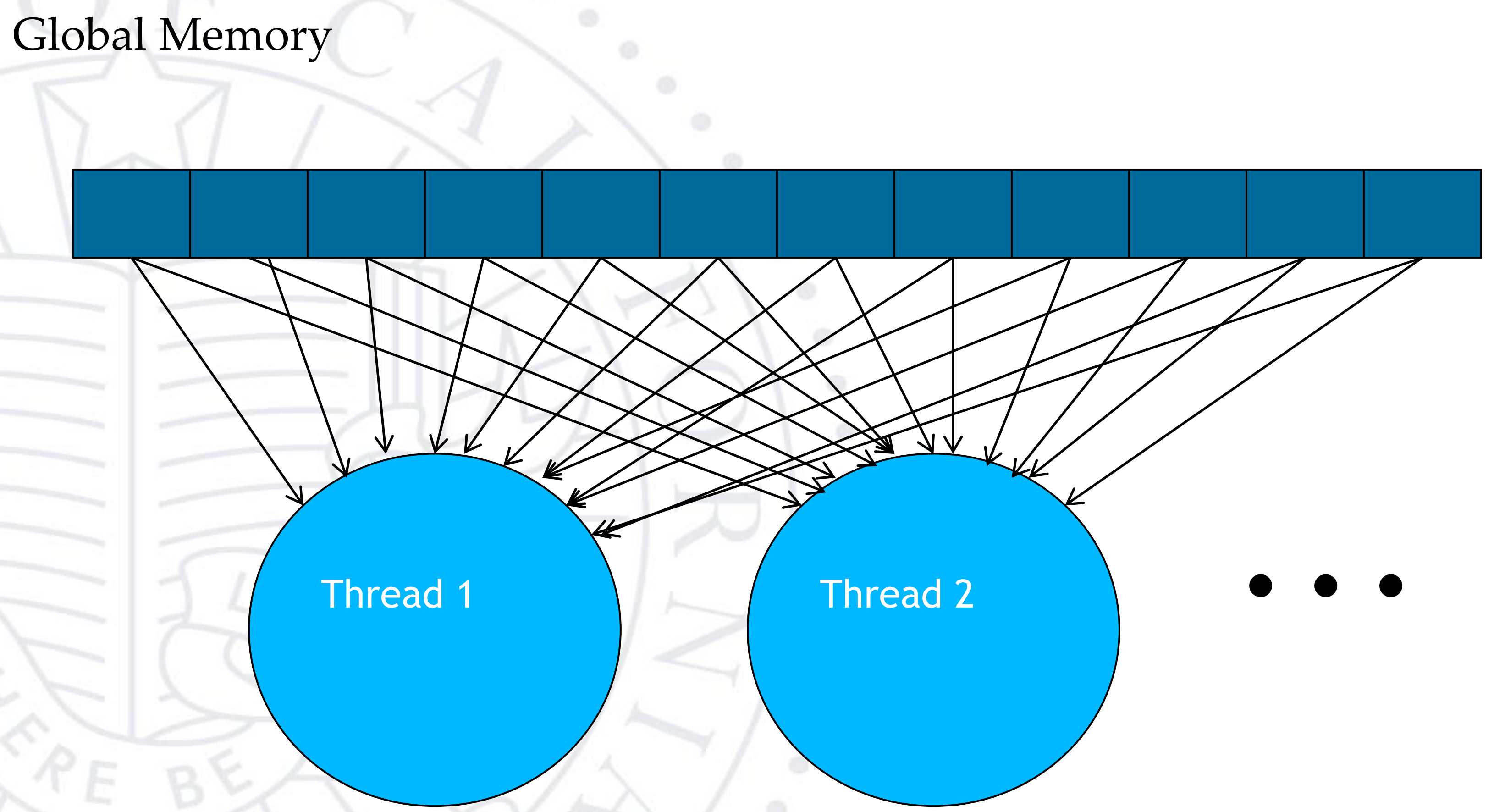


Warp Divergence

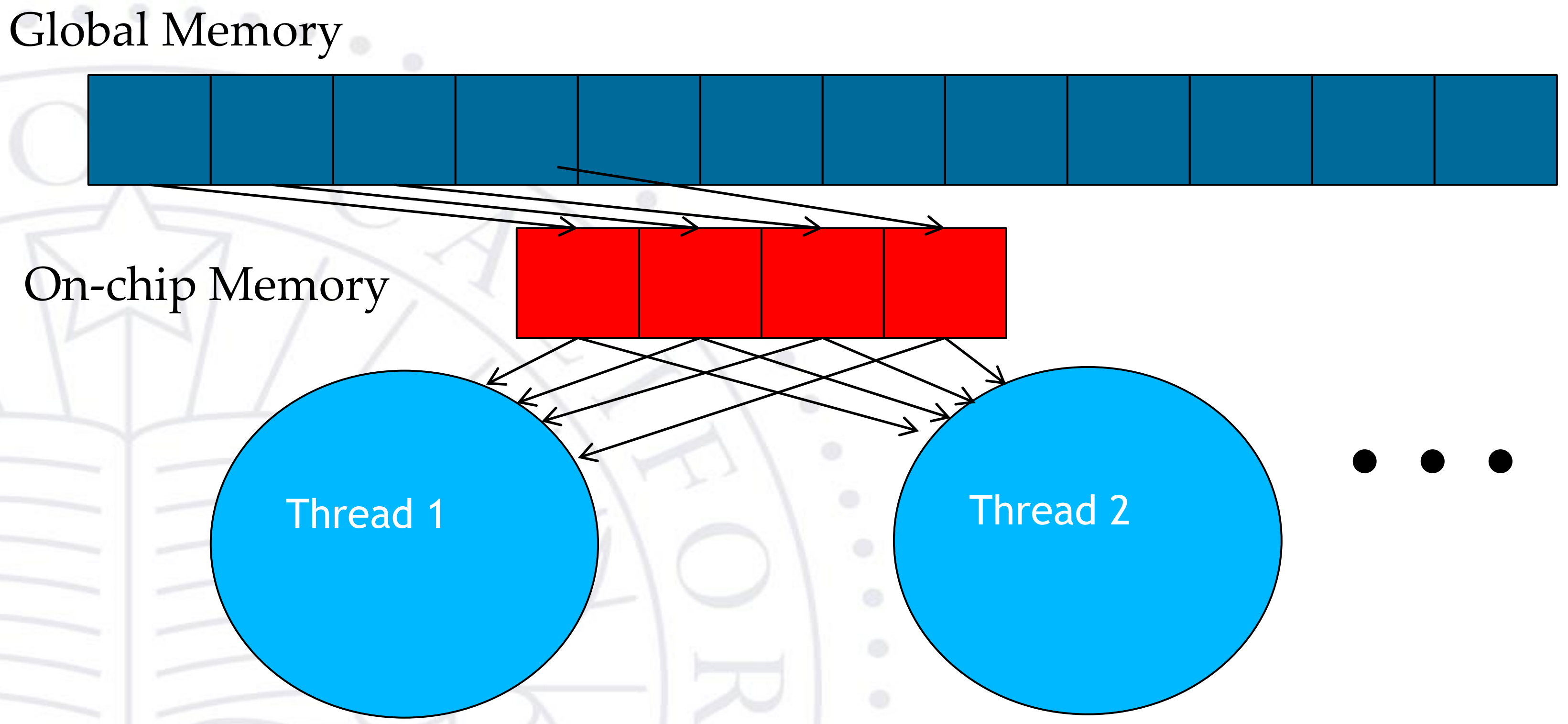
- Be able to calculate the number of warps that exhibit divergence for a particular input and block size
- Spot and modify code to reduce the amount of divergence
  - Pad outer bounds with 0 and get rid of any control instructions
  - Resize block or change thread access pattern to land on warp boundaries
  - Compact active threads to contiguous warps (reduction implementation)

Shared memory

Accessing memory is expensive, reduce the number of global memory loads



Shared Memory



Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time

Shared Memory

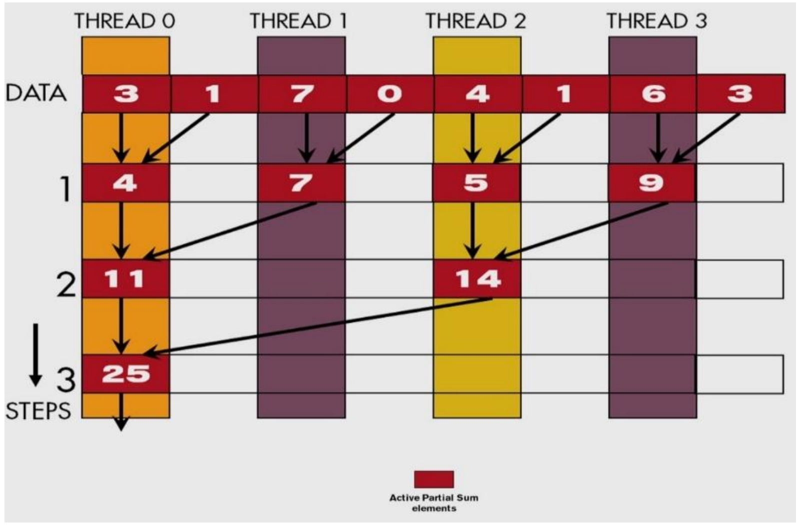
- Declare with __Shared__ var[size]
- Load into shared var then read from it
- Shared memory is only useful if you access it multiple times
- How to use it with tiling

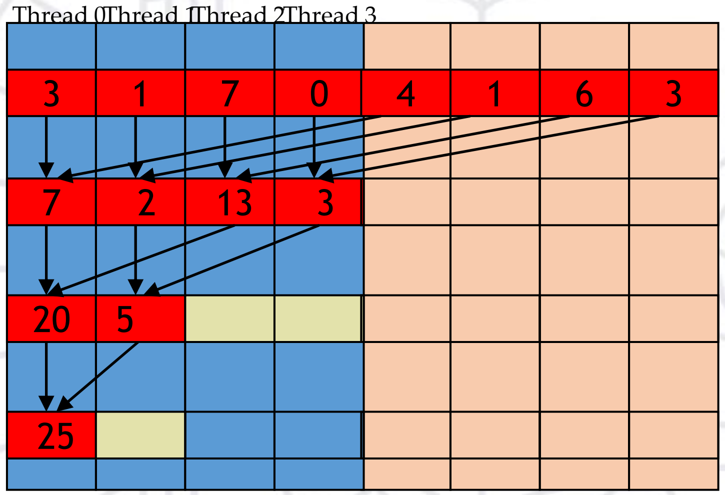
Synchronization

- __syncthreads() synchronizes all threads within a thread block
- Cannot synchronize within conditional statements, this will create a deadlock
- Some examples when to sync
  - Loading data into shared memory, computation depends on previous iterations, writing back to global memory
- To synchronize across thread blocks, need to do a cudaDeviceSynchronize on the host side.
- This means threads blocks are only “synchronized” once the kernel completes

Reduction

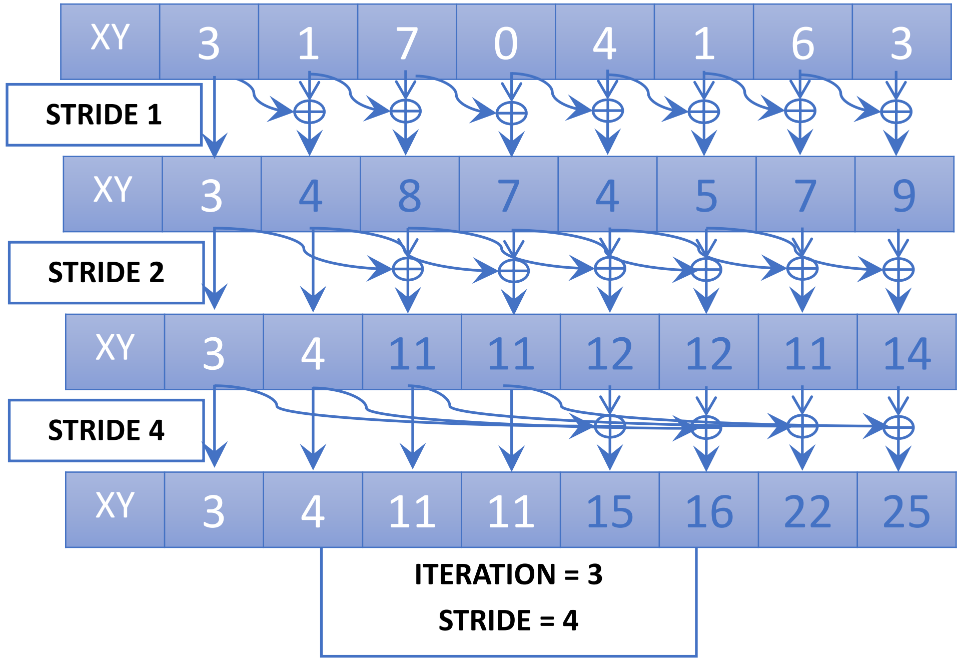
- Parallel reduction uses tree algorithm for O(logn)
- Two implementations
  - Understand the difference in implementation and performance
- Understand as an example of warp divergence, memory coalescing, and thread synchronization

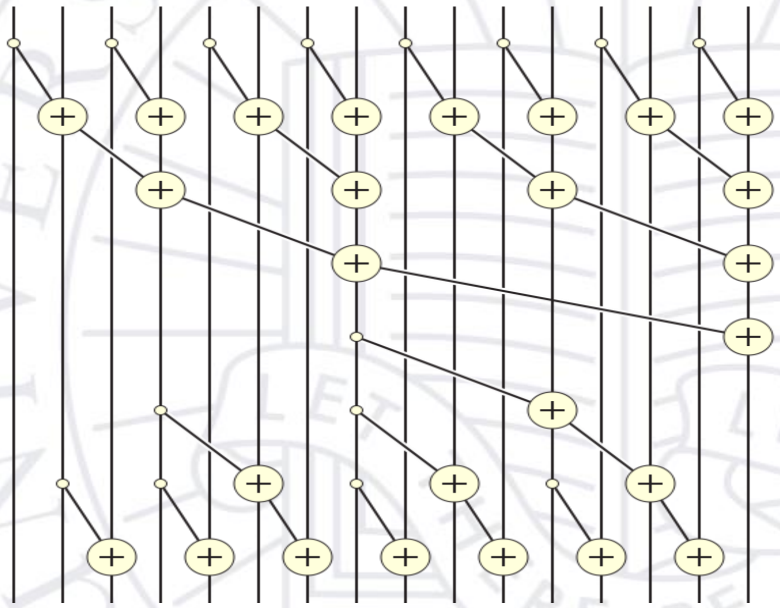




Scan

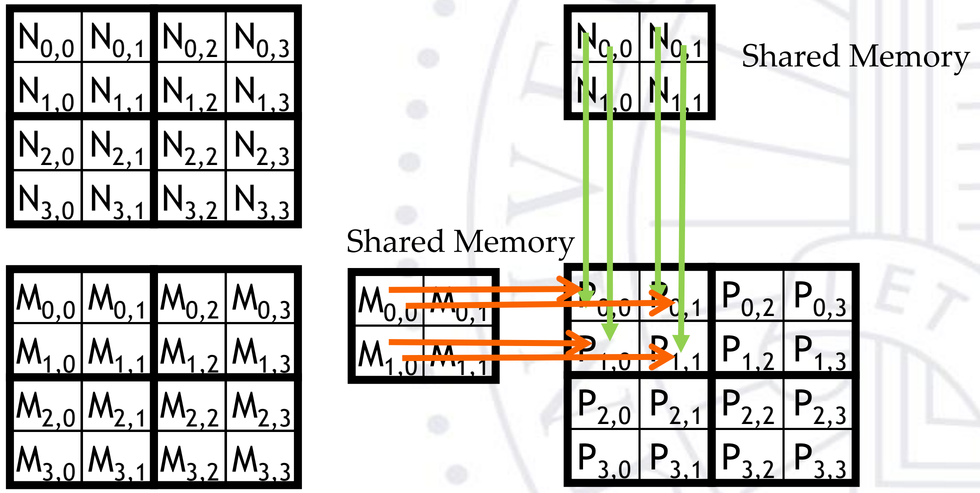
- Parallel scan either strided array or tree algorithm
- Two implementations
  - Understand the difference in implementation and performance
- Understand as an example of work efficiency and thread synchornization





Tiled Matrix Multiplication

- Great example of tiling algorithm, use of shared memory, and thread synchronization
- Relation between tile size and block size
- Number of tiled phases for any height and width of matrix
- 2D Thread and block ids



$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Shared Memory
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
Shared Memory
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Midterm Question 7

Game of Life kernel

- Where __syncthreads() was placed was the issue
- Synchronization cannot be placed inside if else statements because it might be possible that not all threads in the block go down that path so this would stall the entire program
- ALSO, the iteration loop should be on the outside of the kernel. This is because ALL cells in the grid must be updated before you move onto the next iteration. This requires a device level synchronization to synchronize all thread blocks. This can only be done with cudaDeviceSynchronzie at the host level.
- I gave the most leeway with these next two questions since it may be confusing and it was the end of a really long test

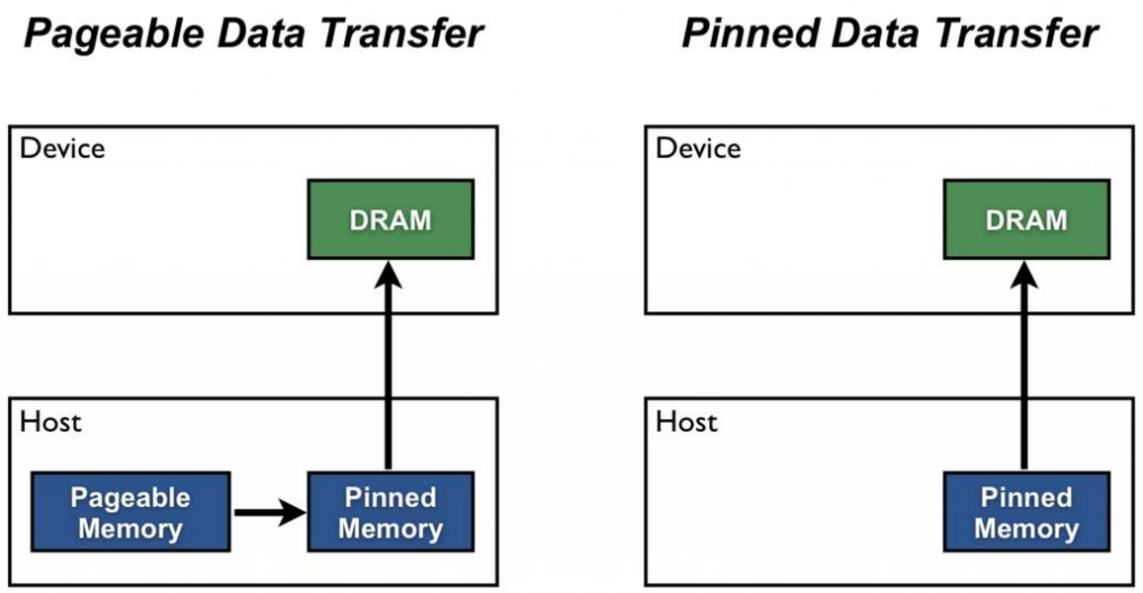
Midterm Question 8

Game of Life kernel

- Some improvements included reducing the number of if else cases so there is less control flow
- Using shared memory since cells are used across many threads
- Any other improvements that you provided I accepted, but I did not accept just stating that we could improve memory coalescing, etc…
- I’m looking for the actual changes that will improve those behaviors

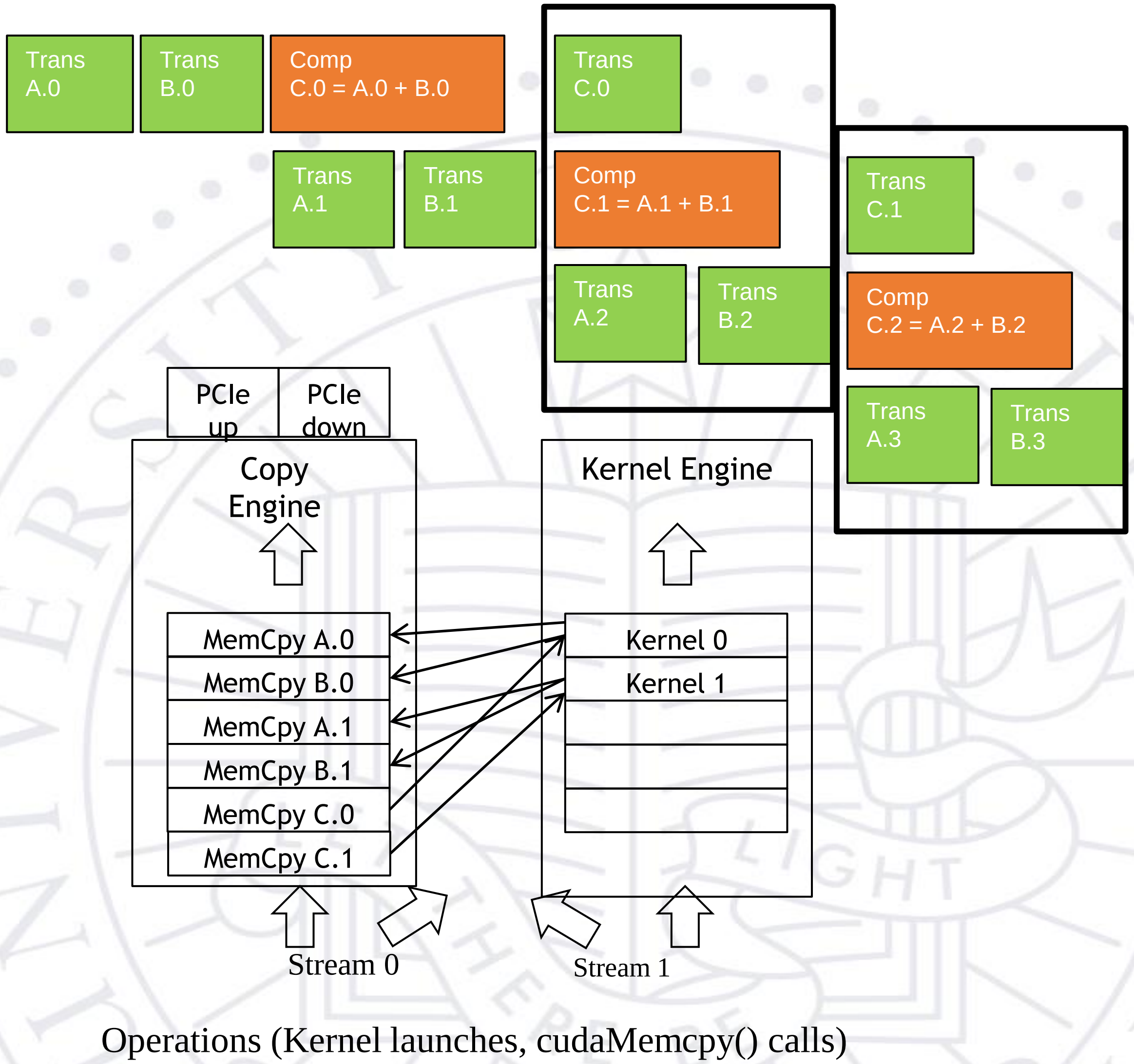
Unified Memory

- Pageable memory vs pinned memory
  - How it affects performance, number of memory copies, consumes physical resources
- Unified Memory
- Single pointer for host and device memory
  - Transfers are now handled at the driver level
- On demand paging
  - Pages are swapped from host to device whenever they are needed
- What type of applications can benefit from unified memory and on demand paging?



Streams

- Streams allow parallel execution of kernels and memory copy
- Streams are then put into fifo queues for copy and kernels
- Allow for pipelined overlap of copy and computation
- What type of applications benefit from streams? What type would not benefit from streams? Overheads of two kernels competing for the same physical resources?



Operations (Kernel launches, cudaMemcpy() calls)

Atomics

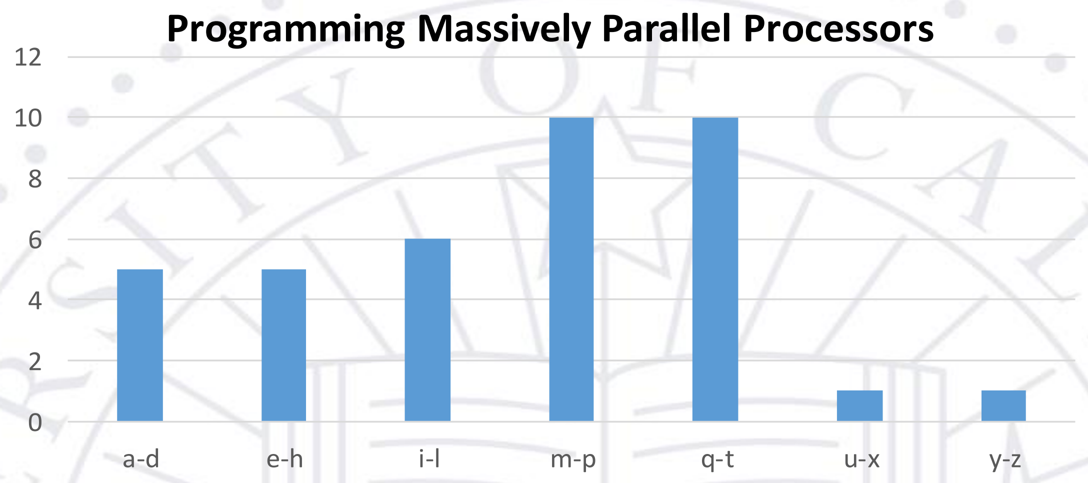
- Data races occur when two or more threads are trying to read-modify-write to the same memory address location
- Atomics are intrinsic instructions, built into the hardware, that ensures only a single thread can perform the read-modify-write operation at once.
  - All threads perform their atomic operations serially on the same location
- Atomics are long latency operations
- Can be used in at L2 or shared memory to shorten latency
- What type of applications benefit from atomics? Overheads of atomics?

```
thread1:   Old ← Mem[x]
           New ← Old + 1
           Mem[x] ← New

                                 thread2:   Old ← Mem[x]
                                            New ← Old + 1
                                            Mem[x] ← New
```

Histogram

- Counting total number of objects based on some value or feature
- Can be parallelized through sectioned indexing or interleaved indexing
  - Understand the memory coalescing consequence of each method
- Atomics must be used to increment global bin counters
- Improve performance through privatization
- Understand as an example of privatization & shared memory, atomics at different memory levels, indexing to improve memory coalescing



Dynamic Parallelism

- Device side kernel launches
- Per thread kernel launch with the <<<>>> syntax
- Launch is non blocking
- Device side cudaDeviceSynchronize() forces the calling thread to wait until the kernel that it launched is finished
- What type of applications and use cases does dynamic parallelism enable? How is this more efficient than host side launches?