INTRODUCTION TO CUDA

Agenda

- Why GPUs?
- Parallel Processing
- CUDA
- Example
- GPU Memory
- Advanced Techniques
- Issues
- Conclusion

- Moore’s Law
  - Concurrency, not speed
  - Hardware support
- Memory bandwidth
- Programmability

Why GPUs?

Image from: CUDA Programming Guide

TECHNICAL SPECIFICATIONS	TESLA K10[a]	TESLA K20	TESLA K20X
Peak double precision floating point performance (board)	0.19 teraflops	1.17 teraflops	1.31 teraflops
Peak single precision floating point performance (board)	4.58 teraflops	3.52 teraflops	3.95 teraflops
Number of GPUs	2 x GK104s	1 x GK110	
Number of CUDA cores	2 x 1536	2496	2688
Memory size per board (GDDR5)	8 GB	5 GB	6 GB
Memory bandwidth for board (ECC off)[b]	320 GBytes/sec	208 GBytes/sec	250 GBytes/sec
GPU computing applications	Seismic, image, signal processing, video analytics	CFD, CAE, financial computing, computational chemistry and physics, data analytics, satellite imaging, weather modeling	
Architecture features	SMX	SMX, Dynamic Parallelism, Hyper-Q	
System	Servers only	Servers and Workstations	Servers only

[a] Tesla K10 specifications are shown as aggregate of two GPUs.
[b] With ECC on, 12.5% of the GPU memory is used for ECC bits. So, for example, 6 GB total memory yields 5.25 GB of user available memory with ECC on.

Why GPUs?

- Raw computing power
- Cost ($ per FLOP)
- Ubiquitous
  - Stable – Graphics & Games
- Practical (space, noise, power)
- Power (performance per watt)

- Shared Memory
  - Multi-core
  - Multi-socket
- Distributed Memory
  - Clusters
  - LCFs
- Accelerators
  - GPUs
  - FPGAs
- Asynchronous Operations

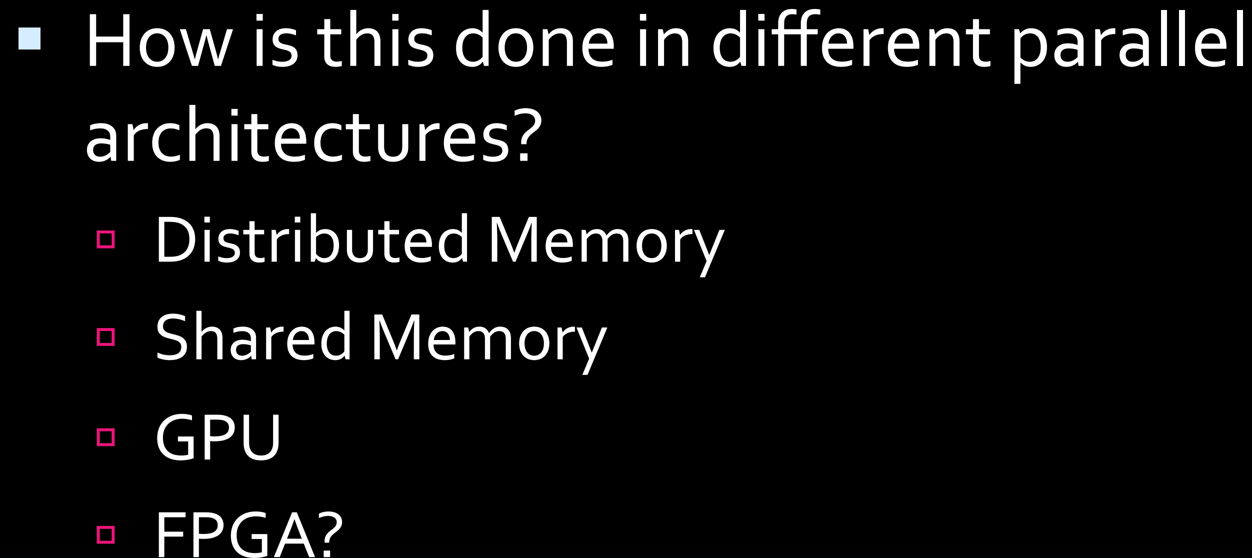
- Example: Reduction

Parallel Processing

- Example: Reduction

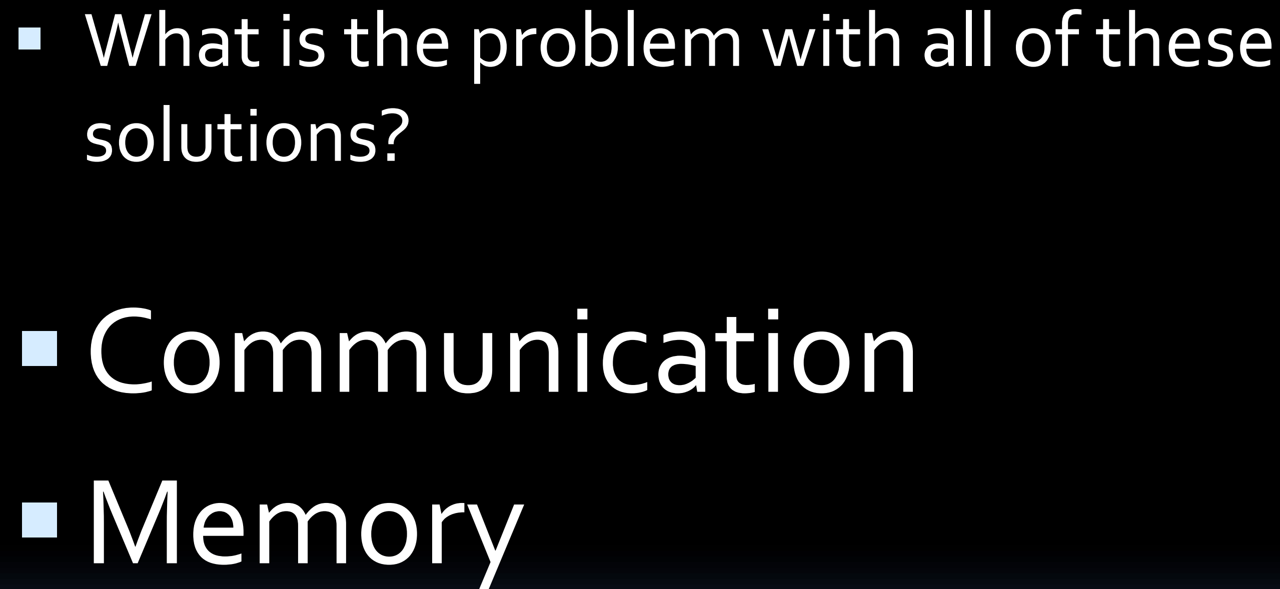
- Each processor does a portion
- Sync
- Repeat

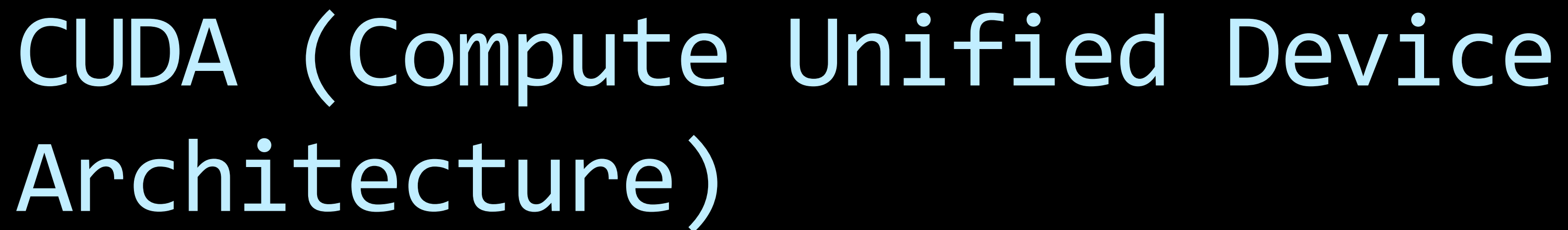
Parallel Processing


How is this done in different parallel architectures?
Distributed Memory
Shared Memory
GPU
FPGA?

- What is the problem with all of these solutions?

Parallel Processing


What is the problem with all of these solutions?
Communication
Memory



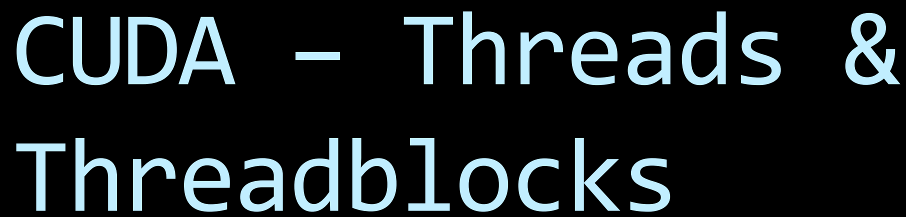
CUDA (Compute Unified Device Architecture)

- Single Instruction Multiple Data (SIMD)
  - Each processor executes the **same** instruction, at the same time
  - Each processor operates on different data
- Thousands of concurrent threads
- Hardware support
- Reduced functionality cores
- Limited number of functional units
- Quickly evolving

GPU Architecture

- Cores are organized into SMs (streaming multiprocessors)
- Multiple SMs per GPU
- Certain elements are **shared** on and SM:
  - Special function units
  - Double precision units
  - Register file
  - Caches
  - Texture elements

Image from http://benchmarkreviews.com/index.php?option=com_content&task=view&id=440&Itemid=63&limit=1&limitstart=4

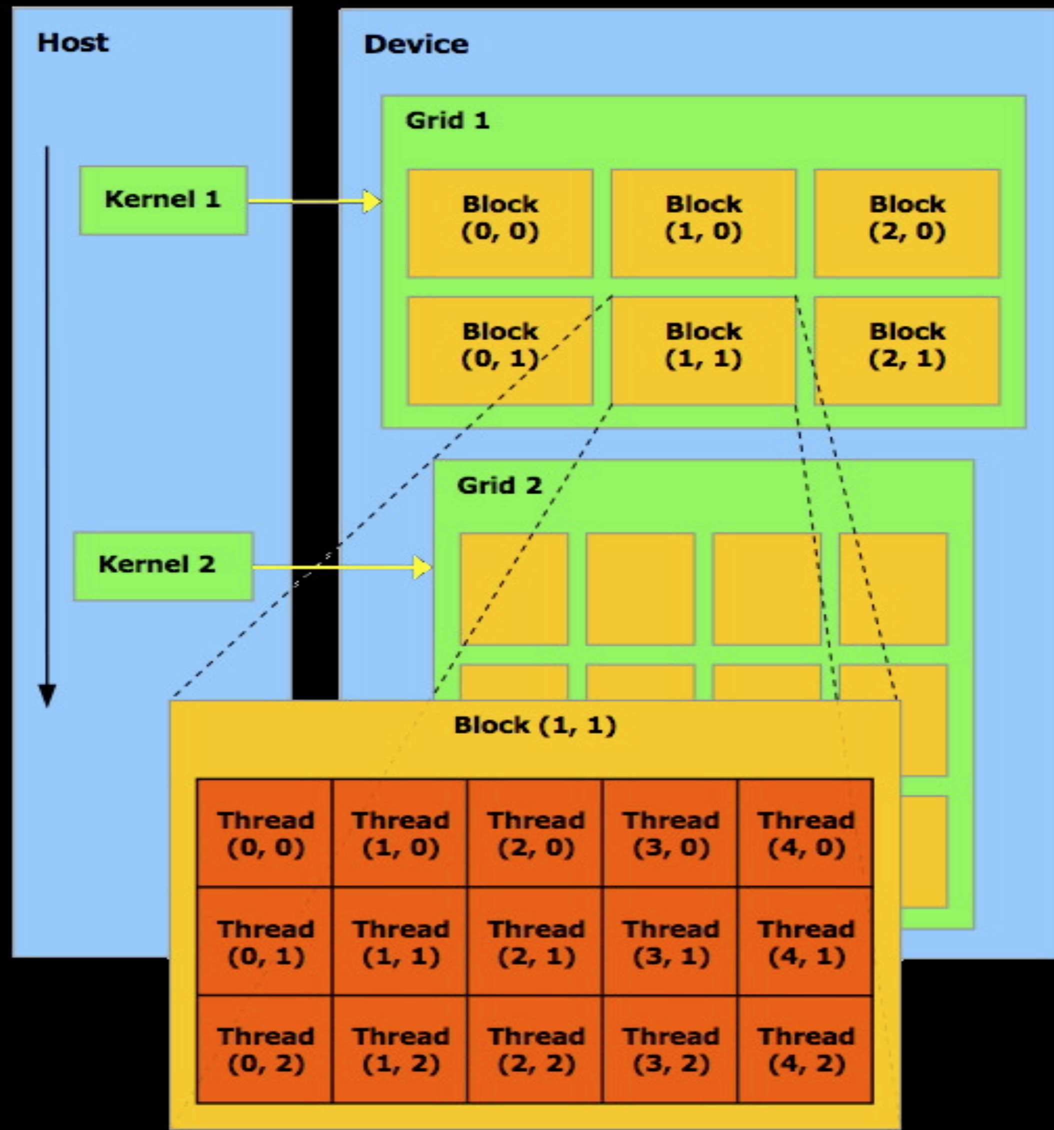


CUDA – Threads & Threadblocks

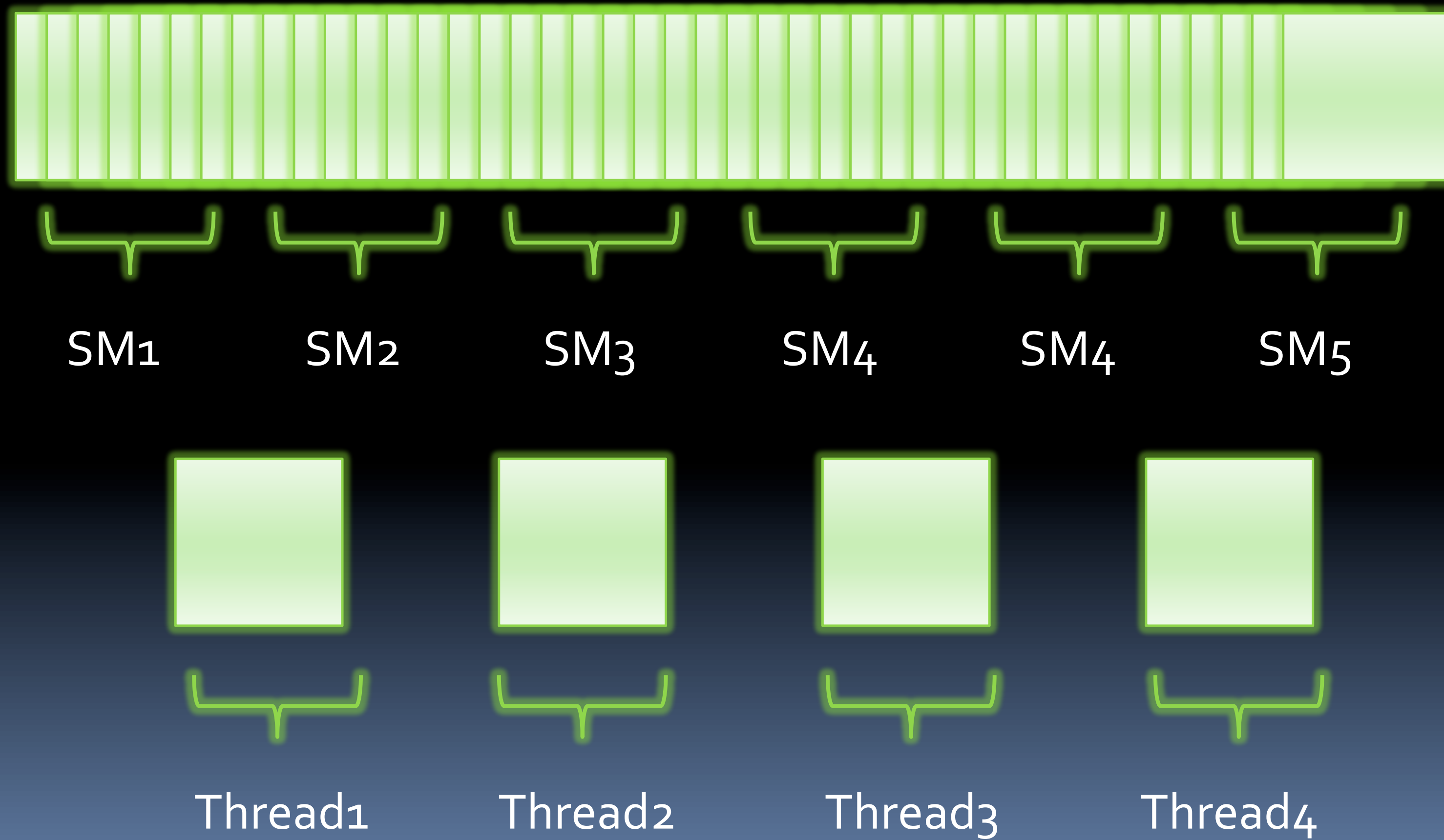
- Individual execution units are called **threads**.
  - Threads are also grouped into **warps** and **half-warps**.
  - A warp – 32 threads
- Threads are grouped into **thread blocks**.
- Thread blocks are grouped into a grid.
- Functions that execute on the GPU are called **kernels**.
- No communication between thread blocks

CUDA – Threads & Threadblocks

Image from: CUDA Programming Guide



CUDA – Threads & Threadblocks



CUDA First Example

- Vector Addition:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

CUDA First Example Revisited

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

CUDA Second Example

- Previous examples all 1D thread blocks and grid – what about 2D data?

CUDA Second Example

- Previous examples all 1D thread blocks and grid – what about 2D data?

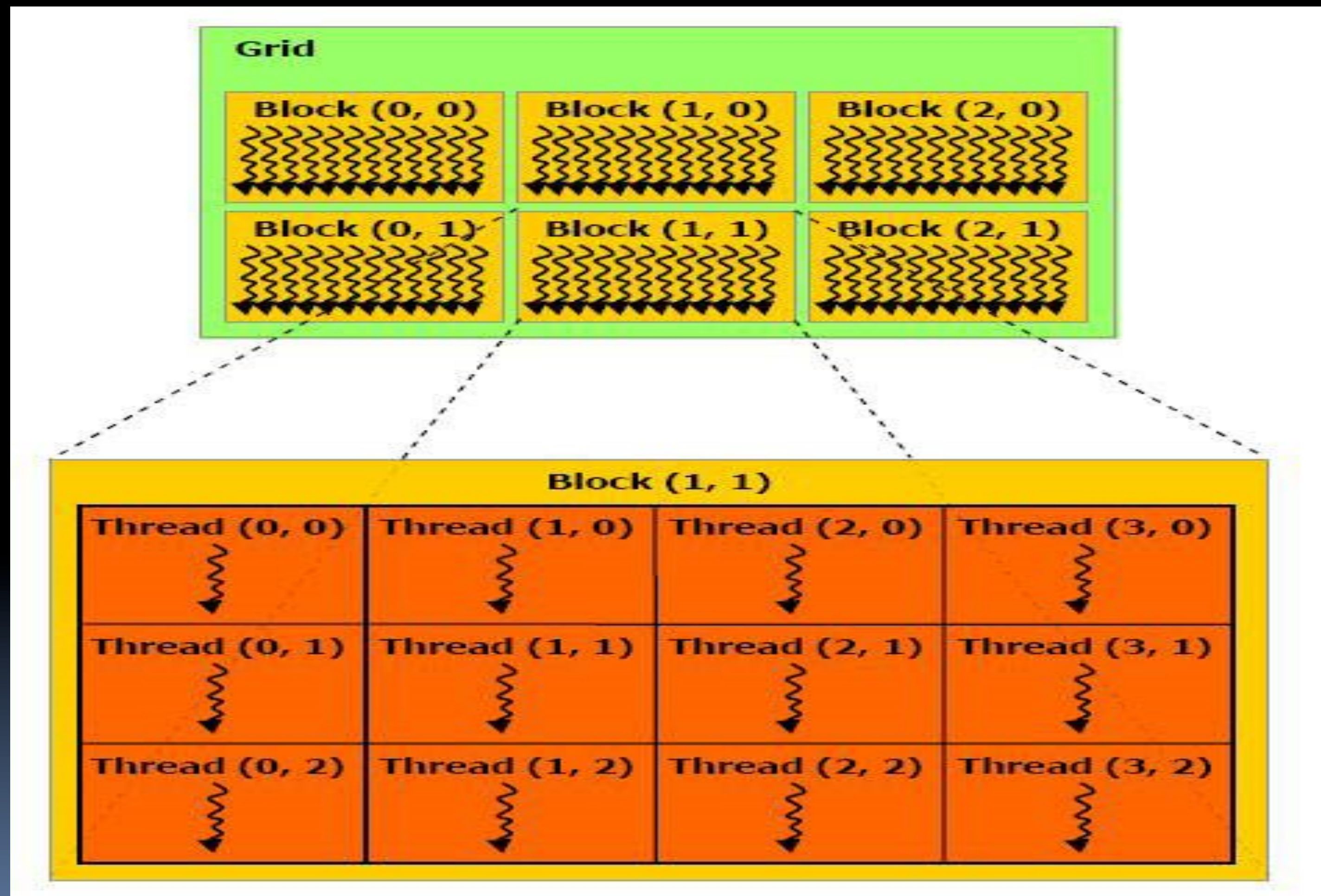


Image from: CUDA Programming Guide

CUDA Second Example: Matrix Addition

- Matrices are two dimensional

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i < N && j < N)
      C[i][j] = A[i][j] + B[i][j];
}
```

GPU Memory

- Probably the biggest difference from traditional CPUs
- High bandwidth – but only if accessed correctly
- Lots of restrictions
- May be one of the biggest obstacles to adoption

GPU Memory

- Different regions:
  - Registers
  - Global
  - Shared
  - Local
  - Texture
  - Constant

GPU Memory

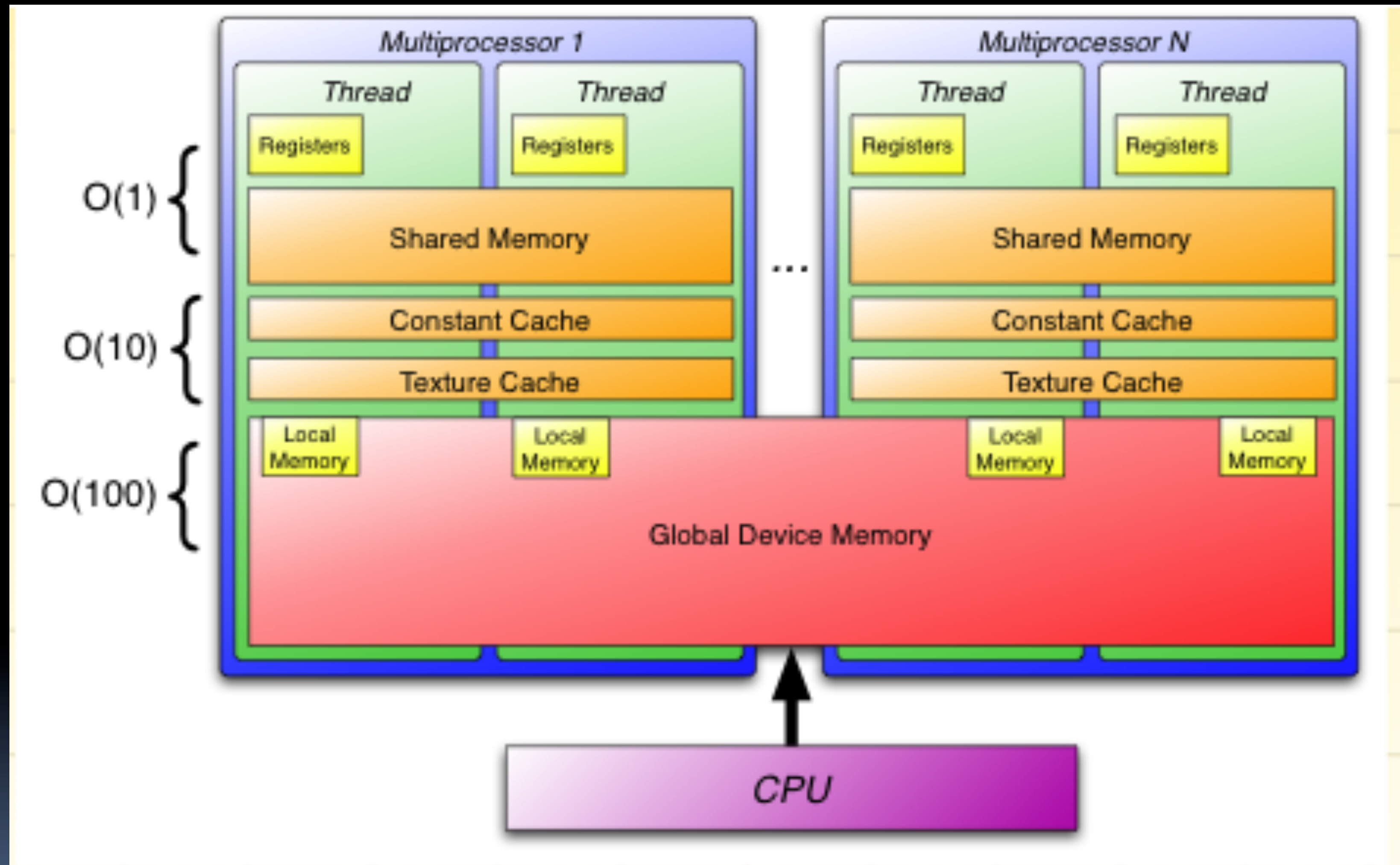
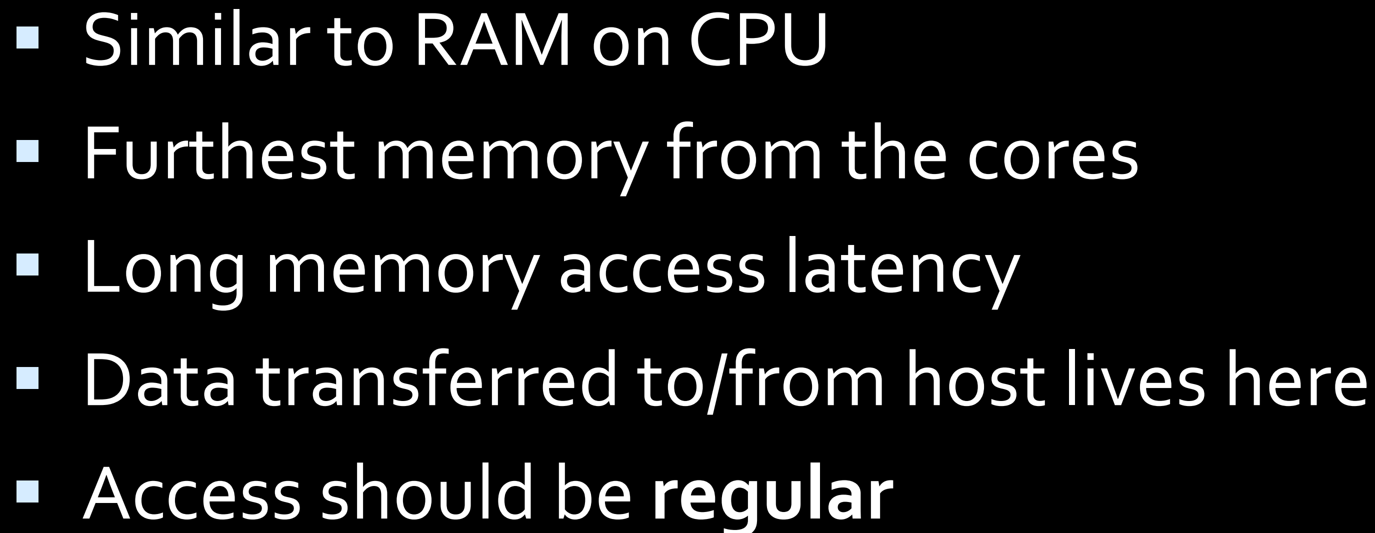


Image from: http://people.sc.fsu.edu/~jburkardt/latex/fdi_2009/gpu_memory.png

GPU Memory - Global


Similar to RAM on CPU
Furthest memory from the cores
Long memory access latency
Data transferred to/from host lives here
Access should be regular

GPU Memory - Global

- Number of **aligned** memory transactions for each version of CUDA HW.

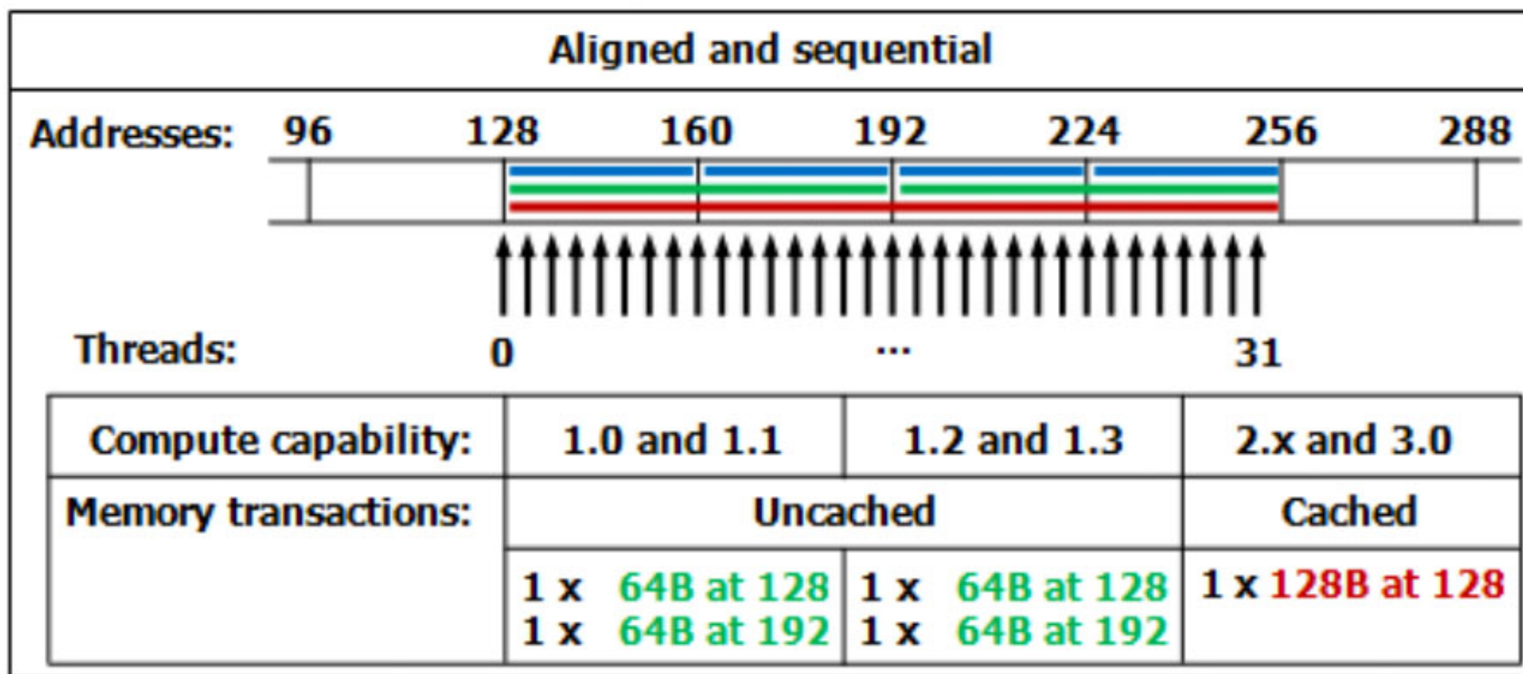
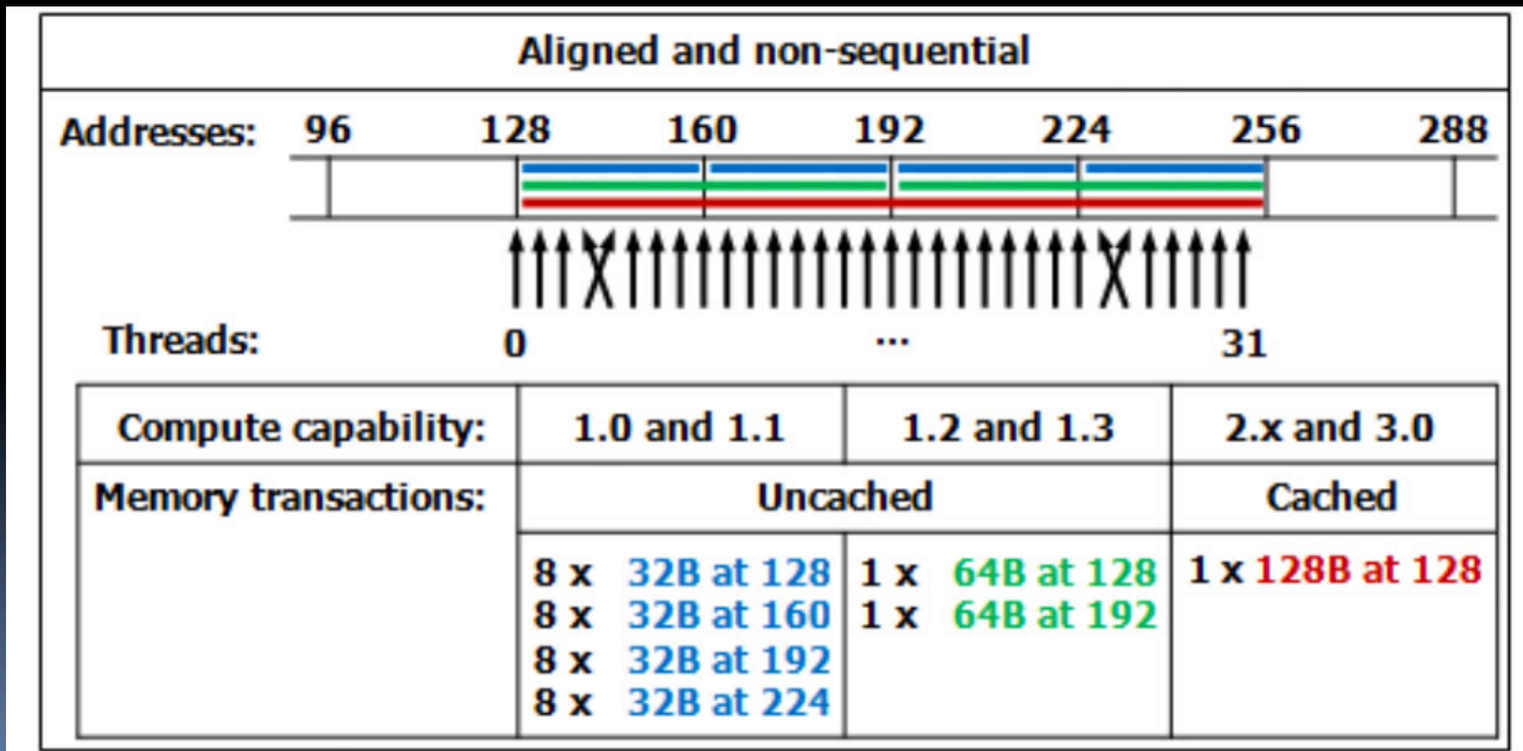


Image from CUDA Programming Guide

GPU Memory - Global

- Number of **aligned non-sequential** memory transactions for each version of CUDA HW.

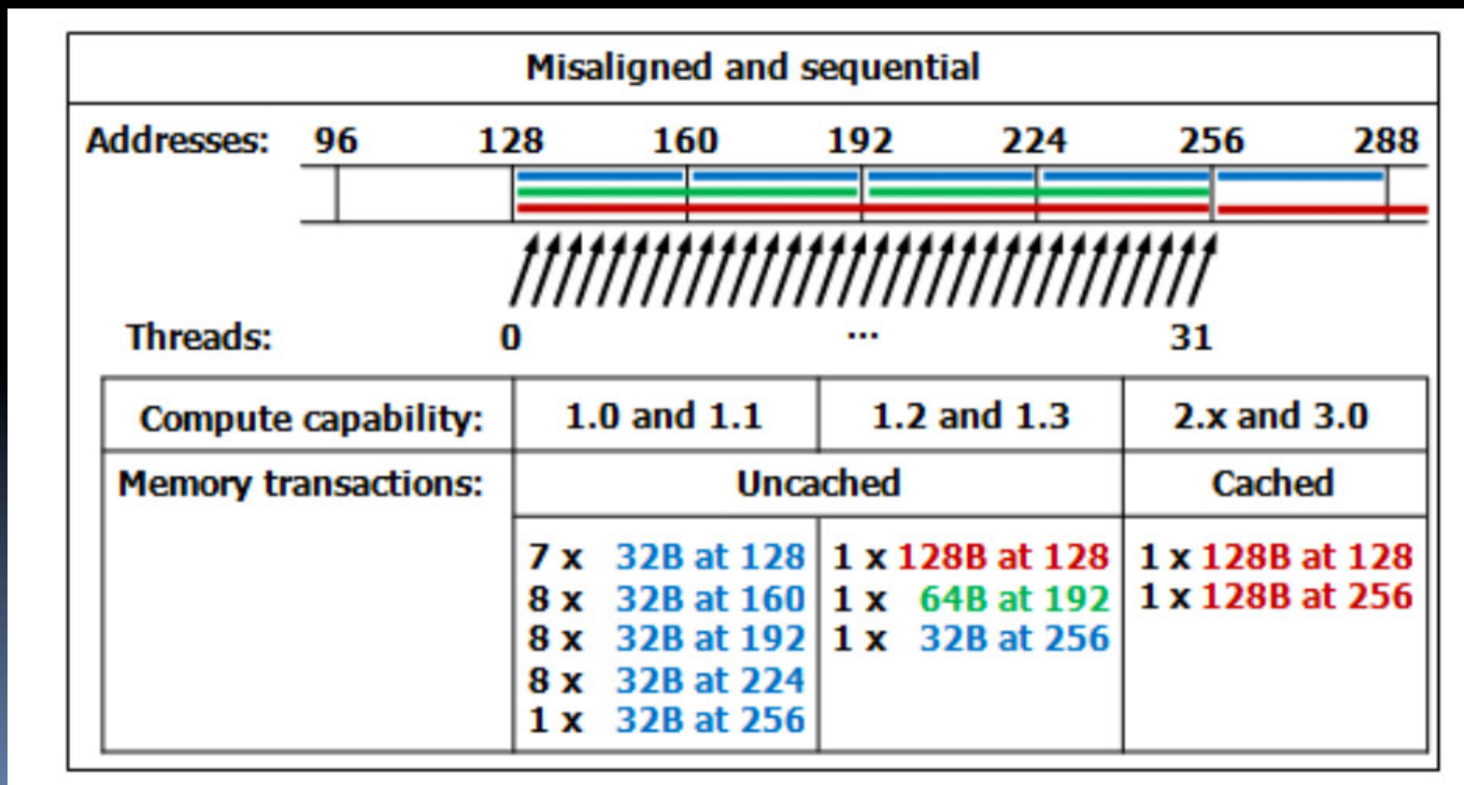


Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Image from CUDA Programming Guide

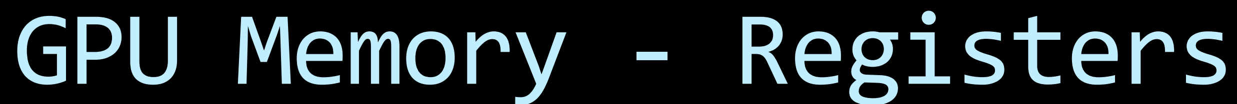
GPU Memory - Global

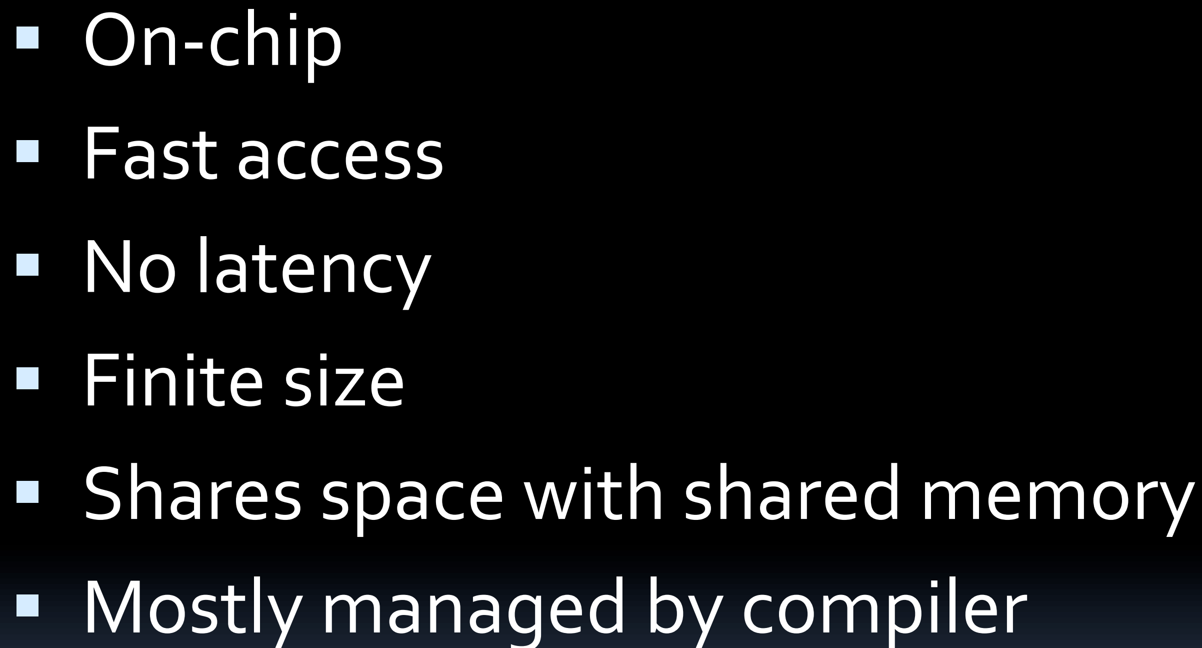
- Number of **misaligned non-sequential** memory transactions for each version of CUDA HW.


Misaligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Image from CUDA Programming Guide


GPU Memory - Registers


On-chip
Fast access
No latency
Finite size
Shares space with shared memory
Mostly managed by compiler

- On-chip (SM), very fast
- User managed cache
- Required below compute capability < 1.2
- Explicit use in later versions still recommended
- Shares space with register file
- Perfect for repeatedly accessed data
- Example?

GPU Memory – Shared Example: Matrix Multiplication

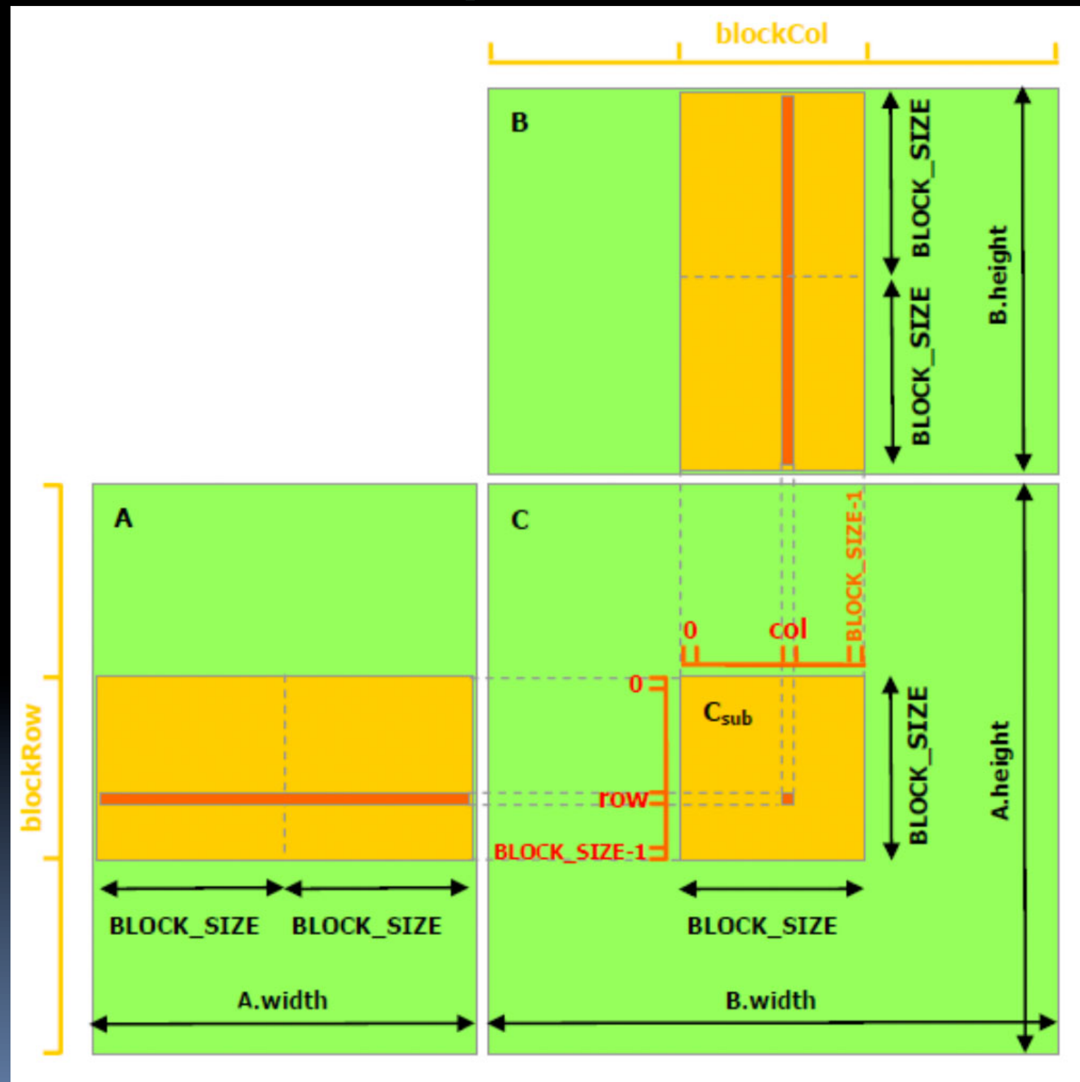
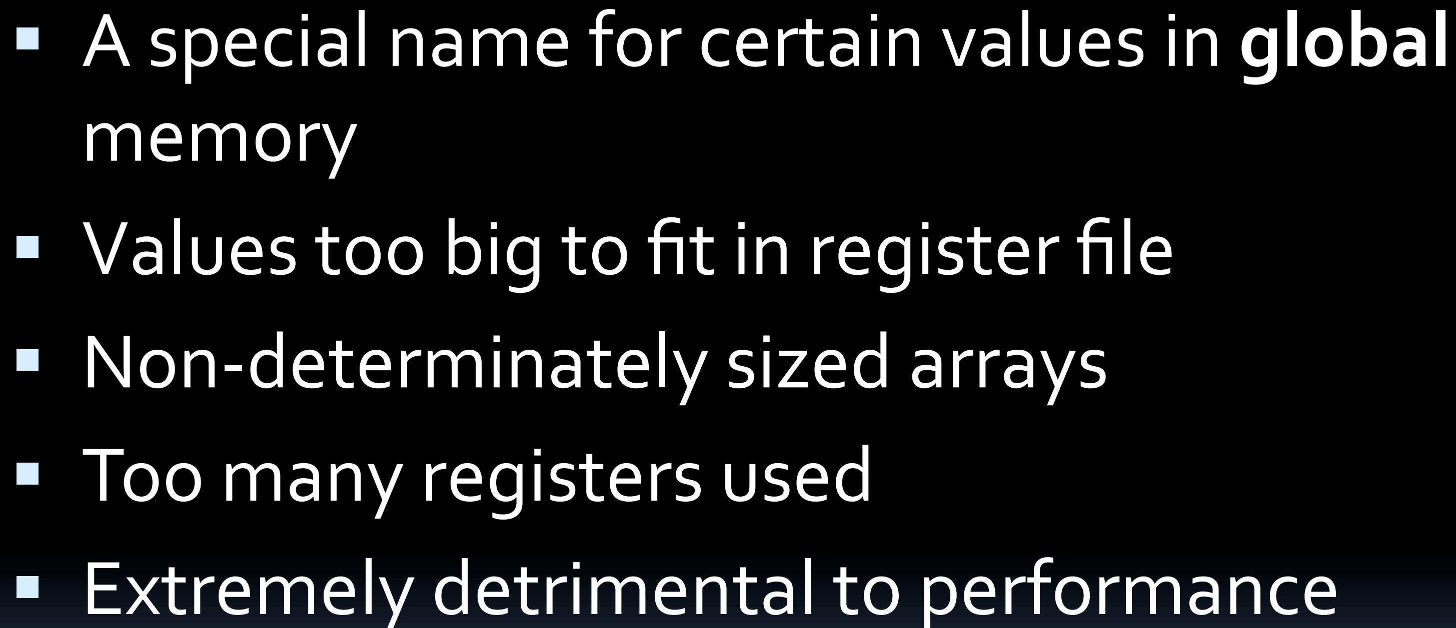
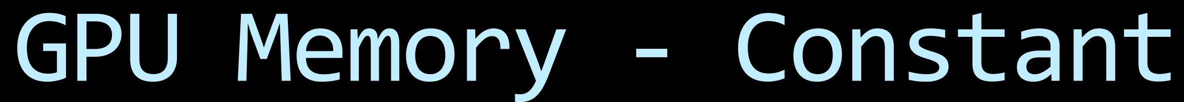


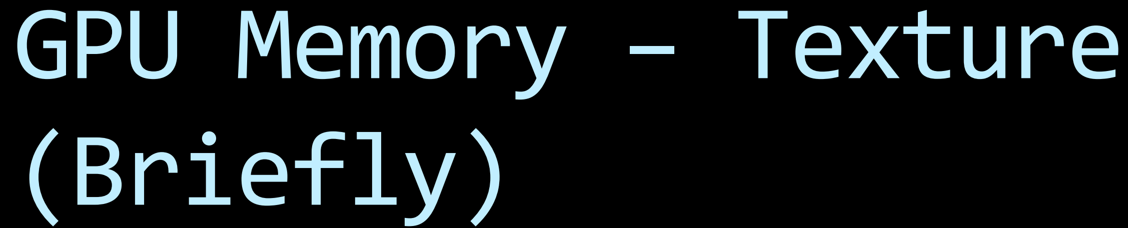
Image from CUDA Programming Guide

GPU Memory - Local


A special name for certain values in global memory
Values too big to fit in register file
Non-determinately sized arrays
Too many registers used
Extremely detrimental to performance



- Fast, on-chip memory region for read-only values
- Small
- Coalescence is different: all threads must access **same** value
- Useful for constants, or small read-only data



GPU Memory – Texture (Briefly)

- Comes from graphics operations
- Cached reads
- Cache writes not coherent
- Spatial locality is important
- Certain hardware assisted operations (interpolation)

GPU Memory - Latency

- Global memory accesses take several hundred cycles
- Once a warp makes a request to global memory it is swapped out and the next warp is loaded in
- By the time the time all warps have made a request the data for the first request will have arrived and the first warp can continue

- GPU device must be saturated
- There must be enough work to do
- Hide latency

- Atomic operations
- Memory bound kernels
- Arithmetic intensity
- Redundancy calculations
- Memory halos
- Index calculations
- Memory **shape**

Advanced Techniques – Atomic Operations

- Back to the reduction
- Every step cuts the data size by half
- Multiple threads write to same location
- Serializes execution
- Use with Care

```
// Thread zero of each block signals that it is done

unsigned int value = atomicInc(&count, gridDim.x);
```

- Useful metrics to determine performance
- FLOPs per read
- How many times is each global memory value used?
- Bus utilization
  - Memory transactions sometimes transmit more bytes than required on the bus
  - Want to minimize number of unused bytes across the bus

Advanced Techniques – Index Calculations

- Recall 2-D case:

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
```

- Approximately 15 instructions for each index
- This is calculated for **every** thread
- For 1M data elements, 15M instructions just for indices

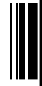
Advanced Techniques – Index Calculations

- After first index calculation (15 instructions), calculate another, relative to the first, with a **single** instruction
  - int i = blockIdx.x * blockDim.x + threadIdx.x;
  - int j = i + BLOCK_DIM
- Now two indices calculated for 16 instructions
- 15M down to 8M instructions
- Approximately 44% reduction

Issues

- PCI Express Bus
- Memory access
- Bus utilization
- Arithmetic intensity
- Optimization is difficult

- Serious performance improvements possible
- Requires certain data organization and algorithms
- Fits into other parallel methodologies
- Non-trivial implementation
- Constantly evovling



Questions?