NVIDIA®

Tesla GPU Computing

John Nickolls

Outline

- Tesla GPU Computing
- GPU Computing Architecture
- Multithreading and Thread Arrays
- Data Parallel Problem Decomposition
- Parallel Memory Sharing
- Transparent Scalability
- CUDA C Programming Model
- Summary

NVIDIA Tesla

Scalable High Density Computing

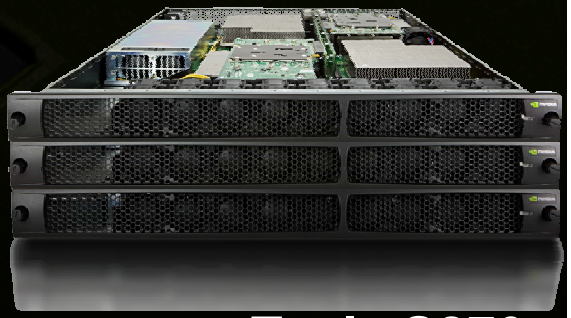
Massively Multi-threaded Parallel Computing

Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture is a scalable parallel computing platform
- In laptops, desktops, workstations, servers
- 8-series GPUs deliver 50 to 200 GFLOPS on compiled parallel C applications
- GPU parallel performance pulled by the insatiable demands of PC game market
- GPU parallelism is doubling every year
- Programming model scales transparently
- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism

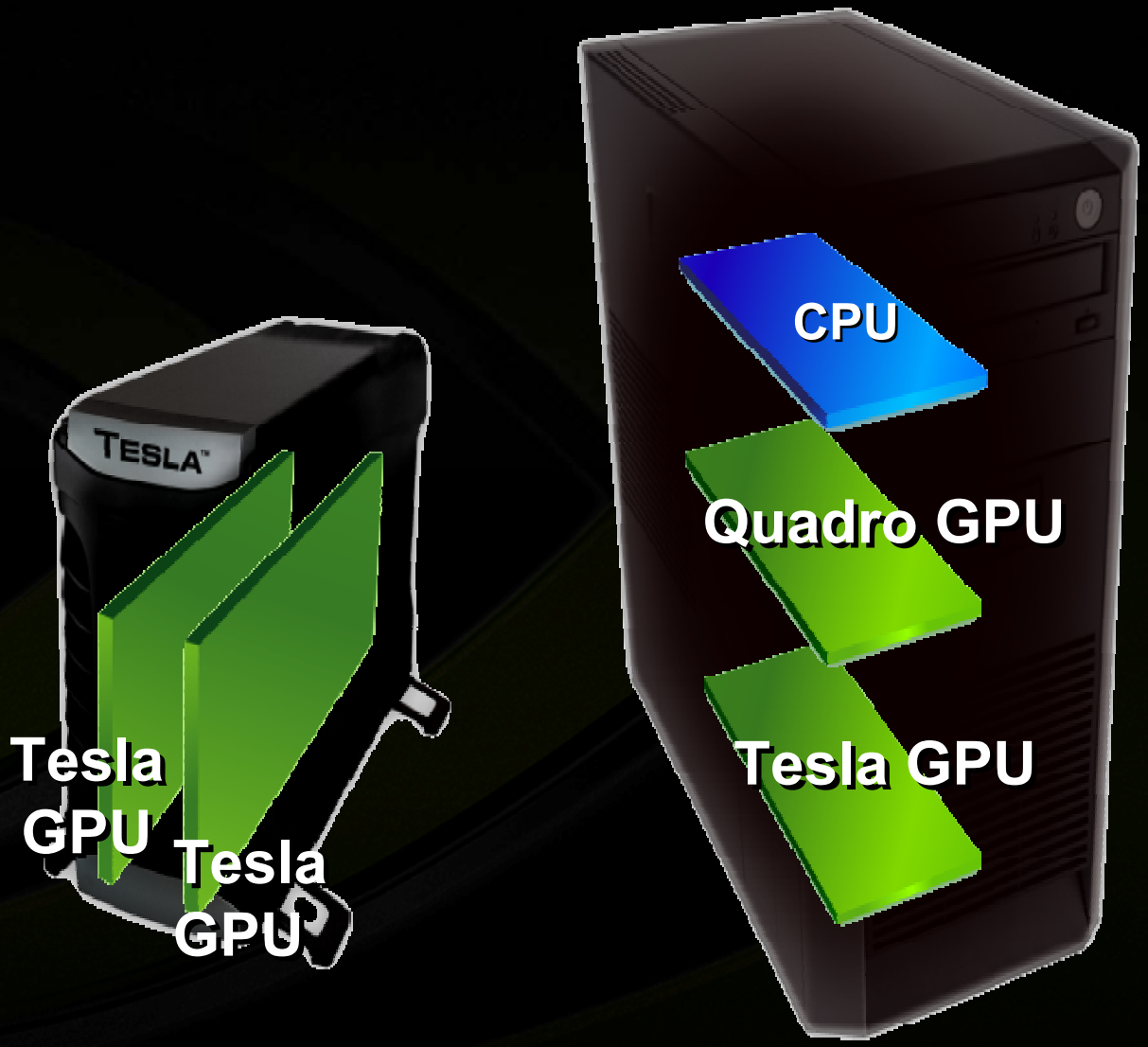
GeForce 8800

Tesla D870

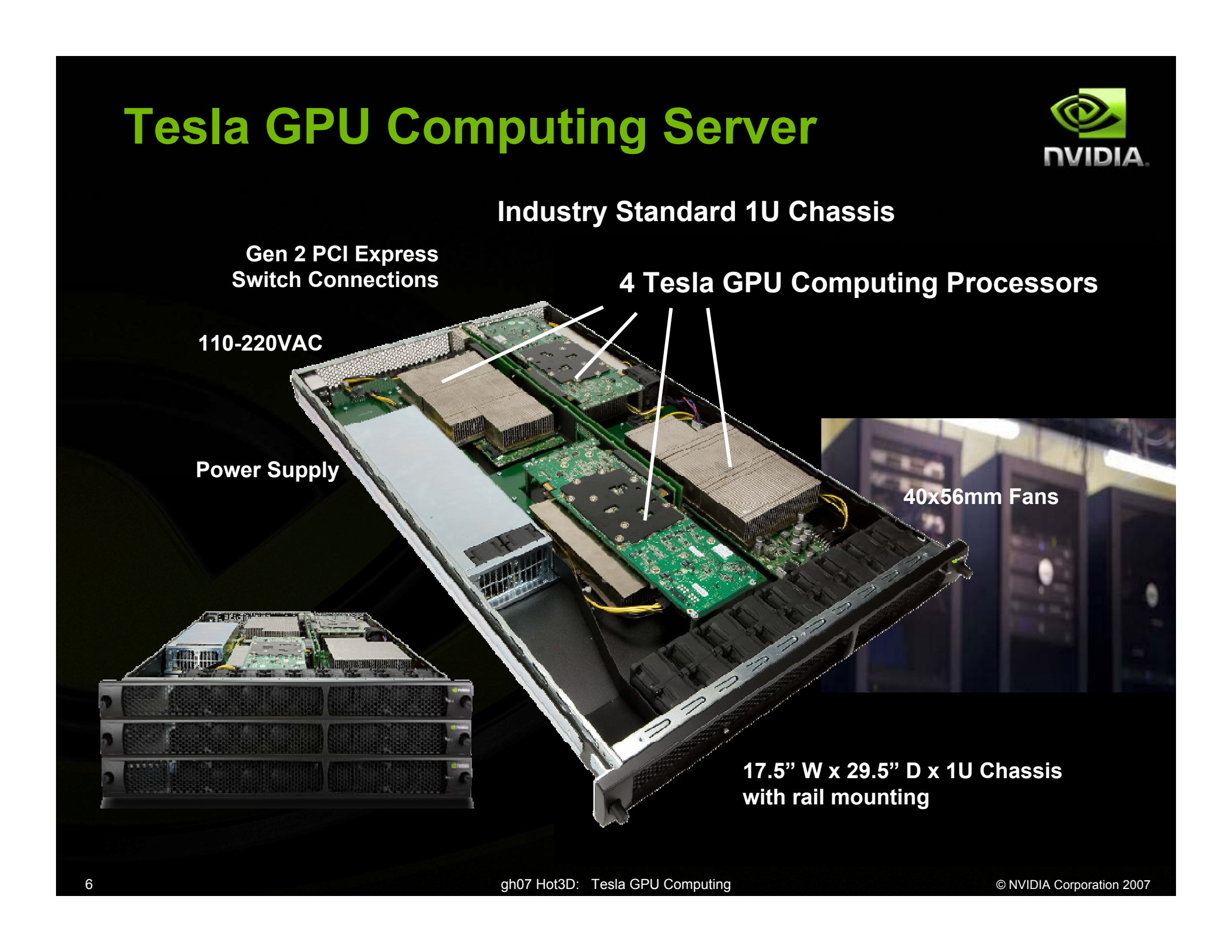


Tesla S870

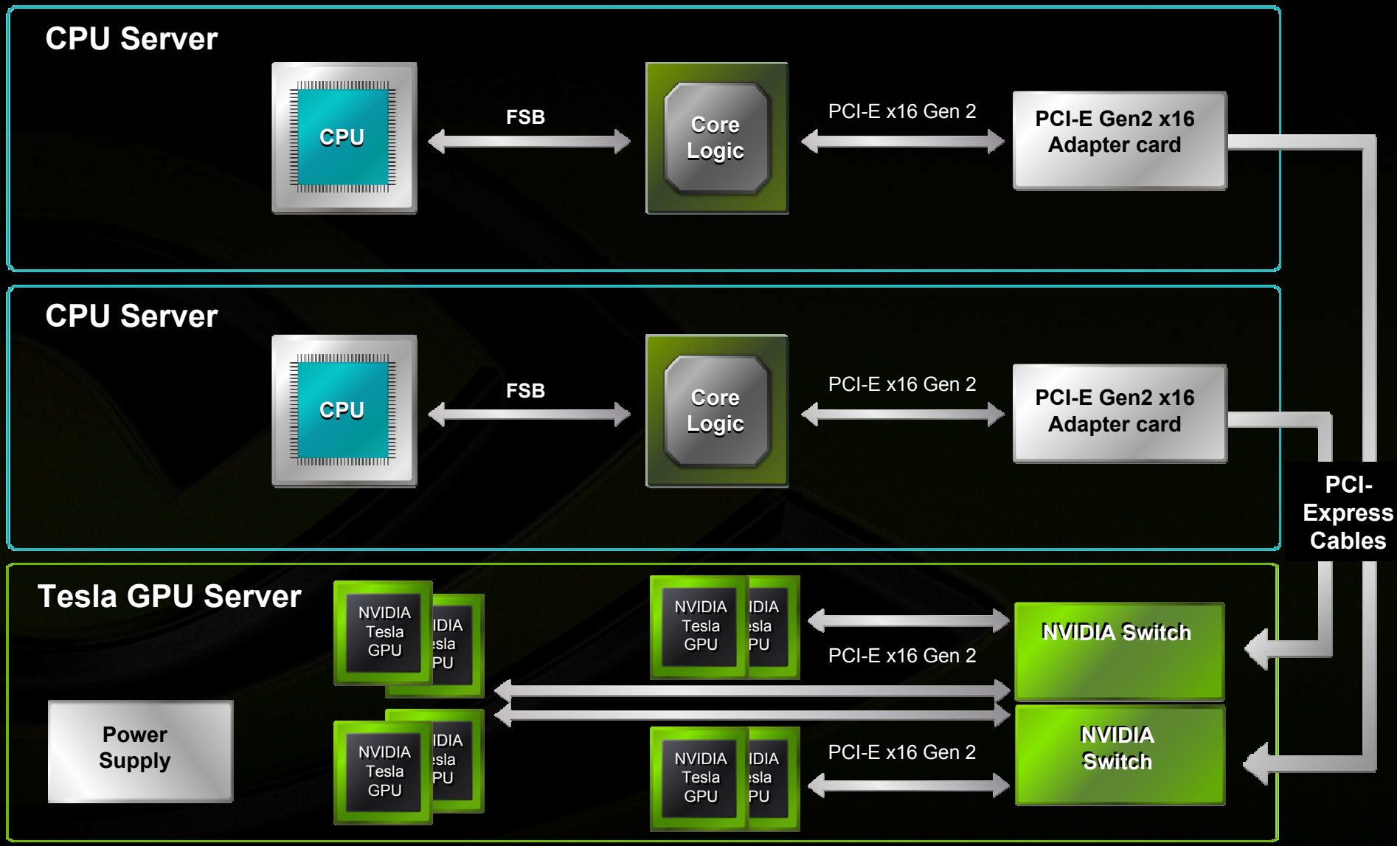
Workstation HPC



Tesla GPU Computing Server

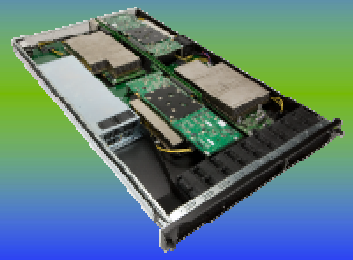


Tesla GPU Computing Server



Tesla System Solutions

Server



GPU Computing Server

Tesla S870
- **4** x 8-Series GPUs
- 550W typical (800W max)
- over 500 gigaflops per GPU
- 1U height

Work Station

Deskside Supercomputer

Tesla D870
- **2** x 8-Series GPU
- 550W Max
- over 500 gigaflops per GPU

GPU Computing Processor

Tesla C870
- **1** x 8-Series GPU
- 170W max
- over 500 gigaflops per GPU

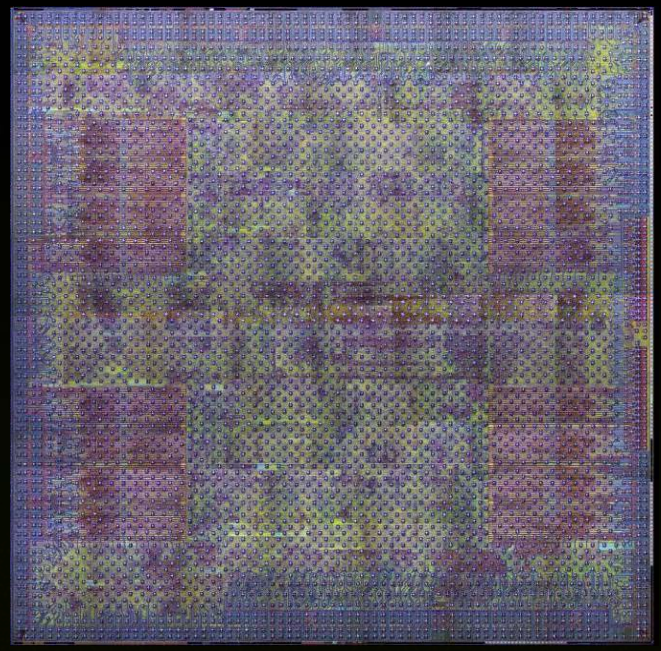
Tesla C870 GPU Implementation

- 681 million transistors
- 470 mm² in 90 nm CMOS

- 128 thread processors
- 518 GFLOPS peak
- 1.35 GHz processor clock

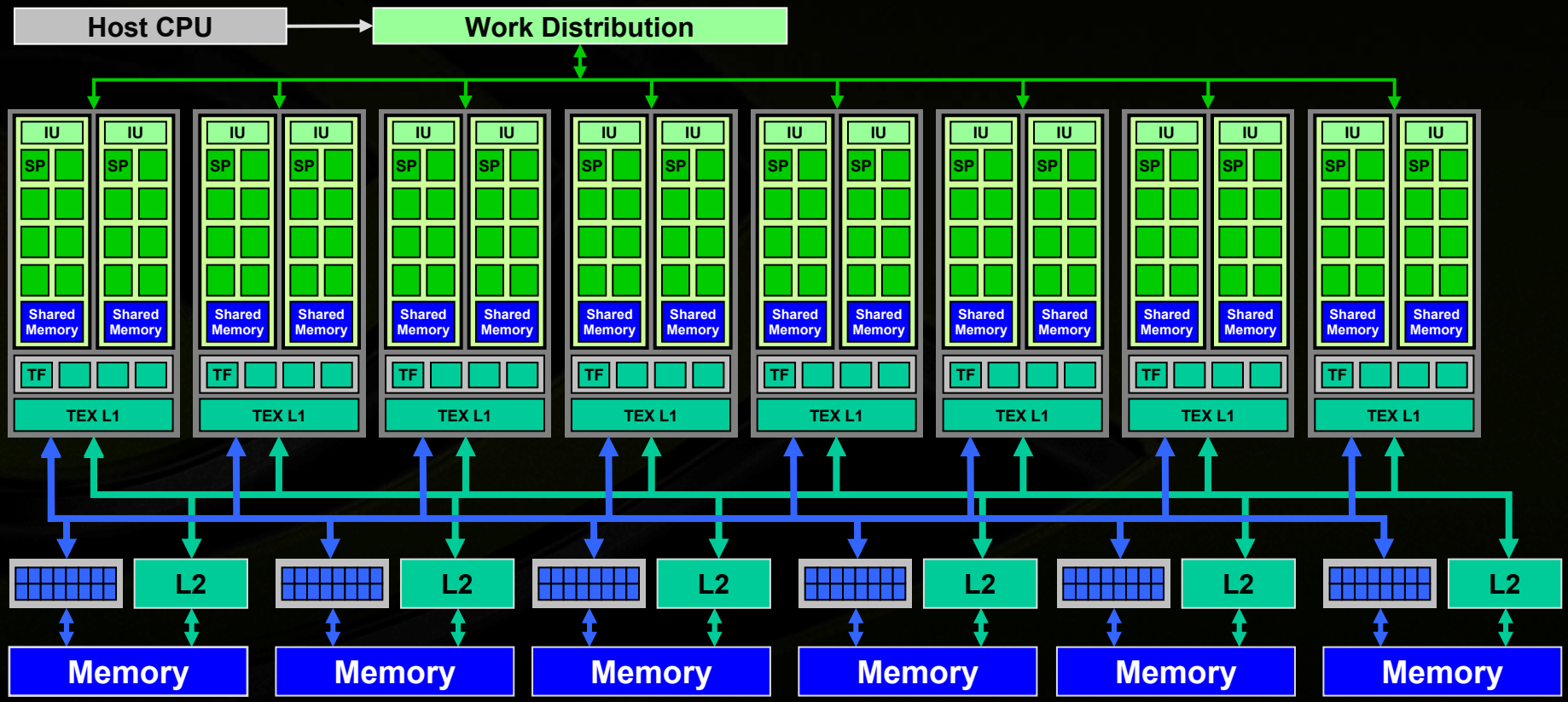
- 1.5 GB DRAM
- 76 GB/s peak
- 800 MHz GDDR3 clock
- 384 pin DRAM interface

- ATX form factor card
- PCI Express x16
- 170 W max with DRAM

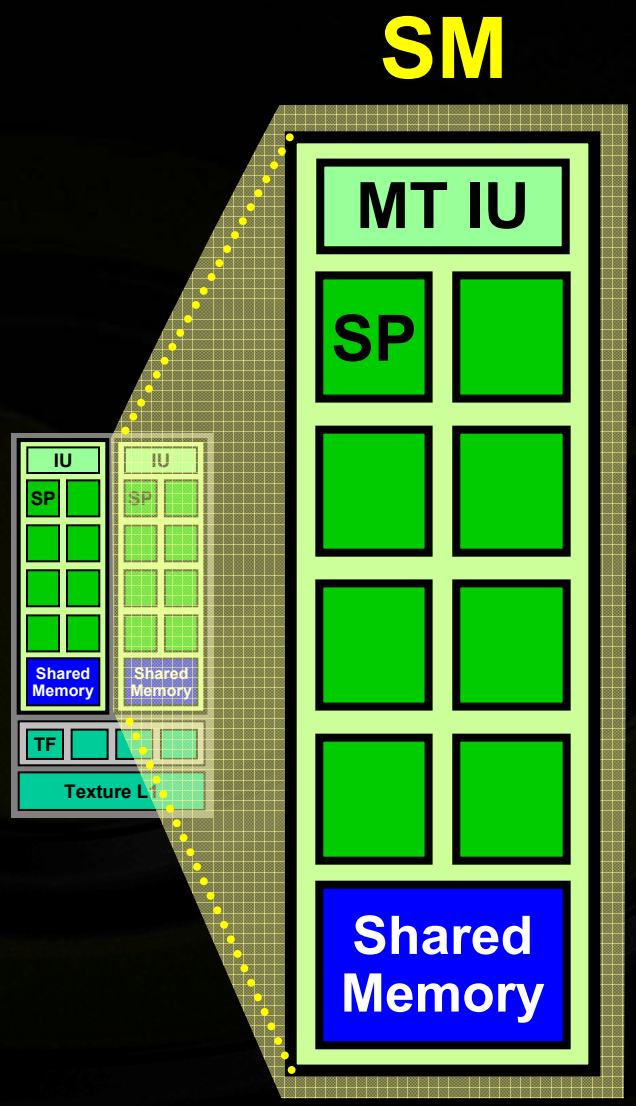


Tesla GPU Computing Processor

- Massively multithreaded parallel computing platform
- 12,288 concurrent threads, hardware managed
- 128 SP Thread Processor cores at 1.35 GHz == 518 GFLOPS peak
- GPU Computing features enable C on Graphics Processing Unit



SM Multithreaded Multiprocessor



- SM has 8 SP Thread Processors
  - 32 GFLOPS peak at 1.35 GHz
  - IEEE 754 32-bit floating point
  - 32-bit integer
- SM has 2 SFU Special Function Units
- Scalar ISA
  - Memory load/store
  - Texture fetch
  - Branch, call, return
  - Barrier synchronization instruction
- Multithreaded Instruction Unit
  - 768 Threads, hardware multithreaded
  - 24 SIMD warps of 32 threads
  - Independent MIMD thread execution
  - Hardware thread scheduling
- 16KB Shared Memory
  - Concurrent threads share data
  - Low latency load/store

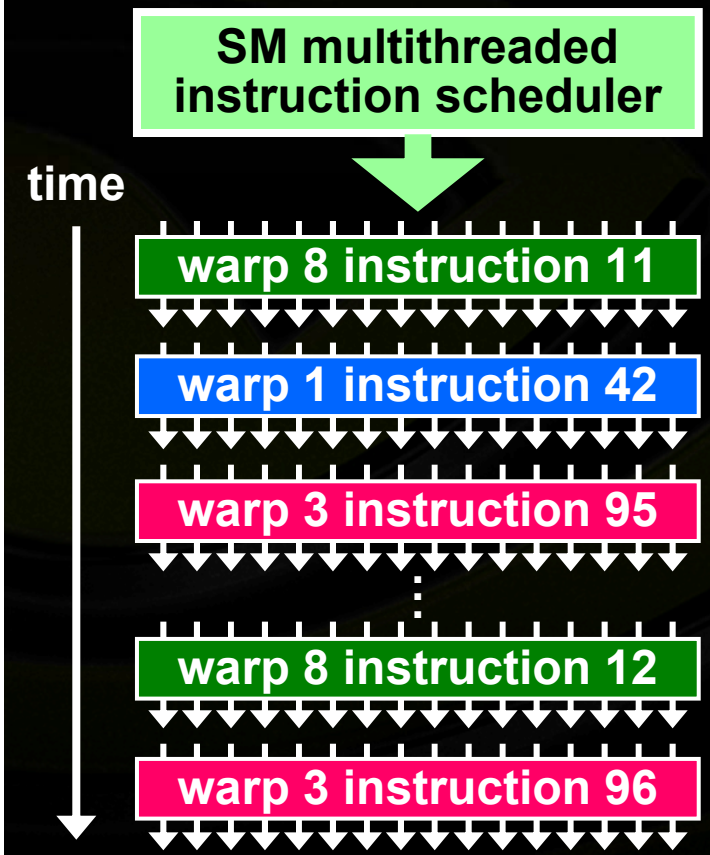
Thread Processor Datapath

- Executes 32-bit IEEE floating point instructions:
  - FADD, FMUL, FMAD, FMIN, FMAX, FSET, F2I, I2F
- Performs 32-bit integer instructions:
  - IADD, IMUL24, IMAD24, IMIN, IMAX, ISET, I2I
  - SHR, SHL, AND, OR, XOR
- Fully pipelined
  - Latency and area optimized
- IEEE 754 compliant FADD, FMUL
  - Round to nearest even, round toward zero
  - Handles special numbers, NaNs, infinities properly
  - Flushes denormal operands and results to zero

Special Function Unit (SFU)

- Executes transcendental function instructions
  - RCP, RSQRT, EXP2, LOG2, SIN, COS
  - 2 SFUs per SM yields ¼ instruction throughput
- Evaluates function approximations
  - Quadratic interpolation with Enhanced Minimax Approximation
  - Interpolates pixel attributes
- Accuracy ranges from 22.5 to 24.0 good bits
  - 1/x in the interval [1,2) is 24 bits, 1 ulp

SM SIMD Multithreaded Execution



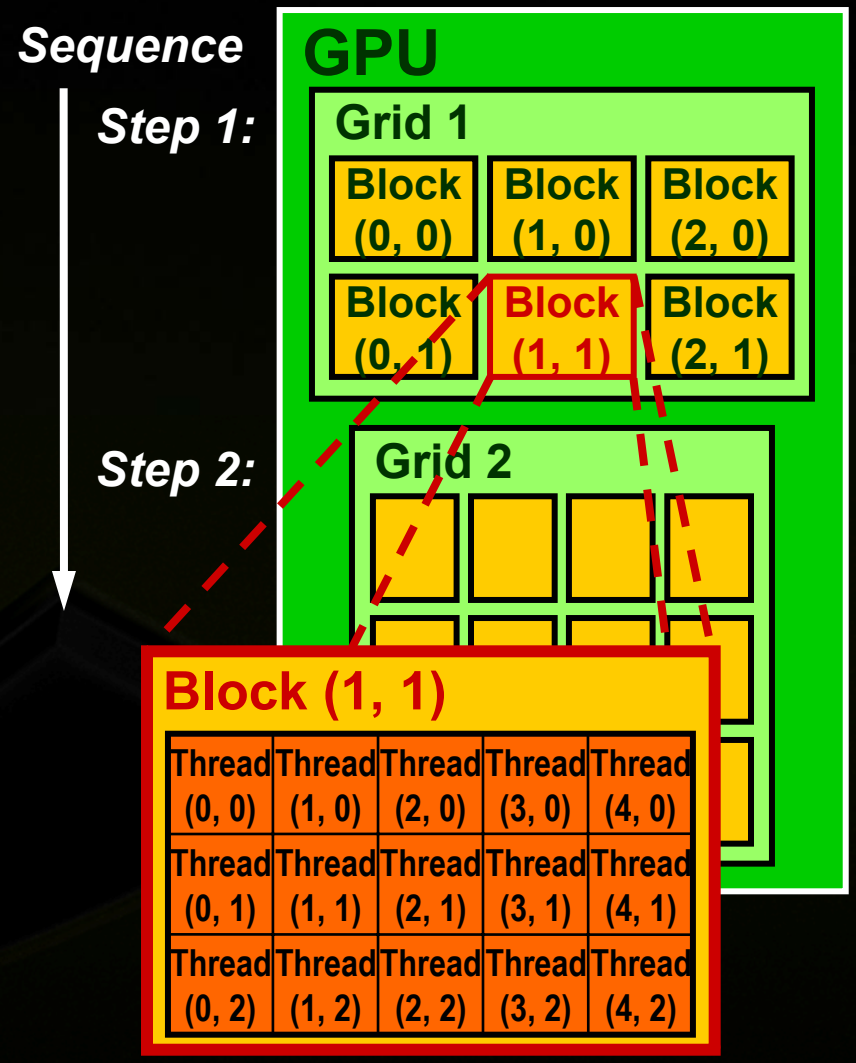
- Weaving: the original parallel thread technology is about 10,000 years old
- **Warp**: the set of 32 parallel threads that execute a SIMD instruction

- SM hardware implements zero-overhead warp and thread scheduling
- Each SM executes up to 768 concurrent threads, as 24 SIMD warps of 32 threads

- Threads can execute independently
- SIMD warp diverges and converges when threads branch independently
- Best efficiency and performance when threads of a warp execute together
- **SIMD across threads (not just data)** gives easy single-thread scalar programming with SIMD efficiency

Programmer Partitions Problem with Data-Parallel Decomposition

- CUDA Programmer partitions problem into Grids, one Grid per sequential problem step
- Programmer partitions Grid into result Blocks computed independently in parallel
- GPU thread array computes result Block
- Programmer partitions Block into elements computed cooperatively in parallel
- GPU thread computes result element

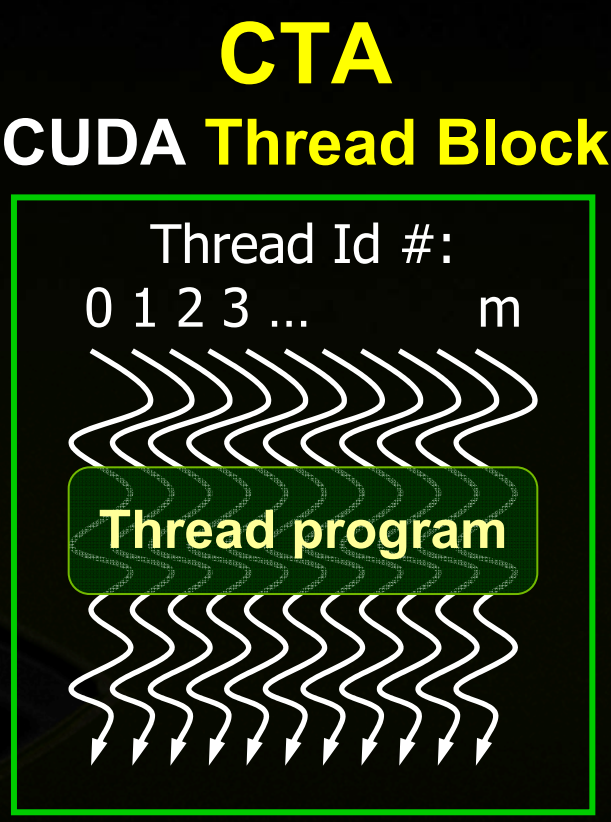


Cooperative Thread Array
CTA Implements CUDA Thread Block

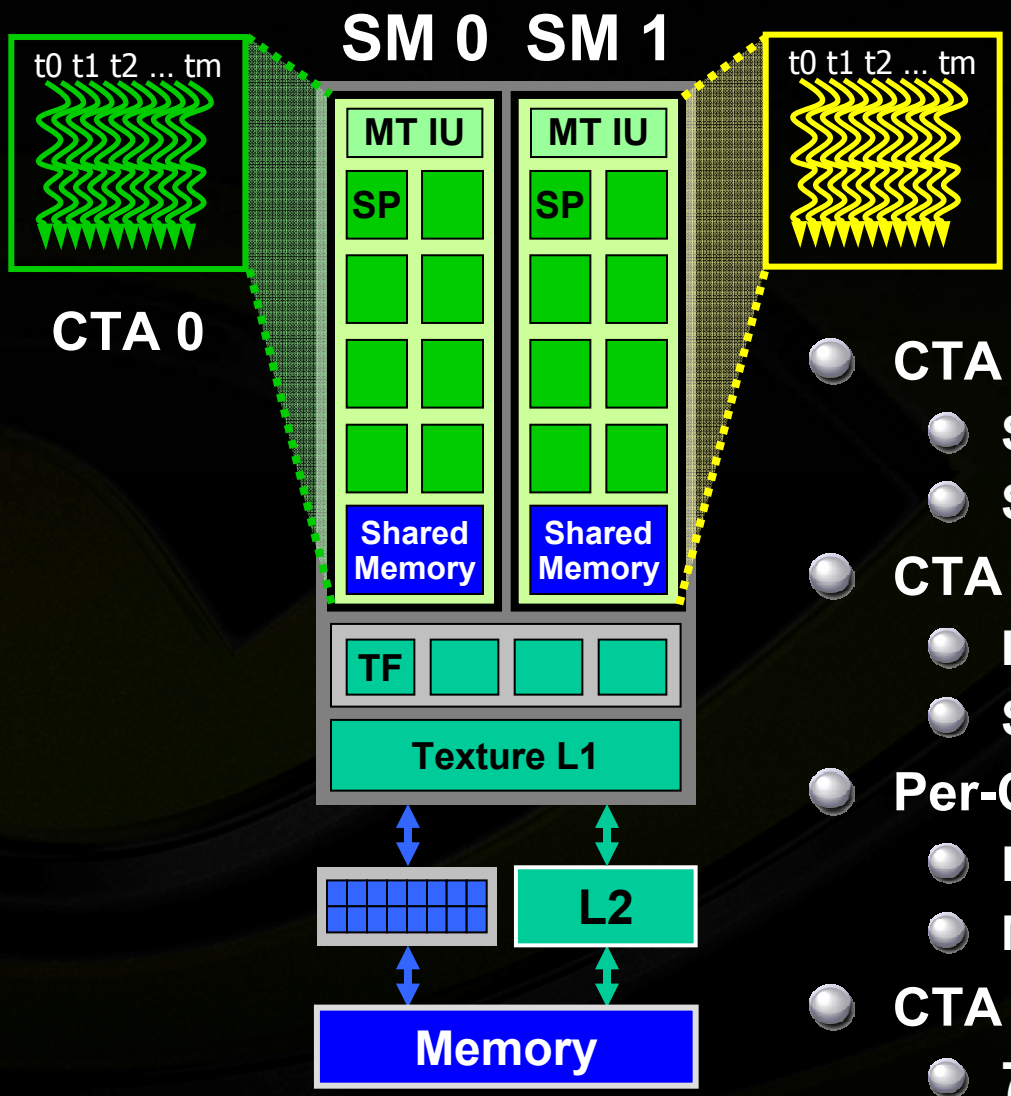
- A CTA is an array of concurrent threads that cooperate to compute a result
- A CUDA thread block is a CTA

- Programmer declares CTA:
  - CTA size 1 to 512 concurrent threads
  - CTA shape 1D, 2D, or 3D
  - CTA dimensions in threads

- CTA threads execute thread program
- CTA threads have thread id numbers
- CTA threads share data and synchronize
- Thread program uses thread id to select work and address shared data

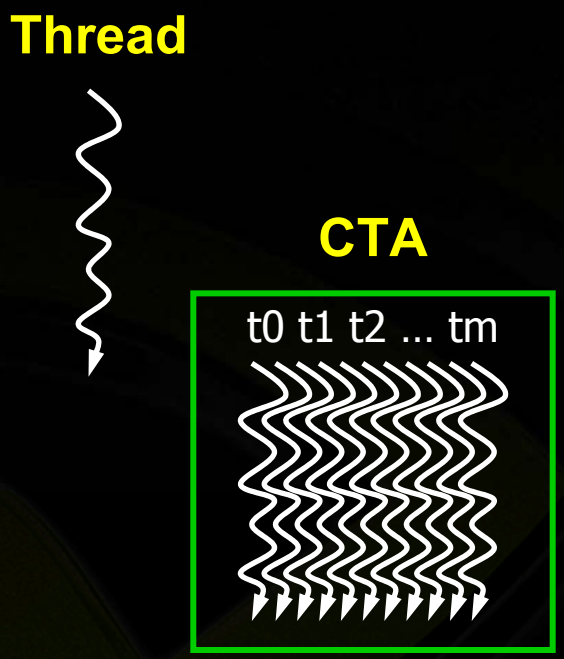


SM Multiprocessor Executes CTAs



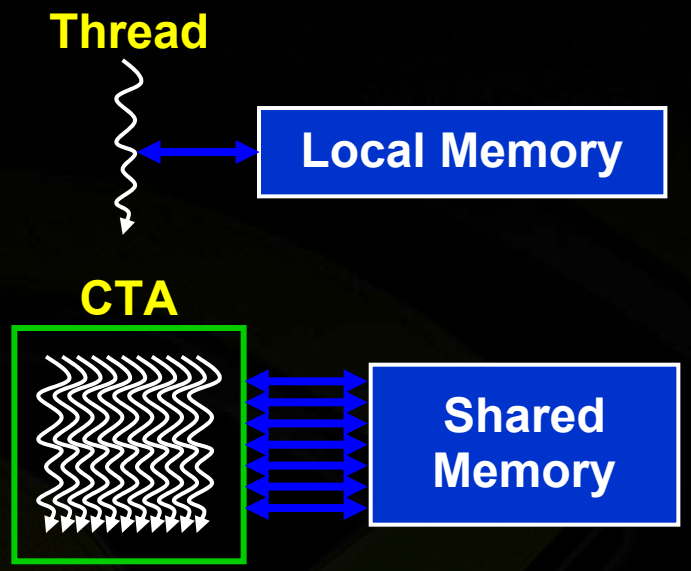
- CTA threads run concurrently
  - SM assigns thread id #s
  - SM manages thread execution
- CTA threads share data & results
  - In Memory and Shared Memory
  - Synchronize at barrier instruction
- Per-CTA Shared Memory
  - Keeps data close to processor
  - Minimize trips to global Memory
- CTA threads access global Memory
  - 76 GB/sec GDDR DRAM

Data Parallel Levels

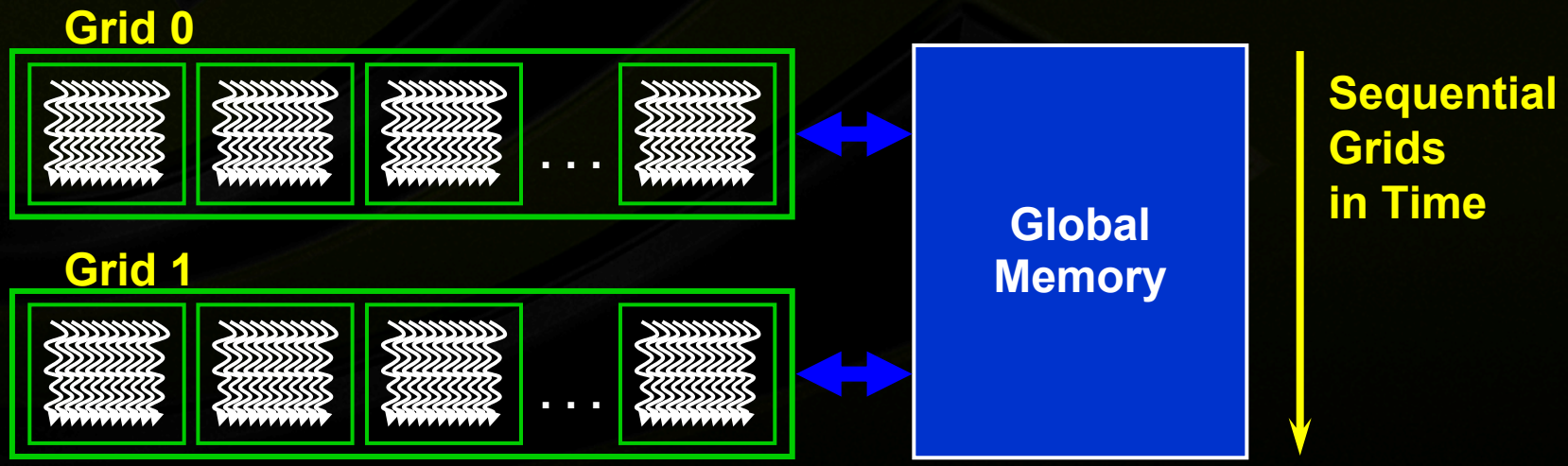


- Thread
  - Computes result elements
  - Thread id number
- CTA – Cooperative Thread Array
  - Computes result Block
  - 1 to 512 threads per CTA
  - CTA (Block) id number
- Grid of CTAs
  - Computes many result Blocks
  - 1 to many CTAs per Grid
- Sequential Grids
  - Compute sequential problem steps

Parallel Memory Sharing



- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-CTA
  - Shared by threads of CTA
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication



How to Scale GPU Computing?

- GPU parallelism varies widely
  - Ranges from 8 cores to many 100s of cores
  - Ranges from 100 to many 1000s of threads
  - GPU parallelism doubles yearly
- Graphics performance scales with GPU parallelism
  - Data parallel mapping of pixels to threads
  - Unlimited demand for parallel pixel shader threads and cores

Challenge:

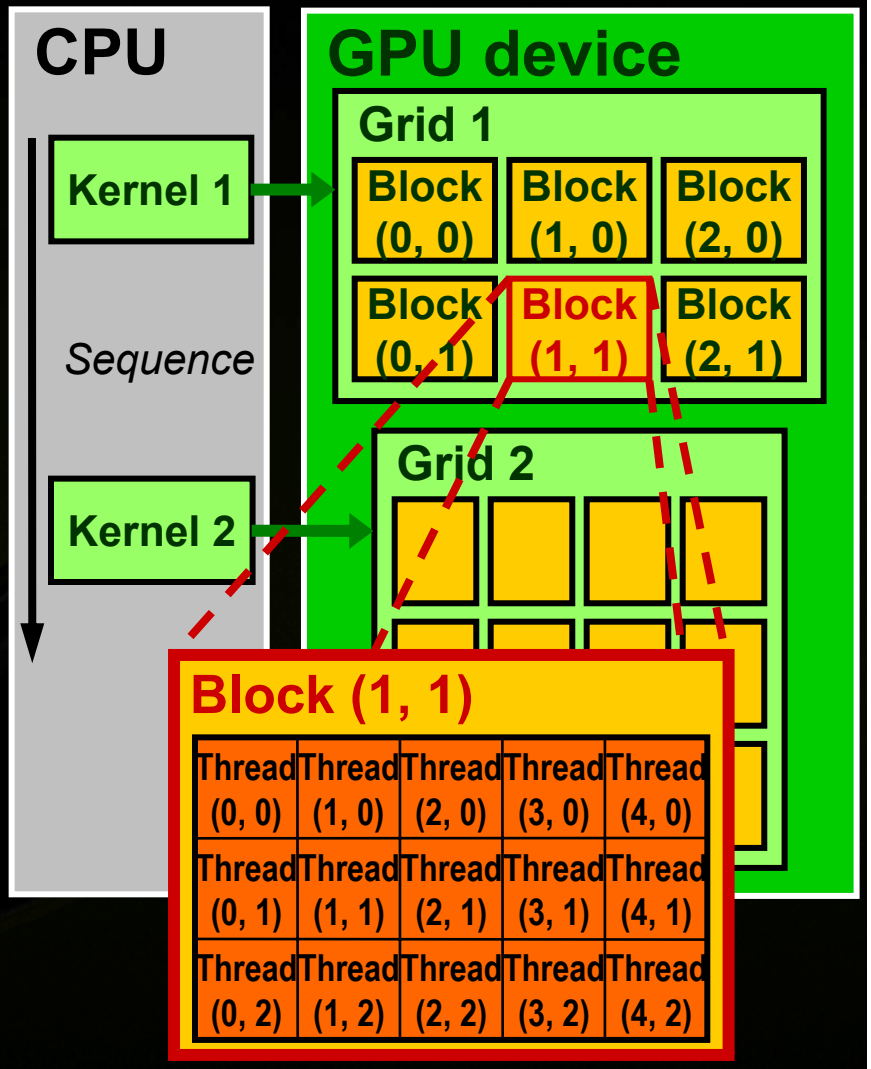
- Scale Computing performance with GPU parallelism
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling

Transparent Scalability

- Programmer uses multi-level data parallel decomposition
  - Decomposes problem into sequential steps (Grids)
  - Decomposes Grid into computing parallel Blocks (CTAs)
  - Decomposes Block into computing parallel elements (threads)
- GPU hardware distributes CTA work to available SM cores
  - GPU balances CTA work load across any number of SM cores
  - SM core executes CTA program that computes Block
- CTA program computes a Block independently of others
  - Enables parallel computing of Blocks of a Grid
  - No communication among Blocks of same Grid
  - Scales one program across any number of parallel SM cores
- Programmer writes one program for all GPU sizes
- Program does not know how many cores it uses
- Program executes on GPU with any number of cores

CUDA Programming Model: Grids, Blocks, and Threads

- Execute a sequence of kernels on GPU computing device
- A kernel executes as a Grid of thread blocks
- A thread block is an array of threads that can cooperate
- Threads within the same block synchronize and share data in Shared Memory
- Execute thread blocks as CTAs on multithreaded multiprocessor SM cores

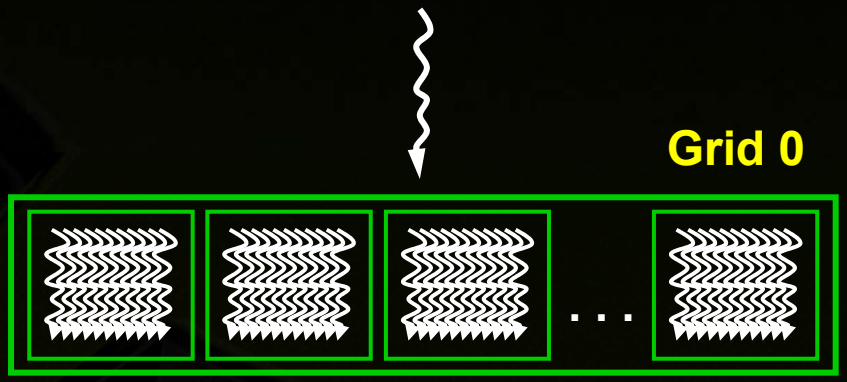


Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks

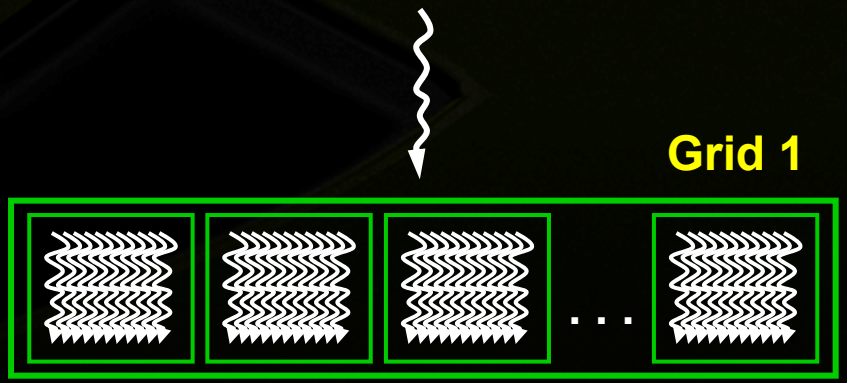
CPU Serial Code

GPU Parallel Kernel
KernelA<<< nBlk, nTid >>>(args);

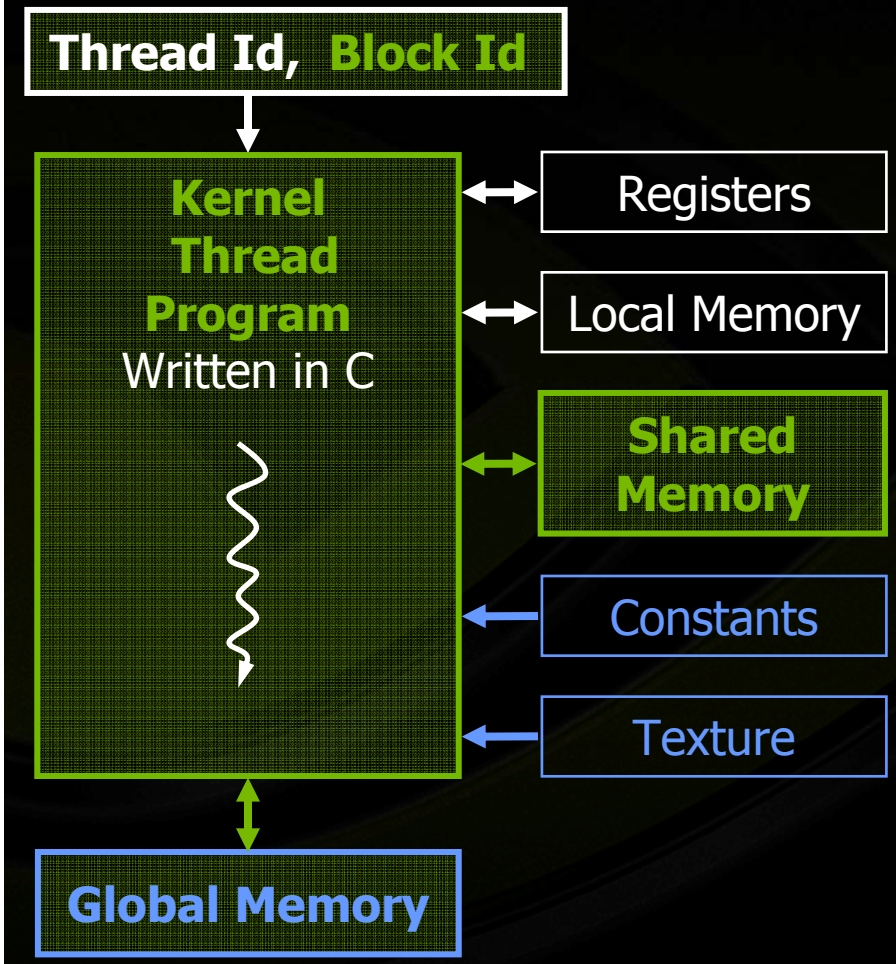


CPU Serial Code

GPU Parallel Kernel
KernelB<<< nBlk, nTid >>>(args);



CUDA Programming Model: Thread Memory Spaces



- Each kernel thread can read:
  - Thread Id — per thread
  - Block Id — per block
  - Constants — per grid
  - Texture — per grid
- Each thread can read and write:
  - Registers — per thread
  - Local memory — per thread
  - Shared memory — per block
  - Global memory — per grid
- Host CPU can read and write:
  - Constants — per grid
  - Texture — per grid
  - Global memory — per grid

Summary

- Tesla GPU Computing Architecture
  - Architecture enables parallel C on GPUs
  - Massively multithreaded – 1000s of threads
  - Executes parallel threads and thread arrays
  - Threads cooperate via Shared and Global memory
  - Scales to any number of parallel processor cores
  - Now on: Tesla C870, D870, S870, GeForce 8800/8600/8500, and Quadro FX 5600/4600
- CUDA Programming model
  - C program for GPU threads
  - Scales transparently to GPU parallelism
  - Compiler, tools, libraries, and driver for GPU Computing

http://www.nvidia.com/Tesla
http://developer.nvidia.com/CUDA