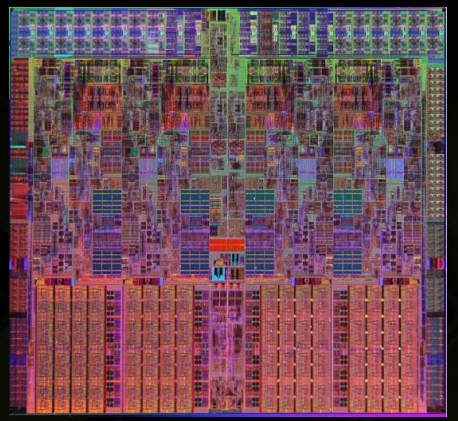
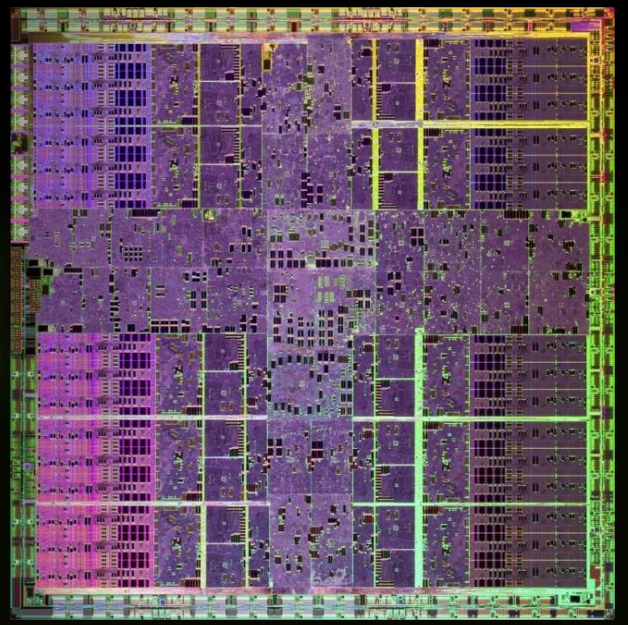
HIGH-PERFORMANCE COMPUTING
WITH CUDA AND TESLA GPUS
Timothy Lanfear, NVIDIA
NVIDIA

What is GPU Computing?

© NVIDIA Corporation 2009

What is GPU Computing?





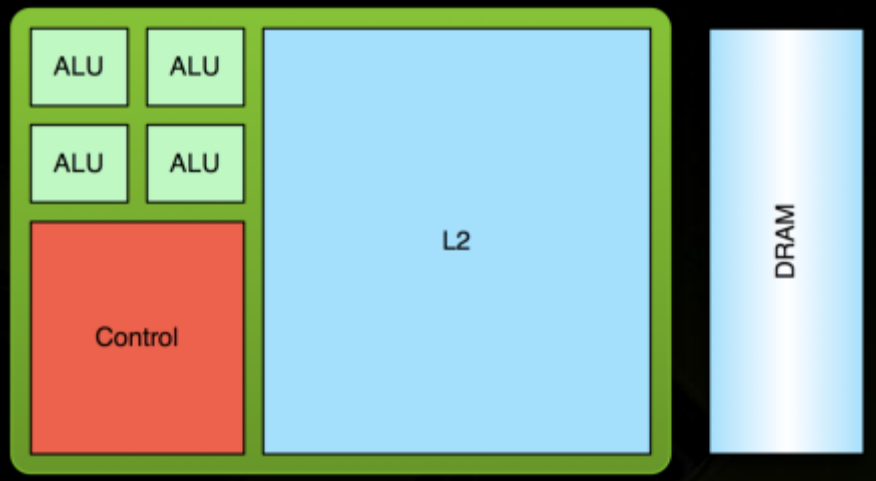
x86 PCIe bus GPU

Computing with CPU + GPU

Heterogeneous Computing

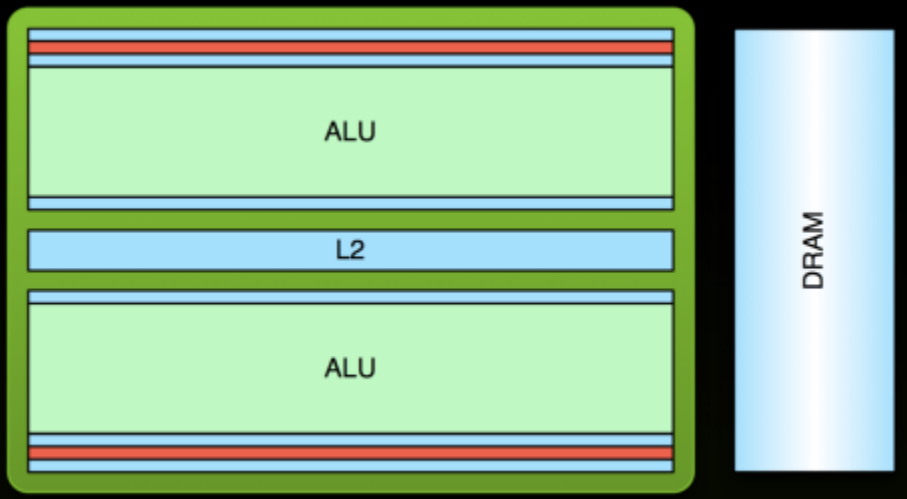
© NVIDIA Corporation 2009

Low Latency or High Throughput?



CPU

- Optimised for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution



GPU

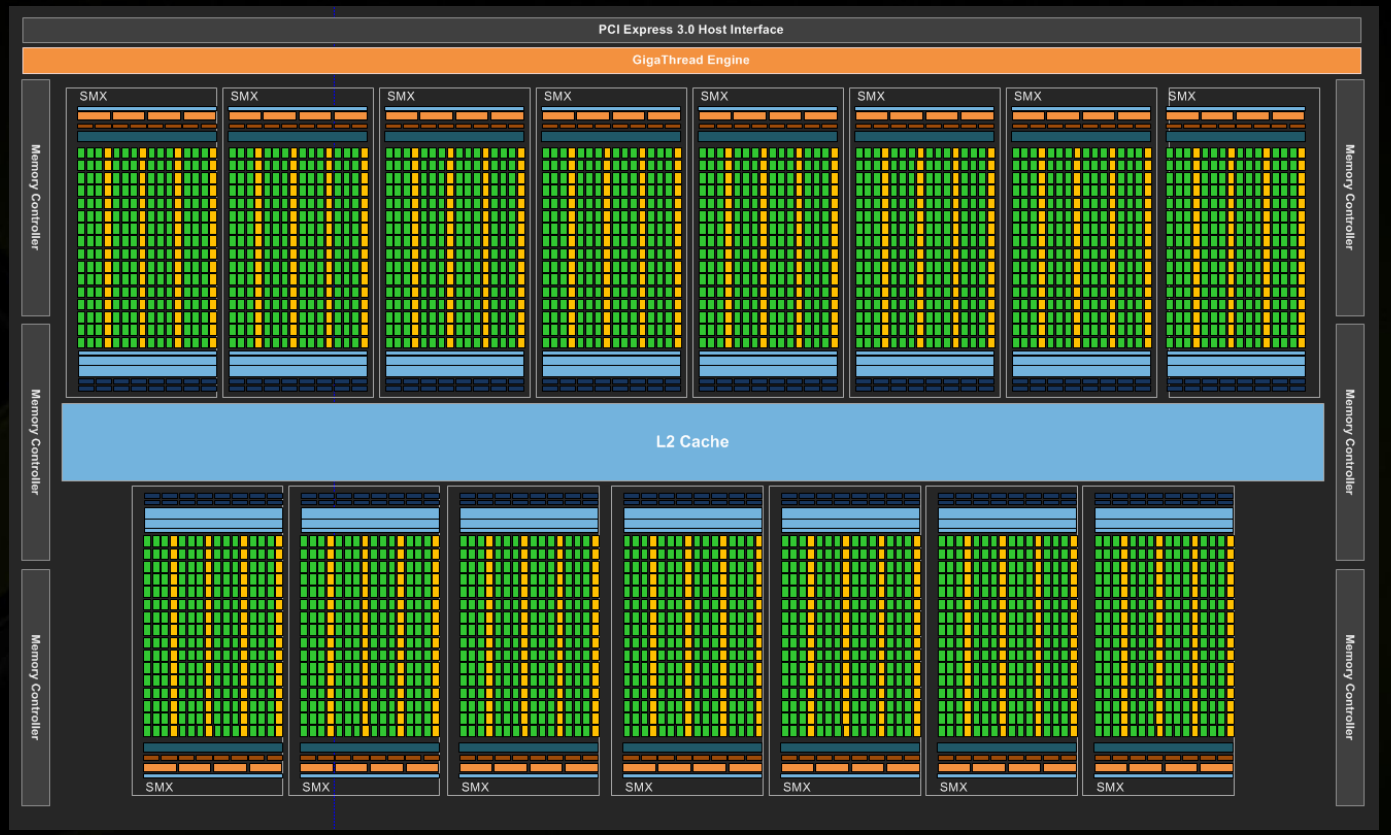
- Optimised for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

© NVIDIA Corporation 2009

Kepler GK110 Block Diagram

Architecture

- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3



© NVIDIA Corporation 2009

Tesla K20: Same Power, 3x DP Perf of Fermi

Product Name	M2090	K20	K20X
Peak Single Precision Peak SGEMM	**1.3 Tflops** 0.87 TF	**3.52 TF** 2.61 TF	**3.95 TF** 2.90 TF
Peak Double Precision Peak DGEMM	**0.66 Tflops** 0.43 TF	**1.17 TF** 1.10 TF	**1.32 TF** 1.22 TF
Memory size	6 GB	5 GB	6 GB
Memory BW (ECC off)	177.6 GB/s	208 GB/s	250 GB/s
PCIe Gen	Gen2	Gen2	Gen2
New CUDA Features	-	GPUDirect w/ RDMA, Hyper-Q, Dynamic Parallelism	
ECC Features	DRAM, Caches & Reg Files		
# CUDA Cores	512	2496	2688
Total Board Power	225W	225W	235W

3x Double Precision

Hyper-Q, Dynamic Parallelism

CFD, FEA, Finance, Physics

© NVIDIA Corporation 2009

Tesla K10: Same Power, 2x Perf of Fermi

Product Name	M2090	K10	
GPU Architecture	Fermi	Kepler GK104	
# of GPUs	1	2	
		Board	Per GPU
Single Precision Flops	**1.3 TF**	**4.58 TF**	**2.29 TF**
Double Precision Flops	0.66 TF	0.190 TF	0.095 TF
# CUDA Cores	512	3072	1536
Memory size	**6 GB**	**8 GB**	**4GB**
Memory BW (ECC off)	**177.6 GB/s**	**320 GB/s**	**160GB/s**
PCI-Express	Gen 2: 8 GB/s	Gen 3: 16 GB/s	

3x Single Precision

1.8x Memory Bandwidth

Seismic, Imaging, Signal, Molecular Dynamics

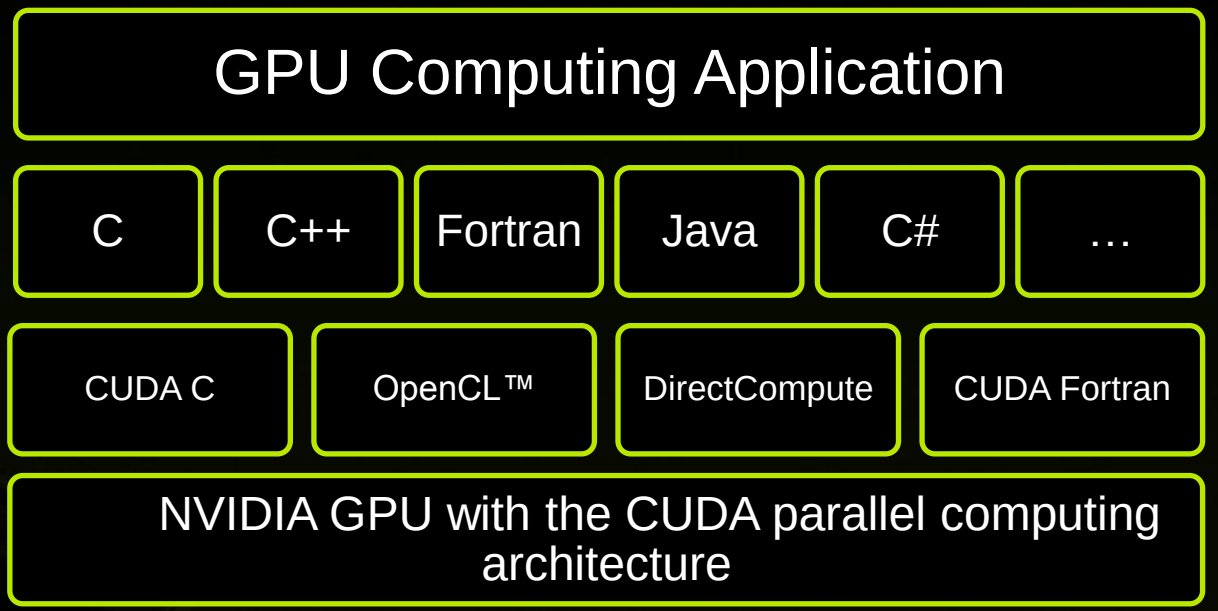
© NVIDIA Corporation 2009

CUDA Architecture

© NVIDIA Corporation 2009

CUDA Parallel Computing Architecture

- **Parallel computing architecture and programming model**
- **Includes a CUDA C compiler, support for OpenCL and DirectCompute**
- **Architected to natively support multiple computational interfaces (standard languages and APIs)**

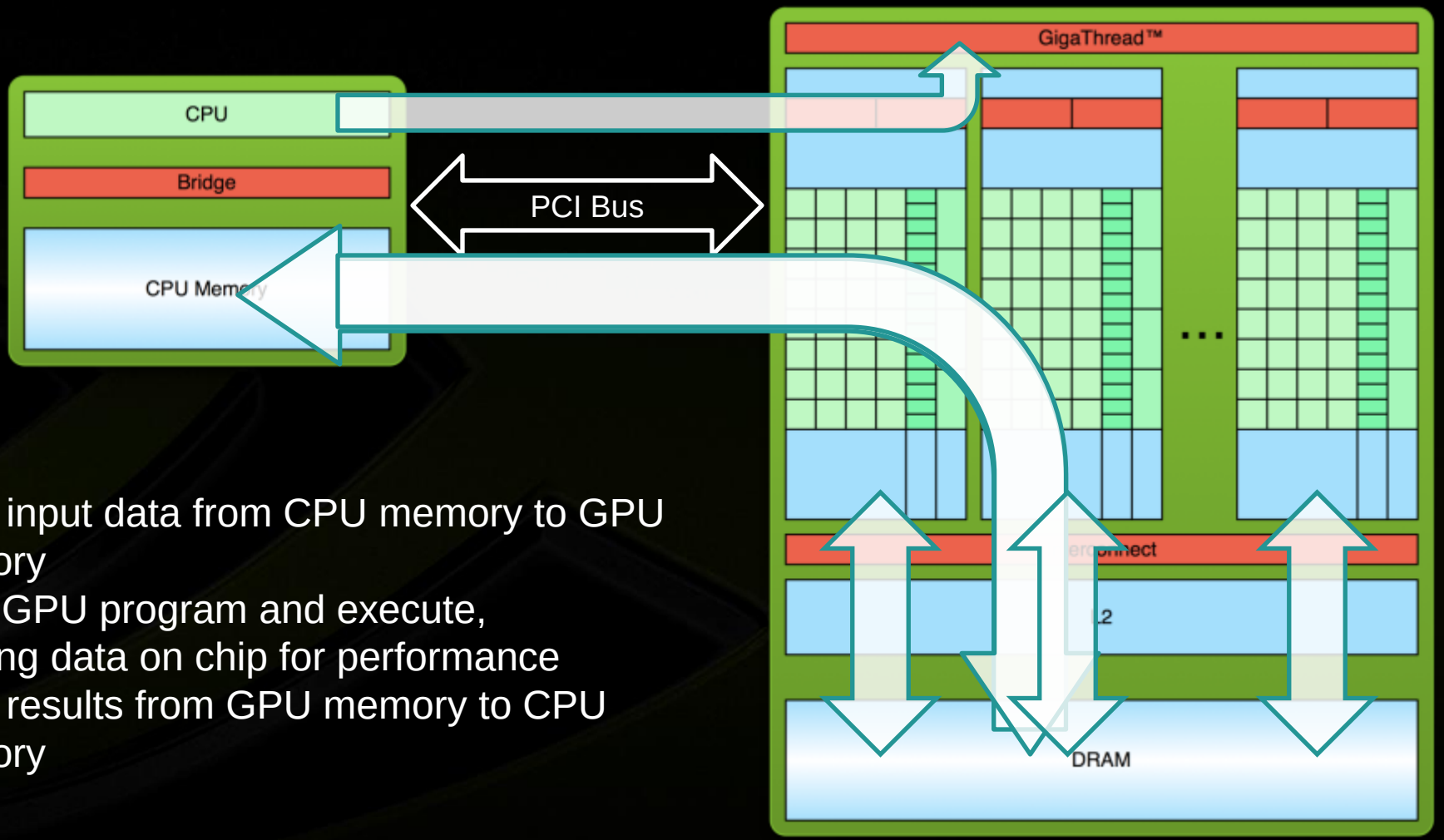


© NVIDIA Corporation 2009

CUDA Programming Model

© NVIDIA Corporation 2009

Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

© NVIDIA Corporation 2009

CUDA Kernels

- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads

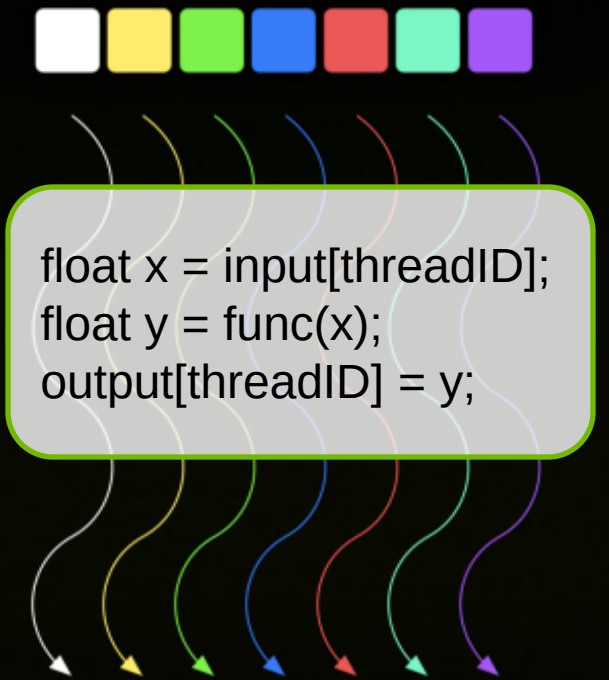
- CUDA threads:
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously

CPU	Host	Executes functions
GPU	Device	Executes kernels

© NVIDIA Corporation 2009

CUDA Kernels: Parallel Threads

- A **kernel** is an array of threads, executed in parallel
- All threads execute the same code
- Each thread has an ID
  - Select input/output data
  - Control decisions



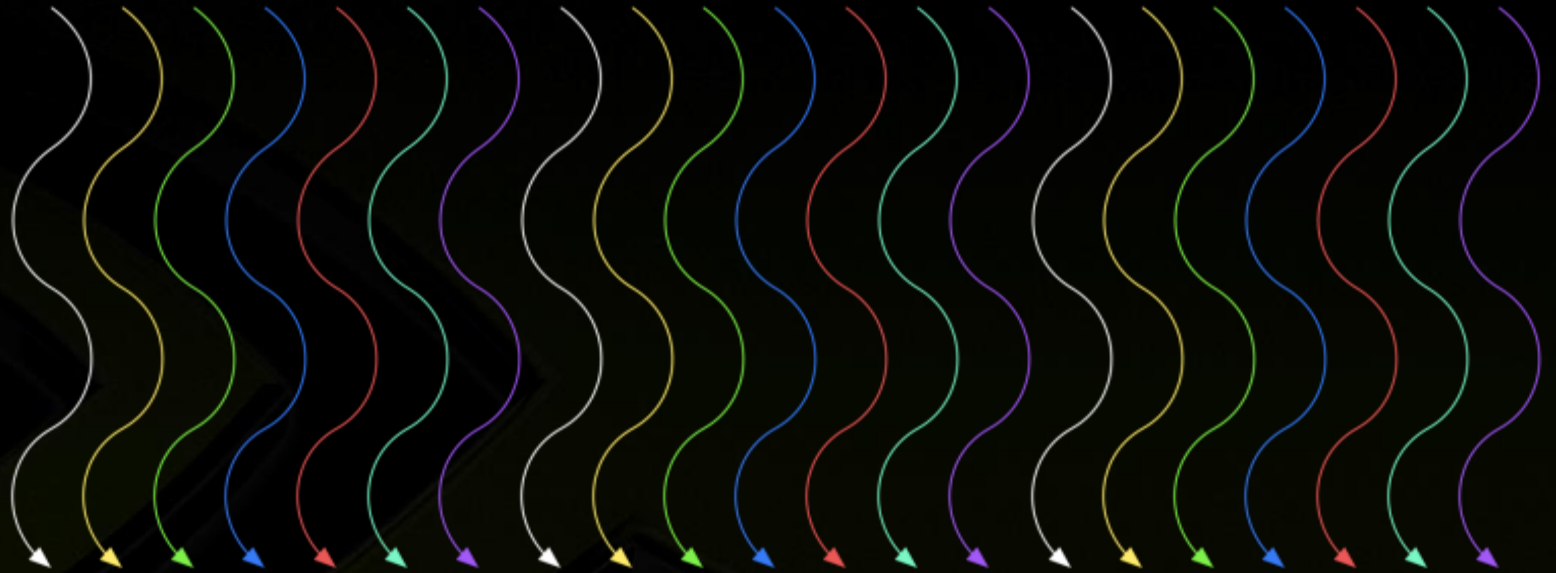
© NVIDIA Corporation 2009

Key Idea of CUDA

- Write a single-threaded program parameterized in terms of the thread ID.
- Use the thread ID to select a subset of the data for processing, and to make control flow decisions.
- Launch a number of threads, such that the ensemble of threads processes the whole data set.

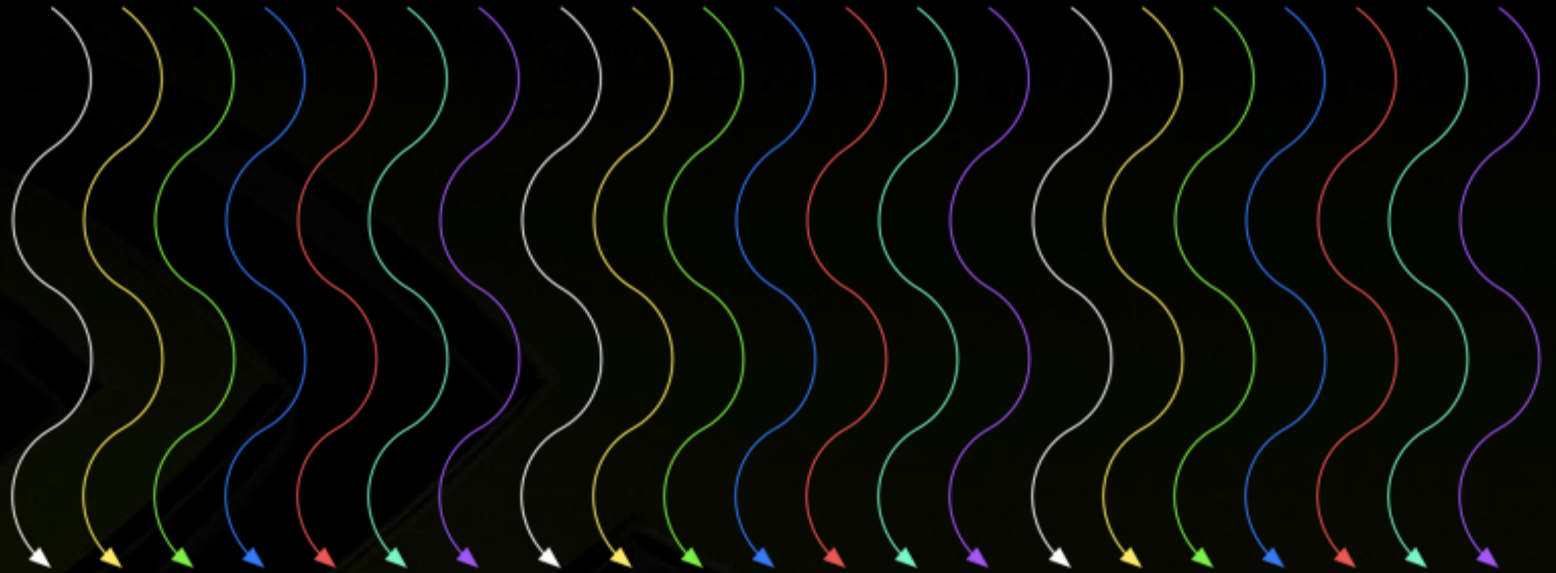
© NVIDIA Corporation 2009

CUDA Kernels: Subdivide into Blocks



© NVIDIA Corporation 2009

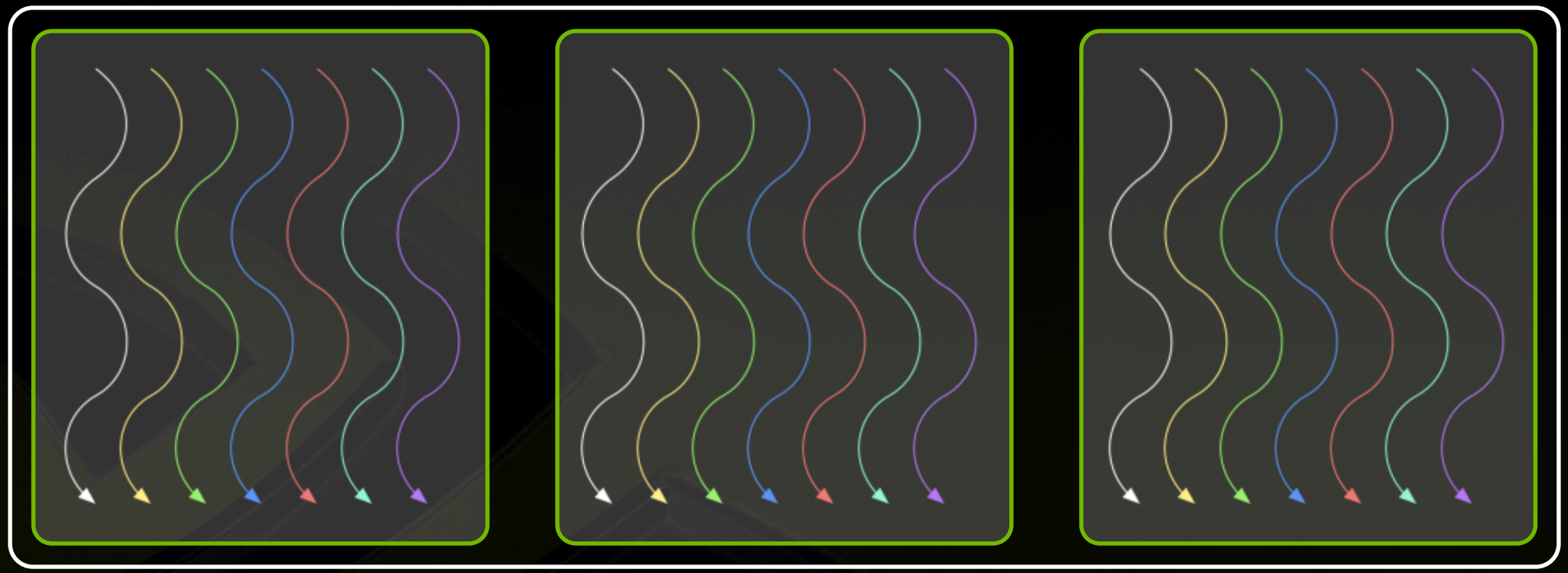
CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks

© NVIDIA Corporation 2009

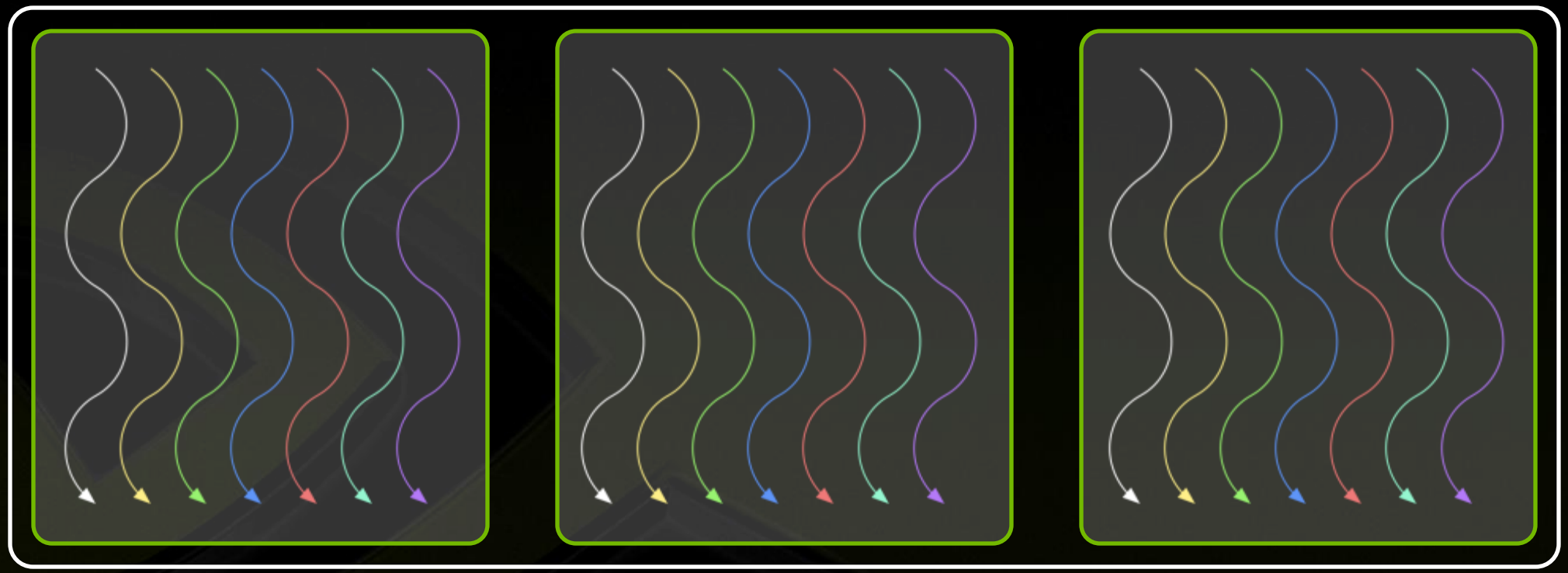
CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid

© NVIDIA Corporation 2009

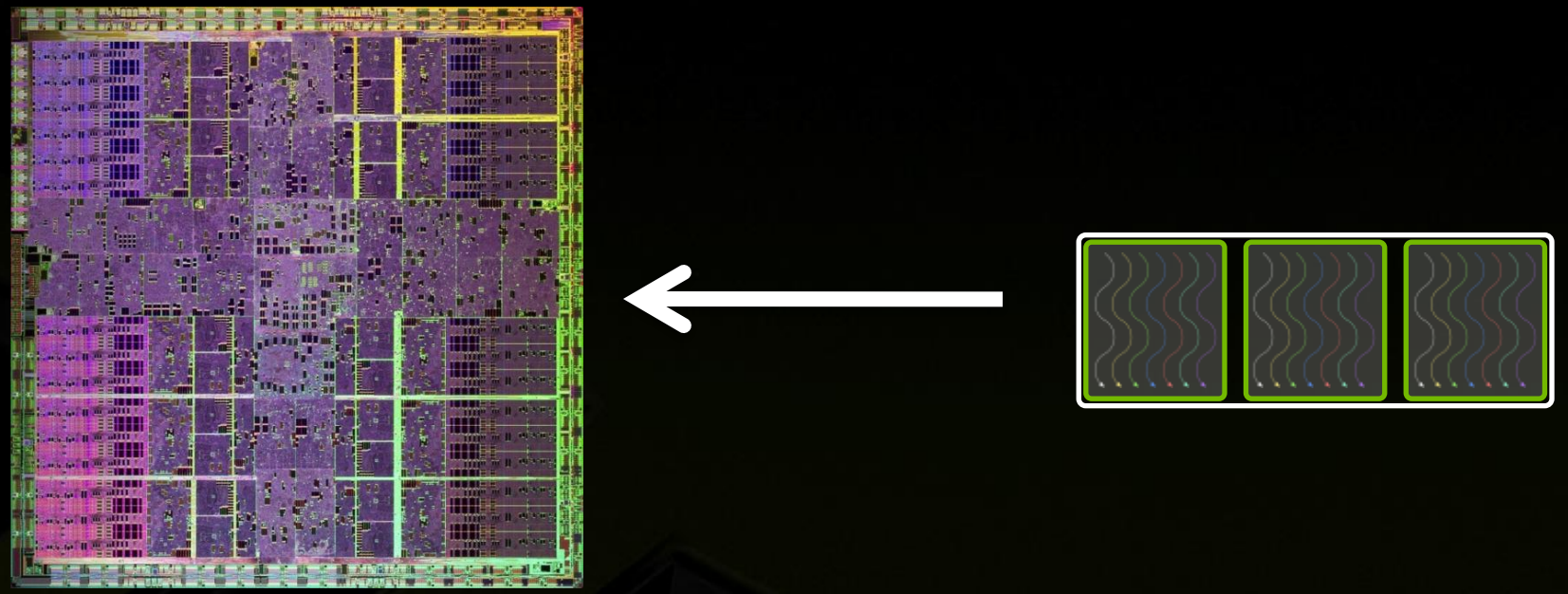
CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

© NVIDIA Corporation 2009

CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

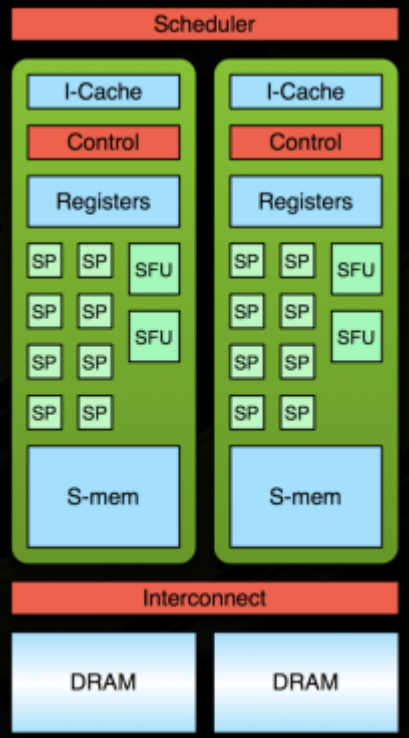
© NVIDIA Corporation 2009

Communication Within a Block

- Threads may need to cooperate
  - Memory accesses
  - Share results
- Cooperate using shared memory
  - Accessible by all threads within a block
- Restriction to “within a block” permits scalability
  - Fast communication between N threads is not feasible when N large

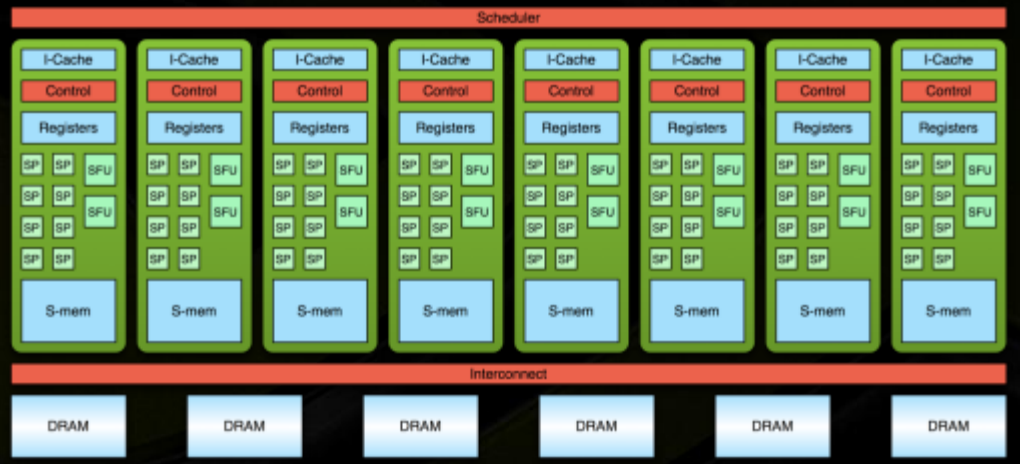
© NVIDIA Corporation 2009

Transparent Scalability – G84



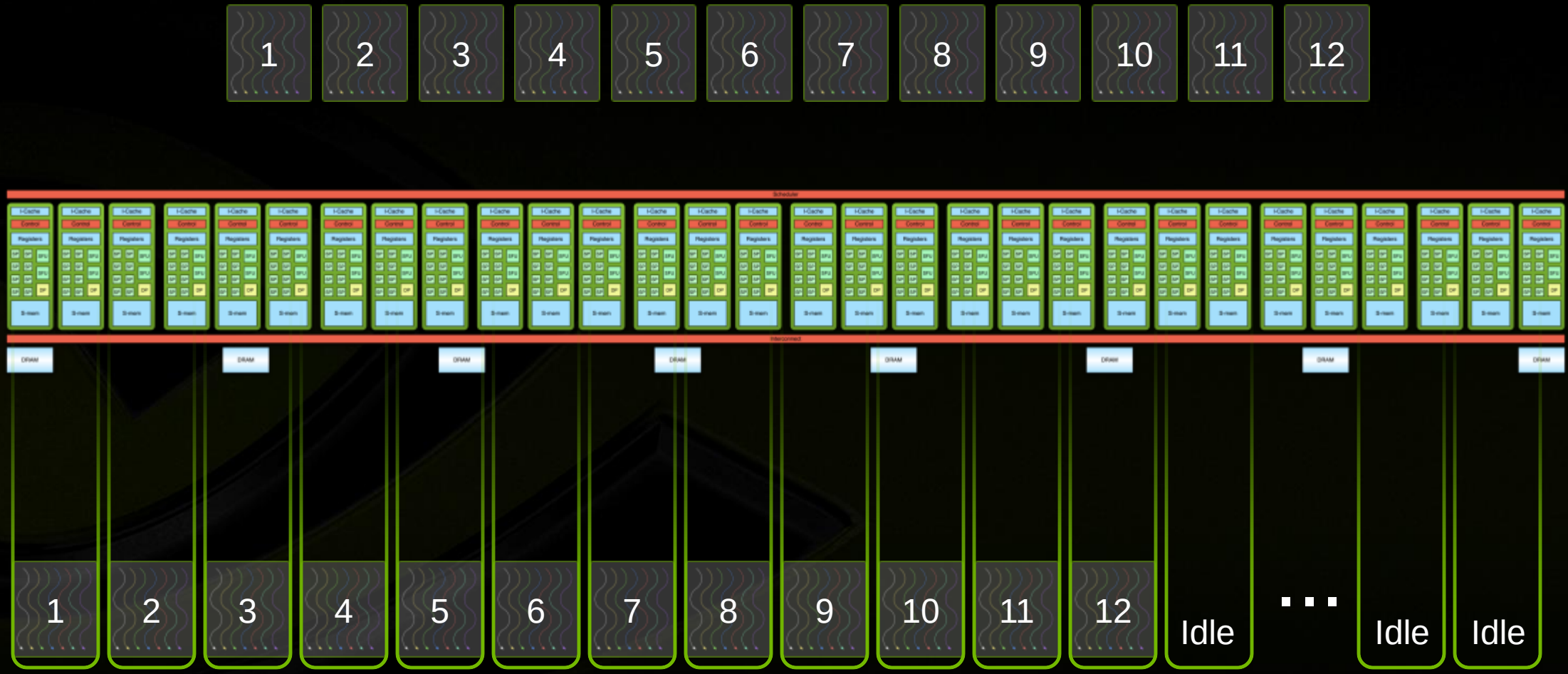
© NVIDIA Corporation 2009

Transparent Scalability – G80



© NVIDIA Corporation 2009

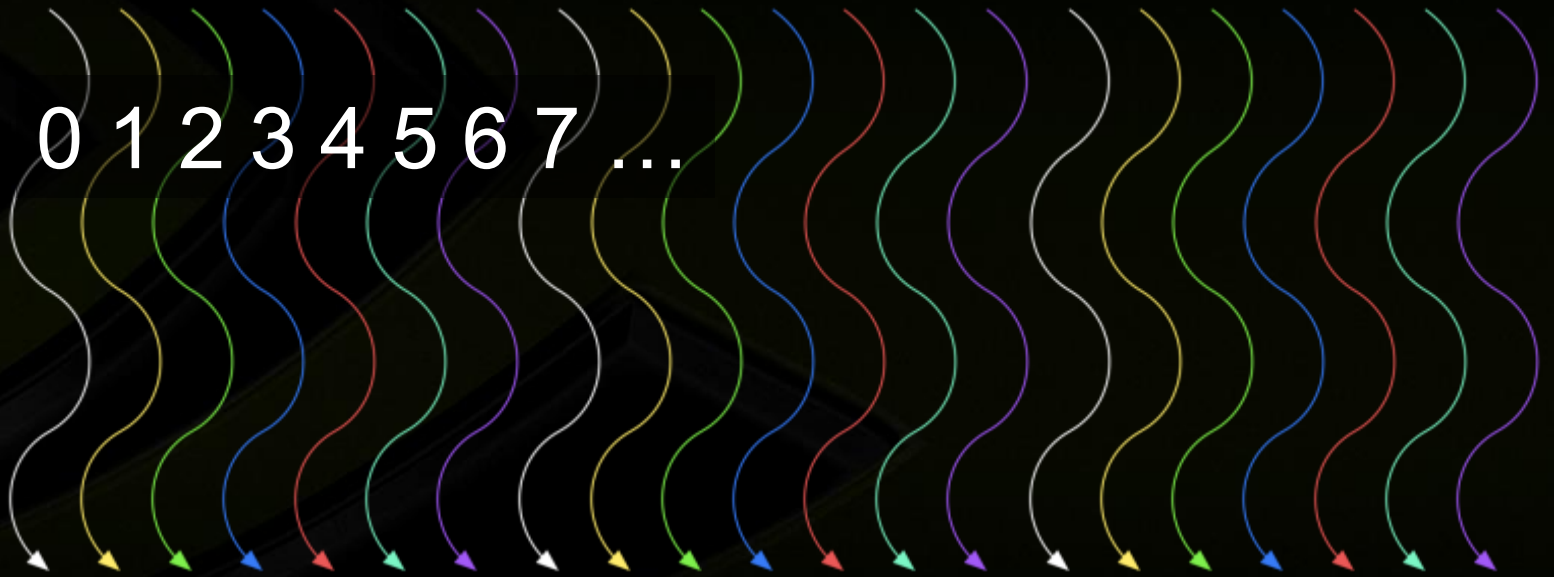
Transparent Scalability – GT200



© NVIDIA Corporation 2009

Numbering of Threads

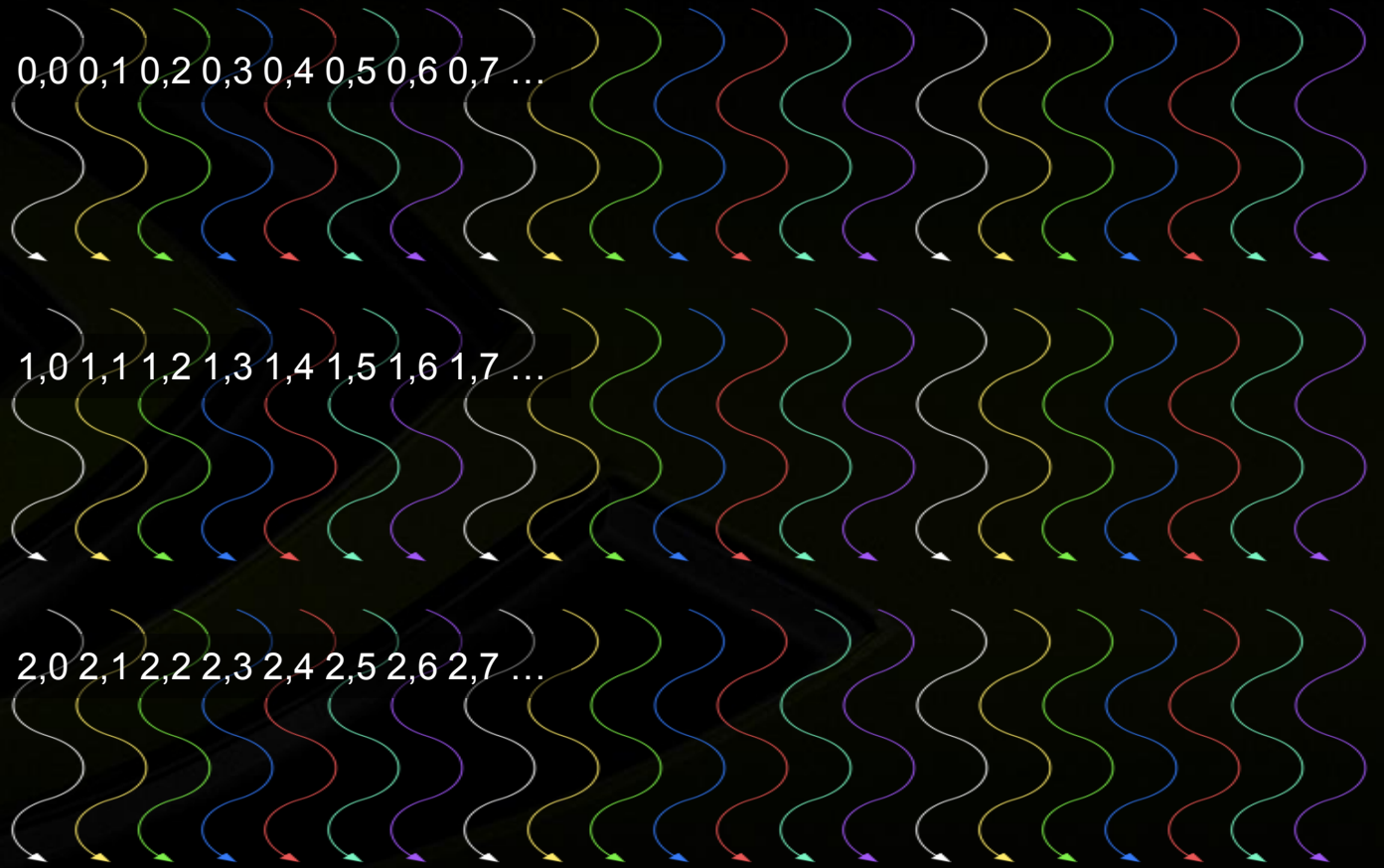
1-dimensional indexing



© NVIDIA Corporation 2009

Numbering of Threads

2-dimensional indexing



© NVIDIA Corporation 2009

Numbering of Threads

Or 3-dimensional indexing

0,0,0 0,0,1 0,0,2 0,0,3 0,0,4 0,0,5 0,0,6 0,0,7 …
0,1,0 0,1,1 0,1,2 0,1,3 0,1,4 0,1,5 0,1,6 0,1,7 …

▪ ▪ ▪

1,0,0 1,0,1 1,0,2 1,0,3 1,0,4 1,0,5 1,0,6 1,0,7 …
1,1,0 1,1,1 1,1,2 1,1,3 1,1,4 1,1,5 1,1,6 1,1,7 …

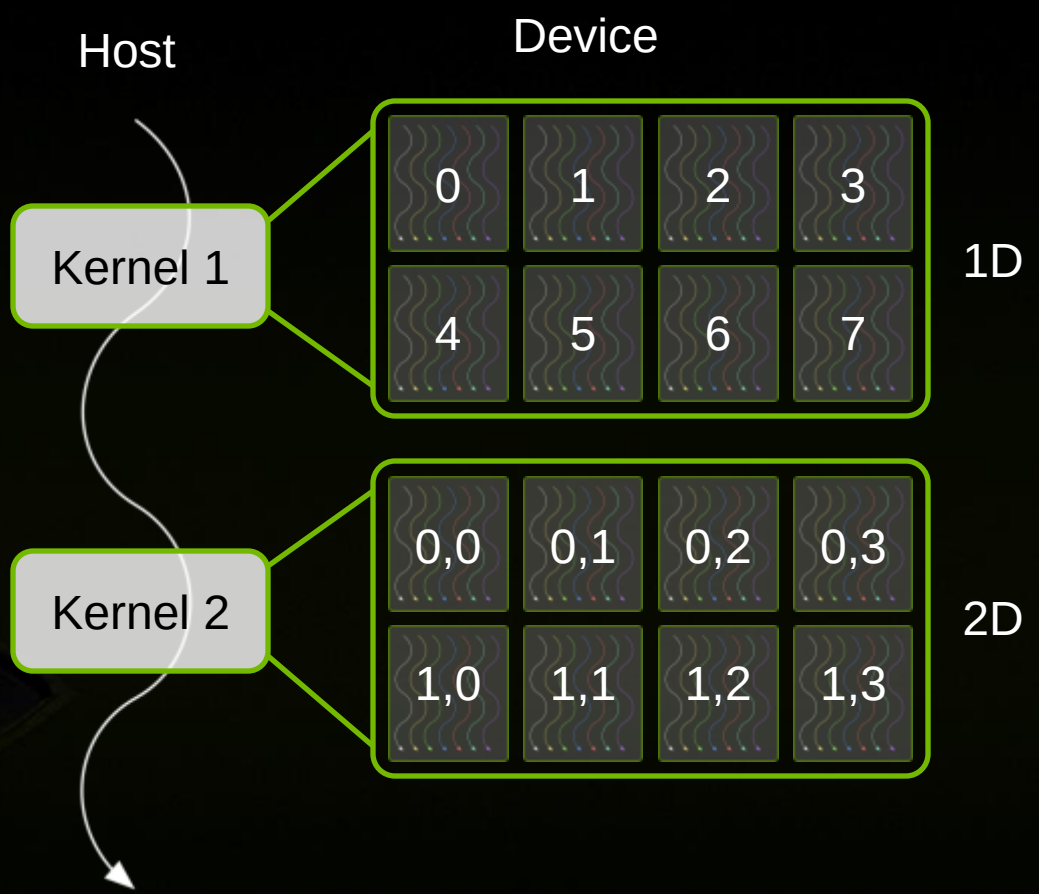
© NVIDIA Corporation 2009

Numbering of Blocks

© NVIDIA Corporation 2009

CUDA Programming Model - Summary

- A kernel executes as a grid of thread blocks
- A block is a batch of threads
  - Communicate through shared memory
- Each block has a block ID
- Each thread has a thread ID



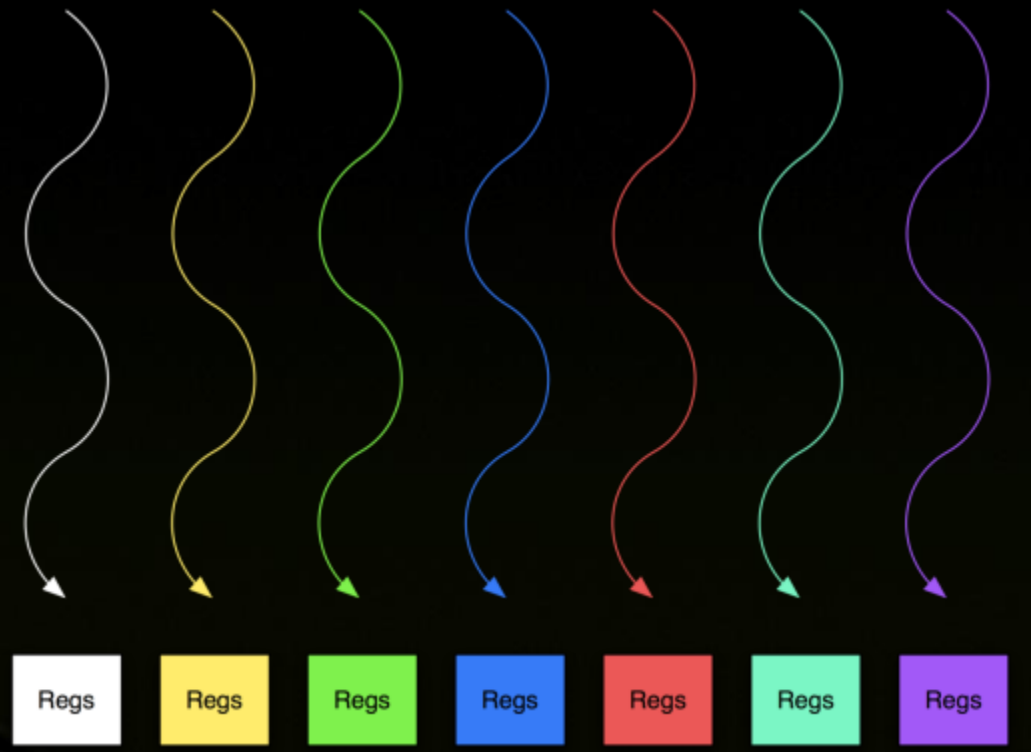
© NVIDIA Corporation 2009

Memory Model

© NVIDIA Corporation 2009

Memory hierarchy

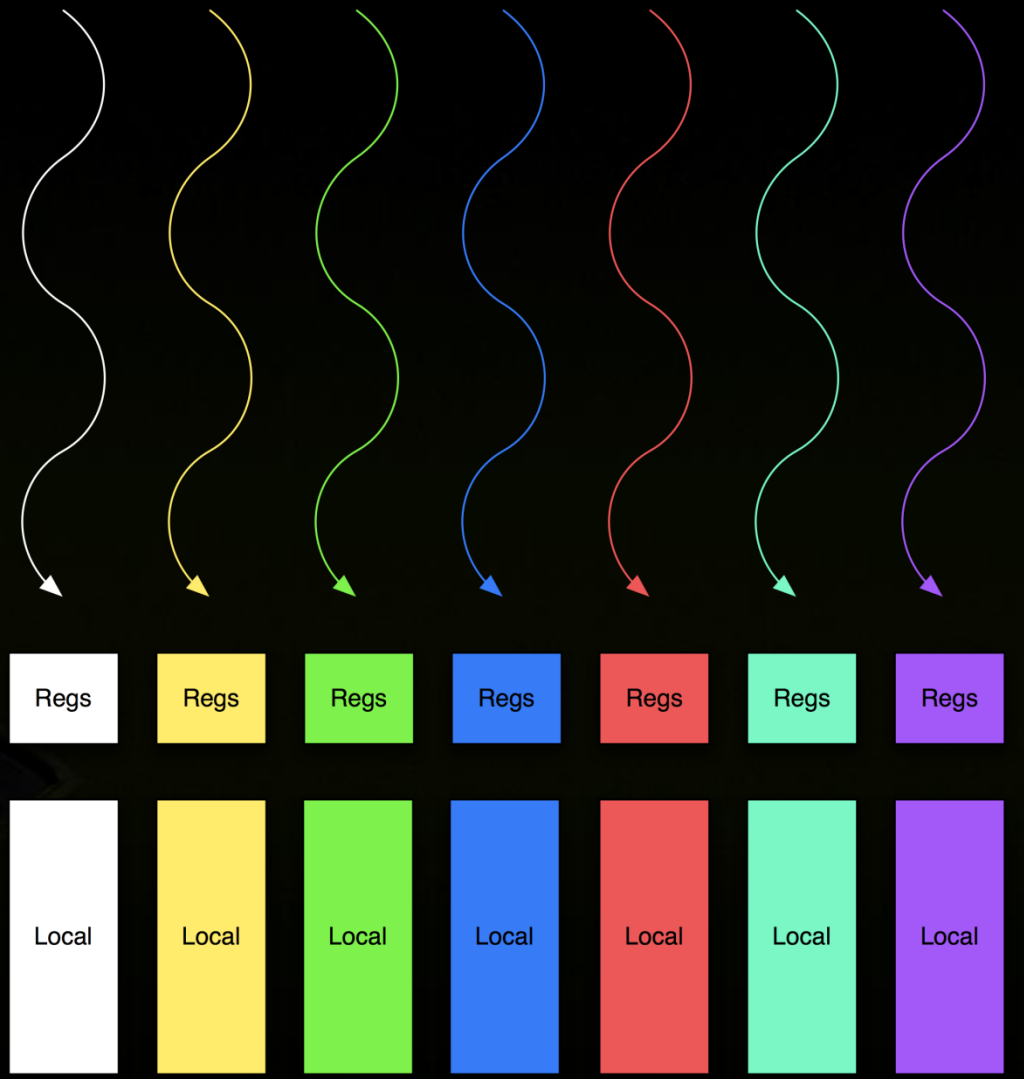
- Thread:
  - Registers



© NVIDIA Corporation 2009

Memory hierarchy

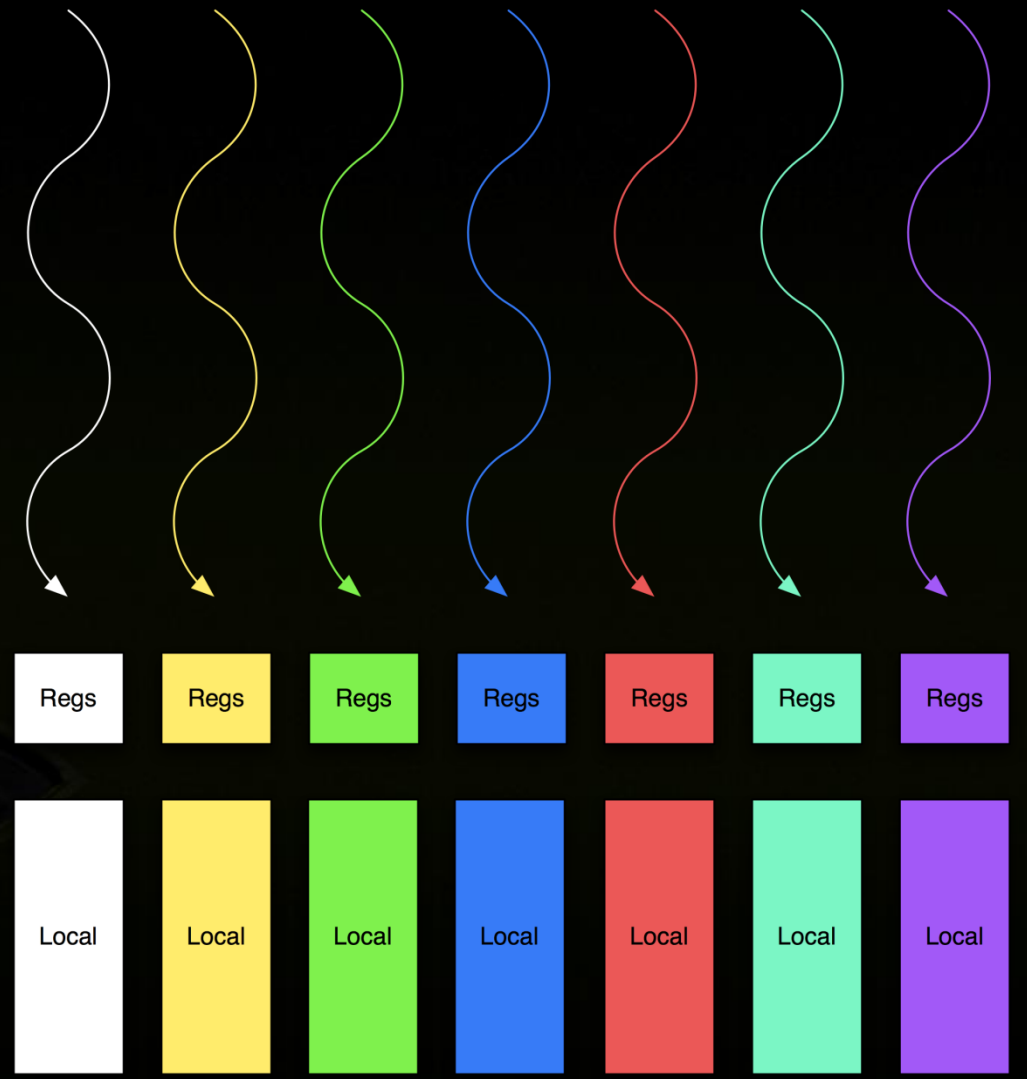
- Thread:
  - Registers
- Thread:
  - Local memory



© NVIDIA Corporation 2009

Memory hierarchy

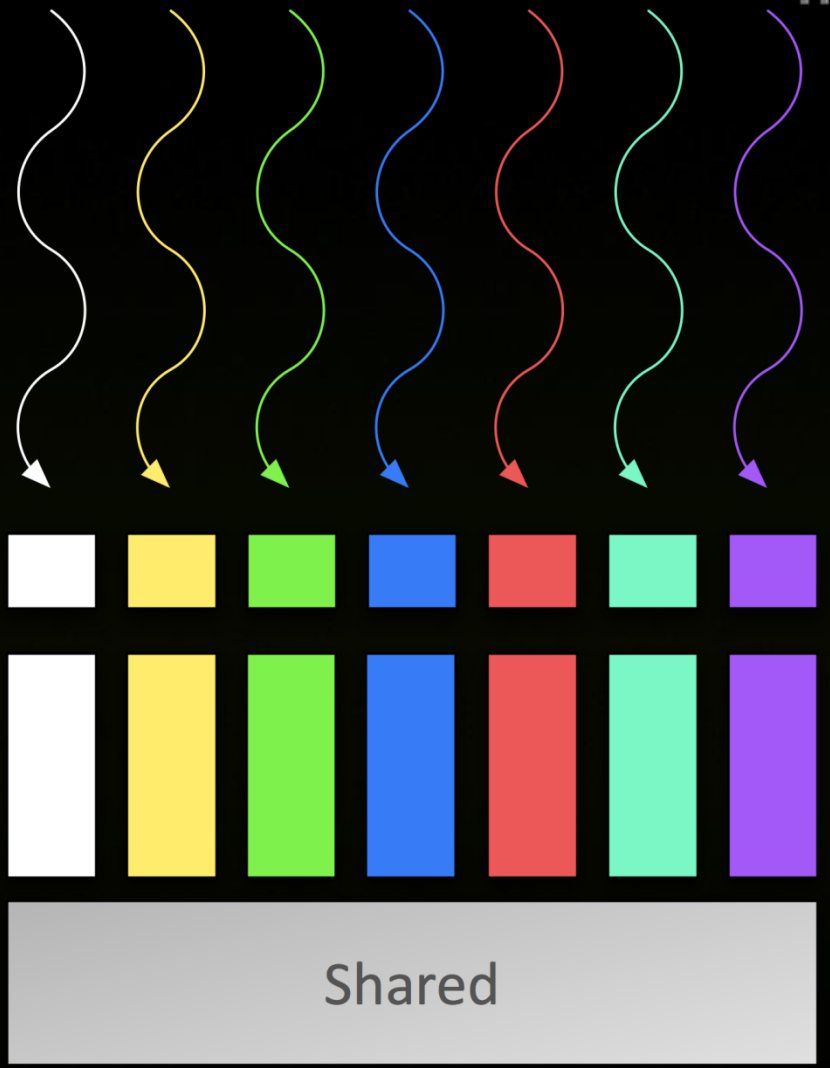
- Thread:
  - Registers
- Thread:
  - Local memory
- Block of threads:
  - Shared memory



© NVIDIA Corporation 2009

Memory hierarchy

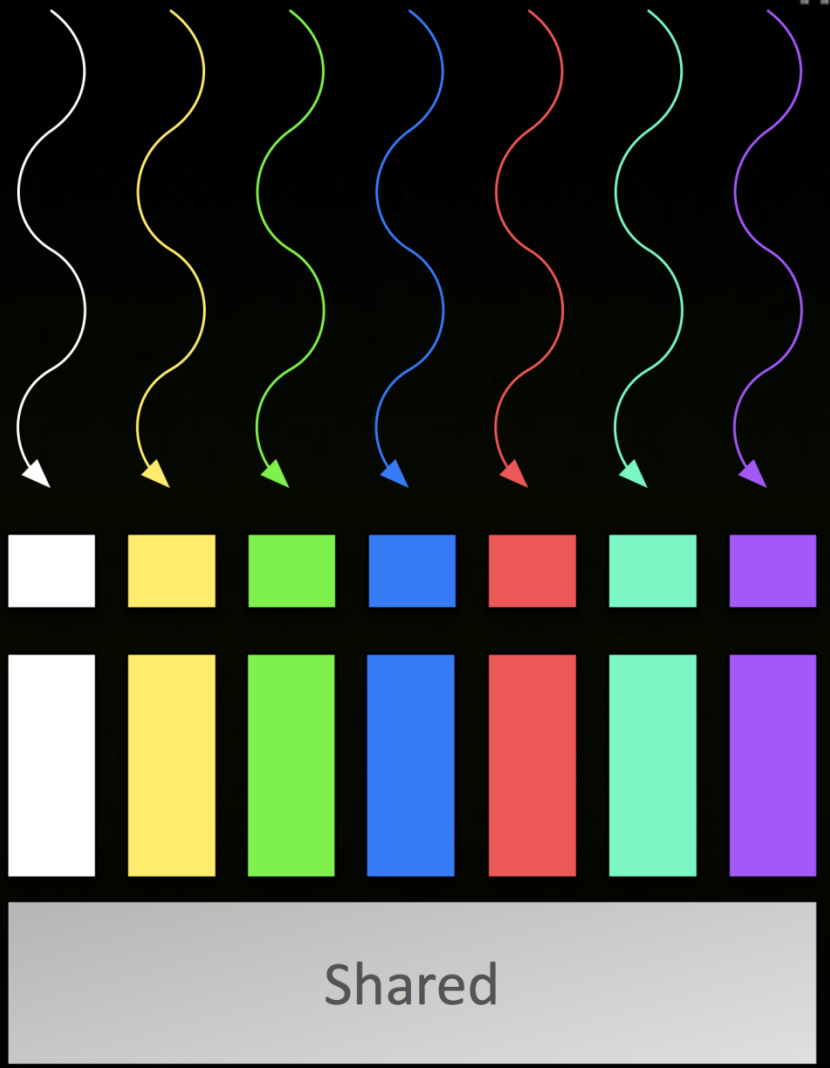
- Thread:
  - Registers
- Thread:
  - Local memory
- Block of threads:
  - Shared memory



© NVIDIA Corporation 2009

Memory hierarchy

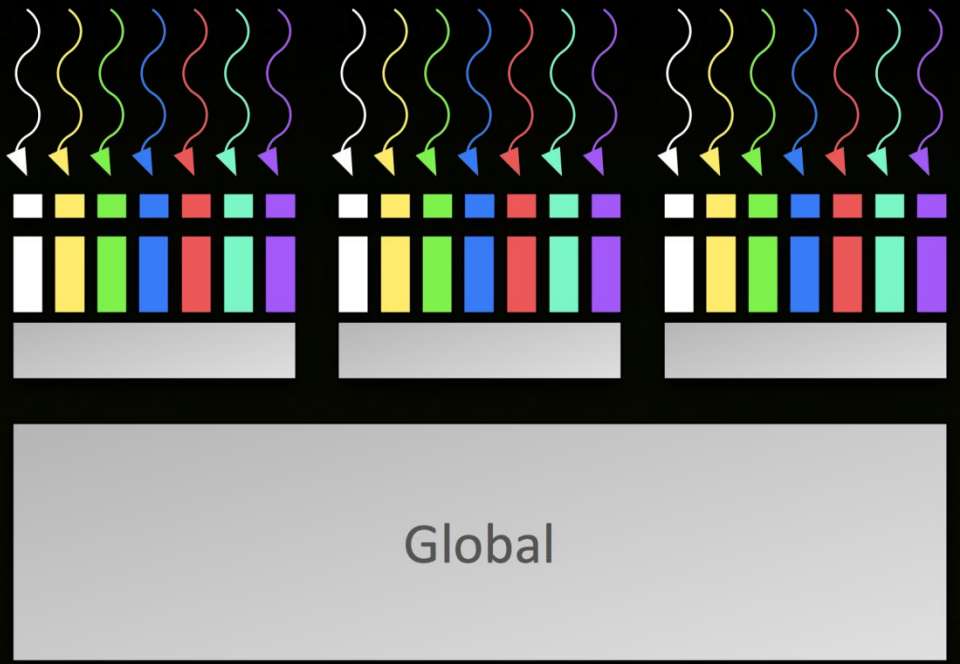
- Thread:
  - Registers
- Thread:
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory



© NVIDIA Corporation 2009

Memory hierarchy

- Thread:
  - Registers
- Thread:
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory



© NVIDIA Corporation 2009

Memory Spaces

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

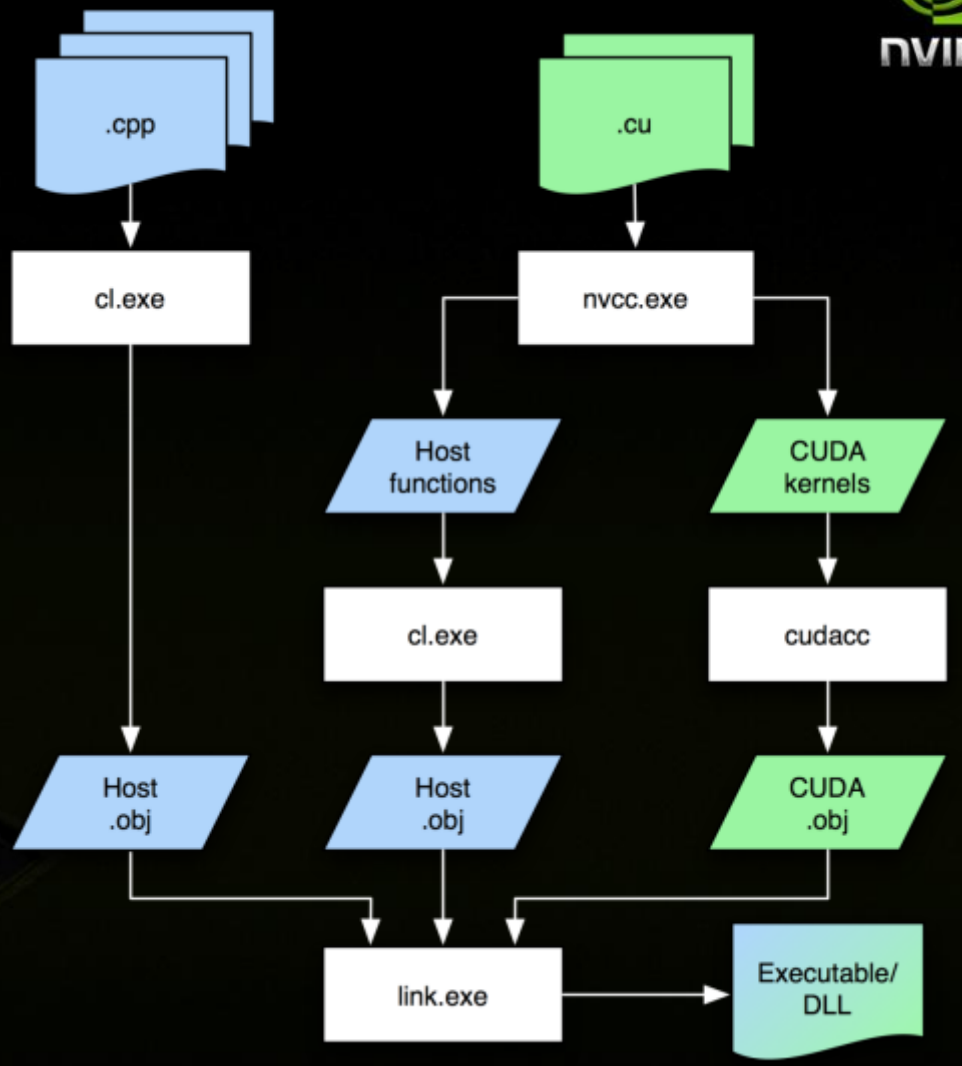
© NVIDIA Corporation 2009

COMPILATION

© NVIDIA Corporation 2009

Visual Studio

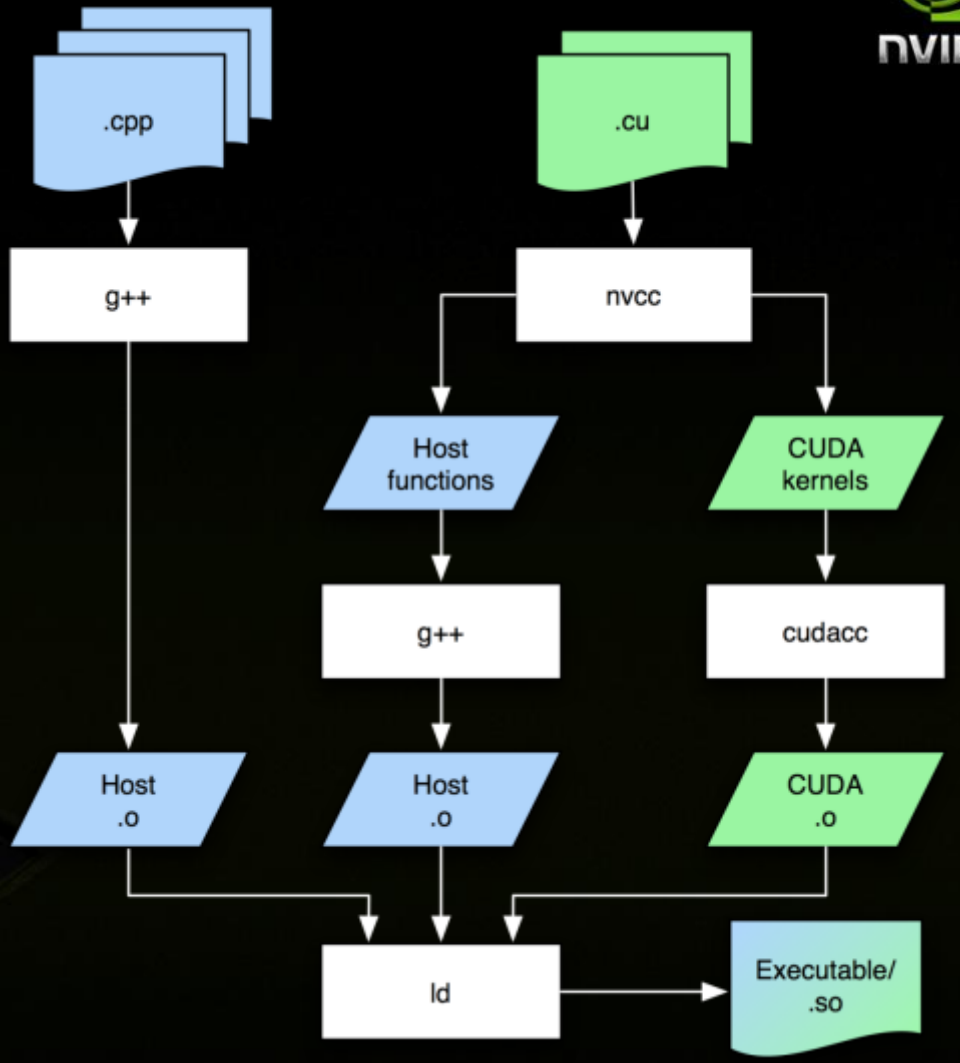
- Separate file types
  - .c/.cpp for host code
  - .cu for device/mixed code
- Compilation rules: cuda.rules
  - Syntax highlighting
  - Intellisense
- Integrated debugger and profiler: Nsight



© NVIDIA Corporation 2009

Linux

- Separate file types
  - .c/.cpp for host code
  - .cu for device/mixed code
- Typically makefile driven
- cuda-gdb, Allinea DDT, TotalView for debugging
- CUDA Visual Profiler



© NVIDIA Corporation 2009

Compilation Commands

- nvcc <filename>.cu [-o <executable>]
  - Builds release code
- nvcc –g <filename>.cu
  - Builds debug CPU code
- nvcc –G <filename>.cu
  - Builds debug GPU code
- nvcc –O <level> <filename>.cu
  - Builds optimised GPU code

© NVIDIA Corporation 2009

Exercise 0: Run a Simple Program

- Log on to test system
- Compile and run pre-written CUDA program — deviceQuery

```
CUDA Device Query (Runtime API) version (CUDART static linking)
There are 2 devices supporting CUDA

 Device 0: "Quadro FX 570M"
Device 0: "Tesla C1060"
  Major revision number:                         1
  Minor revision number:                         1
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 268107776 bytes
  Total amount of global memory:                 4294705152 bytes
  Number of multiprocessors:                     4
  Number of multiprocessors:                     30
  Number of cores:                               32
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Total number of registers available per block: 16384
  Warp size:                                     32
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    0.95 GHz
  Clock rate:                                    1.44 GHz
  Concurrent copy and execution:                 Yes
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Exclusive (only
one host thread at a time can use this device)
 Test PASSED

 Press ENTER to exit...
```

© NVIDIA Corporation 2009

CUDA C Programming Language

© NVIDIA Corporation 2009

CUDA C — C with Runtime Extensions

- **Device management:**
  `cudaGetDeviceCount(), cudaGetDeviceProperties()`
- **Device memory management:**
  `cudaMalloc(), cudaFree(), cudaMemcpy()`
- **Texture management:**
  `cudaBindTexture(), cudaBindTextureToArray()`
- **Graphics interoperability:**
  `cudaGLMapBufferObject(), cudaD3D9MapVertexBuffer()`

© NVIDIA Corporation 2009

CUDA C — C with Language Extensions

- Function qualifiers

```
__global__ void MyKernel() {}        // call from host, execute on GPU
__device__ float MyDeviceFunc() {}   // call from GPU, execute on GPU
__host__   int HostFunc() {}         // call from host, execute on host
```

- Variable qualifiers

```
__device__   float MyGPUArray[32];     // in GPU memory space
__constant__ float MyConstArray[32];  // write by host; read by GPU
__shared__   float MySharedArray[32]; // shared within thread block
```

- Built-in vector types

```
int1, int2, int3, int4
float1, float2, float3, float4
double1, double2
etc.
```

© NVIDIA Corporation 2009

CUDA C — C with Language Extensions

- Execution configuration

```
dim3 dimGrid(100, 50);              // 5000 thread blocks
dim3 dimBlock(4, 8, 8);             // 256 threads per block
MyKernel <<< dimGrid, dimBlock >>> (...); // Launch kernel
```

- Built-in variables and functions valid in device code:

```
dim3 gridDim;                       // Grid dimension
dim3 blockDim;                      // Block dimension
dim3 blockIdx;                      // Block index
dim3 threadIdx;                     // Thread index
void __syncthreads();               // Thread synchronization
```

© NVIDIA Corporation 2009

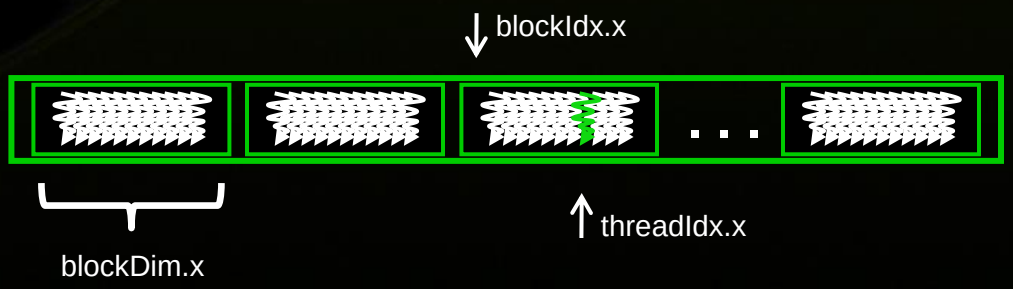
SAXPY: Device Code

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
```

Parallel C Code



© NVIDIA Corporation 2009

SAXPY: Host Code

```
// Allocate two N-vectors h_x and h_y
int size = N * sizeof(float);
float* h_x = (float*)malloc(size);
float* h_y = (float*)malloc(size);

// Initialize them...

// Allocate device memory
float* d_x; float* d_y;
cudaMalloc((void**)&d_x, size));
cudaMalloc((void**)&d_y, size));

// Copy host memory to device memory
cudaMemcpy(d_x, h_x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, size, cudaMemcpyHostToDevice);

// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (N + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(N, 2.0, d_x, d_y);

// Copy result back from device memory to host memory
cudaMemcpy(h_y, d_y, size, cudaMemcpyDeviceToHost);
```

© NVIDIA Corporation 2009

Exercise 1: Move Data between Host and GPU

- Start from the "cudaMallocAndMemcpy" template.
- Part 1: Allocate memory for pointers *d_a* and *d_b* on the device.
- Part 2: Copy *h_a* on the host to *d_a* on the device.
- Part 3: Do a device to device copy from *d_a* to *d_b.*
- Part 4: Copy *d_b* on the device back to *h_a* on the host.
- Part 5: Free *d_a* and *d_b* on the host.
- Bonus: Experiment with *cudaMallocHost* in place of *malloc* for allocating *h_a.*

© NVIDIA Corporation 2009

Launching a Kernel

- **Call a kernel with**

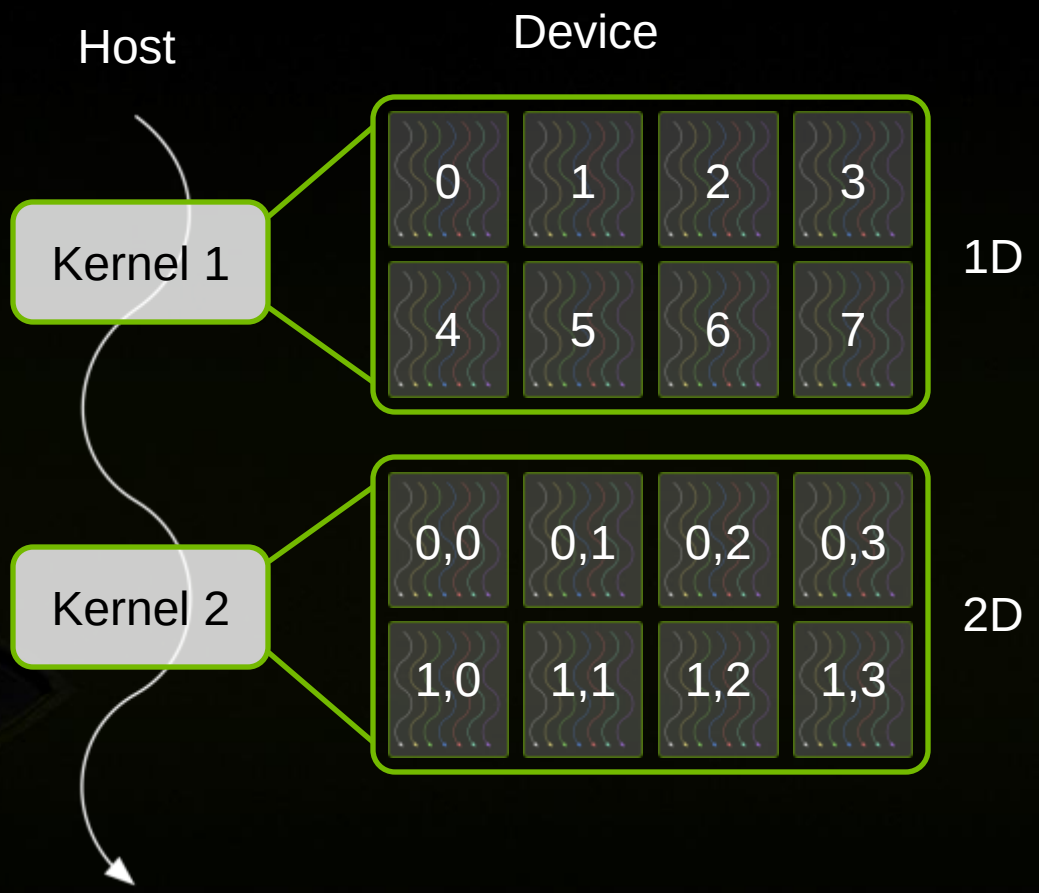
```
Func <<<Dg,Db,Ns,S>>> (params);
dim3 Dg(mx,my,1); // grid spec
dim3 Db(nx,ny,nz); // block spec
size_t Ns; // shared memory
cudaStream_t S; // CUDA stream
```

- **Execution configuration is passed to kernel with built-in variables**

```
dim3 gridDim, blockDim, blockIdx,
     threadIdx;
```

- **Extract components with**

```
threadIdx.x, threadIdx.y,
threadIdx.z, etc.
```



© NVIDIA Corporation 2009

Exercise 2: Launching Kernels

- Start from the "myFirstKernel" template.
- Part1: Allocate device memory for the result of the kernel using pointer *d_a.*
- Part2: Configure and launch the kernel using a 1-D grid of 1-D thread blocks.
- Part3: Have each thread set an element of *d_a* as follows:

```
idx = blockIdx.x*blockDim.x + threadIdx.x
d_a[idx] = 1000*blockIdx.x + threadIdx.x
```

- Part4: Copy the result in *d_a* back to the host pointer *h_a.*
- Part5: Verify that the result is correct.

© NVIDIA Corporation 2009

Exercise 3: Reverse Array, Single Block

- Given an input array $\{a_0, a_1, \ldots, a_{n-1}\}$ in pointer *d_a*, store the reversed array $\{a_{n-1}, a_{n-2}, \ldots, a_0\}$ in pointer *d_b*
- Start from the "reverseArray_singleblock" template
- Only one thread block launched, to reverse an array of size
  N = numThreads = 256 elements
- Part 1 (of 1): All you have to do is implement the body of the kernel "reverseArrayBlock()"
- Each thread moves a single element to reversed position
  - Read input from *d_a* pointer
  - Store output in reversed location in *d_b* pointer

© NVIDIA Corporation 2009

Exercise 4: Reverse Array, Multi-Block

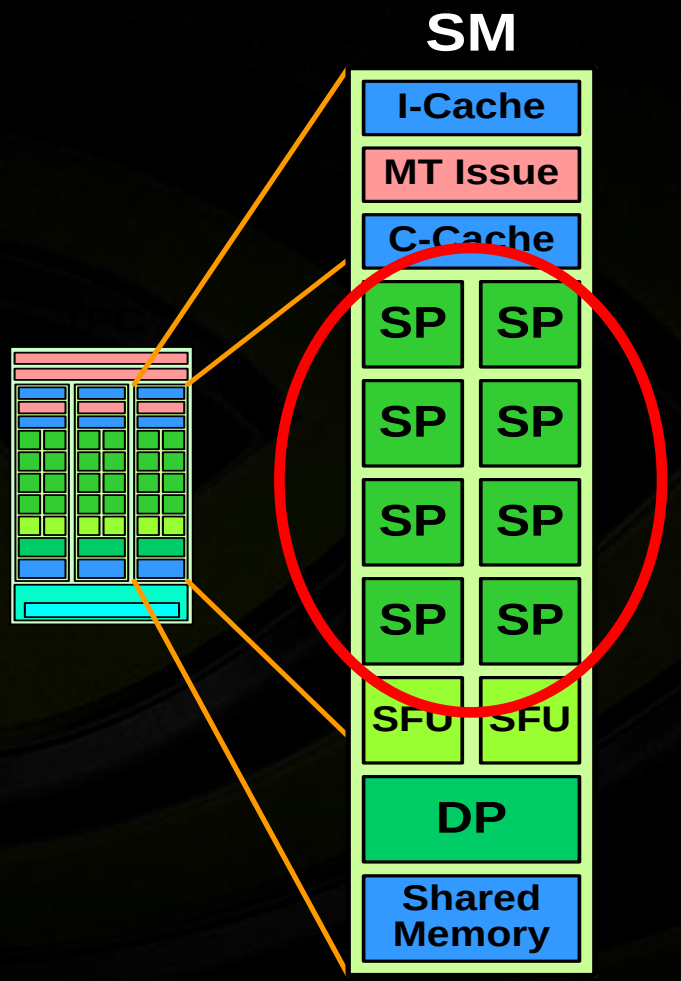
- Given an input array $\{a_0, a_1, \ldots, a_{n-1}\}$ in pointer *d_a*, store the reversed array $\{a_{n-1}, a_{n-2}, \ldots, a_0\}$ in pointer *d_b*
- Start from the "reverseArray_multiblock" template
- Multiple 256-thread blocks launched
  - To reverse an array of size N, N/256 blocks
- Part 1: Compute the number of blocks to launch
- Part 2: Implement the kernel reverseArrayBlock()
- Note that now you must compute both
  - The reversed location within the block
  - The reversed offset to the start of the block

© NVIDIA Corporation 2009

Performance Considerations

© NVIDIA Corporation 2009

Single-Instruction, Multiple-Thread Execution

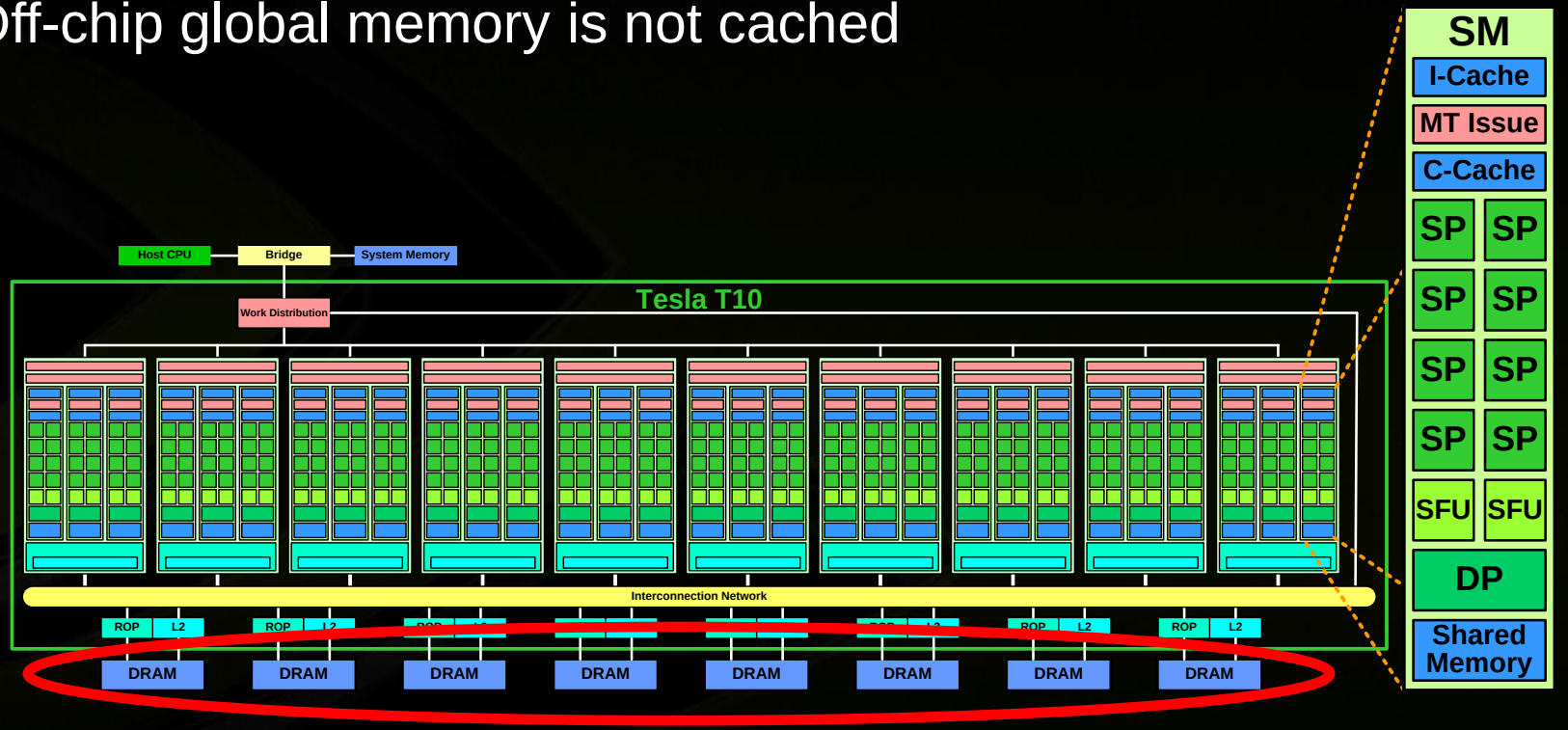


- Warp: set of 32 parallel threads that execute together in single-instruction, multiple-thread mode (SIMT) on a streaming multiprocessor (SM)
- SM hardware implements zero-overhead warp and thread scheduling
- Threads can execute independently
- SIMT warp diverges and converges when threads branch independently
- Best efficiency and performance when threads of a warp execute together, so no penalty if all threads in a warp take same path of execution
- Each SM executes up to 1024 concurrent threads, as 32 SIMT warps of 32 threads

© NVIDIA Corporation 2009

Global Memory

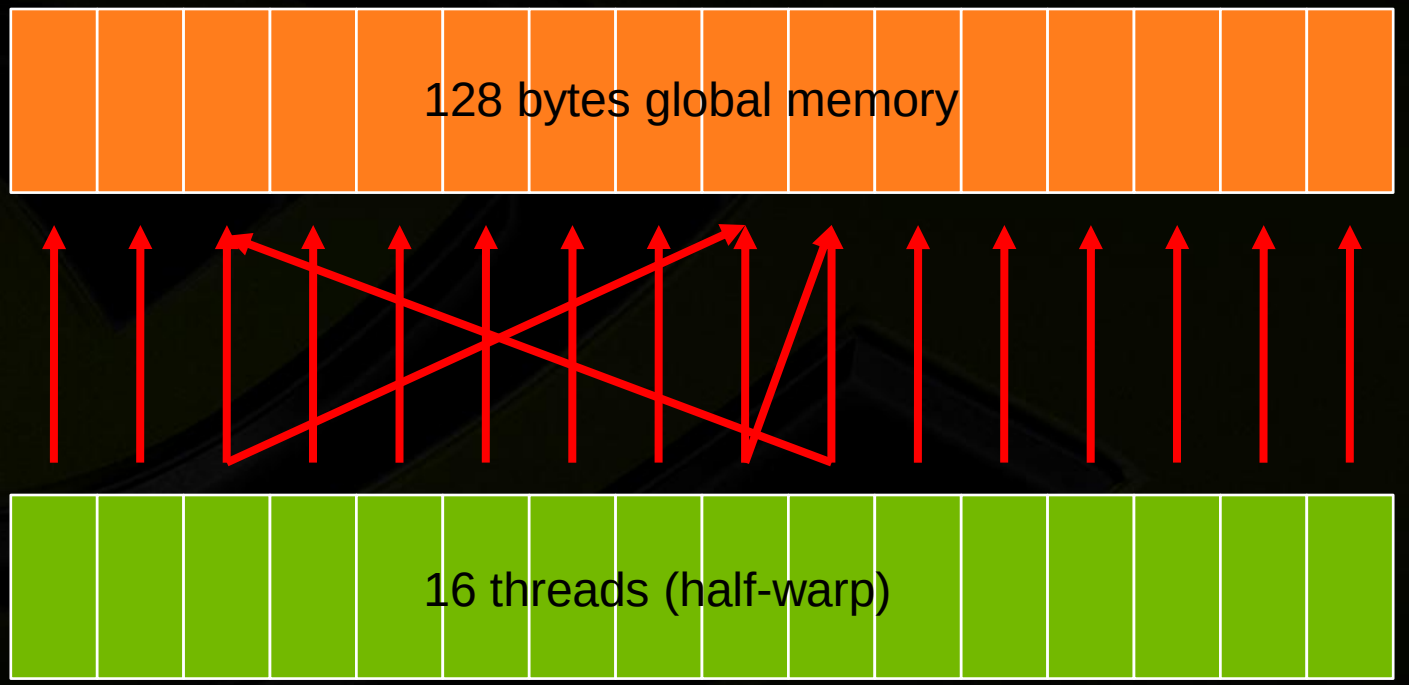
Off-chip global memory is not cached



© NVIDIA Corporation 2009

Efficient Access to Global Memory

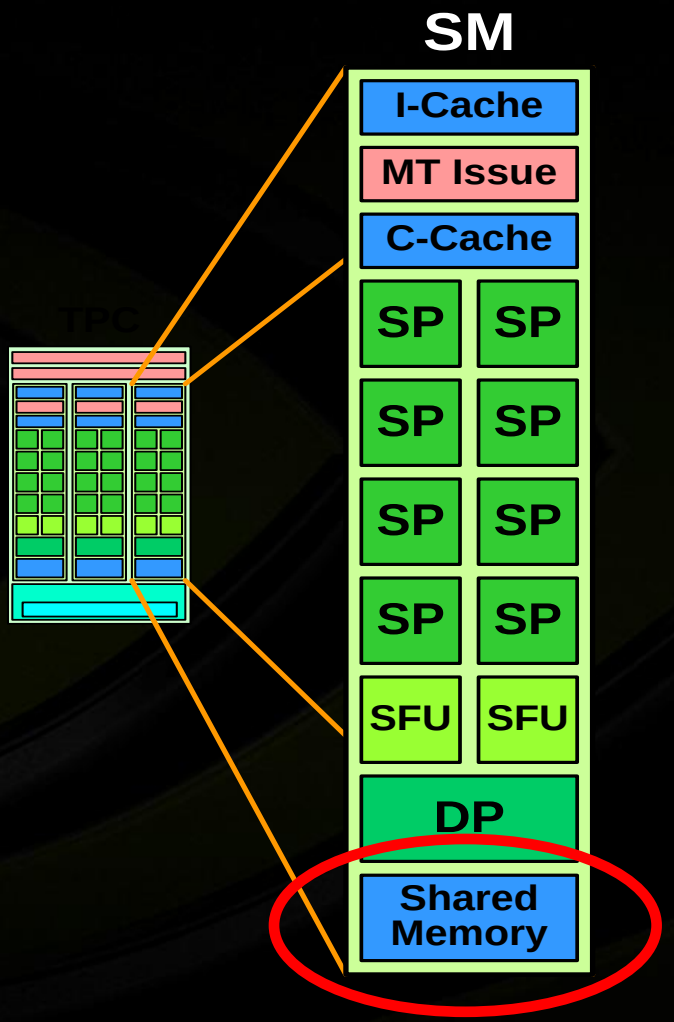
Single memory transaction (coalescing) for some memory addressing patterns



- Linear pattern
- Not all need participate
- Anywhere in block OK

© NVIDIA Corporation 2009

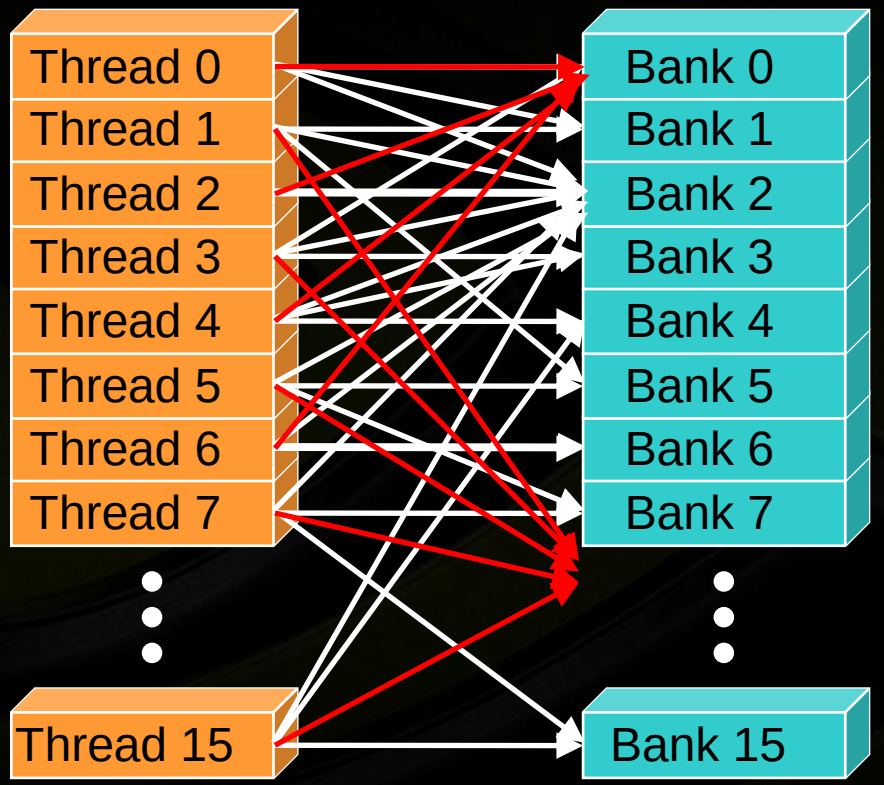
Shared Memory



- More than 1 Tbyte/sec aggregate memory bandwidth
- Use it
  - As a cache
  - To reorganize global memory accesses into coalesced pattern
  - To share data between threads
- 16 kbytes per SM

© NVIDIA Corporation 2009

Shared Memory Bank Conflicts

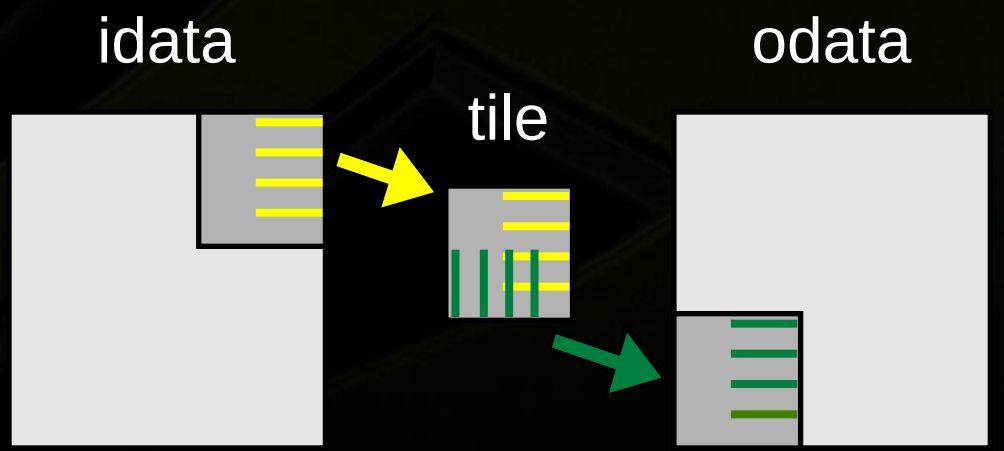


- Successive 32-bit words assigned to different banks
- Simultaneous access to the same bank by threads in a half-warp causes conflict and serializes access
- Linear access pattern
- Permutation
- Broadcast (from one address)
- Conflict, stride 8
  for(i=0; i<n; i+=8) { … = a[i] }

© NVIDIA Corporation 2009

Matrix Transpose

- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires __syncthreads() since threads write data read by other threads
- Pad shared memory array to avoid bank conflicts



© NVIDIA Corporation 2009

Matrix Transpose

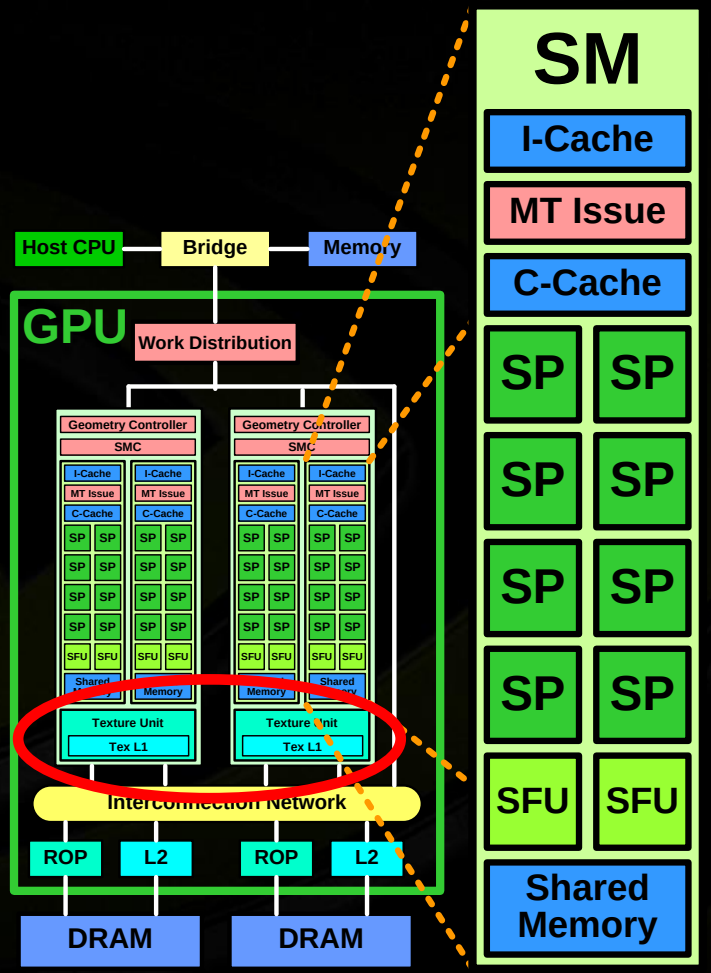
- There are further optimisations: see the New Matrix Transpose SDK example.

© NVIDIA Corporation 2009

Other GPU Memories

© NVIDIA Corporation 2009

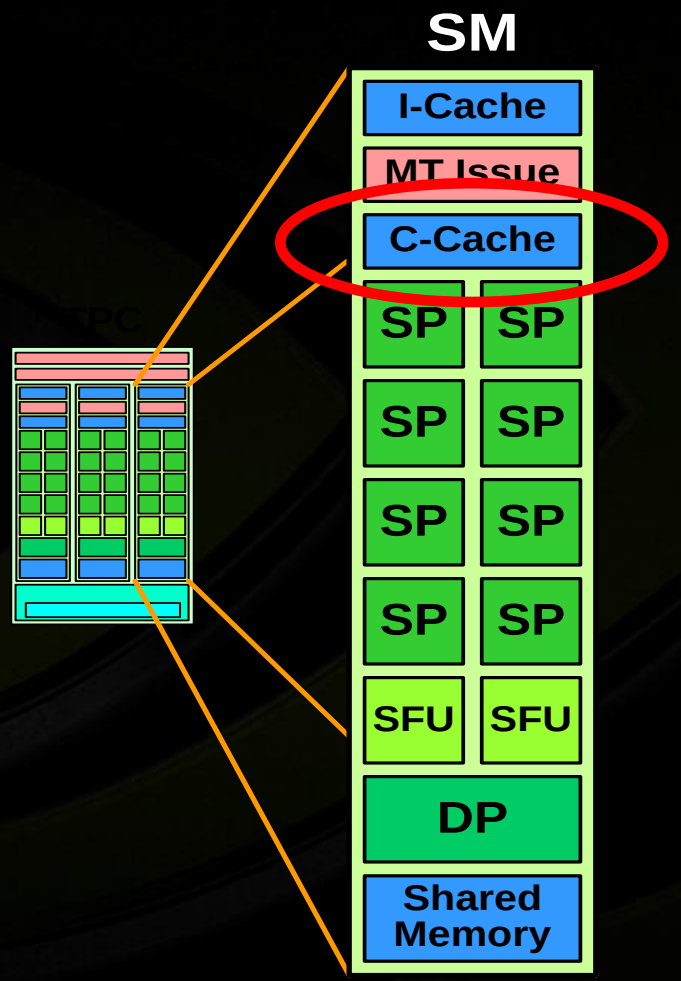
Texture Memory



- Texture is an object for reading data
- Data is cached
- Host actions
  - Allocate memory on GPU
  - Create a texture memory reference object
  - Bind the texture object to memory
  - Clean up after use
- GPU actions
  - Fetch using texture references text1Dfetch(), tex1D(), tex2D(), tex3D()

© NVIDIA Corporation 2009

Constant Memory



- Write by host, read by GPU
- Data is cached
- Useful for tables of constants

© NVIDIA Corporation 2009

Execution Configuration

© NVIDIA Corporation 2009

Execution Configuration

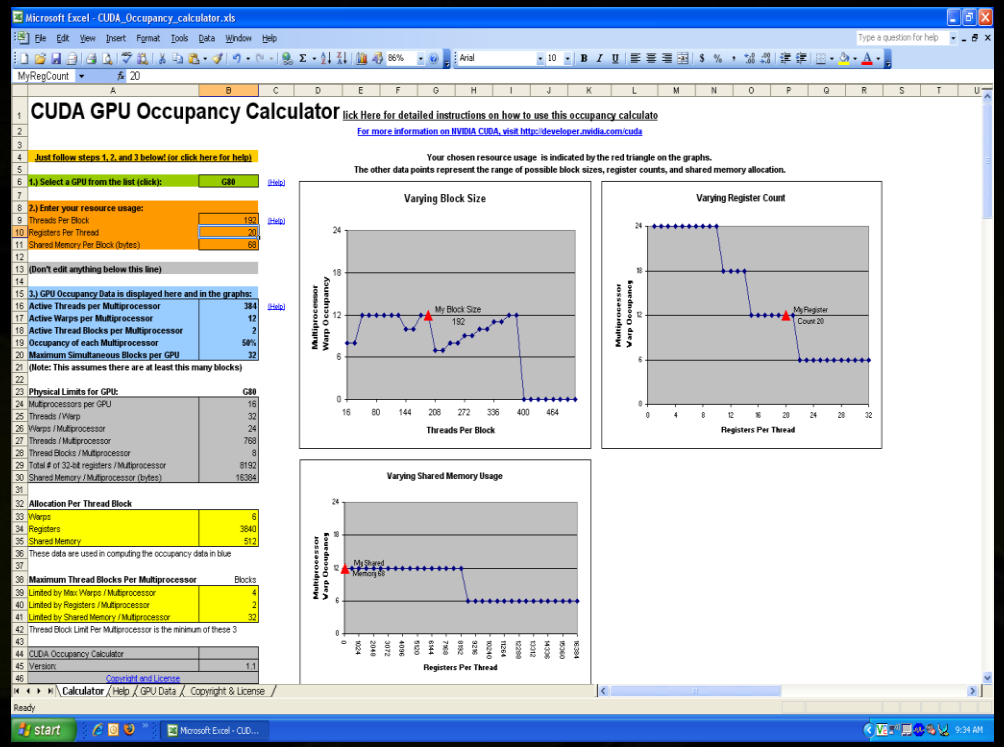
```
vectorAdd <<< BLOCKS, THREADS_PER_BLOCK >>> (N, 2.0, d_x, d_y);
```

- How many blocks?
  - At least one block per SM to keep every SM occupied
  - At least two blocks per SM so something can run if block is waiting for a synchronization to complete
  - Many blocks for scalability to larger and future GPUs
- How many threads?
  - At least 192 threads per SM to hide read after write latency of 11 cycles (not necessarily in same block)
  - Use many threads to hide global memory latency
  - Too many threads exhausts registers and shared memory
  - Thread count a multiple of warp size
  - Typically, between 64 and 256 threads per block

```
x = y + 5;
z = x + 3;
```

© NVIDIA Corporation 2009

Occupancy Calculator



$$\text{occupancy} = \frac{\text{blocks per SM} \times \text{threads per block}}{\text{maximum threads per SM}}$$

- Occupancy calculator shows trade-offs between thread count, register use, shared memory use
- Low occupancy is bad
- Increasing occupancy doesn't always help

© NVIDIA Corporation 2009

Debugging and Profiling

© NVIDIA Corporation 2009

Debugging

- nvcc flags
  - –debug (-g)

    Generate debug information for host code
  - --device-debug <level> (-G <level>)

    Generate debug information for device code, plus also specify the optimisation level for the device code in order to control its ‘debuggability’. Allowed values for this option: 0,1,2,3
- Debug with

  cuda-gdb a.out
- Usual gdb commands available

© NVIDIA Corporation 2009

Debugging

- Additional commands in cuda-gdb
  - thread — Display the current host and CUDA thread of focus.
  - thread <<<(TX,TY,TZ)>>> — Switch to the CUDA thread at specified coordinates
  - thread <<<(BX,BY),(TX,TY,TZ)>>> — Switch to the CUDA block and thread at specified coordinates
  - info cuda threads — Display a summary of all CUDA threads that are currently resident on the GPU
  - info cuda threads all — Display a list of each CUDA thread that is currently resident on the GPU
  - info cuda state — Display information about the current CUDA state.
- next and step advance all threads in a warp, except at _syncthreads() where all warps continue to an implicit barrier following sync

© NVIDIA Corporation 2009

Parallel Nsight

Nsight Parallel Debugger

GPU source code debugging

Variable & memory inspection

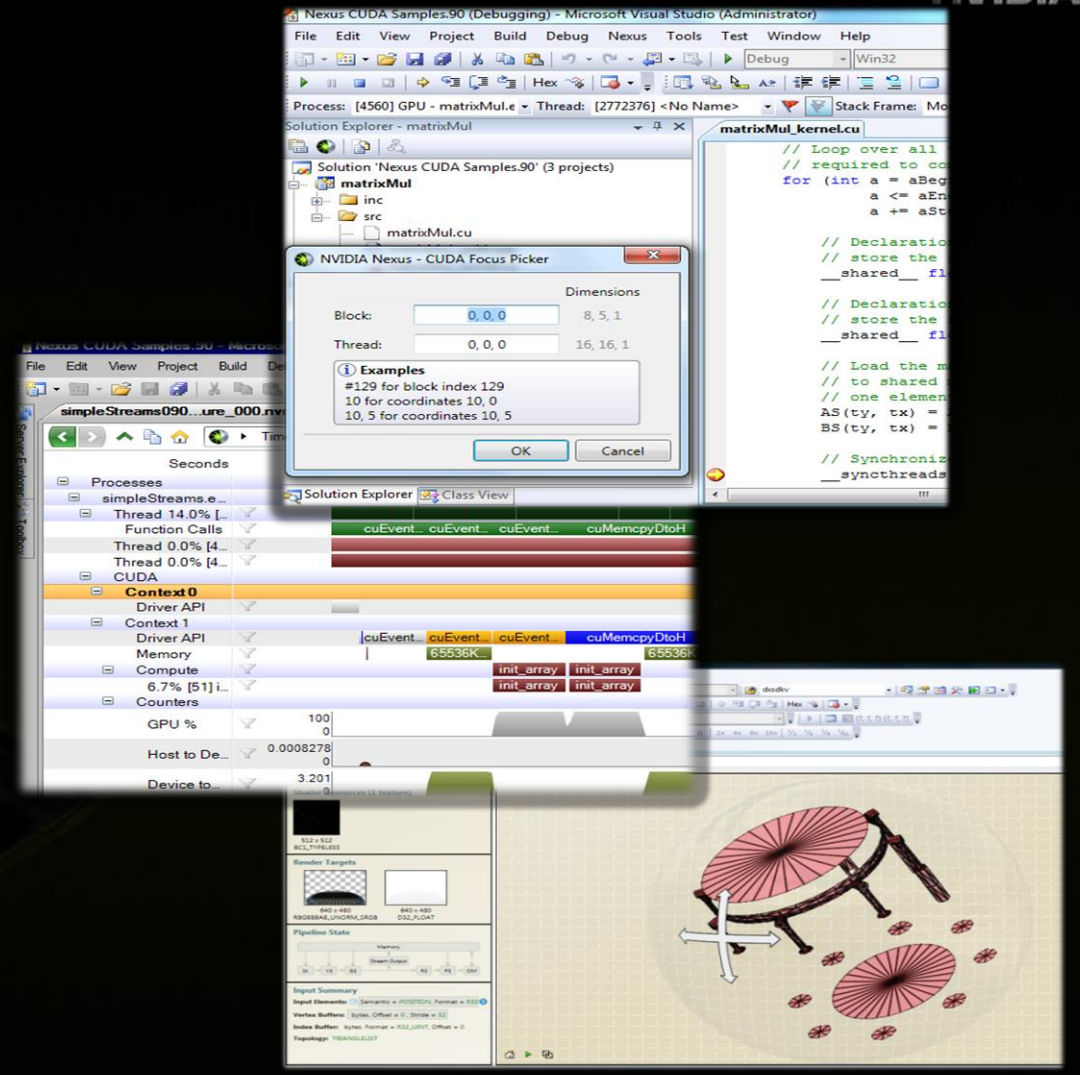
Nsight Analyzer

Platform-level Analysis

For the CPU and GPU

Nsight Graphics Inspector

Visualize and debug graphics content



© NVIDIA Corporation 2009

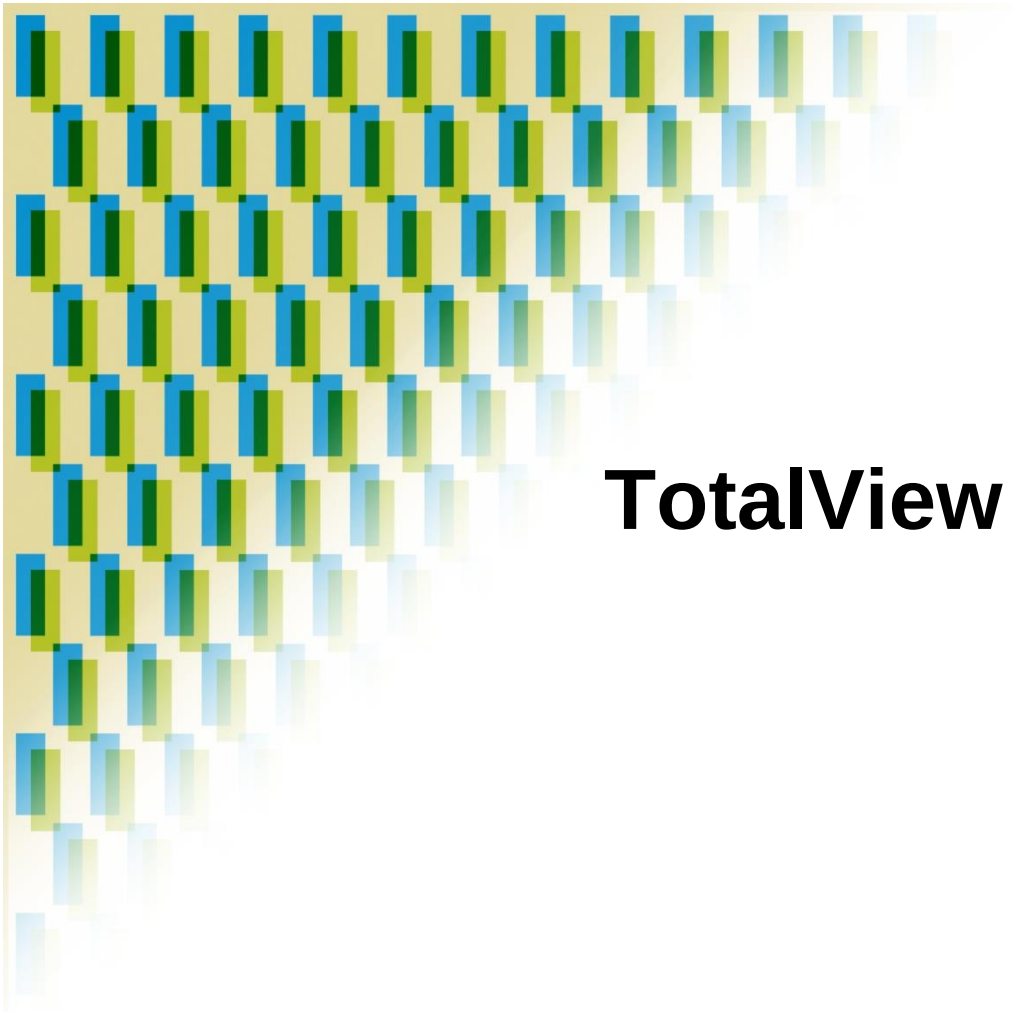
Allinea DDT

GPU Debugging

Making it easy

Allinea DDT — CUDA Enabled

TotalView for CUDA

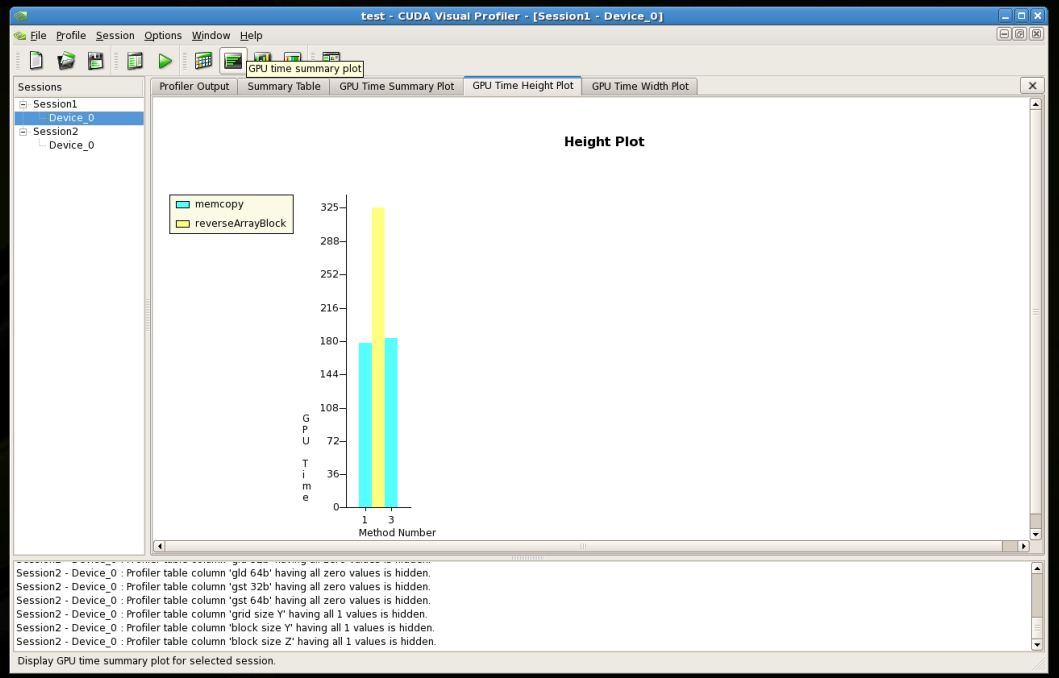


TotalView for CUDA

TOTALVIEW
TECHNOLOGIES

CUDA Visual Profiler

- cudaprof
- Documentation in $CUDA/cudaprof/doc/cudaprof.html



© NVIDIA Corporation 2009

CUDA Visual Profiler

- Open a new project
- Select session settings through dialogue
- Execute CUDA program by clicking Start button
- Various views of collected data available
- Results of different runs stored in sessions for easy comparison
- Project can be saved

© NVIDIA Corporation 2009

Miscellaneous Topics

© NVIDIA Corporation 2009

Expensive Operations

- sin(), exp() etc.; faster, less accurate versions are __sin(), __exp() etc.
- Integer division and modulo; avoid if possible; replace with bit shift operations for powers of 2
- Branching where threads of warp take differing paths of control flow

© NVIDIA Corporation 2009

Host to GPU Data Transfers

- PCI Express Gen2, 8 Gbytes/sec peak
- Use page-locked (pinned) memory for maximum bandwidth between GPU and host
- Data transfer host-GPU and GPU-host can overlap with computation both on host and GPU

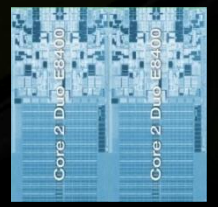
© NVIDIA Corporation 2009

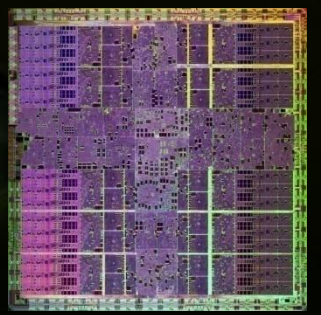
Application Software (written in C)

CUDA Libraries

cuFFT | cuBLAS | cuDPP

CPU Hardware	CUDA Compiler	CUDA Tools
1U PCI-E Switch	C Fortran	Debugger Profiler





4 cores

240 cores

© NVIDIA Corporation 2009

On-line Course

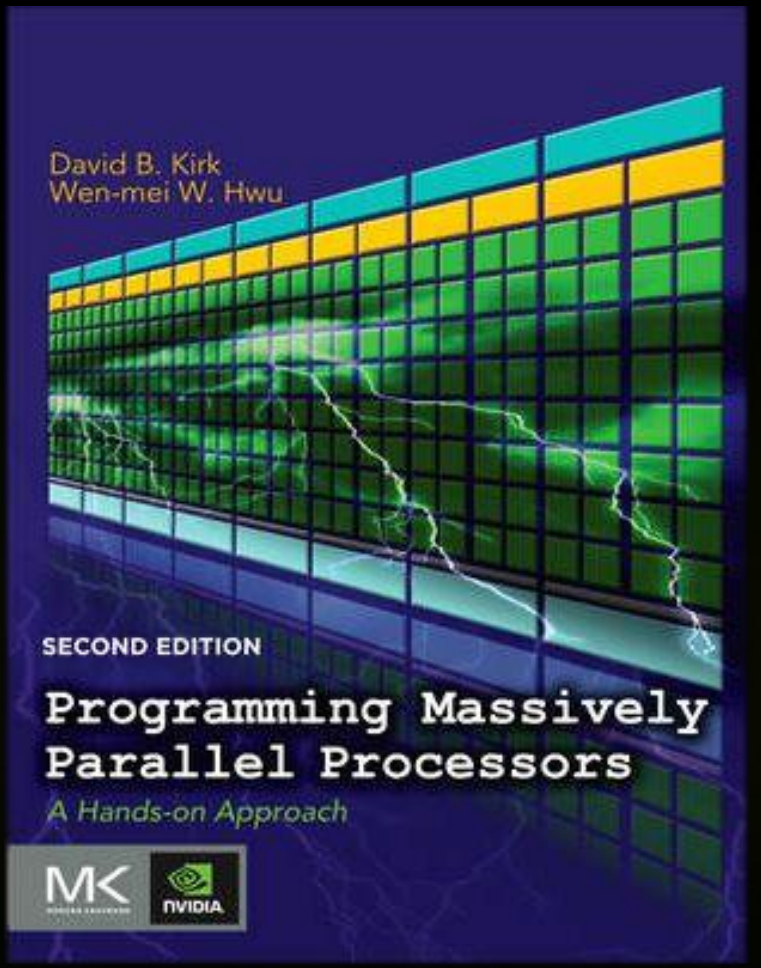
- Introduction to Parallel Programming
  https://www.udacity.com/course/cs344
- Learn the fundamentals of parallel computing with the GPU and the CUDA programming environment! In this class, you'll learn about parallel programming by coding a series of image processing algorithms, such as you might find in Photoshop or Instagram. You'll be able to program and run your assignments on high-end GPUs, even if you don't own one yourself.

© NVIDIA Corporation 2009

GPU Programming Text Book

David Kirk (NVIDIA) Wen-mei Hwu (UIUC)

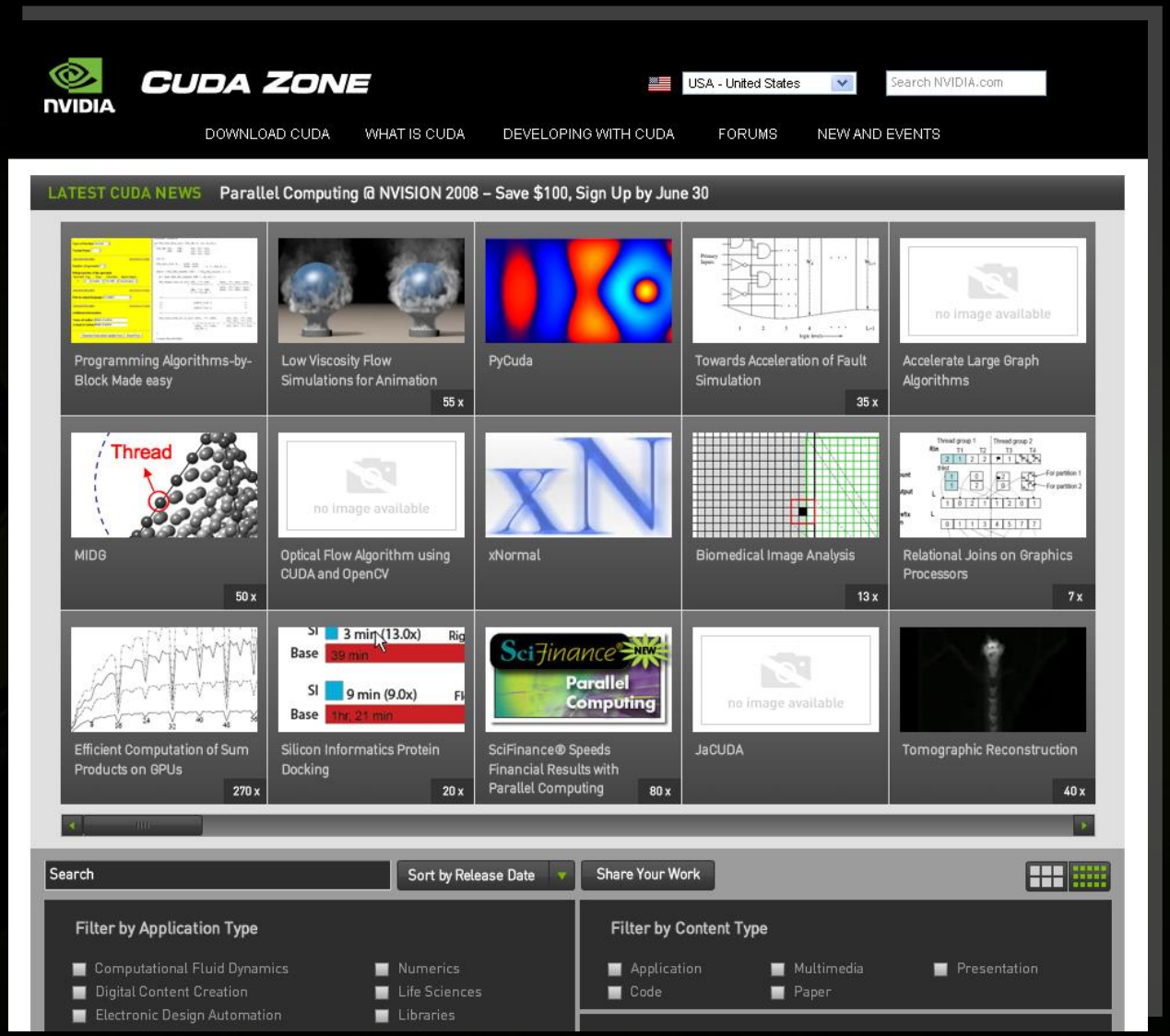
Chapter 1: Introduction
Chapter 2: History of GPU Computing
Chapter 3: Introduction to Data Parallelism and CUDA C
Chapter 4: Data-Parallel Execution Model
Chapter 5: CUDA Memories
Chapter 6: Performance Considerations
Chapter 7: Floating-Point Considerations
Chapter 8: Parallel Patterns: Convolution
Chapter 9: Parallel Patterns: Prefix Sum
Chapter 10: Parallel Patterns: Sparse Matrix-Vector Multiplication
Chapter 11: Application Case Study I - Advanced MRI Reconstruction
Chapter 12: Application Case Study II – Molecular Visualization and Analysis
Chapter 13: Parallel Programming and Computational Thinking
Chapter 14: A Brief Introduction to OpenCL
Chapter 15: Parallel Programming with OpenACC
Chapter 16: Thrust: A Productivity-Orientated Library for CUDA
Chapter 17: CUDA Fortran
Chapter 18: An Introduction to C++ AMP
Chapter 19: Programming a Heterogeneous Computing Cluster
Chapter 20: CUDA Dynamic Parallelism
Chapter 21: Conclusion and Future Outlook



© NVIDIA Corporation 2009

CUDA Zone: www.nvidia.com/CUDA

- CUDA Toolkit
  - Compiler
  - Libraries
- CUDA SDK
  - Code samples
- CUDA Profiler
- Forums
- Resources for CUDA developers



© NVIDIA Corporation 2009

CUDA Registered Developer Program

All GPGPU developers should become NVIDIA Registered Developers

Benefits include:

- Early Access to Pre-Release Software
  - Beta software and libraries
- Submit & Track Issues and Bugs
  - Interact directly with NVIDIA QA engineers
- New benefits from 2011
  - Exclusive Q&A Webinars with NVIDIA Engineering
  - Exclusive deep dive CUDA training webinars
  - In-depth engineering presentations on pre-release software

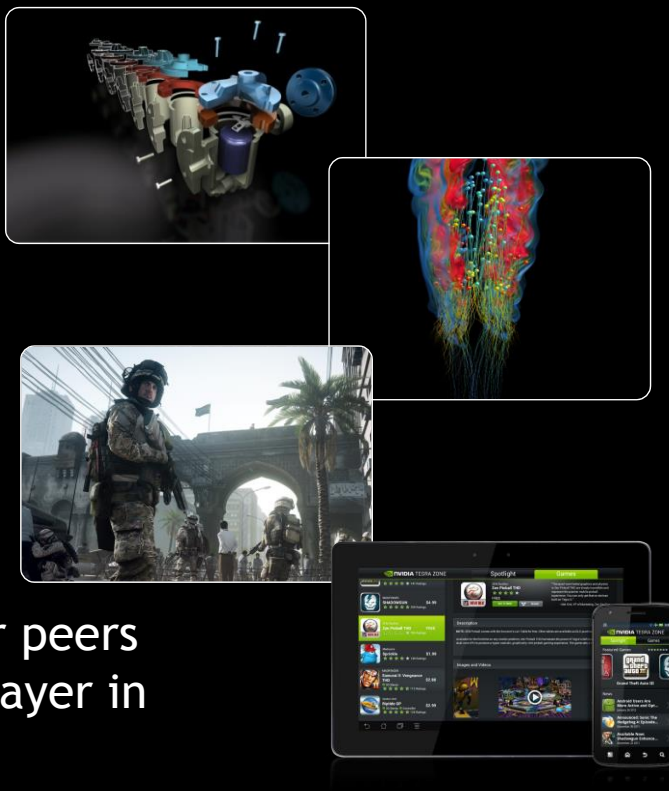
Sign up Now: www.nvidia.com/ParallelDeveloper

GPU TECHNOLOGY CONFERENCE

GPU Technology Conference 2013
March 18-21 | San Jose, CA

Why attend GTC?

GTC advances global awareness of the dramatic changes we're seeing in science and research, graphics, cloud computing, game development, and mobile computing, and how the GPU is central to innovation in all areas.



Ways to participate

- Submit a Research Poster – share your work and gain exposure as a thought leader
- Register – learn from the experts and network with your peers
- Exhibit/Sponsor – promote your organization as a key player in the GPU ecosystem

Visit www.gputechconf.com