CUDA Technical Training

Volume I:

Introduction to CUDA Programming

Prepared and Provided by NVIDIA

Q2 2008

Table of Contents

Introduction to GPU Computing

Parallel Computing’s Golden Age

- 1980s, early `90s: a golden age for parallel computing
  - Particularly *data-parallel computing*
- Machines
  - Connection Machine, MasPar, Cray
  - True supercomputers: incredibly exotic, powerful, expensive
- Algorithms, languages, & programming models
  - Solved a wide variety of problems
  - Various parallel algorithmic models developed
  - P-RAM, V-RAM, circuit, hypercube, etc.

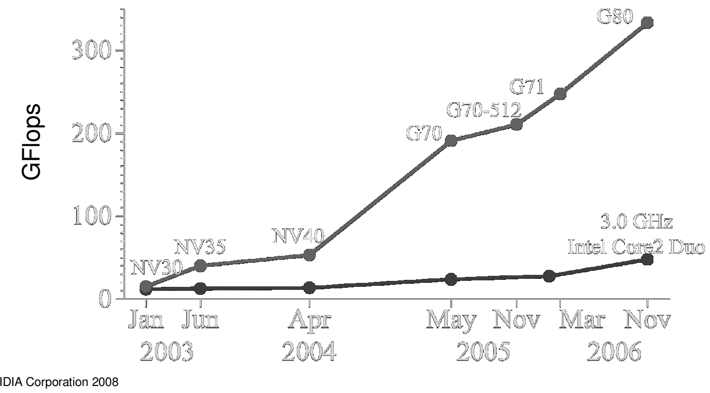
Parallel Computing’s Dark Age

- But…impact of data-parallel computing limited
  - Thinking Machines sold 7 CM-1s (100s of systems total)
  - MasPar sold ~200 systems
- Commercial and research activity subsided
  - Massively-parallel machines replaced by clusters of ever-more powerful commodity microprocessors
  - Beowulf, Legion, grid computing, …

Massively parallel computing loses momentum to inexorable advance of commodity technology

Enter the GPU

- GPUs are massively multithreaded manycore chips
  - Hundreds of cores, thousands of concurrent threads
  - Easily the most computationally intense workload on PCs
  - Huge economies of scale



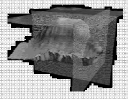
Enter CUDA

- Compute Unified Device Architecture
- Co-designed hardware and software to expose the computational horsepower of NVIDIA GPUs for GPU computing
- Software
  - Small set of extensions to C language
  - Low learning curve
- Hardware
  - Shared memory – scalable thread cooperation

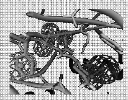
CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is a compute device
  - serves as a coprocessor for the host CPU
  - has its own device memory on the card
  - executes many threads in parallel
- Parallel kernels run a single program in many threads
- GPU threads are extremely lightweight
  - Thread creation and context switching are essentially free
- GPU expects 1000’s of threads for full utilization

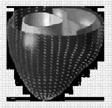
GPU Computing Example Markets



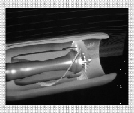
Computational Geoscience



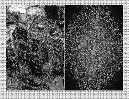
Computational Chemistry



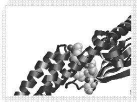
Computational Medicine



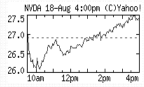
Computational Modeling



Computational Science



Computational Biology



Computational Finance

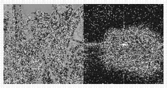
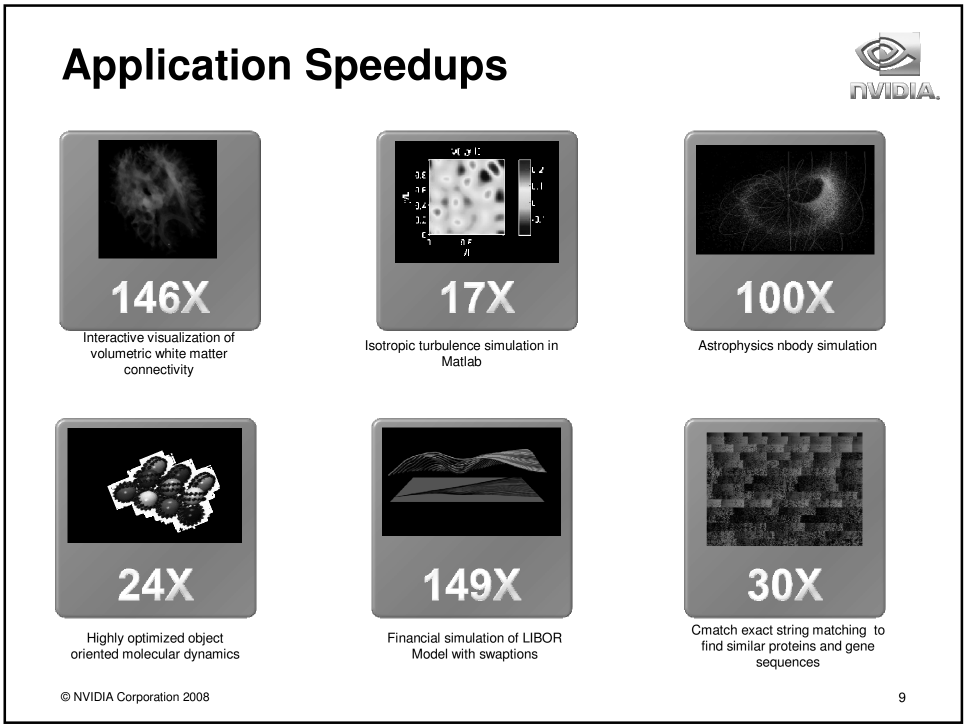


Image Processing

GPU Computing Sweet Spots

- Applications:
  - High arithmetic intensity:
    Dense linear algebra, PDEs, n-body, finite difference, …
  - High bandwidth:
    Sequencing (virus scanning, genomics), sorting, database…
  - Visual computing:
    Graphics, image processing, tomography, machine vision…



Application Speedups

146X

Interactive visualization of volumetric white matter connectivity

17X

Isotropic turbulence simulation in Matlab

100X

Astrophysics nbody simulation

24X

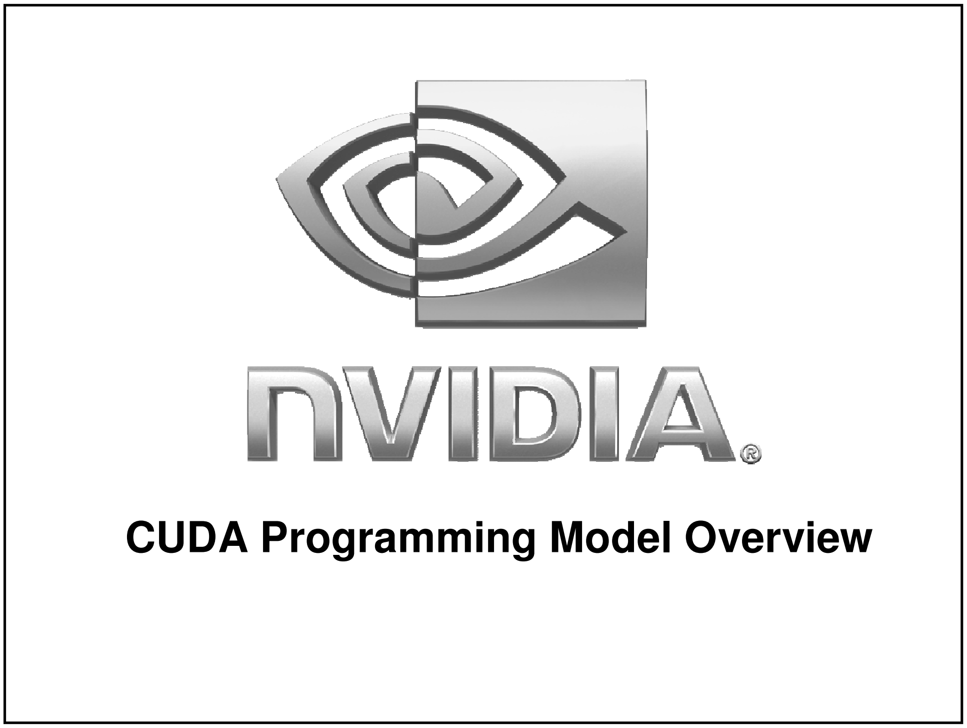
Highly optimized object oriented molecular dynamics

149X

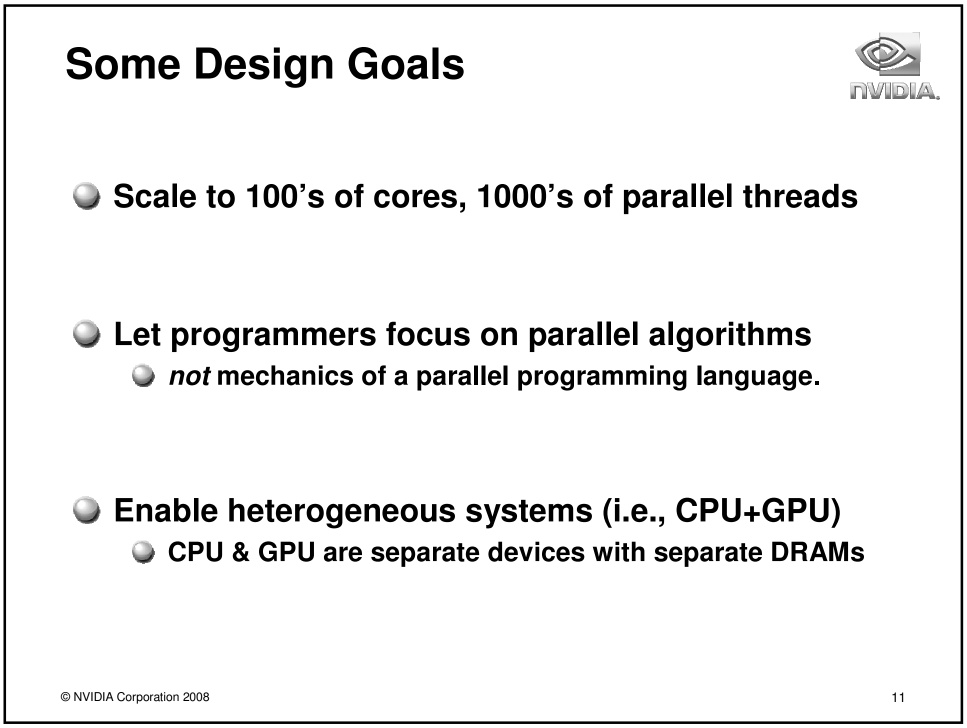
Financial simulation of LIBOR Model with swaptions

30X

Cmatch exact string matching to find similar proteins and gene sequences

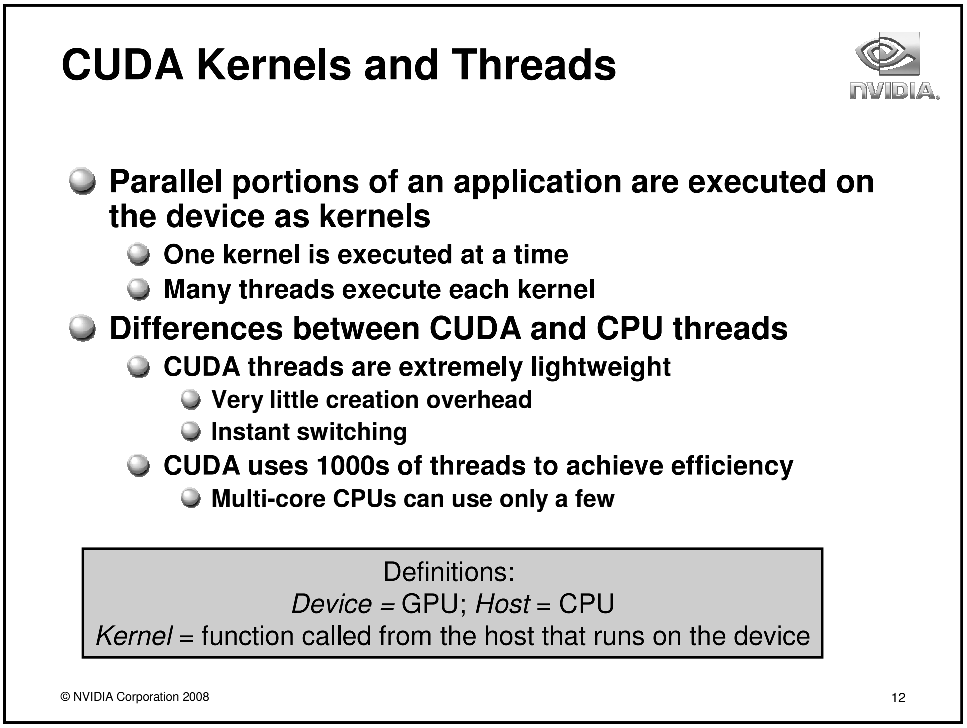


CUDA Programming Model Overview



Some Design Goals

- Scale to 100’s of cores, 1000’s of parallel threads
- Let programmers focus on parallel algorithms
  - *not* mechanics of a parallel programming language.
- Enable heterogeneous systems (i.e., CPU+GPU)
  - CPU & GPU are separate devices with separate DRAMs



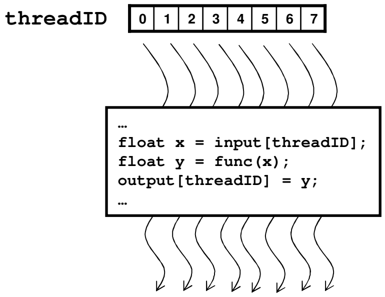
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions:
Device = GPU; *Host* = CPU
Kernel = function called from the host that runs on the device

Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

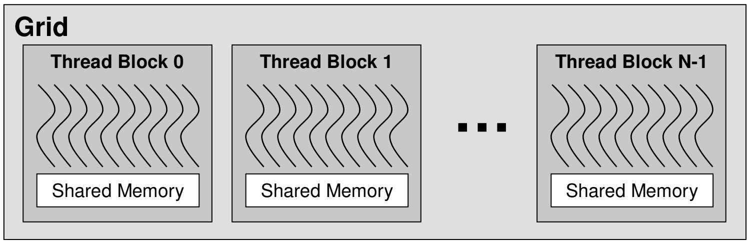
Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Drastic bandwidth reduction
- Thread cooperation is a powerful feature of CUDA

- Cooperation between a monolithic array of threads is not scalable
  - Cooperation within smaller batches of threads is scalable

Thread Batching

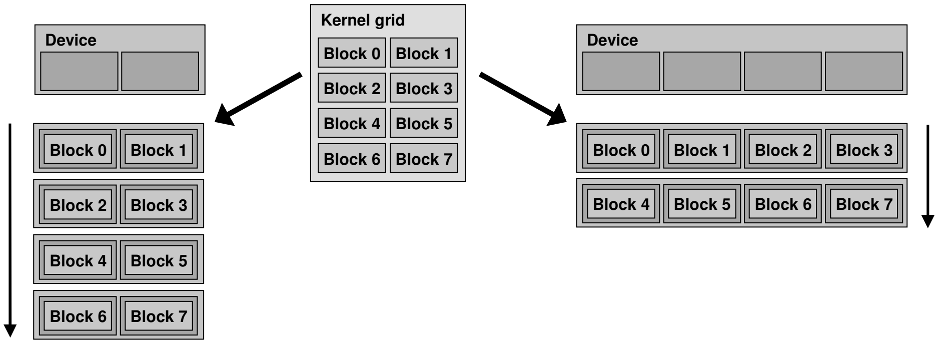
- Kernel launches a grid of thread blocks



  - Threads within a block cooperate via shared memory
  - Threads in different block cannot cooperate
- Allows programs to *transparently scale* to different GPUs

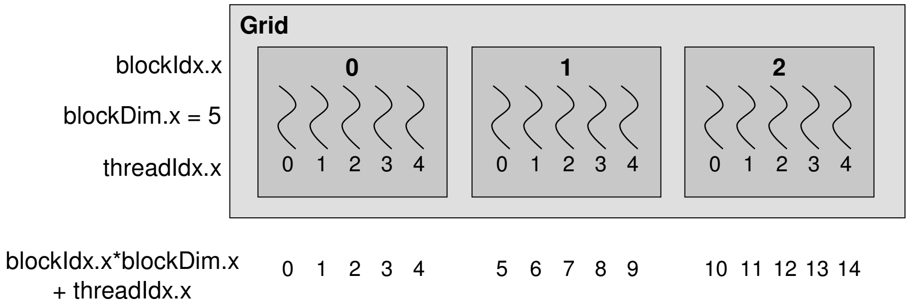
Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
  - A kernel scales across parallel multiprocessors



Data Decomposition

- Often want each thread in kernel to access a different element of an array
- Each thread has access to:
  - threadIdx.x **- thread ID within block**
  - blockIdx.x **- block ID within grid**
  - blockDim.x **- number of threads per block**

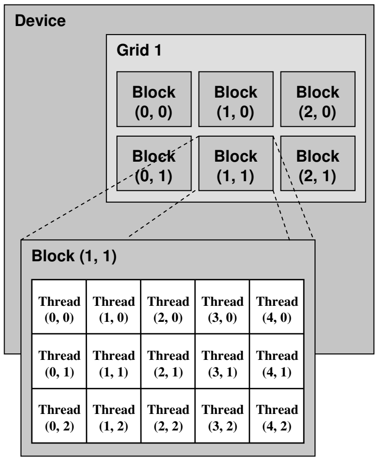


Grid

	0	1	2
blockIdx.x	0	1	2
blockDim.x = 5			
threadIdx.x	0 1 2 3 4	0 1 2 3 4	0 1 2 3 4
blockIdx.x*blockDim.x + threadIdx.x	0 1 2 3 4	5 6 7 8 9	10 11 12 13 14

Multidimensional IDs

- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes



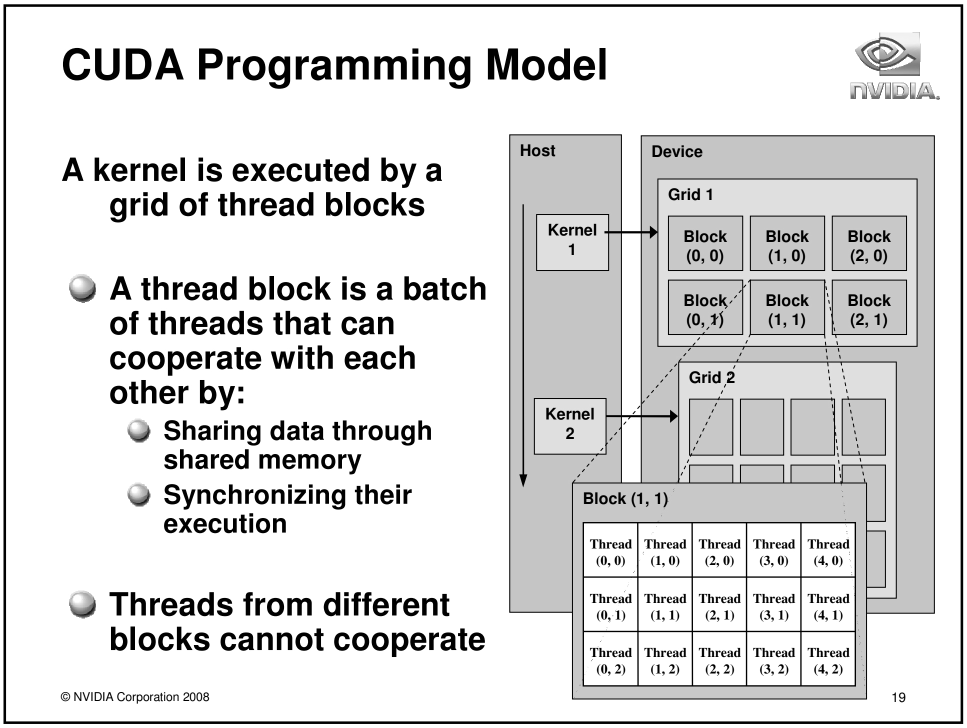
Device

Grid 1

Block (0, 0)	Block (1, 0)	Block (2, 0)
Block (0, 1)	Block (1, 1)	Block (2, 1)

Block (1, 1)

Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)

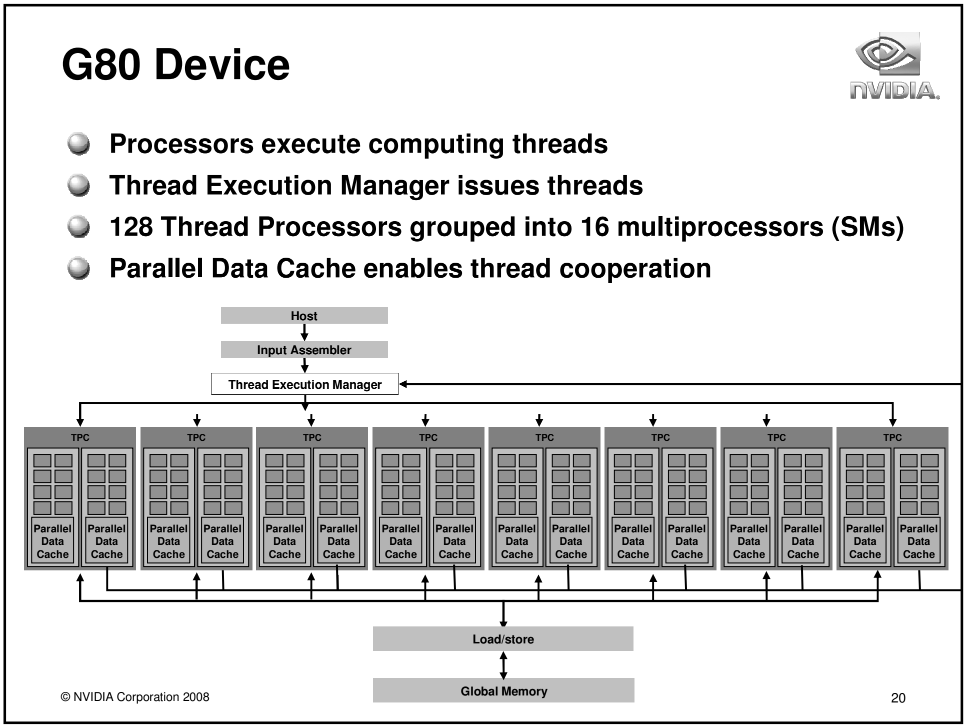


CUDA Programming Model

A kernel is executed by a grid of thread blocks

- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate

Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)

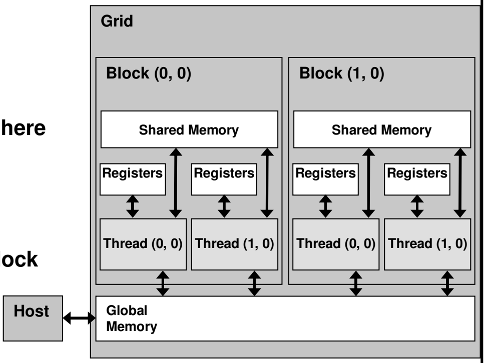


G80 Device

- Processors execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors grouped into 16 multiprocessors (SMs)
- Parallel Data Cache enables thread cooperation

Kernel Memory Access

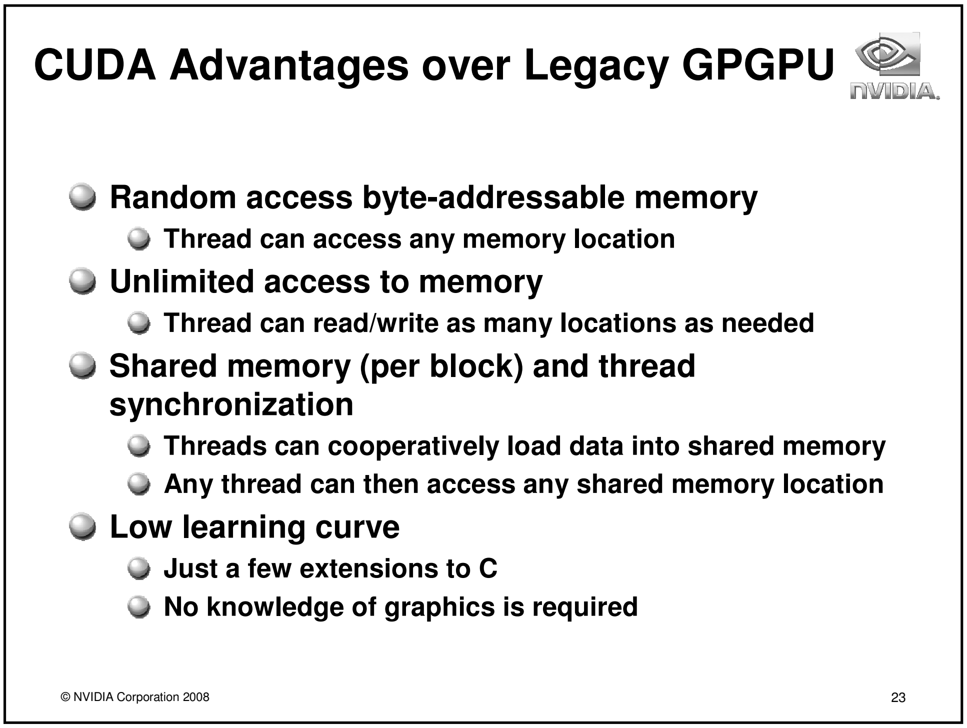
- Registers
- Global Memory
  - Kernel input and output data reside here
  - Off-chip, large
  - Uncached
- Shared Memory
  - Shared among threads in a single block
  - On-chip, small
  - As fast as registers



- The host can read & write global memory but not shared memory

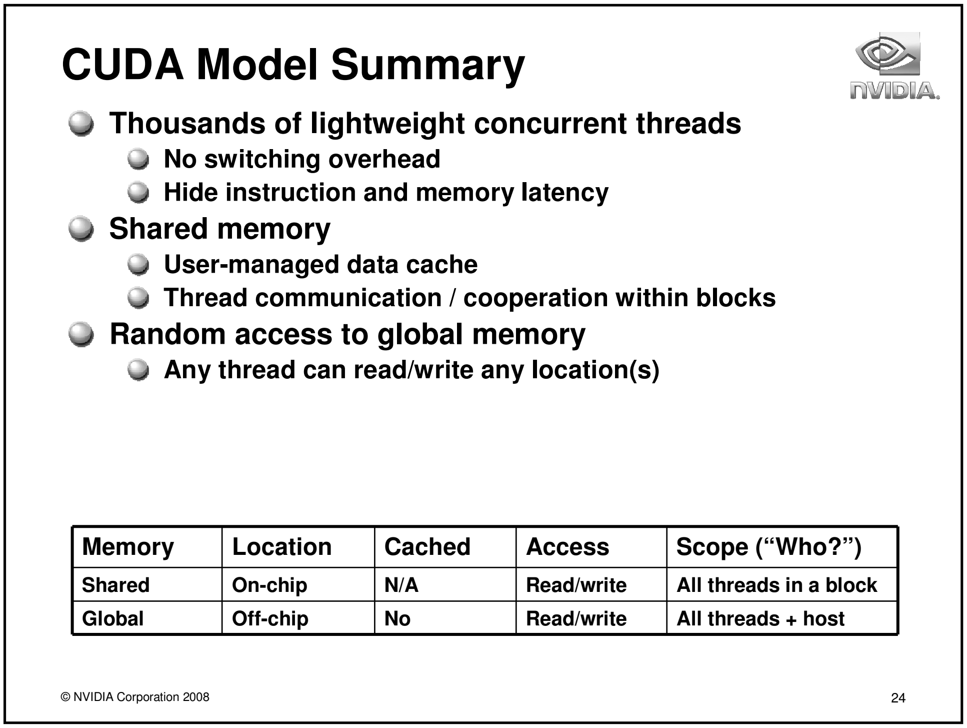
Execution Model

- Kernels are launched in grids
  - One kernel executes at a time
- A block executes on one multiprocessor
  - Does not migrate
- Several blocks can reside concurrently on one multiprocessor
  - Number is limited by multiprocessor resources
    - Register file is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks



CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
  - Thread can access any memory location
- Unlimited access to memory
  - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization
  - Threads can cooperatively load data into shared memory
  - Any thread can then access any shared memory location
- Low learning curve
  - Just a few extensions to C
  - No knowledge of graphics is required



CUDA Model Summary

- Thousands of lightweight concurrent threads
  - No switching overhead
  - Hide instruction and memory latency
- Shared memory
  - User-managed data cache
  - Thread communication / cooperation within blocks
- Random access to global memory
  - Any thread can read/write any location(s)

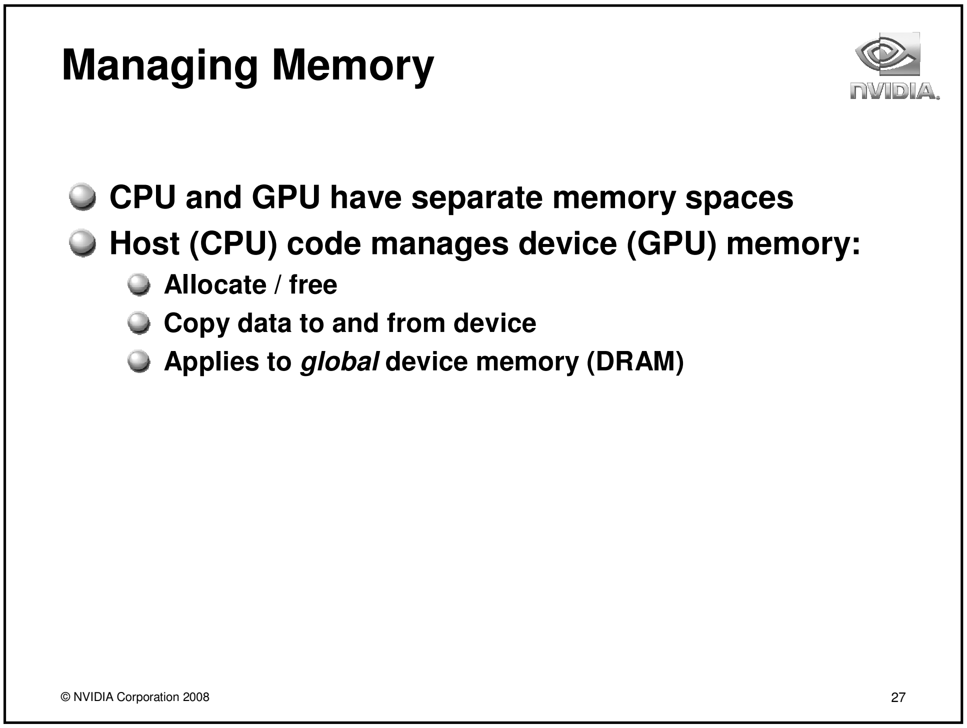
Memory	Location	Cached	Access	Scope ("Who?")
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host

CUDA Programming

The Basics

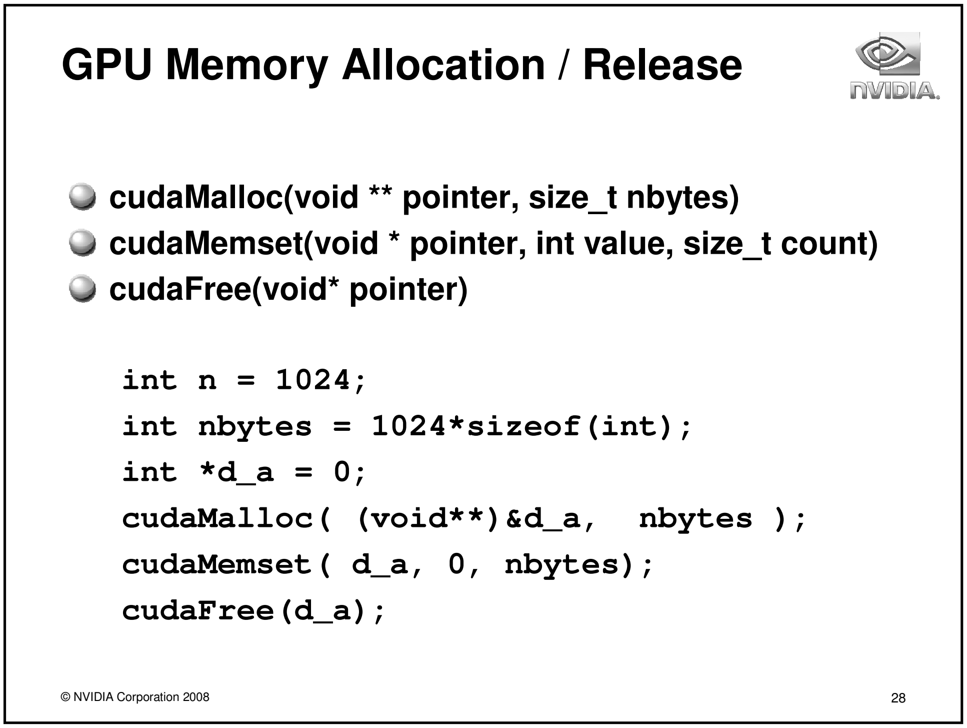
Outline of CUDA Basics

- Basics to set up and execute GPU code:
  - GPU memory management
  - GPU kernel launches
  - Some specifics of GPU code
- Some additional features:
  - Vector types
  - Synchronization
  - Checking CUDA errors
- NOTE: only the basic features are covered
  - See the Programming Guide for many more API functions
  - More in Optimization section



Managing Memory

- CPU and GPU have separate memory spaces
- Host (CPU) code manages device (GPU) memory:
  - Allocate / free
  - Copy data to and from device
  - Applies to *global* device memory (DRAM)



GPU Memory Allocation / Release

- cudaMalloc(void ** pointer, size_t nbytes)
- cudaMemset(void * pointer, int value, size_t count)
- cudaFree(void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
  - `direction` specifies locations (host or device) of `src` and `dst`
  - Blocks CPU thread: returns after the copy is complete
  - Doesn’t start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice

CUDA Exercises

- We have provided skeletons and solutions for hands-on CUDA exercises
- In each exercise, you have to implement the missing portions of the code
  - Finished when you compile and run the program and get the output “Correct!”
- Solutions are included in the “solution” folder of each exercise

Compiling the Code: Windows

- Open the <project>.sln file in Microsoft Visual Studio
  - Build the project
  - Four configuration choices:
    - Release,Debug,EmuRelease, EmuDebug
- To debug your code build EmuDebug configuration
  - Can set breakpoints inside kernels (__global__ or __device__ functions)
  - Can debug the code as normal, even printf!
  - One CPU thread per GPU thread
  - Threads not actually in parallel on GPU

Compiling the Code: Linux

```
nvcc <filename>.cu [-o <executable>]
```

- Builds release mode

```
nvcc -g <filename>.cu
```

- Builds debug (device) mode
- Can debug host code but not device code (runs on GPU)

```
nvcc -deviceemu <filename>.cu
```

- Builds device emulation mode
- All code runs on CPU, but no debug symbols

```
nvcc -deviceemu -g <filename>.cu
```

- Builds debug device emulation mode
- All code runs on CPU, with debug symbols
- Debug using gdb or other linux debugger

Exercise 1: Copying between host and device

- **Start from the "cudaMallocAndMemcpy" template.**
- **Part1: Allocate memory for pointers *d_a* and *d_b* on the device.**
- **Part2: Copy *h_a* on the host to *d_a* on the device.**
- **Part3: Do a device to device copy from *d_a* to *d_b*.**
- **Part4: Copy *d_b* on the device back to *h_a* on the host.**
- **Part5: Free *d_a* and *d_b* on the host.**

Executing Code on the GPU

- **Kernels are C functions with some restrictions**
  - **Can only access GPU memory**
  - **Must have void return type**
  - **No variable number of arguments ("varargs")**
  - **Not recursive**
  - **No static variables**
- **Function arguments automatically copied from CPU to GPU memory**

Function Qualifiers

- **__global__ : invoked from within host (CPU) code,**
  cannot be called from device (GPU) code
  must return void

- **__device__ : called from other GPU functions,**
  cannot be called from host (CPU) code

- **__host__ : can only be executed by CPU, called from host**

- **__host__ and __device__ qualifiers can be combined**
  - **Sample use: overloading operators**
  - **Compiler will generate both CPU and GPU code**

Launching kernels

- **Modified C function call syntax:**

```
kernel<<<dim3 grid, dim3 block>>>(…)
```

- **Execution Configuration ("<<< >>>"):**
  - **grid dimensions: x and y**
  - **thread-block dimensions: x, y, and z**

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

CUDA Built-in Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block

Minimal Kernels

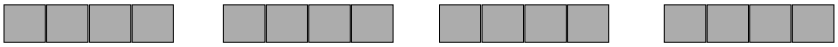
```
__global__ void minimal( int* d_a)
{
    *d_a = 13;
}

__global__ void assign( int* d_a, int value)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    d_a[idx] = value;
}
```

Common Pattern!

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let’s assume N=16, blockDim=4 -> 4 blocks



blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

int idx = blockDim.x * blockId.x + threadIdx.x;
will map from local index threadIdx to global index

NB: blockDim should be >= 32 in real code, this is just an example

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
        for (int idx = 0; idx<N; idx++)
            a[idx] = a[idx] + b;
}


void main()
{
   .....
      increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx < N)
            a[idx] = a[idx] + b;
}


void main()
{
   …
      dim3 dimBlock (blocksize);
      dim3 dimGrid( ceil( N / (float)blocksize) );
      increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Minimal Kernel for 2D data

```
__global__ void assign2D(int* d_a, int w, int h, int value)
{
    int iy = blockDim.y * blockIdx.y + threadIdx.y;
    int ix = blockDim.x * blockIdx.x + threadIdx.x;
    int idx = iy * w + ix;

    d_a[idx] = value;
}
...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```

Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- cudaMemcpy() is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete

Example: Host Code

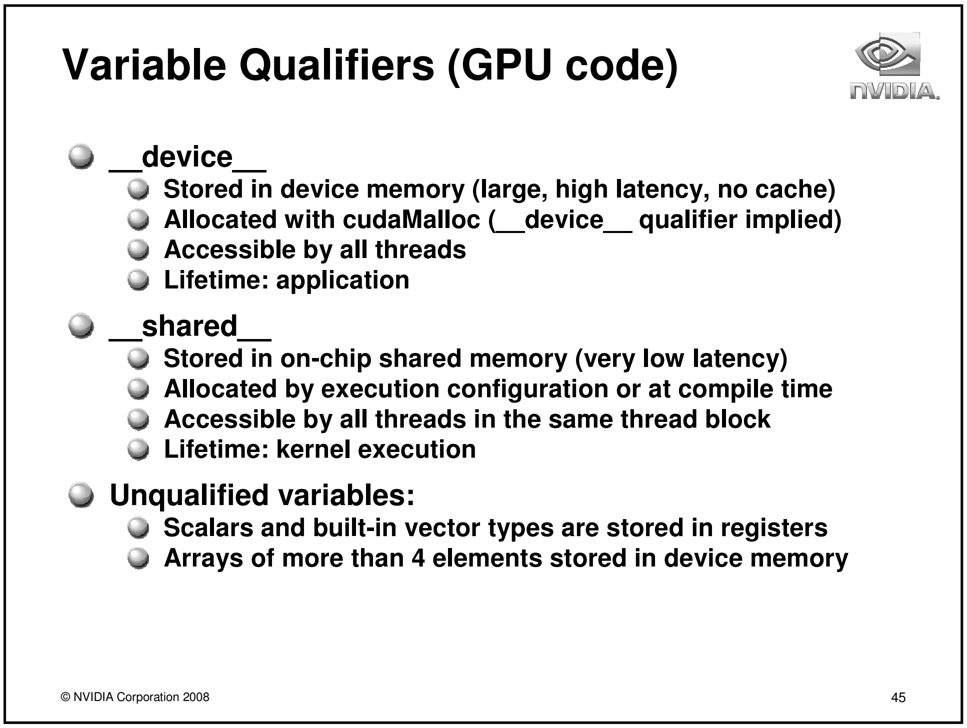
```
// allocate host memory
int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);


// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);


// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```

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:**

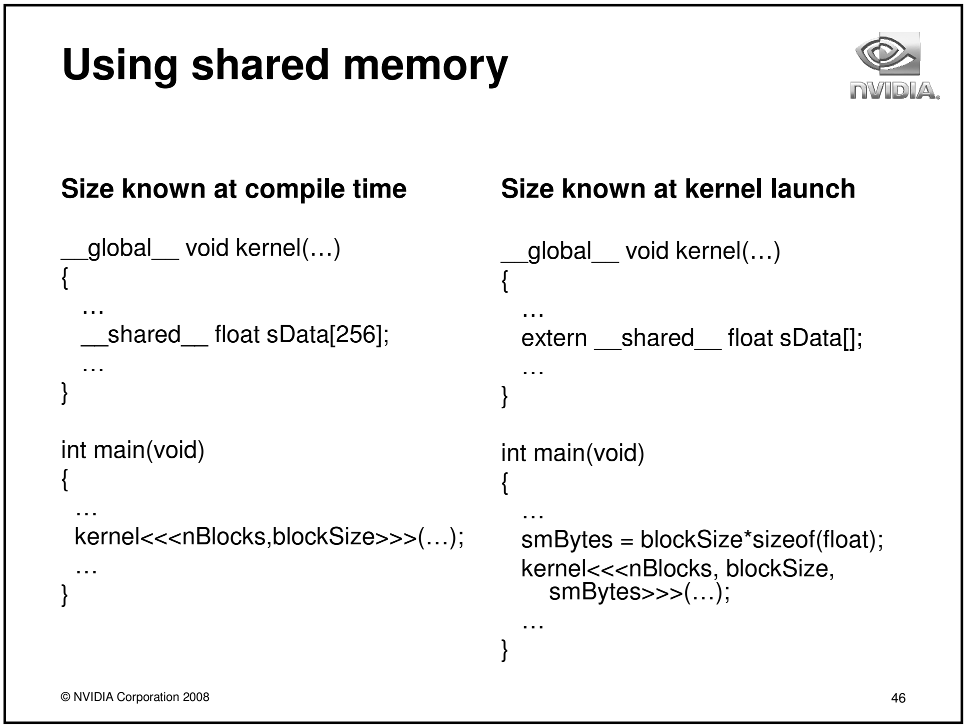
```
idx = blockIdx.x*blockDim.x + threadIdx.x
d_a[idx] = 1000*blockIdx.x + threadIdx.x
```

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



Variable Qualifiers (GPU code)

- **__device__**
  - **Stored in device memory (large, high latency, no cache)**
  - **Allocated with cudaMalloc (__device__ qualifier implied)**
  - **Accessible by all threads**
  - **Lifetime: application**
- **__shared__**
  - **Stored in on-chip shared memory (very low latency)**
  - **Allocated by execution configuration or at compile time**
  - **Accessible by all threads in the same thread block**
  - **Lifetime: kernel execution**
- **Unqualified variables:**
  - **Scalars and built-in vector types are stored in registers**
  - **Arrays of more than 4 elements stored in device memory**



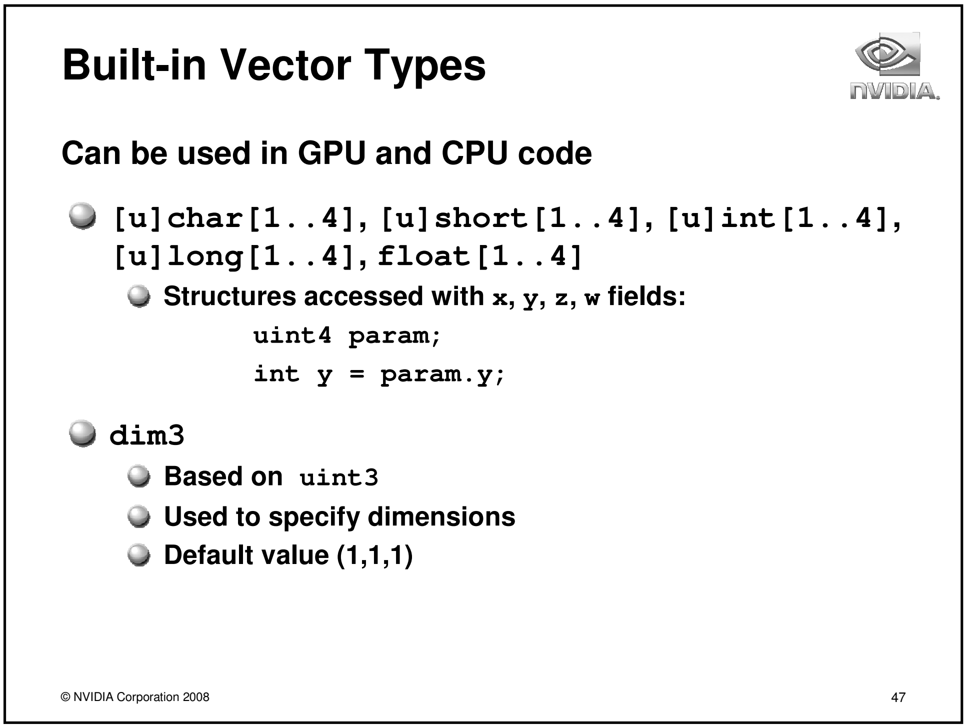
Using shared memory

Size known at compile time

```
__global__ void kernel(…)
{
   …
   __shared__ float sData[256];
   …
}

int main(void)
{
  …
  kernel<<<nBlocks,blockSize>>>(…);
  …
}
```

Size known at kernel launch

```
__global__ void kernel(…)
{
   …
   extern __shared__ float sData[];
   …
}

int main(void)
{
   …
   smBytes = blockSize*sizeof(float);
   kernel<<<nBlocks, blockSize,
       smBytes>>>(…);
   …
}
```



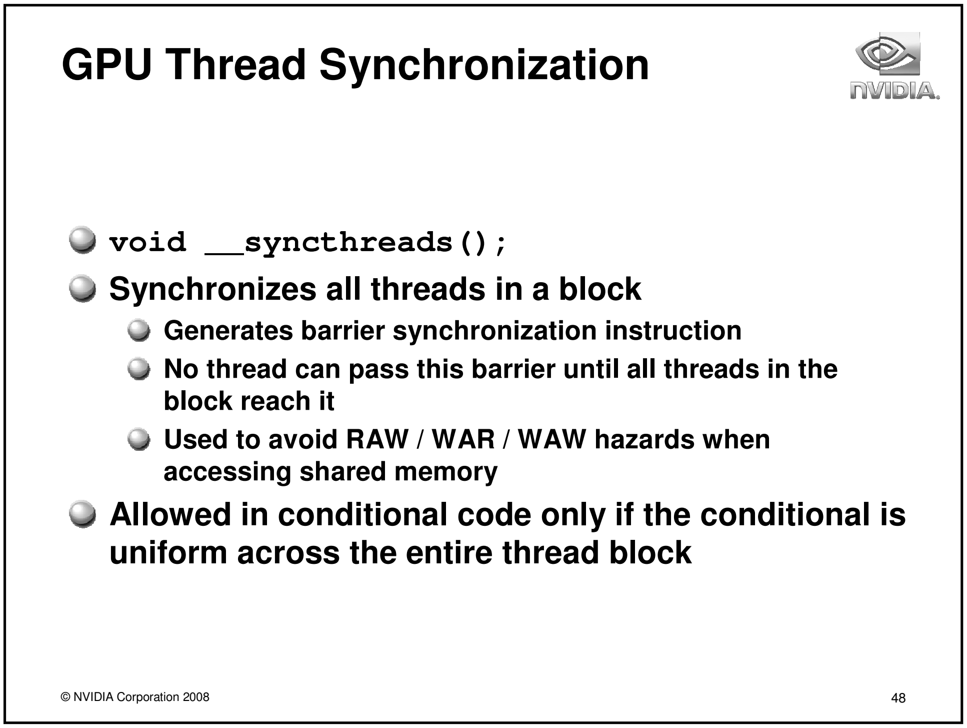
Built-in Vector Types

Can be used in GPU and CPU code

- `[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]`
  - Structures accessed with `x`, `y`, `z`, `w` fields:

```
uint4 param;
int y = param.y;
```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions
  - Default value (1,1,1)



GPU Thread Synchronization

- `void __syncthreads();`
- Synchronizes all threads in a block
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block

GPU Atomic Integer Operations

- Requires hardware with compute capability 1.1
  - G80 = Compute capability 1.0
  - G84/G86/G92 = Compute capability 1.1
- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap

CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - Except for kernel launches
  - cudaError_t type
- cudaError_t cudaGetLastError(void)
  - Returns the code for the last error (no error has a code)
  - Can be used to get error from kernel execution
- char* cudaGetErrorString(cudaError_t code)
  - Returns a null-terminated character string describing the error

```
printf(“%s\n”, cudaGetErrorString( cudaGetLastError() ) );
```

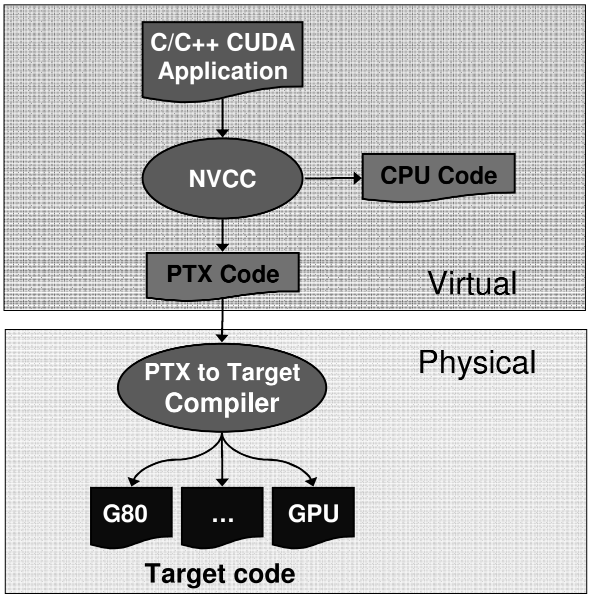
Exercise 3: Reverse Array (single block)

- Given an input array {a_0, a_1, …, a_{n-1}} in pointer *d_a*, store the reversed array {a_{n-1}, a_{n-2}, …, 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

Exercise 4: Reverse Array (multiblock)

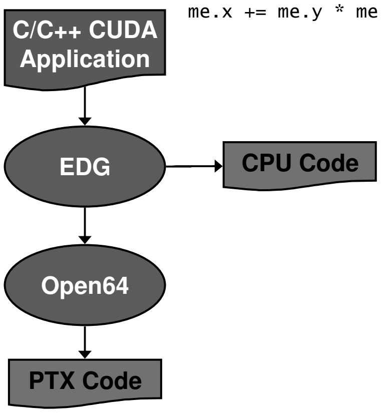
- Given an input array {a_0, a_1, …, a_{n-1}} in pointer *d_a*, store the reversed array {a_{n-1}, a_{n-2}, …, a_0} in pointer *d_b*
- Start from the "reverseArray_multiblock" template
- Multiple 256-thread blocks launched
  - To reverse an array of size N, N/256 blocks
- Part 1: Compute the number of 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

Compiling CUDA



NVCC & PTX Virtual Machine

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```



- EDG
  - Separate GPU vs. CPU code
- Open64
  - Generates GPU PTX assembly
- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```

Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
  - Either C code (CPU Code)
    - That must then be compiled with the rest of the application using another tool
  - Or PTX object code directly
- An executable with CUDA code requires:
  - The CUDA core library (`cuda`)
  - The CUDA runtime library (`cudart`)
    - if runtime API is used
    - loads cuda library

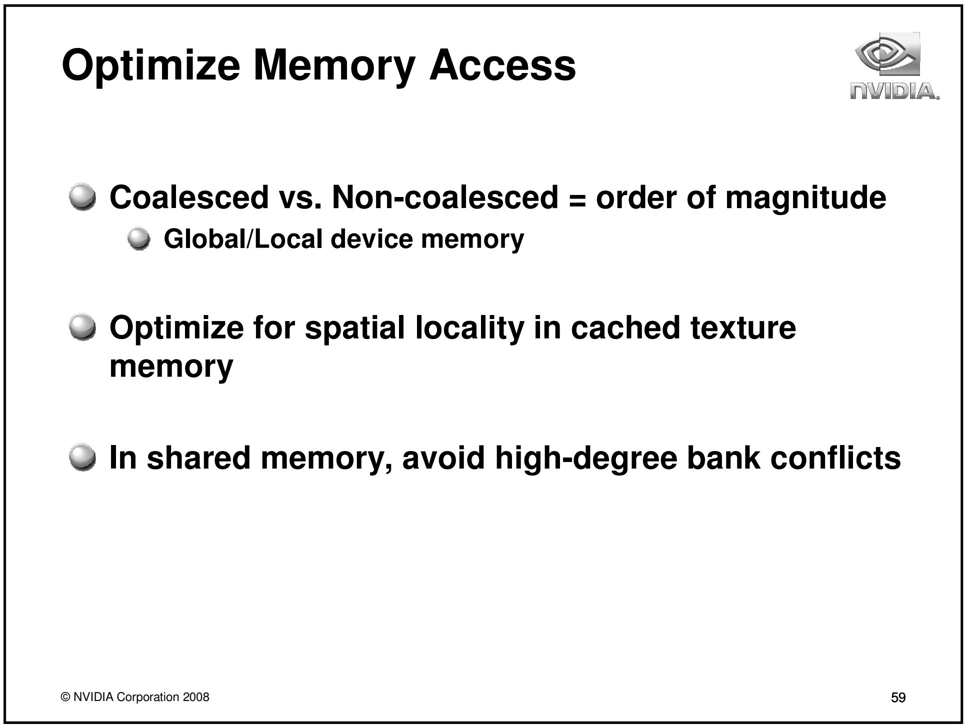
Performance Optimization

Outline

- Overview
- G8x Hardware
- Memory Optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

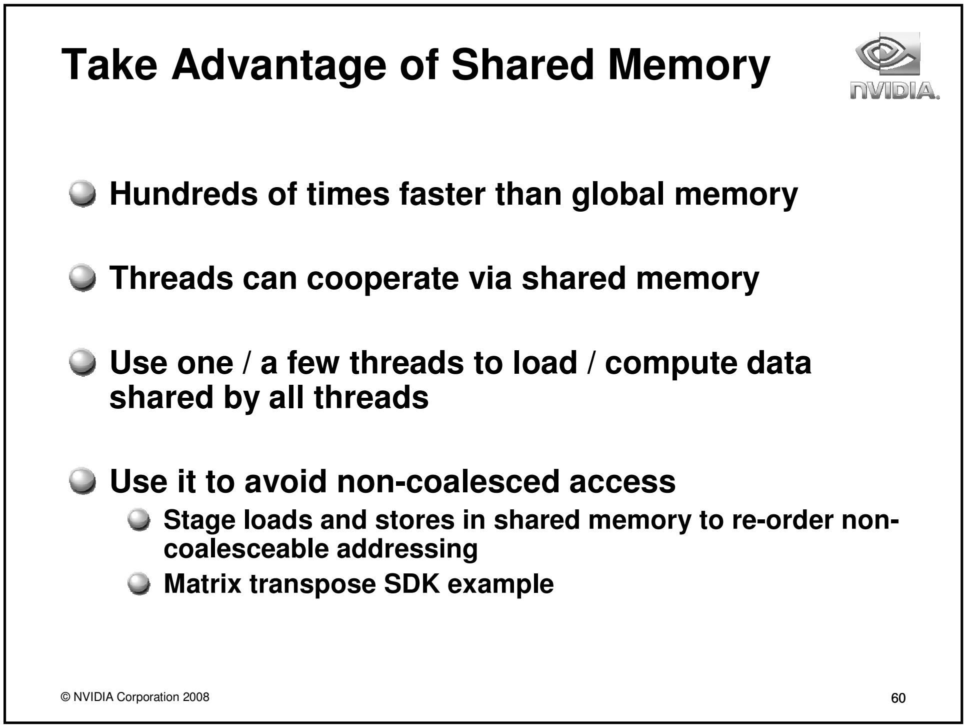
Optimize Algorithms for the GPU

- Maximize independent parallelism
- Maximize arithmetic intensity (math/bandwidth)
- Sometimes it's better to recompute than to cache
  - GPU spends its transistors on ALUs, not memory
- Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host



Optimize Memory Access

- Coalesced vs. Non-coalesced = order of magnitude
  - Global/Local device memory
- Optimize for spatial locality in cached texture memory
- In shared memory, avoid high-degree bank conflicts



Take Advantage of Shared Memory

- Hundreds of times faster than global memory
- Threads can cooperate via shared memory
- Use one / a few threads to load / compute data shared by all threads
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - Matrix transpose SDK example

Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

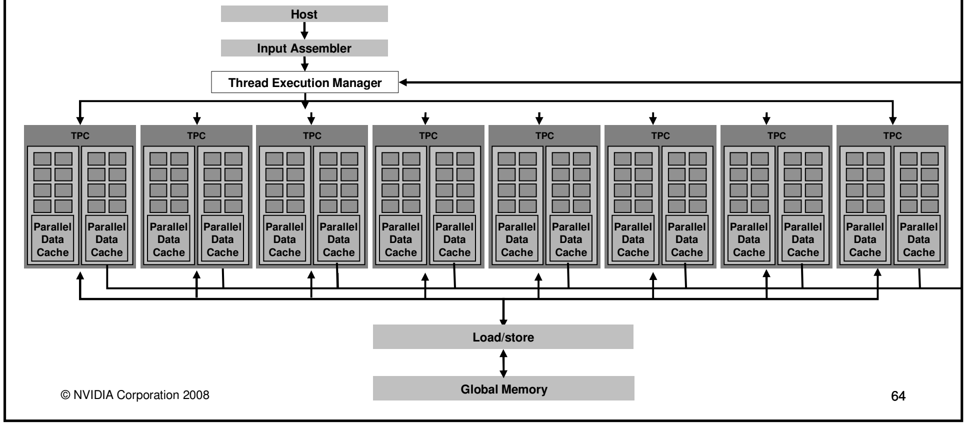
G8x Hardware

Terminology

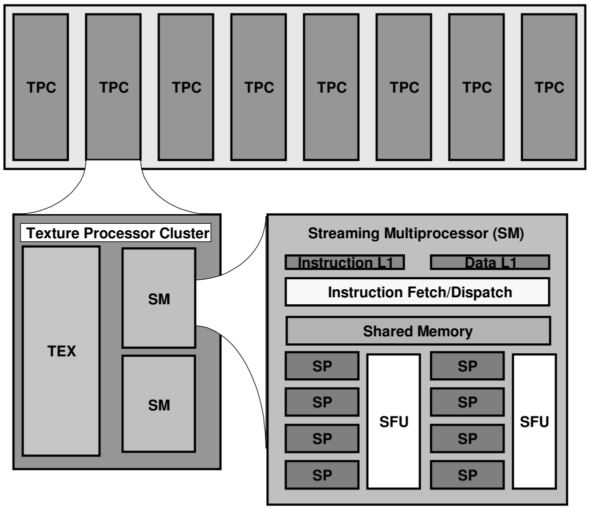
- Thread: concurrent code and associated state executed on the CUDA device (in parallel with other threads)
  - The unit of parallelism in CUDA
  - Note difference from CPU threads: creation cost, resource usage, and switching cost of GPU threads is much smaller
- Warp: a group of threads executed *physically* in parallel (SIMD)
  - *Half-warp*: the first or second half of a warp of threads
- Thread Block: a group of threads that are executed together and can share memory on a single multiprocessor
- Grid: a group of thread blocks that execute a single CUDA kernel *logically* in parallel on a single GPU

G80 Device

- Processors execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors
- Parallel Data Cache accelerates processing

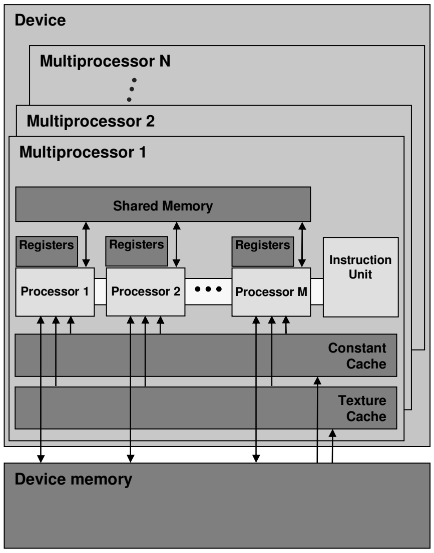


Texture Processor Cluster (TPC)



Memory Architecture

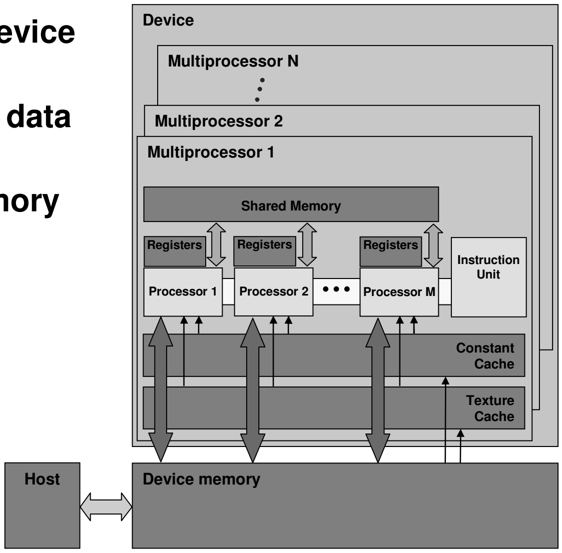
- The global, constant, and texture spaces are regions of device memory
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
    - Where the shared memory space resides
  - A read-only constant cache
    - To speed up access to the constant memory space
  - A read-only texture cache
    - To speed up access to the texture memory space



Memory Optimizations

Memory optimizations

- Optimizing host-device data transfers
- Coalescing global data accesses
- Using shared memory effectively



Host-Device Data Transfers

- Device memory to host memory bandwidth much lower than device memory to device bandwidth
  - 4GB/s peak (PCI-e x16 Gen 1) vs. 76 GB/s peak (Tesla C870)
- Minimize transfers
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to host memory
- Group transfers
  - One large transfer much better than many small ones

Page-Locked Data Transfers

- cudaMallocHost() allows allocation of page-locked ("pinned") host memory
- Enables highest cudaMemcpy performance
  - 3.2 GB/s on PCI-e x16 Gen1
  - 5.2 GB/s on PCI-e x16 Gen2
- See the "bandwidthTest" CUDA SDK sample
- Use with caution!!
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits

Asynchronous memory copy

- Asynchronous host-device memory copy for pinned memory (allocated with “cudaMallocHost” in C) frees up CPU on all CUDA capable devices
- Overlap implemented by using a stream
- Stream = Sequence of operations that execute in order
- Stream API:
  - 0 = default stream
  - `cudaMemcpyAsync(dst, src, size, direction, 0);`

Overlap kernel and memory copy

- Concurrent execution of a kernel and a host ←→ device memory copy for pinned memory
  - Devices with compute capability >= 1.1 (G84 and up)
  - Available as a preview feature in CUDA toolkit v1.1
  - Overlaps kernel execution in one stream with a memory copy from another stream
- Stream API:

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, dir, stream1);  }
kernel<<<grid, block, 0, stream2>>>(…);         }
cudaStreamQuery(stream2);                   overlapped
```

Global and Shared Memory

- Global memory not cached on G8x GPUs
  - High latency, but launching more threads hides latency
  - Important to minimize accesses
  - Coalesce global memory accesses (more later)
- Shared memory is on-chip, very high bandwidth
  - Low latency
  - Like a user-managed per-multiprocessor cache
  - Try to minimize or avoid bank conflicts (more later)

Texture and Constant Memory

- Texture partition is cached
  - Uses the texture cache also used for graphics
  - Optimized for 2D spatial locality
  - Best performance when threads of a warp read locations that are close together in 2D
- Constant memory is cached
  - 4 cycles per address read within a single warp
    - Total cost 4 cycles if all threads in a warp read same address
    - Total cost 64 cycles if all threads read different addresses

Global Memory Reads/Writes

- Global memory is not cached on G8x
- Highest latency instructions: 400-600 clock cycles
- Likely to be a performance bottleneck
- Optimizations can greatly increase performance

Loading and storing global memory

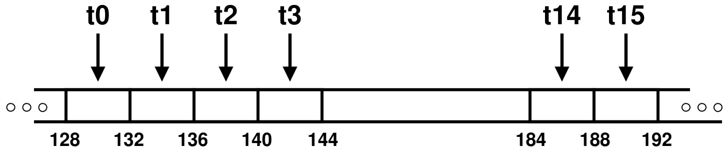
- Use -ptx flag of nvcc to inspect instructions:

```
                         ld.global.f32      $f1, [$rd4+0];       // id:74
4 byte load and store    …
                         st.global.f32      [$rd4+0], $f2;       // id:75
                         …
                         ld.global.v2.f32   {$f3,$f5}, [$rd7+0];              //
8 byte load and store    …
                         st.global.v2.f32   [$rd7+0], {$f4,$f6};              //
                         …
                         ld.global.v4.f32   {$f7,$f9,$f11,$f13}, [$rd10+0];            //
16 byte load and store   …
                         st.global.v4.f32   [$rd10+0], {$f8,$f10,$f12,$f14};           //
```

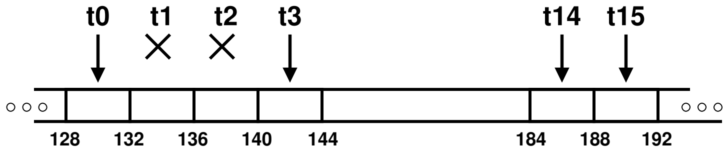
Coalescing

- A coordinated read by a half-warp (16 threads)
- A contiguous region of global memory:
  - 64 bytes - each thread reads a word: int, float, …
  - 128 bytes - each thread reads a double-word: int2, float2, …
  - 256 bytes – each thread reads a quad-word: int4, float4, …
- Additional restrictions:
  - Starting address for a region must be a multiple of region size
  - The *k*th thread in a half-warp must access the *k*th element in a block being read
- Exception: not all threads must be participating
  - Predicated access, divergence within a halfwarp

Coalesced Access: Reading floats

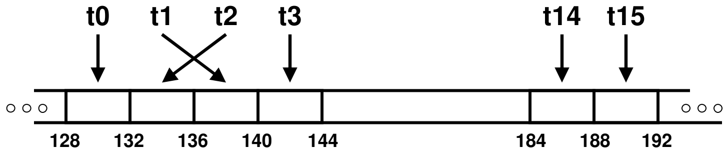


All threads participate

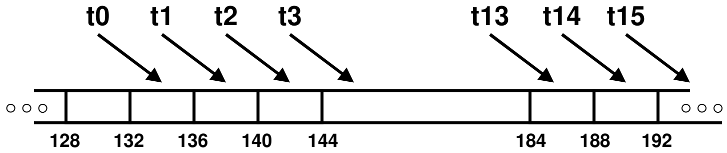


Some Threads Do Not Participate

Uncoalesced Access: Reading floats



Permuted Access by Threads



Misaligned Starting Address (not a multiple of 64)

Coalescing: Timing Results

- Experiment:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs
- 12K blocks x 256 threads:
  - 356µs – coalesced
  - 357µs – coalesced, some threads don’t participate
  - 3,494µs – permuted/misaligned thread access

Hands On: Array Reversal Revisited

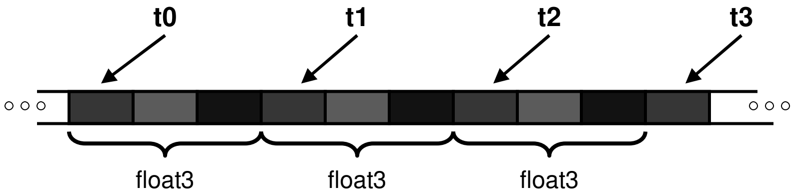
- **Considering the limitations on memory coalescing, analyze the data access patterns in your implementation.**
- **What, if anything, could be done to improve the data access pattern?**

Uncoalesced float3 Code

```
__global__ void accessFloat3(float3 *d_in, float3 d_out)
{
   int index = blockIdx.x * blockDim.x + threadIdx.x;
   float3 a = d_in[index];

   a.x += 2;
   a.y += 2;
   a.z += 2;

   d_out[index] = a;
}
```

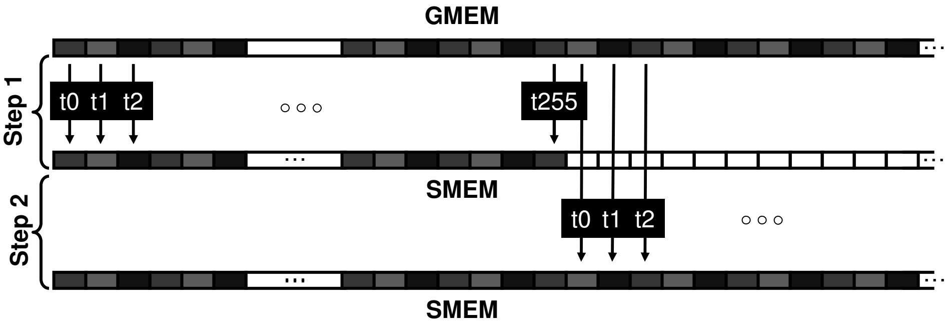
Uncoalesced Access: float3 Case

- float3 is 12 bytes
- Each thread ends up executing 3 reads
  - sizeof(float3) ≠ 4, 8, or 16
  - Half-warp reads three 64B non-contiguous regions



First read

Coalescing float3 Access



Similarly, Step3 starting at offset 512

Coalesced Access: float3 Case

- Use shared memory to allow coalescing
  - Need sizeof(float3)*(threads/block) bytes of SMEM
  - Each thread reads 3 scalar floats:
    - Offsets: 0, (threads/block), 2*(threads/block)
    - These will likely be processed by other threads, so sync
- Processing
  - Each thread retrieves its float3 from SMEM array
    - Cast the SMEM pointer to (float3*)
    - Use thread ID as index
  - Rest of the compute code does not change!

Coalesced float3 Code

```
__global__ void accessInt3Shared(float *g_in, float *g_out)
{
    int index = 3 * blockIdx.x * blockDim.x + threadIdx.x;
    __shared__ float s_data[256*3];
    s_data[threadIdx.x]         = g_in[index];
    s_data[threadIdx.x+256] = g_in[index+256];
    s_data[threadIdx.x+512] = g_in[index+512];
    __syncthreads();
    float3 a = ((float3*)s_data)[threadIdx.x];

    a.x += 2;
    a.y += 2;
    a.z += 2;

    ((float3*)s_data)[threadIdx.x] = a;
    __syncthreads();
    g_out[index]         = s_data[threadIdx.x];
    g_out[index+256] = s_data[threadIdx.x+256];
    g_out[index+512] = s_data[threadIdx.x+512];
}
```

Read the input through SMEM

Compute code is not changed

Write the result through SMEM

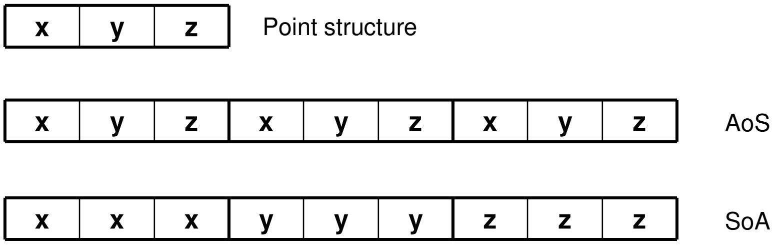
Coalescing: Timing Results

- Experiment:
  - Kernel: read a float, increment, write back
  - 3M floats (12MB)
  - Times averaged over 10K runs
- 12K blocks x 256 threads reading floats:
  - 356µs – coalesced
  - 357µs – coalesced, some threads don't participate
  - 3,494µs – permuted/misaligned thread access
- 4K blocks x 256 threads reading float3s:
  - 3,302µs – float3 uncoalesced
  - 359µs – float3 coalesced through shared memory

Coalescing: Structures of size ≠ 4, 8, or 16 Bytes

- Use a Structure of Arrays (SoA) instead of Array of Structures (AoS)

- If SoA is not viable:
  - Force structure alignment: __align(X), where X = 4, 8, or 16
  - Use SMEM to achieve coalescing



x	y	z							Point structure
x	y	z	x	y	z	x	y	z	AoS
x	x	x	y	y	y	z	z	z	SoA

Coalescing: Summary

- Coalescing greatly improves throughput
- Critical to memory-bound kernels
- Reading structures of size other than 4, 8, or 16 bytes will break coalescing:
  - Prefer Structures of Arrays over AoS
  - If SoA is not viable, read/write through SMEM
- Additional resources:
  - Aligned Types SDK Sample

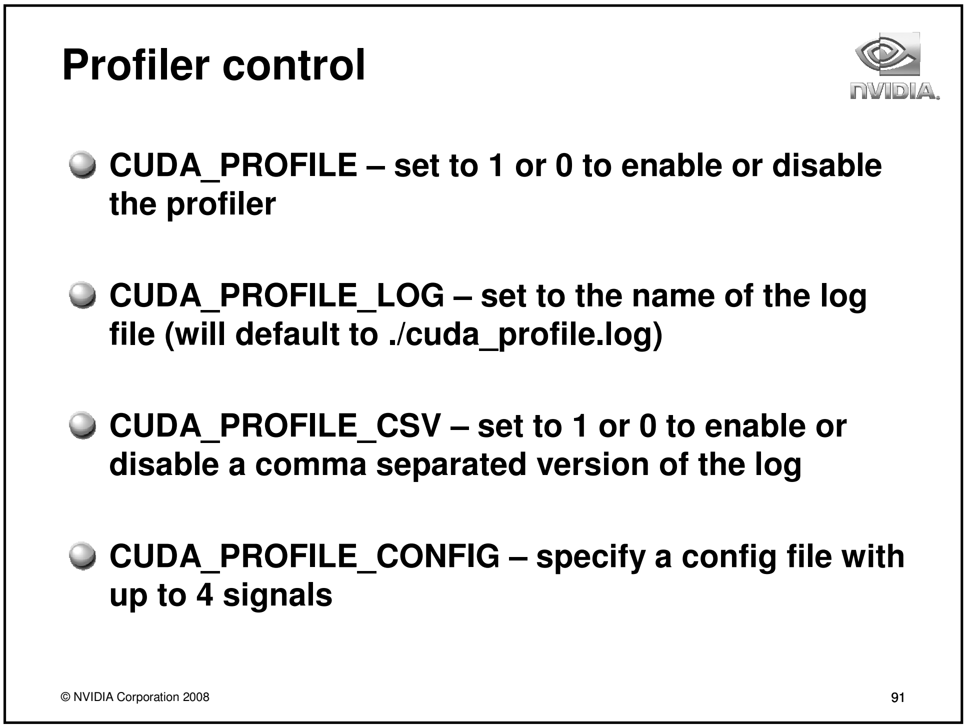
Profiler Signals

- Events are tracked with hardware counters on signals in the chip:
  - timestamp
  - gld_incoherent
  - gld_coherent
  - gst_incoherent
  - gst_coherent

    Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent)
  - local_load
  - local_store

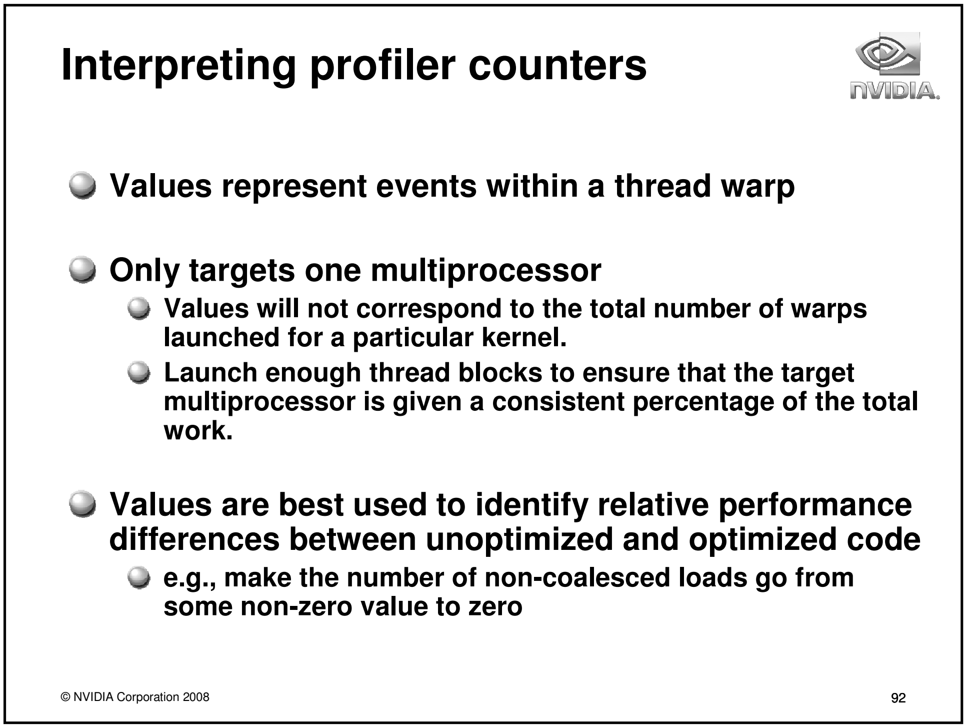
    Local loads/stores
  - branch
  - divergent_branch

    Total branches and divergent branches taken by threads
  - instructions – instruction count
  - warp_serialize – thread warps that serialize on address conflicts to shared or constant memory
  - cta_launched – executed thread blocks



Profiler control

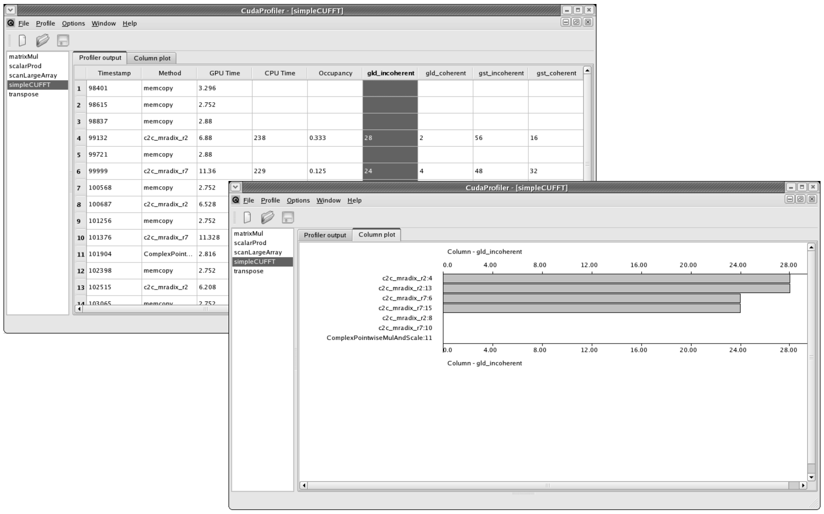
- CUDA_PROFILE – set to 1 or 0 to enable or disable the profiler
- CUDA_PROFILE_LOG – set to the name of the log file (will default to ./cuda_profile.log)
- CUDA_PROFILE_CSV – set to 1 or 0 to enable or disable a comma separated version of the log
- CUDA_PROFILE_CONFIG – specify a config file with up to 4 signals



Interpreting profiler counters

- Values represent events within a thread warp
- Only targets one multiprocessor
  - Values will not correspond to the total number of warps launched for a particular kernel.
  - Launch enough thread blocks to ensure that the target multiprocessor is given a consistent percentage of the total work.
- Values are best used to identify relative performance differences between unoptimized and optimized code
  - e.g., make the number of non-coalesced loads go from some non-zero value to zero

Visual Profiler



CudaProfiler - [simpleCUFFT]

	Timestamp	Method	GPU Time	CPU Time	Occupancy	gld_incoherent	gld_coherent	gst_incoherent	gst_coherent
1	98401	memcopy	3.296						
2	98615	memcopy	2.752						
3	98837	memcopy	2.88						
4	99132	c2c_mradix_r2	6.88	238	0.333	28	2	56	16
5	99721	memcopy	2.88						
6	99999	c2c_mradix_r7	11.36	229	0.125	24	4	48	32
7	100568	memcopy	2.752						
8	100687	c2c_mradix_r2	6.528						
9	101256	memcopy	2.752						
10	101376	c2c_mradix_r7	11.328						
11	101904	ComplexPoint...	2.816						
12	102398	memcopy	2.752						
13	102515	c2c_mradix_r2	6.208						
14	103065	memcopy	2.752						

CudaProfiler - [simpleCUFFT]

Hands On: Using the Profiler

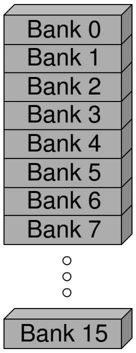
- **Use the profiler (either command line and text configuration files or the visual interface) to confirm the analysis of the data access patterns for your in-place array reversal implementation.**

Shared Memory

- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - See “Matrix Transpose” SDK example

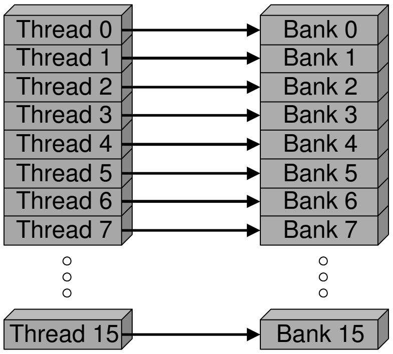
Parallel Memory Architecture

- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized

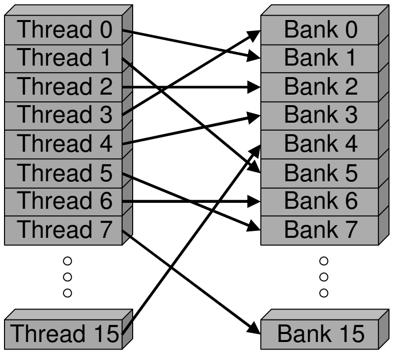


Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

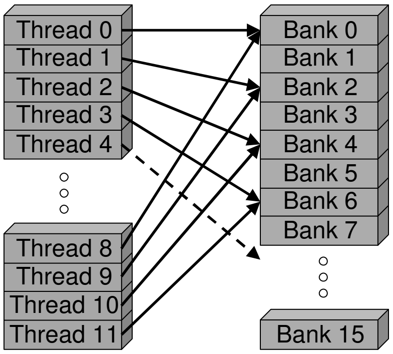


- No Bank Conflicts
  - Random 1:1 Permutation

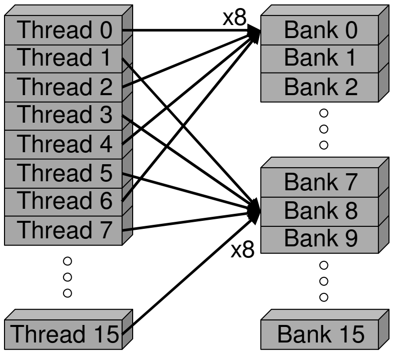


Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8



Shared memory bank conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- Use the bank checker macro in the SDK to check for conflicts
    - warp_serialize signal can usually be used to check for conflicts
- The fast case:
    - If all threads of a half-warp access different banks, there is no bank conflict
    - If all threads of a half-warp read the identical address, there is no bank conflict (broadcast)
- The slow case:
    - Bank Conflict: multiple threads in the same half-warp access the same bank
    - Must serialize the accesses
    - Cost = max # of simultaneous accesses to a single bank

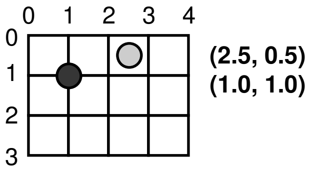
Hands On: Array Reversal Performance

- Improve your array reversal code to access global memory with coalesced loads and stores by using shared memory.

Textures in CUDA

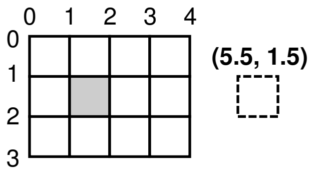
- Texture is an object for *reading* data
- Benefits:
  - Data is cached (optimized for 2D locality)
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear
    - dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable in 1D, 2D, or 3D
    - Using integer or normalized coordinates
- Usage:
  - CPU code binds data to a texture object
  - Kernel reads data by calling a *fetch* function

Texture Addressing



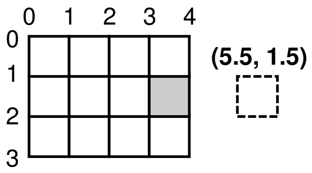
Wrap

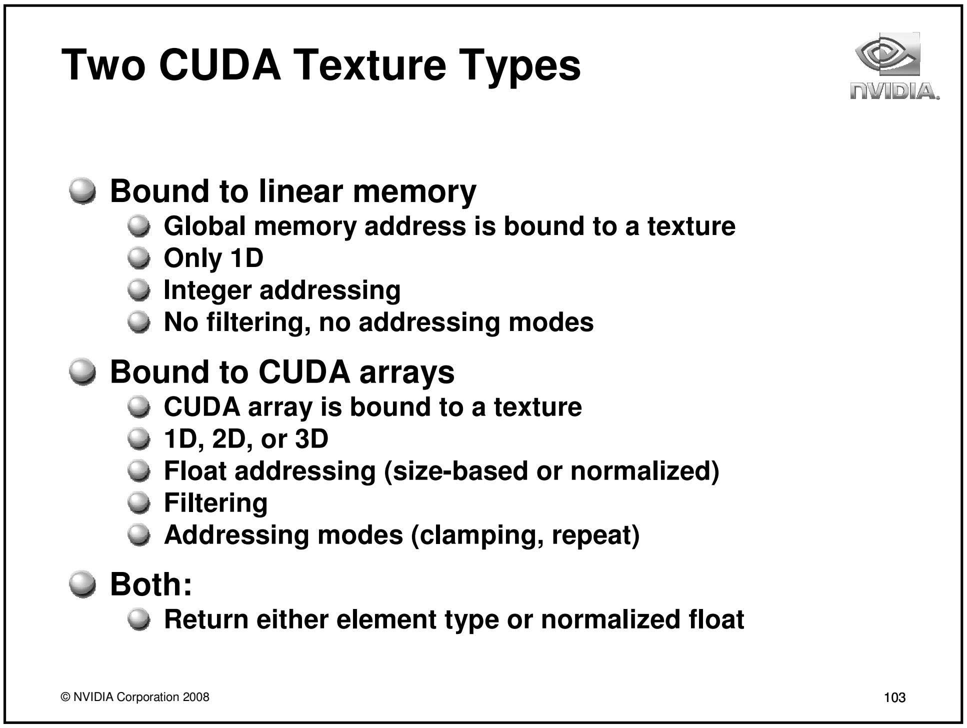
- Out-of-bounds coordinate is wrapped (modulo arithmetic)



Clamp

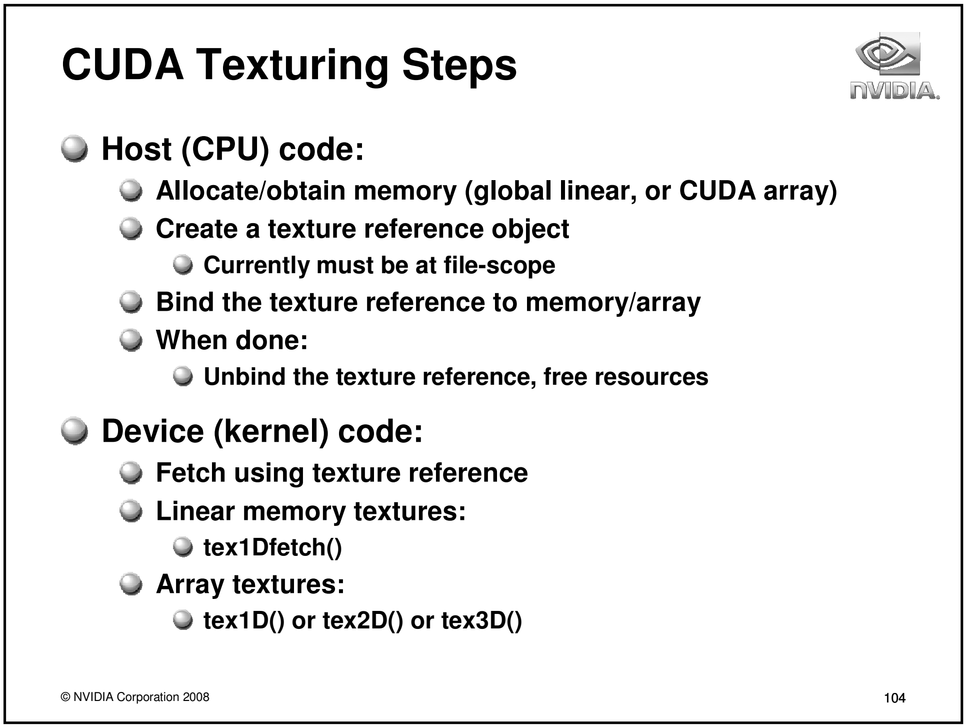
- Out-of-bounds coordinate is replaced with the closest boundary





Two CUDA Texture Types

- Bound to linear memory
  - Global memory address is bound to a texture
  - Only 1D
  - Integer addressing
  - No filtering, no addressing modes
- Bound to CUDA arrays
  - CUDA array is bound to a texture
  - 1D, 2D, or 3D
  - Float addressing (size-based or normalized)
  - Filtering
  - Addressing modes (clamping, repeat)
- Both:
  - Return either element type or normalized float



CUDA Texturing Steps

- Host (CPU) code:
  - Allocate/obtain memory (global linear, or CUDA array)
  - Create a texture reference object
    - Currently must be at file-scope
  - Bind the texture reference to memory/array
  - When done:
    - Unbind the texture reference, free resources
- Device (kernel) code:
  - Fetch using texture reference
  - Linear memory textures:
    - tex1Dfetch()
  - Array textures:
    - tex1D() or tex2D() or tex3D()

Execution Configuration Optimizations

Occupancy

- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy
- Occupancy = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently
- Limited by resource usage:
  - Registers
  - Shared memory

Grid/Block Size Heuristics

- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute
- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren't waiting at a __syncthreads() keep the hardware busy
  - Subject to resource availability – registers, shared memory
- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations

Register Dependency

- Read-after-write register dependency
  - Instruction's result can be read ~11 cycles later
  - Scenarios: CUDA: PTX:

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

```
add.f32   $f3, $f1, $f2
add.f32   $f5, $f3, $f4
```

```
s_data[0] += 3;
```

```
ld.shared.f32  $f3, [$r31+0]
add.f32        $f3, $f3, $f4
```

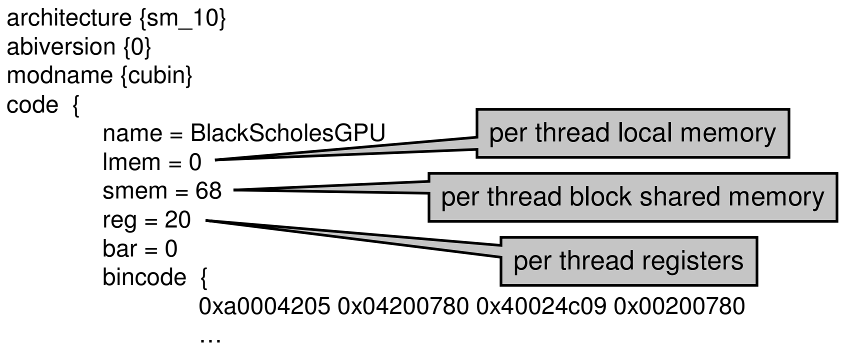
- To completely hide the latency:
  - Run at least 192 threads (6 warps) per multiprocessor
    - At least 25% occupancy
  - Threads do not have to belong to the same thread block

Register Pressure

- Hide latency by using more threads per SM
- Limiting Factors:
  - Number of registers per kernel
    - 8192 per SM, partitioned among concurrent threads
  - Amount of shared memory
    - 16KB per SM, partitioned among concurrent threadblocks
- Compile with –ptxas-options=-v flag
- Use –maxrregcount=N flag to NVCC
  - N = desired maximum registers / kernel
  - At some point “spilling” into LMEM may occur
    - Reduces performance – LMEM is slow

Determining resource usage

- Compile the kernel code with the -cubin flag to determine register usage.
- Open the .cubin file with a text editor and look for the “code” section.



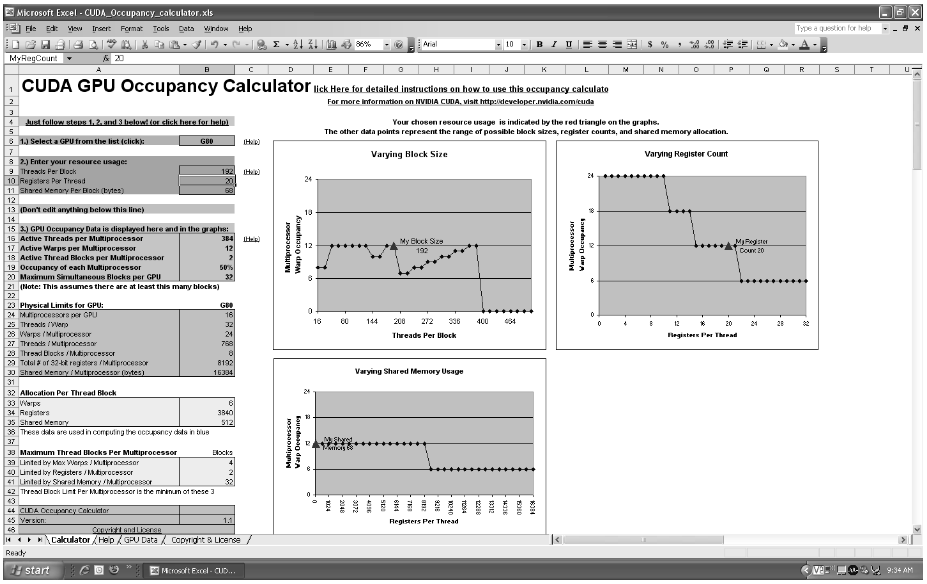
```
architecture {sm_10}
abiversion {0}
modname {cubin}
code  {
        name = BlackScholesGPU
        lmem = 0
        smem = 68
        reg = 20
        bar = 0
        bincode  {
                0xa0004205 0x04200780 0x40024c09 0x00200780
                …
```

per thread local memory

per thread block shared memory

per thread registers

CUDA Occupancy Calculator



Microsoft Excel - CUDA_Occupancy_calculator.xls

MyRegCount fx 20

	A	B	C
1	**CUDA GPU Occupancy Calculator** lick Here for detailed instructions on how to use this occupancy calculato		
2	For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda		
3			
4	**Just follow steps 1, 2, and 3 below! (or click here for help)**		
5			
6	**1.) Select a GPU from the list (click):**	**G80**	(Help)
7			
8	**2.) Enter your resource usage:**		
9	Threads Per Block	192	(Help)
10	Registers Per Thread	20	
11	Shared Memory Per Block (bytes)	68	
12			
13	**(Don't edit anything below this line)**		
14			
15	**3.) GPU Occupancy Data is displayed here and in the graphs:**		
16	**Active Threads per Multiprocessor**	**384**	(Help)
17	**Active Warps per Multiprocessor**	**12**	
18	**Active Thread Blocks per Multiprocessor**	**2**	
19	**Occupancy of each Multiprocessor**	**50%**	
20	**Maximum Simultaneous Blocks per GPU**	**32**	
21	**(Note: This assumes there are at least this many blocks)**		
22			
23	**Physical Limits for GPU:**	**G80**	
24	Multiprocessors per GPU	16	
25	Threads / Warp	32	
26	Warps / Multiprocessor	24	
27	Threads / Multiprocessor	768	
28	Thread Blocks / Multiprocessor	8	
29	Total # of 32-bit registers / Multiprocessor	8192	
30	Shared Memory / Multiprocessor (bytes)	16384	
31			
32	**Allocation Per Thread Block**		
33	Warps	6	
34	Registers	3840	
35	Shared Memory	512	
36	These data are used in computing the occupancy data in blue		
37			
38	**Maximum Thread Blocks Per Multiprocessor**	Blocks	
39	Limited by Max Warps / Multiprocessor	4	
40	Limited by Registers / Multiprocessor	2	
41	Limited by Shared Memory / Multiprocessor	32	
42	Thread Block Limit Per Multiprocessor is the minimum of these 3		
43			
44	CUDA Occupancy Calculator		
45	Version:	1.1	
46	Copyright and License		

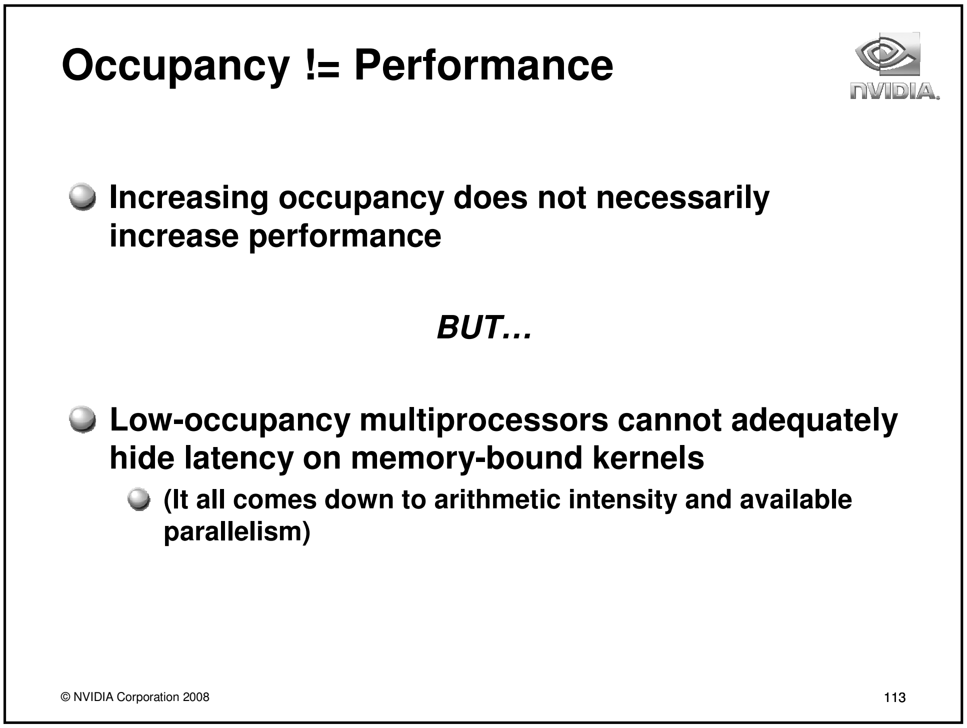
Calculator / Help / GPU Data / Copyright & License

Ready

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Optimizing threads per block

- **Choose threads per block as a multiple of warp size**
  - **Avoid wasting computation on under-populated warps**
- **More threads per block == better memory latency hiding**
- **But, more threads per block == fewer registers per thread**
  - **Kernel invocations can fail if too many registers are used**
- **Heuristics**
  - **Minimum: 64 threads per block**
    - **Only if multiple concurrent blocks**
  - **192 or 256 threads a better choice**
    - **Usually still enough regs to compile and invoke successfully**
  - **This all depends on your computation, so experiment!**

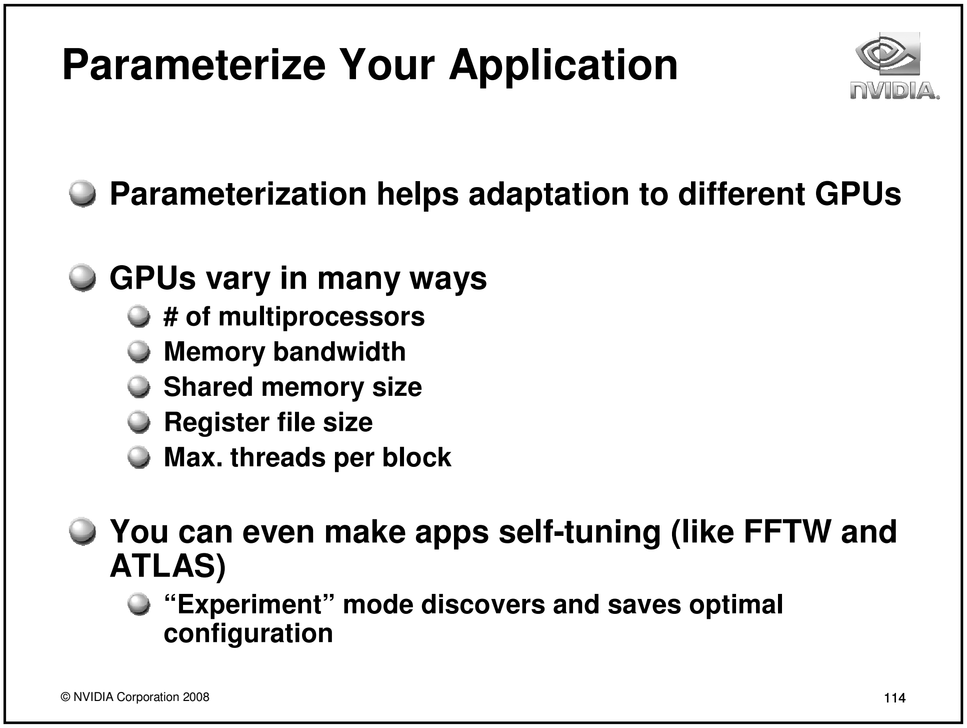


Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

BUT…

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
  - (It all comes down to arithmetic intensity and available parallelism)



Parameterize Your Application

- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
  - # of multiprocessors
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Max. threads per block
- You can even make apps self-tuning (like FFTW and ATLAS)
  - “Experiment” mode discovers and saves optimal configuration

Instruction Optimizations

CUDA Instruction Performance

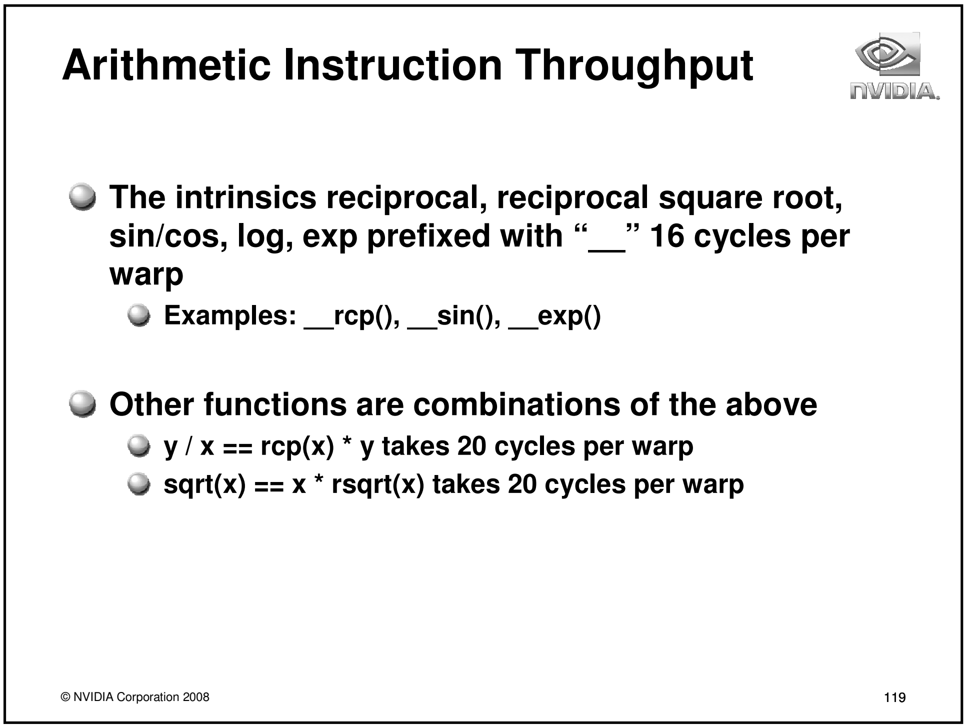
- Instruction cycles (per warp) = sum of
  - Operand read cycles
  - Instruction execution cycles
  - Result update cycles
- Therefore instruction throughput depends on
  - Nominal instruction throughput
  - Memory latency
  - Memory bandwidth
- “Cycle” refers to the multiprocessor clock rate
  - 1.35 GHz on the Tesla C870, for example

Maximizing Instruction Throughput

- Maximize use of high-bandwidth memory
  - Maximize use of shared memory
  - Minimize accesses to global memory
  - Maximize coalescing of global memory accesses
- Optimize performance by overlapping memory accesses with HW computation
  - High arithmetic intensity programs
    - i.e. high ratio of math to memory transactions
  - Many concurrent threads

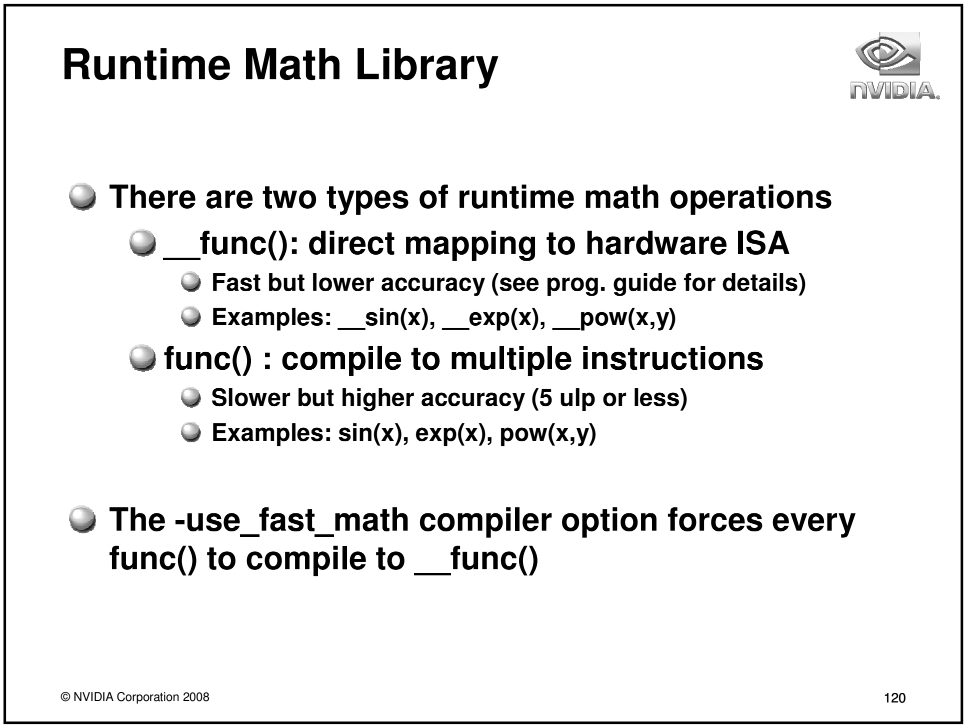
Arithmetic Instruction Throughput

- int and float add, shift, min, max and float mul, mad: 4 cycles per warp
  - int multiply (*) is by default 32-bit
    - requires multiple cycles / warp
  - Use __mul24() / __umul24() intrinsics for 4-cycle 24-bit int multiply
- Integer divide and modulo are more expensive
  - Compiler will convert literal power-of-2 divides to shifts
    - But we have seen it miss some cases
  - Be explicit in cases where compiler can’t tell that divisor is a power of 2!
  - Useful trick: foo % n == foo & (n-1) if n is a power of 2



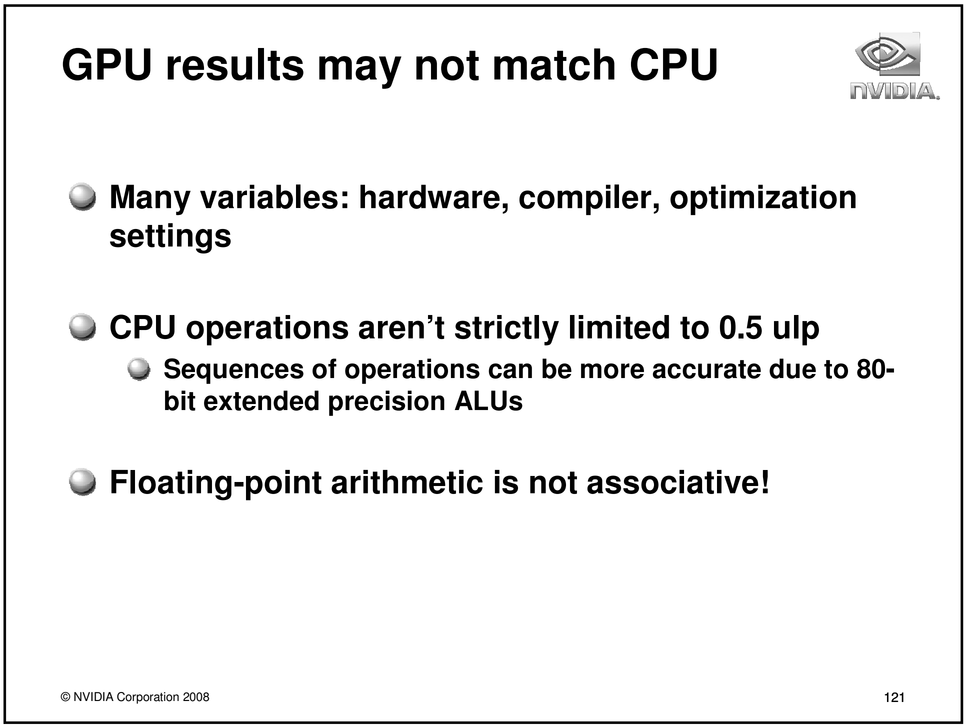
Arithmetic Instruction Throughput

- The intrinsics reciprocal, reciprocal square root, sin/cos, log, exp prefixed with “__” 16 cycles per warp
  - Examples: __rcp(), __sin(), __exp()
- Other functions are combinations of the above
  - y / x == rcp(x) * y takes 20 cycles per warp
  - sqrt(x) == x * rsqrt(x) takes 20 cycles per warp



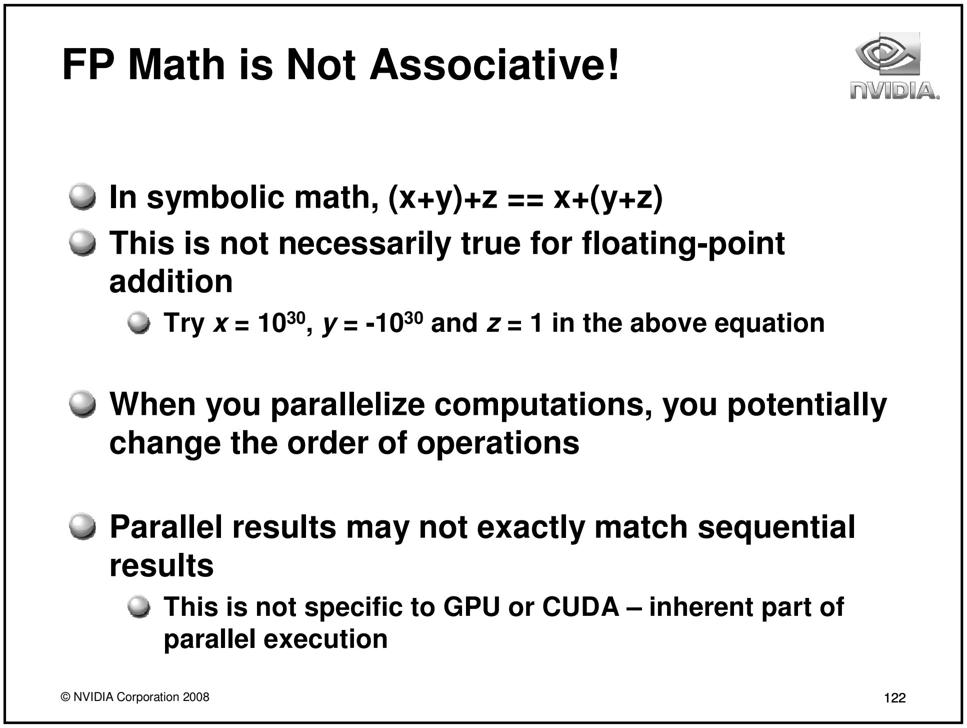
Runtime Math Library

- There are two types of runtime math operations
  - __func(): direct mapping to hardware ISA
    - Fast but lower accuracy (see prog. guide for details)
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - func() : compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: sin(x), exp(x), pow(x,y)
- The -use_fast_math compiler option forces every func() to compile to __func()



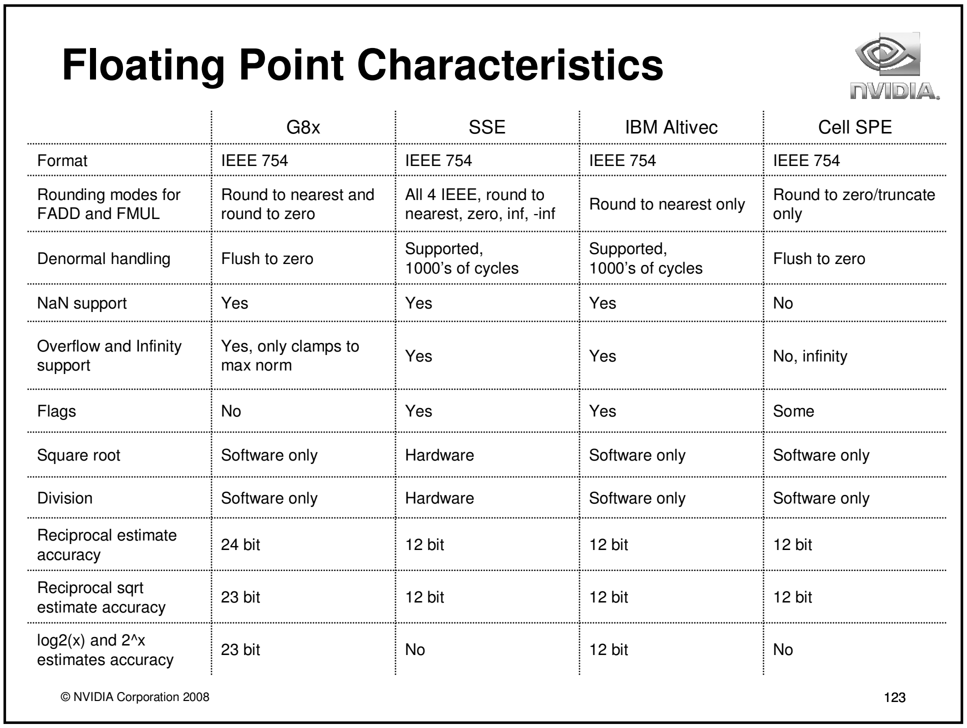
GPU results may not match CPU

- Many variables: hardware, compiler, optimization settings
- CPU operations aren’t strictly limited to 0.5 ulp
  - Sequences of operations can be more accurate due to 80-bit extended precision ALUs
- Floating-point arithmetic is not associative!



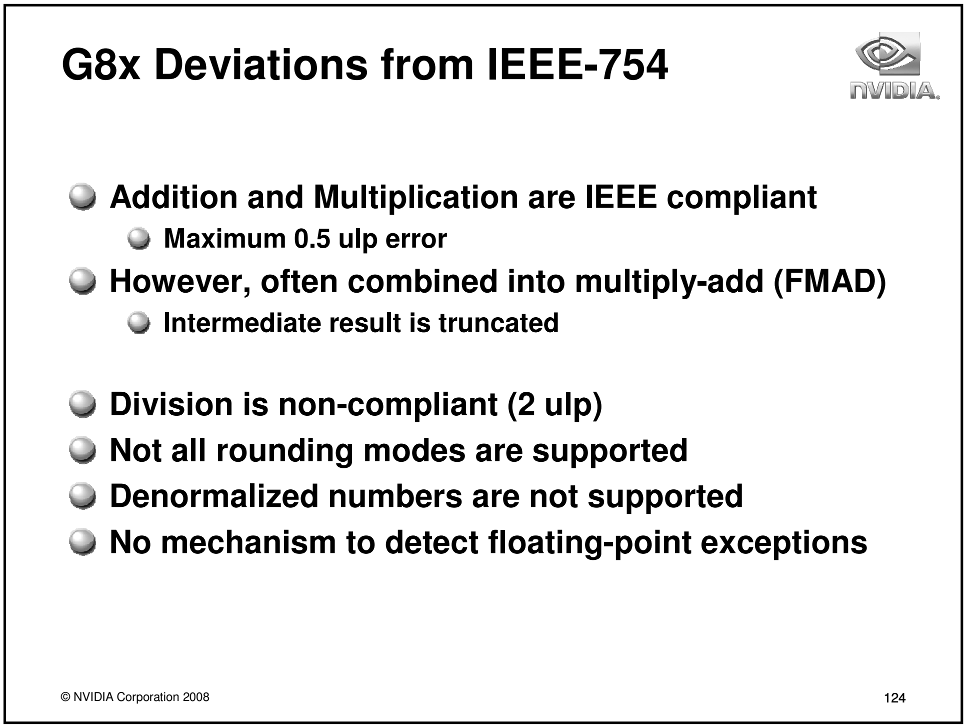
FP Math is Not Associative!

- In symbolic math, (x+y)+z == x+(y+z)
- This is not necessarily true for floating-point addition
  - Try $x = 10^{30}$, $y = -10^{30}$ and $z = 1$ in the above equation
- When you parallelize computations, you potentially change the order of operations
- Parallel results may not exactly match sequential results
  - This is not specific to GPU or CUDA – inherent part of parallel execution



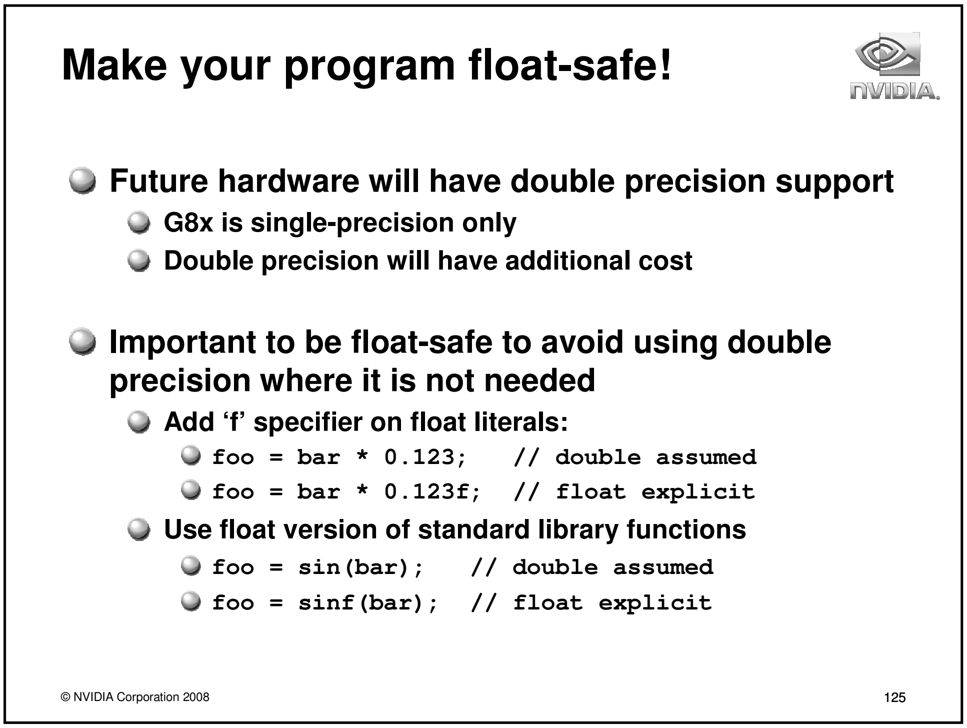
Floating Point Characteristics

	G8x	SSE	IBM Altivec	Cell SPE
Format	IEEE 754	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	Round to nearest and round to zero	All 4 IEEE, round to nearest, zero, inf, -inf	Round to nearest only	Round to zero/truncate only
Denormal handling	Flush to zero	Supported, 1000's of cycles	Supported, 1000's of cycles	Flush to zero
NaN support	Yes	Yes	Yes	No
Overflow and Infinity support	Yes, only clamps to max norm	Yes	Yes	No, infinity
Flags	No	Yes	Yes	Some
Square root	Software only	Hardware	Software only	Software only
Division	Software only	Hardware	Software only	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit	12 bit
log2(x) and 2^x estimates accuracy	23 bit	No	12 bit	No



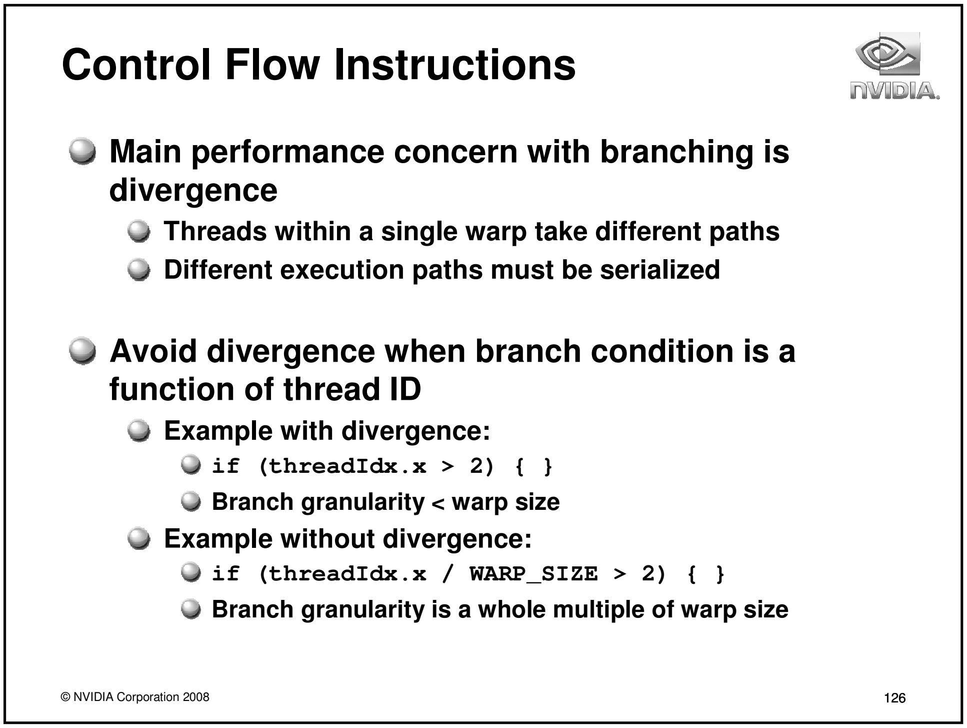
G8x Deviations from IEEE-754

- **Addition and Multiplication are IEEE compliant**
  - **Maximum 0.5 ulp error**
- **However, often combined into multiply-add (FMAD)**
  - **Intermediate result is truncated**
- **Division is non-compliant (2 ulp)**
- **Not all rounding modes are supported**
- **Denormalized numbers are not supported**
- **No mechanism to detect floating-point exceptions**



Make your program float-safe!

- Future hardware will have double precision support
  - G8x is single-precision only
  - Double precision will have additional cost
- Important to be float-safe to avoid using double precision where it is not needed
  - Add ‘f’ specifier on float literals:
    - `foo = bar * 0.123;   // double assumed`
    - `foo = bar * 0.123f;  // float explicit`
  - Use float version of standard library functions
    - `foo = sin(bar);   // double assumed`
    - `foo = sinf(bar);  // float explicit`



Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths must be serialized
- Avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `if (threadIdx.x > 2) { }`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) { }`
    - Branch granularity is a whole multiple of warp size

Summary

- GPU hardware can achieve great performance on data-parallel computations if you follow a few simple guidelines:
  - Use parallelism efficiently
  - Coalesce memory accesses if possible
  - Take advantage of shared memory
  - Explore other memory spaces
    - Texture
    - Constant
  - Reduce bank conflicts

CUDA Libraries

Outline

- CUDA includes 2 widely used libraries
  - CUBLAS: BLAS implementation
  - CUFFT: FFT implementation

CUBLAS

- Implementation of BLAS (Basic Linear Algebra Subprograms) on top of CUDA driver
  - Self-contained at the API level, no direct interaction with CUDA driver
- Basic model for use
  - Create matrix and vector objects in GPU memory space
  - Fill objects with data
  - Call sequence of CUBLAS functions
  - Retrieve data from GPU
- CUBLAS library contains helper functions
  - Creating and destroying objects in GPU space
  - Writing data to and retrieving data from objects

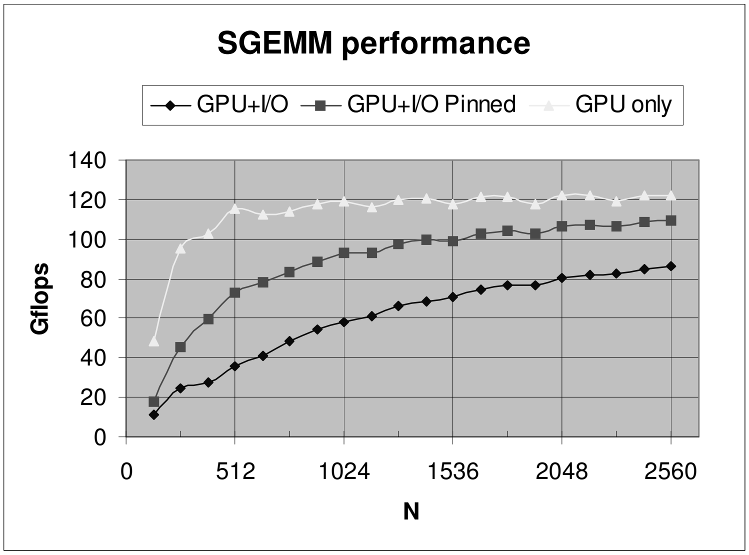
Supported Features

- Single precision BLAS functions
  - Real data
    - Level 1 (vector-vector O(N))
    - Level 2 (matrix-vector O(N^2))
    - Level 3 (matrix-matrix O(N^3))
  - Complex data
    - Level 1
    - CGEMM
- Following BLAS convention, CUBLAS uses column-major storage

Using CUBLAS

- Interface to CUBLAS library is in cublas.h
- Function naming convention
  - cublas + BLAS name
  - Eg., cublasSGEMM
- Error handling
  - CUBLAS core functions do not return error
    - CUBLAS provides function to retrieve last error recorded
  - CUBLAS helper functions do return error
- Implemented using C-based CUDA tool chain
  - Interfacing to C/C++ applications is trivial

CUBLAS performance



cublasInit(), cublasShutdown()

- cublasStatus cublasInit()
  - **Initializes the CUBLAS library**
  - **Allocated hardware resources necessary for accessing the GPU**
  - **Must be called prior to any other CUBLAS API function**
- cublasStatus cublasShutdown()
  - **Releases CPU-side resources used by CUBLAS library**
  - **Release of GPU-side resources may be deferred until application shuts down**

cublasGetError(), cublasAlloc(), cublasFree()

- cublasStatus cublasGetError()
  - **Returns last error that occurred from any of the CUBLAS core functions**
  - **Resets internal error state to** CUBLAS_STATE_SUCCESS
- cublasStatus cublasAlloc(int n, int elemSize,
  void **devPtr)
  - **Creates object in GPU memory for an array of** n **elements**
  - **Each element requires** elemSize **bytes of storage**
  - **Wrapper around** cudaMalloc() **so** devPtr **can be used accordingly**
- cublasStatus cublasFree(const void *devPtr)
  - **Destroys object in GPU space pointer to by** devPtr

cublasSetVector(), cublasGetVector()

- cublasStatus cublasSetVector(int n, int elemSize, const void *x,
  int incx, void *y, int incy)
  - **Copies** n **elements from a vector** x **in CPU memory space to a vector** y **in GPU memory space**
  - **Each element occupies** elemSize **bytes**
  - **Storage spacing between consecutive elements in arrays** x **and** y **is** incx **and** incy**, respectively**
- cublasStatus cublasGetVector(int n, int elemSize, const void *x,
  int incx, void *y, int incy)
  - **Copies** n **elements from a vector** x **in GPU memory space to a vector** y **in CPU memory space**

cublasSetMatrix(), cublasGetMatrix()

- cublasStatus cublasSetMatrix(int rows, int cols, int elemSize,
  const void *A, int lda, void *B, int ldb)
  - **Copies a tile of** rows*cols **elements from a matrix** A **in CPU memory space to a matrix** B **in GPU memory space**
  - **Each element occupies** elemSize **bytes**
  - **Both matrices stored in column-major format, with leading dimensions of** lda **and** ldb **for matrices** A **and** B**, respectively**

- cublasStatus cublasGetMatrix(int rows, int cols, int elemSize,
  const void *A, int lda, void *B, int ldb)
  - **Copies a tile of** rows*cols **elements from a matrix** A **in GPU memory space to a matrix** B **in CPU memory space**

Calling CUBLAS from FORTRAN

- **Fortran-to-C calling conventions are not standardized and differ by platform and toolchain. Differences may include:**
  - **symbol names (capitalization, name decoration)**
  - **argument passing (by value or reference)**
  - **passing of string arguments (length information)**
  - **passing of pointer arguments (size of the pointer)**
  - **returning floating-point or compound data types (for example, single-precision or complex data type)**
- **CUBLAS provides wrapper functions (in the file fortran.c) that need to be compiled with the user preferred toolchain**
  - **Providing source code allows users to make any changes necessary for a particular platform and toolchain.**

Calling CUBLAS from FORTRAN

- **Two interfaces:**
  - **Thunking** (define CUBLAS_USE_THUNKING when compiling fortran.c)
    - Allows interfacing to existing applications without any changes
    - During each call, the wrappers allocate GPU memory, copy source data from CPU memory space to GPU memory space, call CUBLAS, and finally copy back the results to CPU memory space and deallocate the GPGPU memory
    - Intended for light testing due to call overhead
  - **Non-Thunking** (default)
    - Intended for production code
    - Substitute device pointers for vector and matrix arguments in all BLAS functions
    - Existing applications need to be modified slightly to allocate and deallocate data structures in GPGPU memory space (using CUBLAS_ALLOC and CUBLAS_FREE) and to copy data between GPU and CPU memory spaces (using CUBLAS_SET_VECTOR, CUBLAS_GET_VECTOR, CUBLAS_SET_MATRIX, and CUBLAS_GET_MATRIX)

FORTRAN 77 Code example:

```
program matrixmod
implicit none
integer M, N
parameter (M=6, N=5)
real*4 a(M,N)
integer i, j

do j = 1, N
  do i = 1, M
     a(i,j) = (i-1) * M + j
  enddo
enddo

call modify (a, M, N, 2, 3, 16.0, 12.0)

do j = 1, N
  do i = 1, M
     write(*,"(F7.0$)") a(i,j)
  enddo
  write (*,*) ""
enddo

stop
end
```

```
subroutine modify (m, ldm, n, p, q, alpha, beta)
implicit none
integer ldm, n, p, q
real*4 m(ldm,*), alpha, beta

external sscal

call sscal (n-p+1, alpha, m(p,q), ldm)

call sscal (ldm-p+1, beta, m(p,q), 1)

return
end
```

FORTRAN 77 Code example: Non-thunking interface

```
program matrixmod
implicit none
integer M, N, sizeof_real, devPtrA
parameter (M=6, N=5, sizeof_real=4)
real*4 a(M,N)
integer i, j, stat
external cublas_init, cublas_set_matrix,cublas_get_matrix
external cublas_shutdown, cublas_alloc
integer cublas_alloc

do j = 1, N
  do i = 1, M
     a(i,j) = (i-1) * M + j
  enddo
enddo

call cublas_init
stat = cublas_alloc(M*N, sizeof_real, devPtrA)
if (stat .NE. 0) then
   write(*,*) "device memory allocation failed"
   stop
endif

call cublas_set_matrix (M, N, sizeof_real, a, M, devPtrA, M)
call modify (devPtrA, M, N, 2, 3, 16.0, 12.0)
call cublas_get_matrix (M, N, sizeof_real, devPtrA, M, a, M)
call cublas_free(devPtrA)
call cublas_shutdown
```

```
do j = 1, N
  do i = 1, M
     write(*,"(F7.0$)") a(i,j)
  enddo
  write (*,*) ""
enddo

stop
end

#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1)

subroutine modify (devPtrM, ldm, n, p, q, alpha, beta)
implicit none
integer ldm, n, p, q
integer sizeof_real, devPtrM
parameter (sizeof_real=4)
real*4  alpha, beta
call cublas_sscal (n-p+1, alpha,
                   devPtrM+IDX2F(p,q,ldm)*sizeof_real,
                   ldm)
call cublas_sscal (ldm-p+1, beta,
                   devPtrM+IDX2F(p,q,ldm)*sizeof_real,
                   1)
return
end
```

If using fixed format check that the line length is below the 72 column limit !!!

CUFFT

- **The Fast Fourier Transform (FFT) is a divide-and-conquer algorithm for efficiently computing discrete Fourier transform of complex or real-valued data sets.**
- **CUFFT is the CUDA FFT library**
  - **Provides a simple interface for computing parallel FFT on an NVIDIA GPU**
  - **Allows users to leverage the floating-point power and parallelism of the GPU without having to develop a custom, GPU-based FFT implementation**

Supported Features

- 1D, 2D and 3D transforms of complex and real-valued data
- Batched execution for doing multiple 1D transforms in parallel
- 1D transform size up to 8M elements
- 2D and 3D transform sizes in the range [2,16384]
- In-place and out-of-place transforms for real and complex data.

CUFFT Types and Definitions

- cufftHandle
  - Handle type used to store and access CUFFT plans
- cufftResults
  - Enumeration of API function return values
  - Eg. CUFFT_SUCCESS, CUFFT_INVALID_PLAN, etc.

Transform Types

- **Library supports real and complex transforms**
  - CUFFT_C2C, CUFFT_C2R, CUFFT_R2C
- **Directions**
  - CUFFT_FORWARD (-1) and CUFFT_BACKWARD (1)
    - **According to sign of the complex exponential term**
- **Real and imaginary parts of complex input and output arrays are interleaved**
  - cufftComplex **type is defined for this**
- **Real to complex FFTs, output array holds only nonredundant coefficients**
  - **N -> N/2+1**
  - **N0 x N1 x … x Nn -> N0 x N1 x … x (Nn/2+1)**
  - **For in-place transforms the input/output arrays need to be padded**

More on Transforms

- **For 2D and 3D transforms, CUFFT performs transforms in row-major (C-order)**
- **If calling from FORTRAN or MATLAB, remember to change the order of size parameters during plan creation**
- **CUFFT performs un-normalized transforms:**

  IFFT(FFT(A))= length(A)*A
- **CUFFT API is modeled after FFTW. Based on plans, that completely specify the optimal configuration to execute a particular size of FFT**
- **Once a plan is created, the library stores whatever state is needed to execute the plan multiple times without recomputing the configuration**
  - **Works very well for CUFFT, because different kinds of FFTs require different thread configurations and GPU resources**

cufftPlan1d()

cufftResult cufftPlan1d(cufftHandle *plan, int nx, cufftType type, int batch)

- **Creates a 1D FFT** plan **configuration for a specified signal size and data type**
- **The** batch **input parameter tells CUFFT how many 1D transforms to configure**

- **Input:**

plan	**Pointer to a cufftHandle object**
nx	**The transform size (e.g., 256 for a 256-point FFT)**
type	**The transform data type (e.g., CUFFT_C2C)**
batch	**Number of transforms of size nx**

- **Output:**

plan	**Contains a CUFFT 1D plan handle value**

cufftPlan2d()

cufftResult cufftPlan2d(cufftHandle *plan, int nx, int ny, cufftType type)

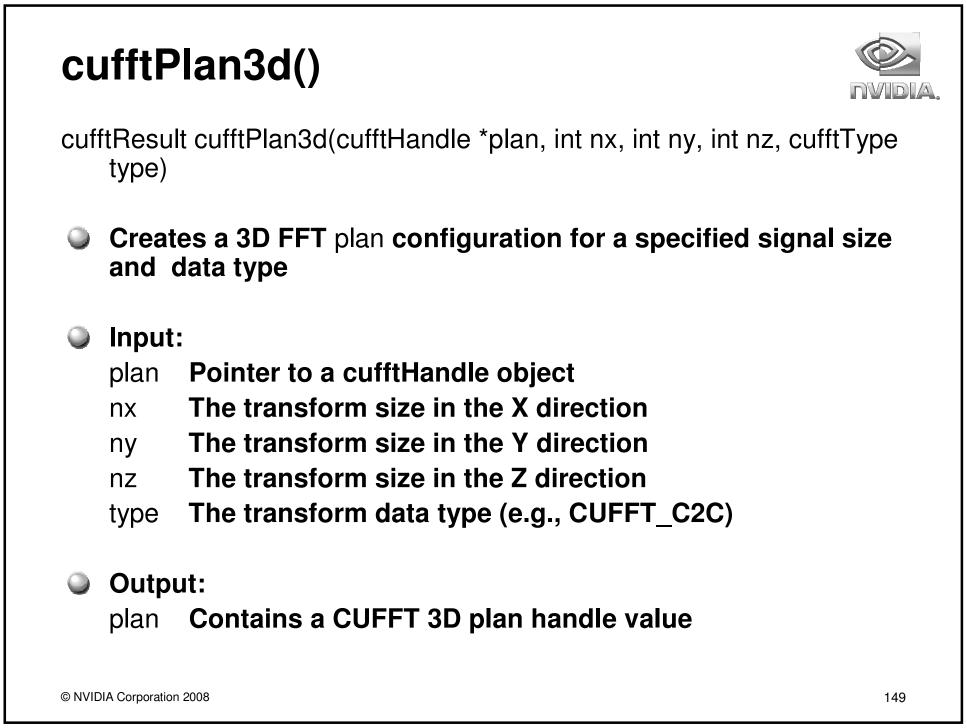
- **Creates a 2D FFT** plan **configuration for a specified signal size and data type**

- **Input:**

plan	**Pointer to a cufftHandle object**
nx	**The transform size in the X direction**
ny	**The transform size in the Y direction**
type	**The transform data type (e.g., CUFFT_C2C)**

- **Output:**

plan	**Contains a CUFFT 2D plan handle value**



cufftPlan3d()

cufftResult cufftPlan3d(cufftHandle *plan, int nx, int ny, int nz, cufftType type)

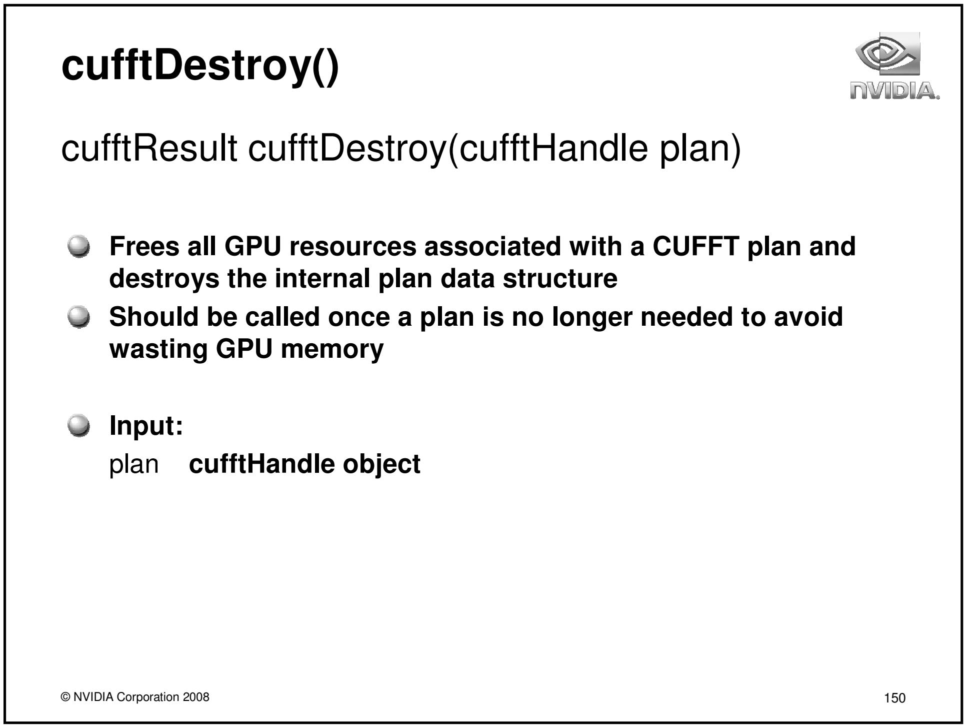
- **Creates a 3D FFT** plan **configuration for a specified signal size and data type**

- **Input:**

plan	**Pointer to a cufftHandle object**
nx	**The transform size in the X direction**
ny	**The transform size in the Y direction**
nz	**The transform size in the Z direction**
type	**The transform data type (e.g., CUFFT_C2C)**

- **Output:**

plan	**Contains a CUFFT 3D plan handle value**



cufftDestroy()

cufftResult cufftDestroy(cufftHandle plan)

- **Frees all GPU resources associated with a CUFFT plan and destroys the internal plan data structure**
- **Should be called once a plan is no longer needed to avoid wasting GPU memory**

- **Input:**

plan	**cufftHandle object**

cufftExecC2C()

```
cufftResult cufftExecC2C(cufftHandle plan, cufftComplex *idata,
                         cufftComplex *odata, int direction)
```

- **Executes a CUFFT complex to complex transform plan**
- **Uses as input data the GPU memory pointed to by the** idata **parameter**
- **Stores the Fourier coefficients in the** odata **array**
  - **If** idata **and** odata **are the same, does an in-place transform**

- **Input:**

plan	**cufftHandle object**
idata	**pointer to input data (in GPU memory) to transform**
odata	**pointer to output data (in GPU memory)**
direction	**direction of transform (**CUFFT_FORWARD **or** CUFFT_BACKWARD**)**

- **Output:**

odata	**contains complex Fourier coefficients**

cufftExecR2C()

```
cufftResult cufftExecR2C(cufftHandle plan, cufftReal *idata,
                         cufftComplex *odata)
```

- **Executes a CUFFT real to complex transform plan**
- **Uses as input data the GPU memory pointed to by the** idata **parameter**
- **Stores non-redundant Fourier coefficients in the** odata **array**
  - **If** idata **and** odata **are the same, does an in-place transform**

- **Input:**

plan	**cufftHandle object**
idata	**pointer to input data (in GPU memory) to transform**
odata	**pointer to output data (in GPU memory)**

- **Output:**

odata	**contains complex Fourier coefficients**

cufftExecC2R()

cufftResult cufftExecC2R(cufftHandle plan, cufftReal *idata,
cufftComplex *odata)

- **Executes a CUFFT complex to real transform plan**
- **Uses as input the GPU memory pointed to by** idata
  - idata **contains only non-redundant complex Fourier coefficients**
- **Stores real output data in the** odata **array**
  - **If** idata **and** odata **are the same, does an in-place transform**

- **Input:**

plan	**cufftHandle object**
idata	**pointer to complex input data (in GPU memory) to transform**
odata	**pointer to real output data (in GPU memory)**

- **Output:**

odata	**contains real-valued output data**

Accuracy and performance

The CUFFT library implements several FFT algorithms, each with different performances and accuracy.

The best performance paths correspond to transform sizes that:

1. **Fit in CUDA’a shared memory**
2. **Are powers of a single factor (e.g. power-of-two)**

If only condition 1 is satisfied, CUFFT uses a more general mixed-radix factor algorithm that is slower and less accurate numerically.

If none of the above conditions is satisfied, CUFFT uses an out-of-place, mixed-radix algorithm that stores all intermediate results in global GPU memory.

One notable exception is for long 1D transforms, where CUFFT uses a distributed algorithm that perform 1D FFT using 2D FFT.

CUFFT does not implement any specialized algorithms for real data, and so there is no direct performance benefit to using real to complex (or complex to real) plans instead of complex to complex. For this release, the real data API exists primarily for convenience

Code example:
1D complex to complex transforms

```
#define NX 256
#define BATCH 10

cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);
…
/* Create a 1D FFT plan. */
 cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);

 /* Use the CUFFT plan to transform the signal in place. */
 cufftExecC2C(plan, data, data, CUFFT_FORWARD);

/* Inverse transform the signal in place. */
 cufftExecC2C(plan, data, data, CUFFT_INVERSE);

/* Note:
   (1) Divide by number of elements in data-set to get back original data
   (2) Identical pointers to input and output arrays implies in-place transformation
 */

 /* Destroy the CUFFT plan. */
   cufftDestroy(plan);

   cudaFree(data);
```

Code example:
2D complex to complex transform

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);
…
/* Create a 1D FFT plan. */
 cufftPlan2d(&plan, NX,NY, CUFFT_C2C);

 /* Use the CUFFT plan to transform the signal out of place. */
 cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* Inverse transform the signal in place. */
 cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

/* Note:
    Different pointers to input and output arrays implies out of place transformation
 */

 /* Destroy the CUFFT plan. */
   cufftDestroy(plan);

   cudaFree(idata), cudaFree(odata);
```

Additional CUDA Topics

Outline

- Texture Functionality
- Fortran Interoperability
- Event API
- Device Management
- Graphics Interoperability

CUDA Texture Functionality

Textures in CUDA

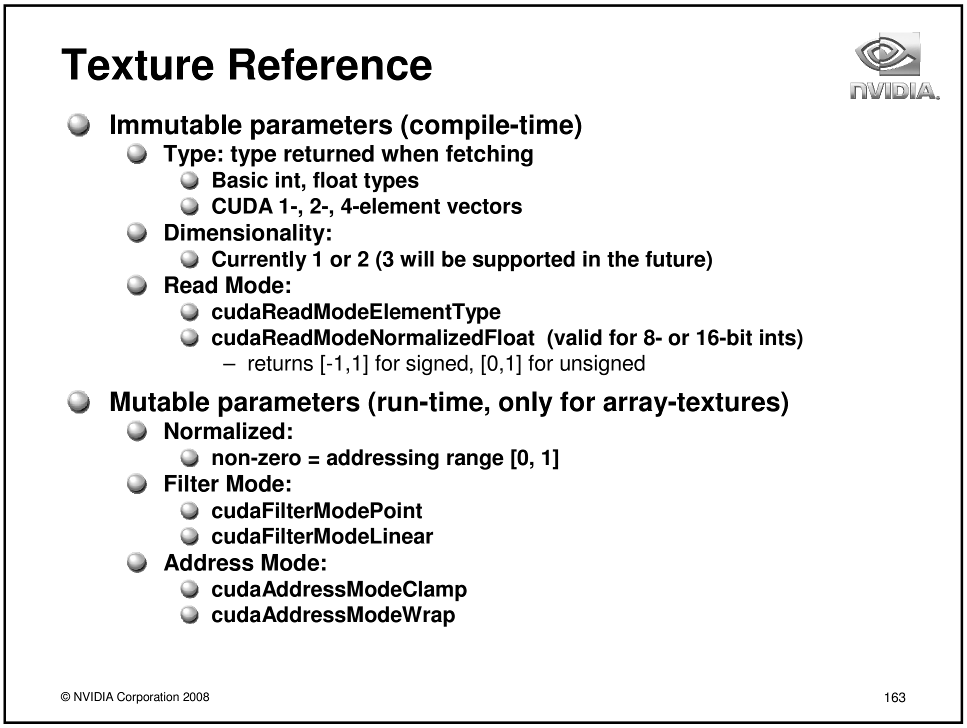
- Different hardware path to memory
- Benefits of CUDA textures:
  - Texture fetches are cached
    - Optimized for 2D locality
  - Textures are addressable in 2D
    - Using integer or normalized coordinates
    - Means fewer addressing calculations in code
  - Provide filtering for free
  - Free wrap modes (boundary conditions)
    - Clamp to edge / repeat
- Limitations of CUDA textures:
  - Read-only
  - Currently either 1D or 2D (3D will be added)
  - 9-bit accuracy of filter weights

Two CUDA Texture Types

- Bound to linear memory
  - Global memory is bound to a texture
  - Only 1D
  - Integer addressing
  - No filtering, no addressing modes
- Bound to CUDA arrays
  - CUDA array is bound to a texture
  - 1D or 2D
  - Float addressing (size-based or normalized)
  - Filtering
  - Addressing modes (clamping, repeat)
- Both:
  - Return either element type or normalized float

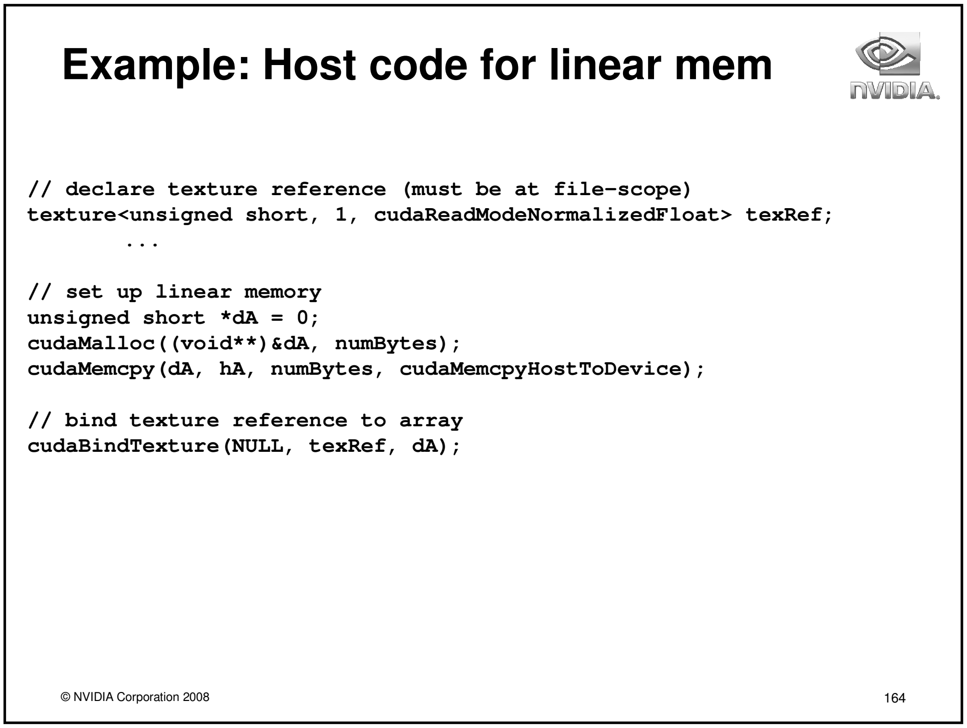
CUDA Texturing Steps

- Host (CPU) code:
  - Allocate/obtain memory (global linear, or CUDA array)
  - Create a texture reference object
    - Currently must be at file-scope
  - Bind the texture reference to memory/array
  - When done:
    - Unbind the texture reference, free resources
- Device (kernel) code:
  - Fetch using texture reference
  - Linear memory textures:
    - tex1Dfetch()
  - Array textures:
    - tex1D() or tex2D()



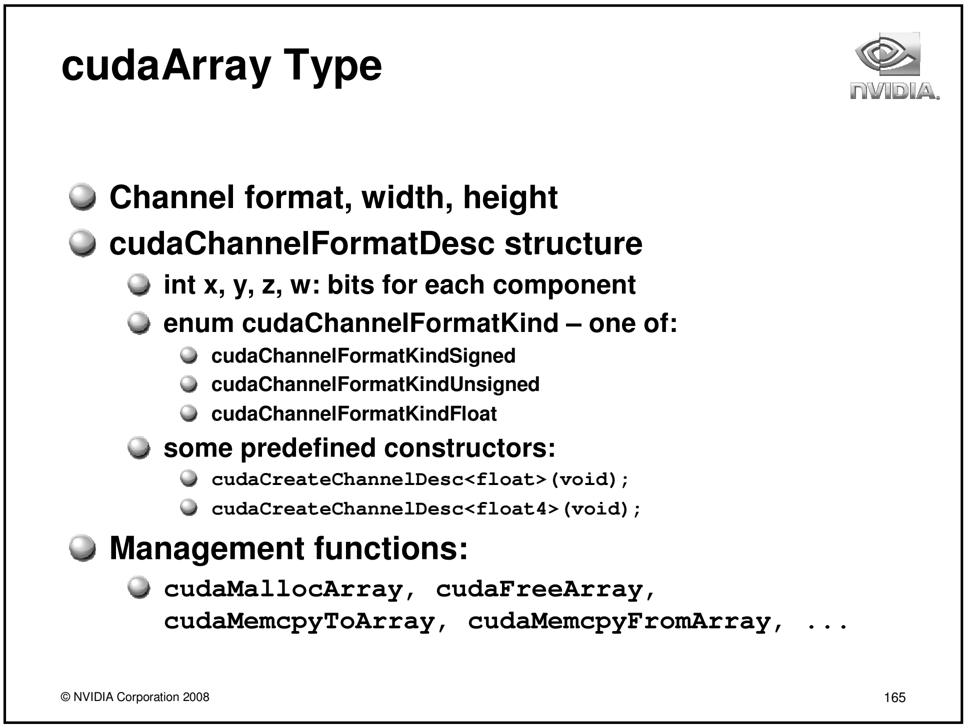
Texture Reference

- Immutable parameters (compile-time)
  - Type: type returned when fetching
    - Basic int, float types
    - CUDA 1-, 2-, 4-element vectors
  - Dimensionality:
    - Currently 1 or 2 (3 will be supported in the future)
  - Read Mode:
    - cudaReadModeElementType
    - cudaReadModeNormalizedFloat (valid for 8- or 16-bit ints)
      - – returns [-1,1] for signed, [0,1] for unsigned
- Mutable parameters (run-time, only for array-textures)
  - Normalized:
    - non-zero = addressing range [0, 1]
  - Filter Mode:
    - cudaFilterModePoint
    - cudaFilterModeLinear
  - Address Mode:
    - cudaAddressModeClamp
    - cudaAddressModeWrap



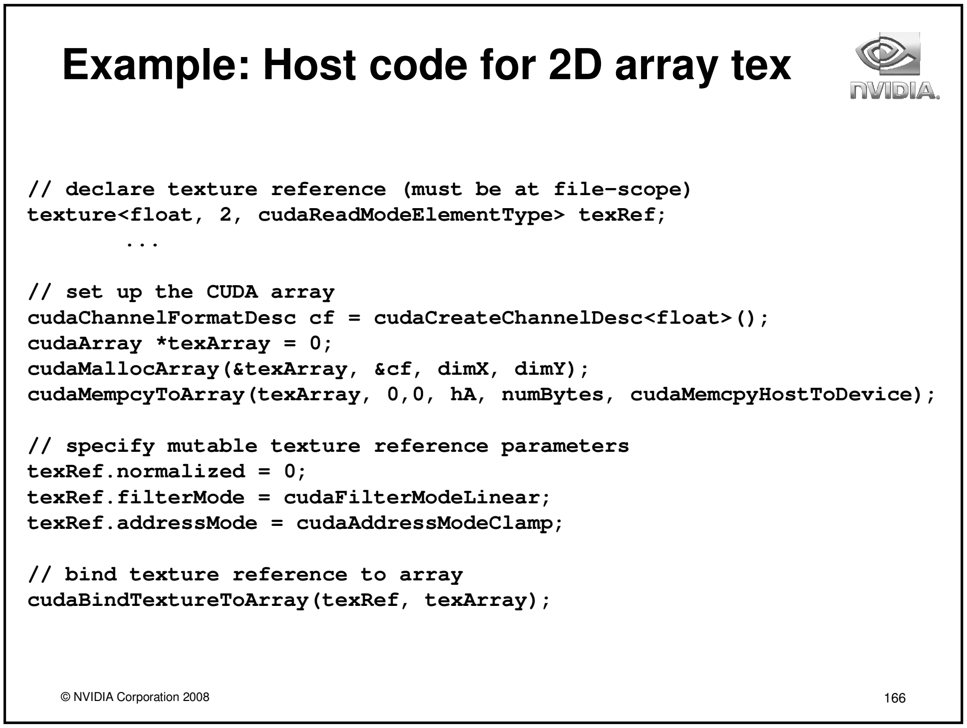
Example: Host code for linear mem

```
// declare texture reference (must be at file-scope)
texture<unsigned short, 1, cudaReadModeNormalizedFloat> texRef;
        ...

// set up linear memory
unsigned short *dA = 0;
cudaMalloc((void**)&dA, numBytes);
cudaMemcpy(dA, hA, numBytes, cudaMemcpyHostToDevice);

// bind texture reference to array
cudaBindTexture(NULL, texRef, dA);
```



cudaArray Type

- Channel format, width, height
- cudaChannelFormatDesc structure
  - int x, y, z, w: bits for each component
  - enum cudaChannelFormatKind – one of:
    - cudaChannelFormatKindSigned
    - cudaChannelFormatKindUnsigned
    - cudaChannelFormatKindFloat
  - some predefined constructors:
    - `cudaCreateChannelDesc<float>(void);`
    - `cudaCreateChannelDesc<float4>(void);`
- Management functions:
  - `cudaMallocArray, cudaFreeArray, cudaMemcpyToArray, cudaMemcpyFromArray, ...`



Example: Host code for 2D array tex

```
// declare texture reference (must be at file-scope)
texture<float, 2, cudaReadModeElementType> texRef;
        ...

// set up the CUDA array
cudaChannelFormatDesc cf = cudaCreateChannelDesc<float>();
cudaArray *texArray = 0;
cudaMallocArray(&texArray, &cf, dimX, dimY);
cudaMempcyToArray(texArray, 0,0, hA, numBytes, cudaMemcpyHostToDevice);

// specify mutable texture reference parameters
texRef.normalized = 0;
texRef.filterMode = cudaFilterModeLinear;
texRef.addressMode = cudaAddressModeClamp;

// bind texture reference to array
cudaBindTextureToArray(texRef, texArray);
```

CUDA Texturing Details

- Linear (bilinear) filtering:
  - Only for textures bound to CUDA arrays
  - Only for textures that return floats
  - Still possible to filter 8- or 16-bit integers:
    - cudaReadModeNormalizedFloat texture reference
    - scale value in the kernel after fetching
- Both run-time and driver API
  - driver API allows half float (16bit) storage
    - fetched values are 32bit
    - will be supported by future run-time API
- It is possible to copy between linear memory and CUDA arrays

CUDA Fortran Interoperability

Fortran examples

- Calling CUBLAS from Fortran
- Using pinned memory in Fortran
- Calling CUDA kernel from Fortran

SGEMM example

```
! Define  3  single precision matrices A, B, C
real , dimension(m1,m1)::      A, B, C
……
! Initialize
……
#ifdef CUBLAS
 ! Call SGEMM in CUBLAS library using THUNKING interface (library takes care of
 ! memory allocation on device and data movement)
  call cublas_SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#else
! Call SGEMM in  host BLAS library
  call SGEMM ('n','n',m1,m1,m1,alpha,A,m1,B,m1,beta,C,m1)
#endif
```

To use the host BLAS routine:

```
g95 –O3 code.f90 –L/usr/local/lib -lblas
```

To use the CUBLAS routine (fortran.c is provided by NVIDIA):

```
gcc -O3 -DCUBLAS_USE_THUNKING -I/usr/local/cuda/include  -c  fortran.c
g95 -O3 -DCUBLAS code.f90 fortran.o -L/usr/local/cuda/lib  -lcublas
```

Pinned memory example

Pinned memory provides a fast PCI-e transfer speed and enables use of streams:
- Allocation needs to be done with cudaMallocHost
- Use new Fortran 2003 features for interoperability with C.

```
use iso_c_binding
! The allocation is performed by C function calls. Define the C pointer as type (C_PTR)
 type(C_PTR) :: cptr_A, cptr_B, cptr_C
! Define Fortran arrays as pointer.
real, dimension(:,:), pointer ::      A, B, C

! Allocating memory with cudaMallocHost.
! The Fortan arrays, now defined as pointers, are then associated with the C pointers using the
! new interoperability defined in iso_c_binding. This is equivalent to allocate(A(m1,m1))
 res = cudaMallocHost ( cptr_A, m1*m1*sizeof(fp_kind) )
 call c_f_pointer ( cptr_A, A, (/ m1, m1 /) )

! Use A as usual.
! See example code for cudaMallocHost interface code
```

Calling CUDA kernels

From Fortran call C function that will call CUDA kernel

```
! Fortran -> C -> CUDA ->C ->Fortran
call cudafunction(c,c2,N)
```

```
/* NB: Fortran subroutine arguments are passed by reference.    */
extern "C" void cudafunction_(cuComplex *a, cuComplex *b,  int *Np)
{
  ...
  int N=*np;
  cudaMalloc ((void **) &a_d , sizeof(cuComplex)*N);
  cudaMemcpy( a_d, a,  sizeof(cuComplex)*N   ,cudaMemcpyHostToDevice);
  dim3 dimBlock(block_size); dim3 dimGrid (N/dimBlock.x); if( N % block_size != 0 ) dimGrid.x+=1;
  square_complex<<<dimGrid,dimBlock>>>(a_d,a_d,N);
   cudaMemcpy( b, a_d, sizeof(cuComplex)*N,cudaMemcpyDeviceToHost);
  cudaFree(a_d);
}
```

```
complex_mul: main.f90 Cuda_function.o
       $(FC) -o complex_mul main.f90 Cuda_function.o -L/usr/local/cuda/lib  -lcudart

Cuda_function.o: Cuda_function.cu
       nvcc -c -O3 Cuda_function.cu
```

CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed
  - asyncAPI sample in CUDA SDK

```
cudaEvent_t start, stop;
cudaEventCreate(&start);           cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start);           cudaEventDestroy(stop);
```

Device Management

- CPU can query and select GPU devices
  - cudaGetDeviceCount(int* count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *current_device)
  - cudaGetDeviceProperties(cudaDeviceProp* prop,
                                     int device)
  - cudaChooseDevice(int *device, cudaDeviceProp* prop)
- Multi-GPU setup:
  - device 0 is used by default
  - one CPU thread can control only one GPU
    - multiple CPU threads can control the same GPU
      - – calls are serialized by the driver

Multiple CPU Threads and CUDA

- CUDA resources allocated by a CPU thread can be consumed only by CUDA calls from the same CPU thread
- Violation Example:
  - CPU thread 2 allocates GPU memory, stores address in *p*
  - thread 3 issues a CUDA call that accesses memory via *p*

CUDA Graphics Interoperability

OpenGL Interoperability

- **OpenGL buffer objects can be mapped into the CUDA address space and then used as global memory**
  - **Vertex buffer objects**
  - **Pixel buffer objects**
- **Direct3D9 Vertex objects can be mapped**
- **Data can be accessed like any other global data in the device code**
- **Image data can be displayed from pixel buffer objects using glDrawPixels / glTexImage2D**
  - **Requires copy in video memory, but still fast**

OpenGL Interop Steps

- **Register a buffer object with CUDA**
  - `cudaGLRegisterBufferObject(GLuint buffObj);`
  - **OpenGL can use a registered buffer only as a source**
  - **Unregister the buffer prior to rendering to it by OpenGL**
- **Map the buffer object to CUDA memory**
  - `cudaGLMapBufferObject(void **devPtr, GLuint buffObj);`
  - **Returns an address in global memory**
  - **Buffer must registered prior to mapping**
- **Launch a CUDA kernel to process the buffer**
- **Unmap the buffer object prior to use by OpenGL**
  - `cudaGLUnmapBufferObject(GLuint buffObj);`
- **Unregister the buffer object**
  - `cudaGLUnregisterBufferObject(GLuint buffObj);`
  - **Optional: needed if the buffer is a render target**
- **Use the buffer object in OpenGL code**

Interop Scenario: Dynamic CUDA-generated texture

- Register the texture PBO with CUDA
- For each frame:
  - Map the buffer
  - Generate the texture in a CUDA kernel
  - Unmap the buffer
  - Update the texture
  - Render the textured object

```
unsigned char *p_d=0;
cudaGLMapBufferObject((void**)&p_d, pbo);
prepTexture<<<height,width>>>(p_d, time);
cudaGLUnmapBufferObject(pbo);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, pbo);
glBindTexture(GL_TEXTURE_2D, texID);
glTexSubImage2D(GL_TEXTURE_2D, 0, 0,0, 256,256,
                GL_BGRA, GL_UNSIGNED_BYTE, 0);
```

Interop Scenario: Frame Post-processing by CUDA

- For each frame:
  - Render to PBO with OpenGL
  - Register the PBO with CUDA
  - Map the buffer
  - Process the buffer with a CUDA kernel
  - Unmap the buffer
  - Unregister the PBO from CUDA

```
unsigned char *p_d=0;
cudaGLRegisterBufferObject(pbo);
cudaGLMapBufferObject((void**)&p_d, pbo);
postProcess<<<blocks,threads>>>(p_d);
cudaGLUnmapBufferObject(pbo);
cudaGLUnregisterBufferObject(pbo);
...
```

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, CUDA and Tesla are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com