NVISION 08

THE WORLD OF VISUAL COMPUTING

# Getting Started with CUDA

Greg Ruetsch, Brent Oster



NVIDIA.

# What is CUDA?

- *CUDA* is a scalable parallel programming model and a software environment for parallel computing
  - Minimal extensions to familiar C/C++ environment
  - Heterogeneous serial-parallel programming model
- NVIDIA's *TESLA* architecture accelerates CUDA
  - Expose the computational horsepower of NVIDIA GPUs
  - Enable *GPU computing*
- CUDA also maps well to multicore CPUs

# Outline

- CUDA programming model
- Basics of CUDA programming
  - Software stack
  - Data management
  - Executing code on the GPU
- CUDA libraries
  - BLAS
  - FFT

# Some Design Goals

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

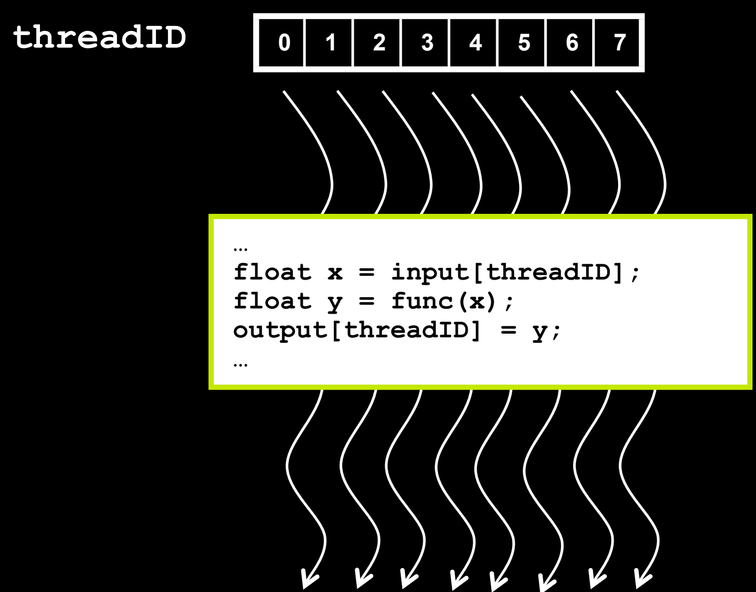
# CUDA Kernels and Threads

- 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 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

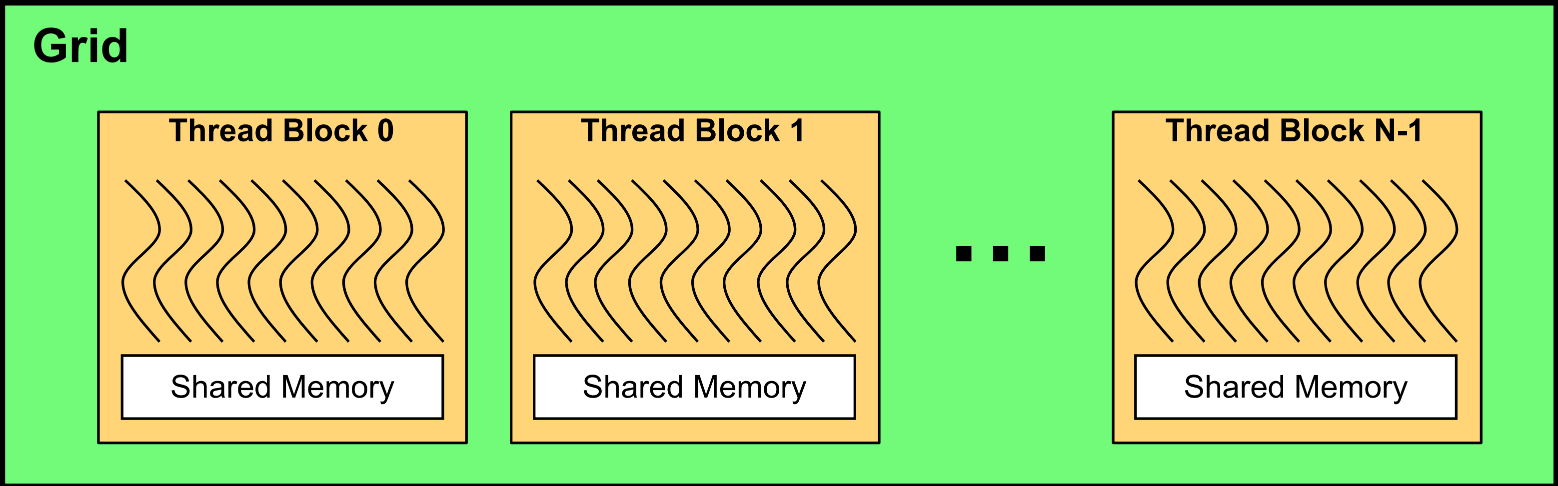


# Thread Cooperation

- 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

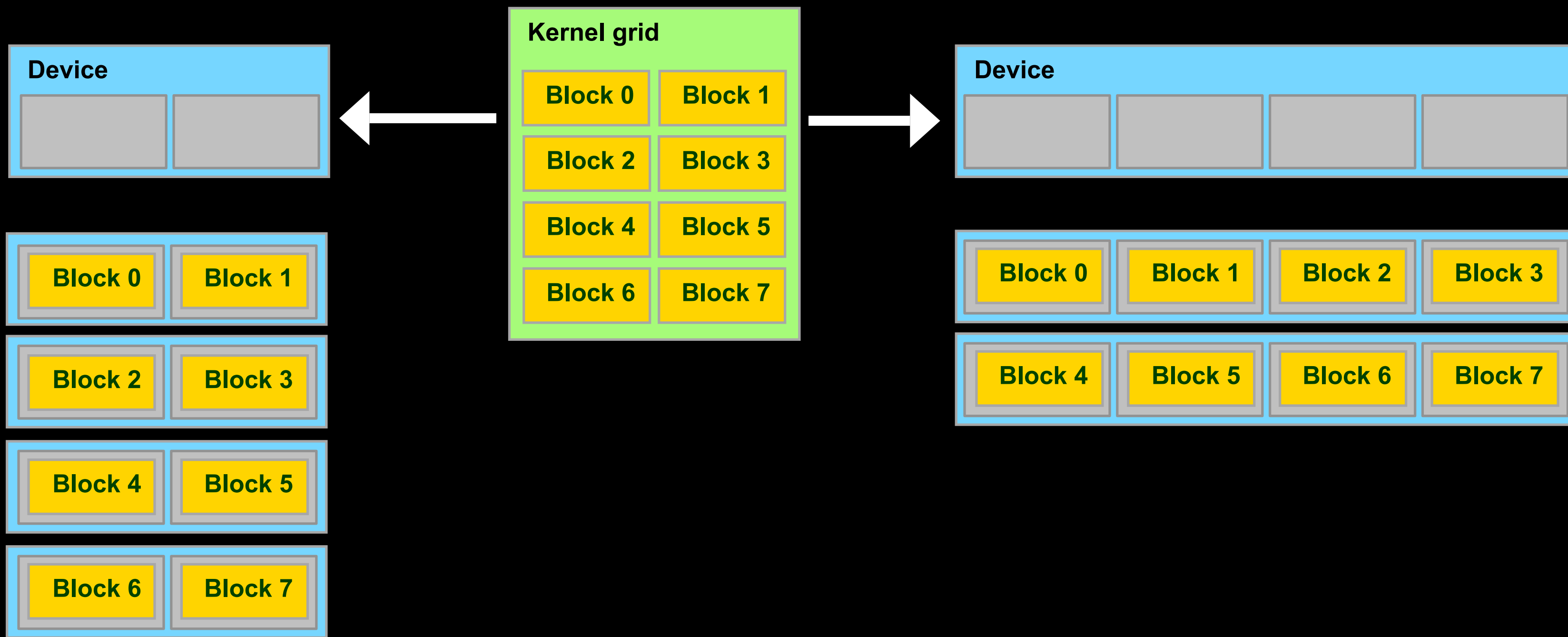
# Thread Batching

- Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate
- Allows programs to *transparently scale* to different GPUs



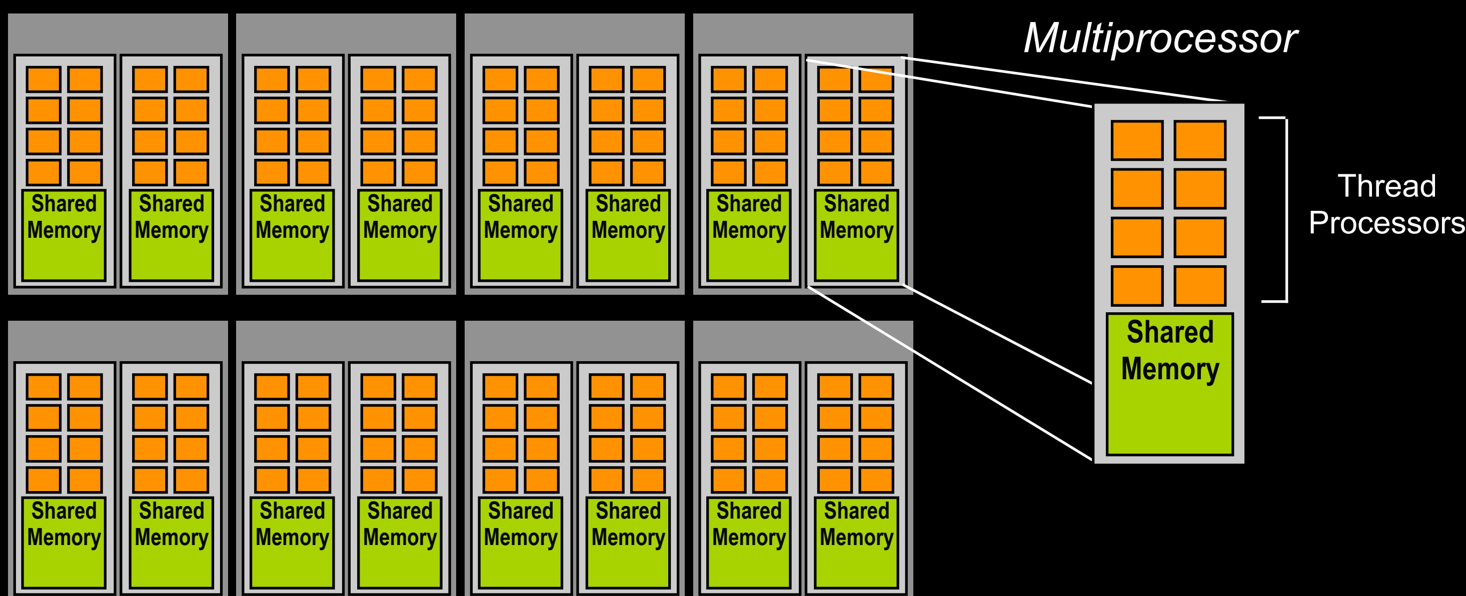
# Transparent Scalability

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



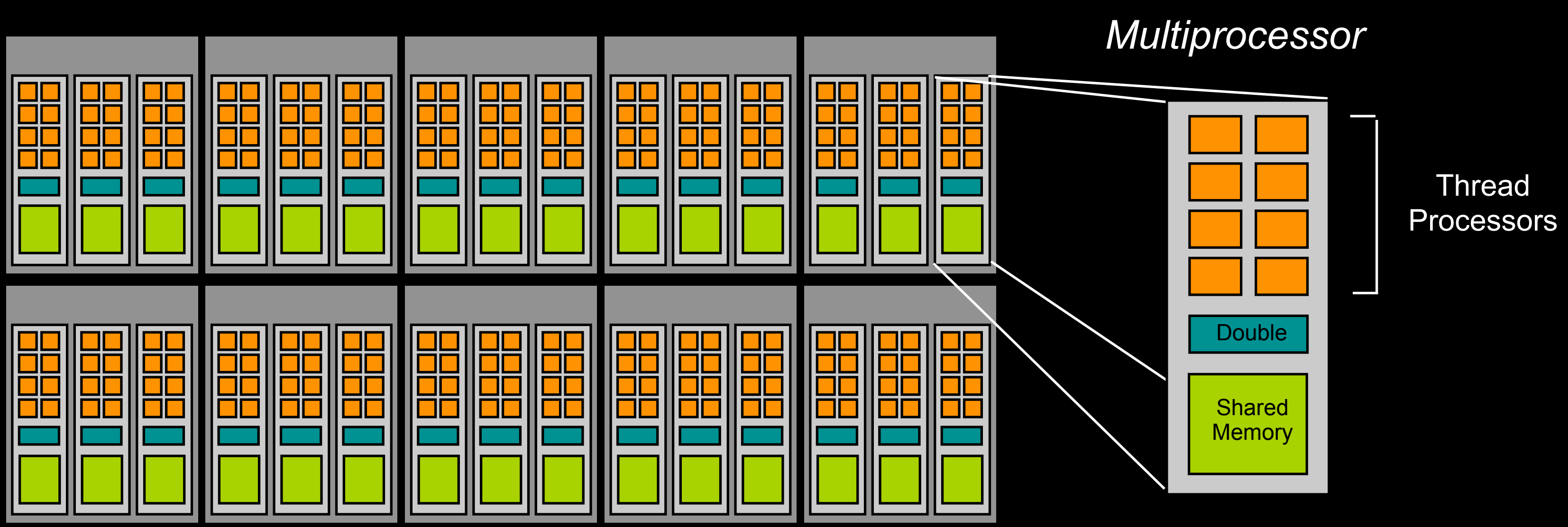
# 8-Series Architecture (G80)

- 128 *thread processors* execute kernel threads
- 16 *multiprocessors*, each contains
  - 8 *thread processors*
  - *Shared memory* enables thread cooperation



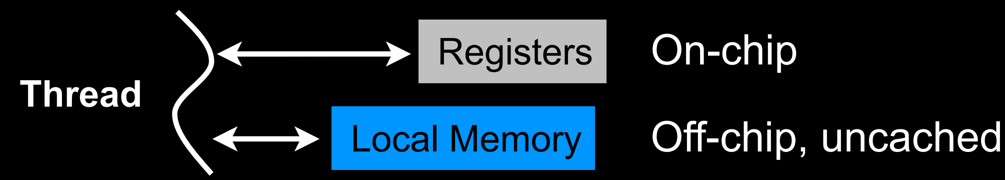
# 10-Series Architecture

- 240 *thread processors* execute kernel threads
- 30 *multiprocessors*, each contains
  - 8 *thread processors*
  - One *double-precision* unit
  - *Shared memory* enables thread cooperation



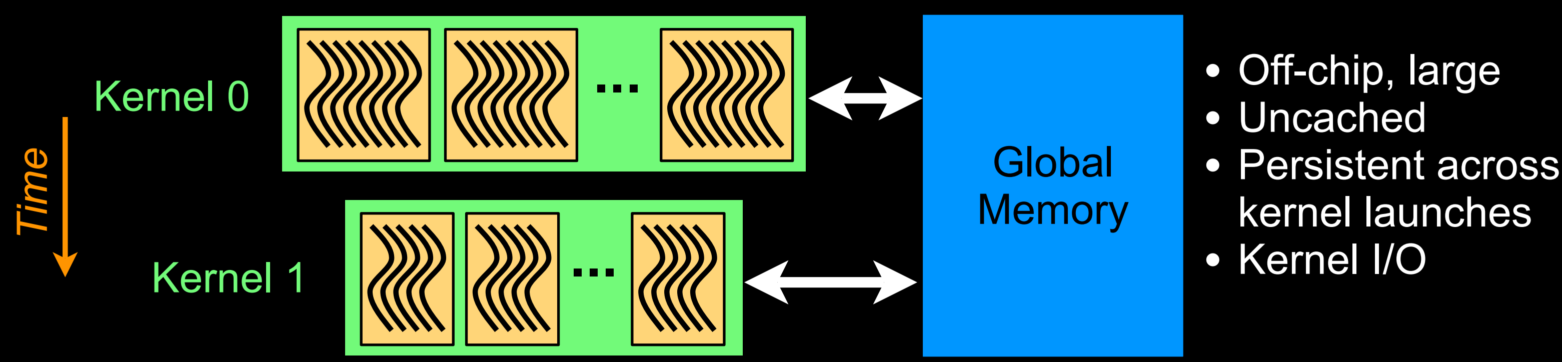
# Kernel Memory Access

- **Per-thread**



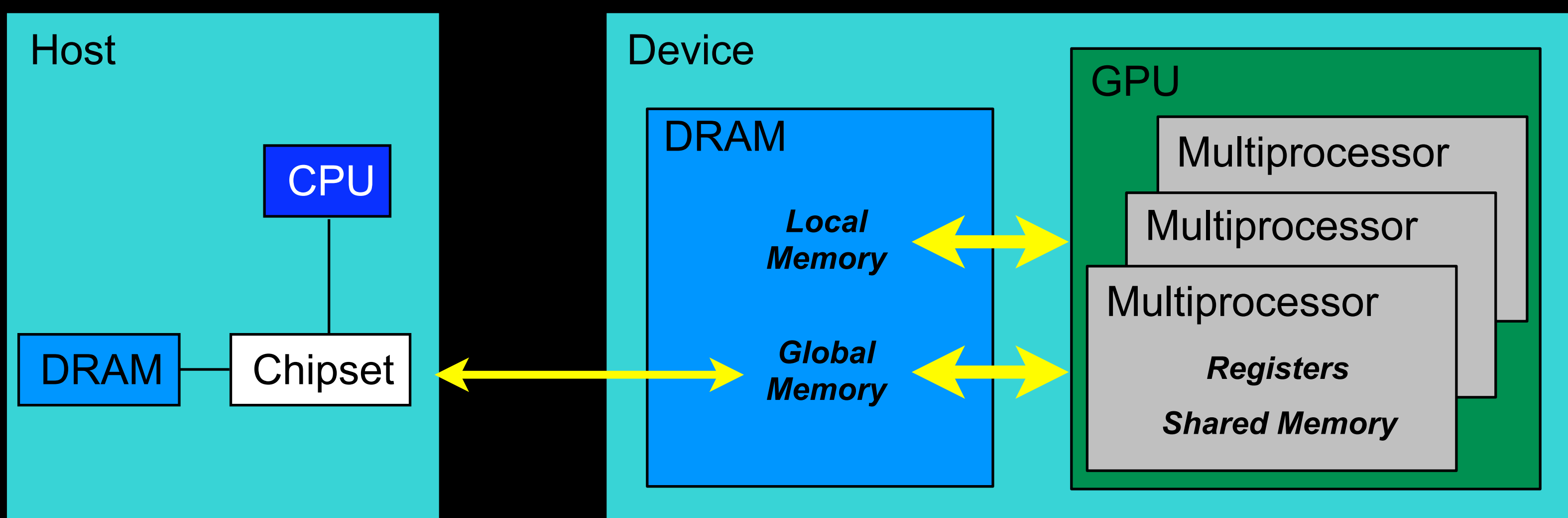

- **Per-block**


- **Per-device**



# Physical Memory Layout

- "Local" memory resides in device DRAM
  - Use registers and shared memory to minimize local memory use
- Host can read and write global memory but not shared memory



# Execution Model

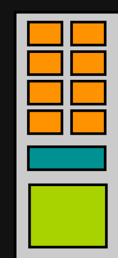
**Software** | **Hardware**

Thread

Thread Processor

Threads are executed by thread processors

Thread Block

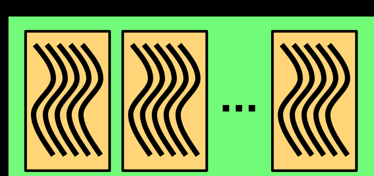


Multiprocessor

Thread blocks are executed on multiprocessors

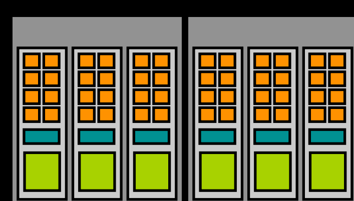
Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)





Grid



Device

A kernel is launched as a grid of thread blocks

Only one kernel can execute on a device at one time

# Key Parallel Abstractions in CUDA

- Trillions of lightweight threads
  - Simple decomposition model
- Hierarchy of concurrent threads
  - Simple execution model
- Lightweight synchronization of primitives
  - Simple synchronization model
- Shared memory model for thread cooperation
  - Simple communication model

# Outline

- CUDA programming model
- *Basics of CUDA programming*
    - *Software stack*
    - Data management
    - Executing code on the GPU
- CUDA libraries
    - BLAS
    - FFT

# CUDA Installation

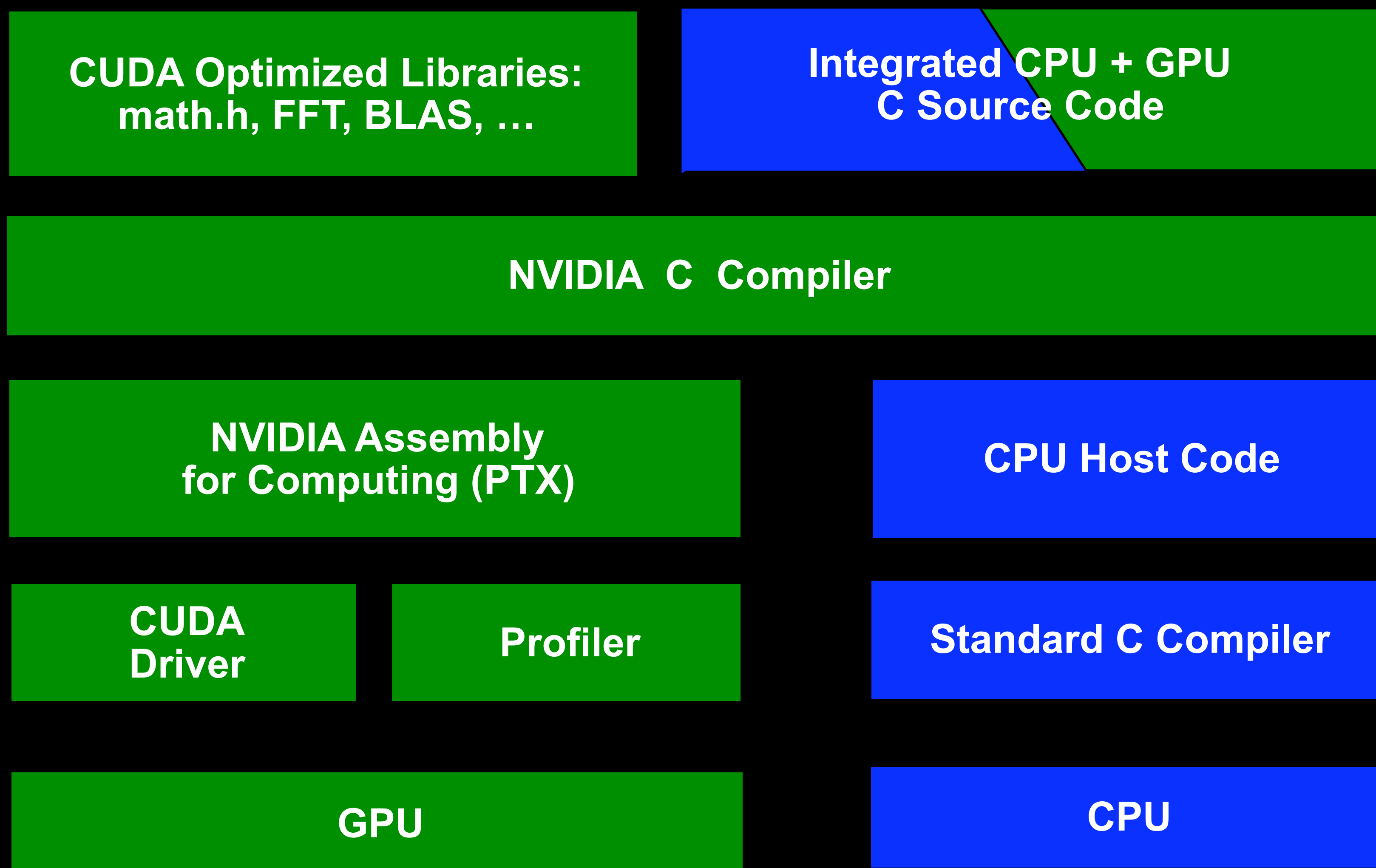
- CUDA installation consists of
  - Driver
  - CUDA Toolkit (compiler, libraries)
  - CUDA SDK (example codes)

NVISION 08
THE WORLD OF VISUAL COMPUTING

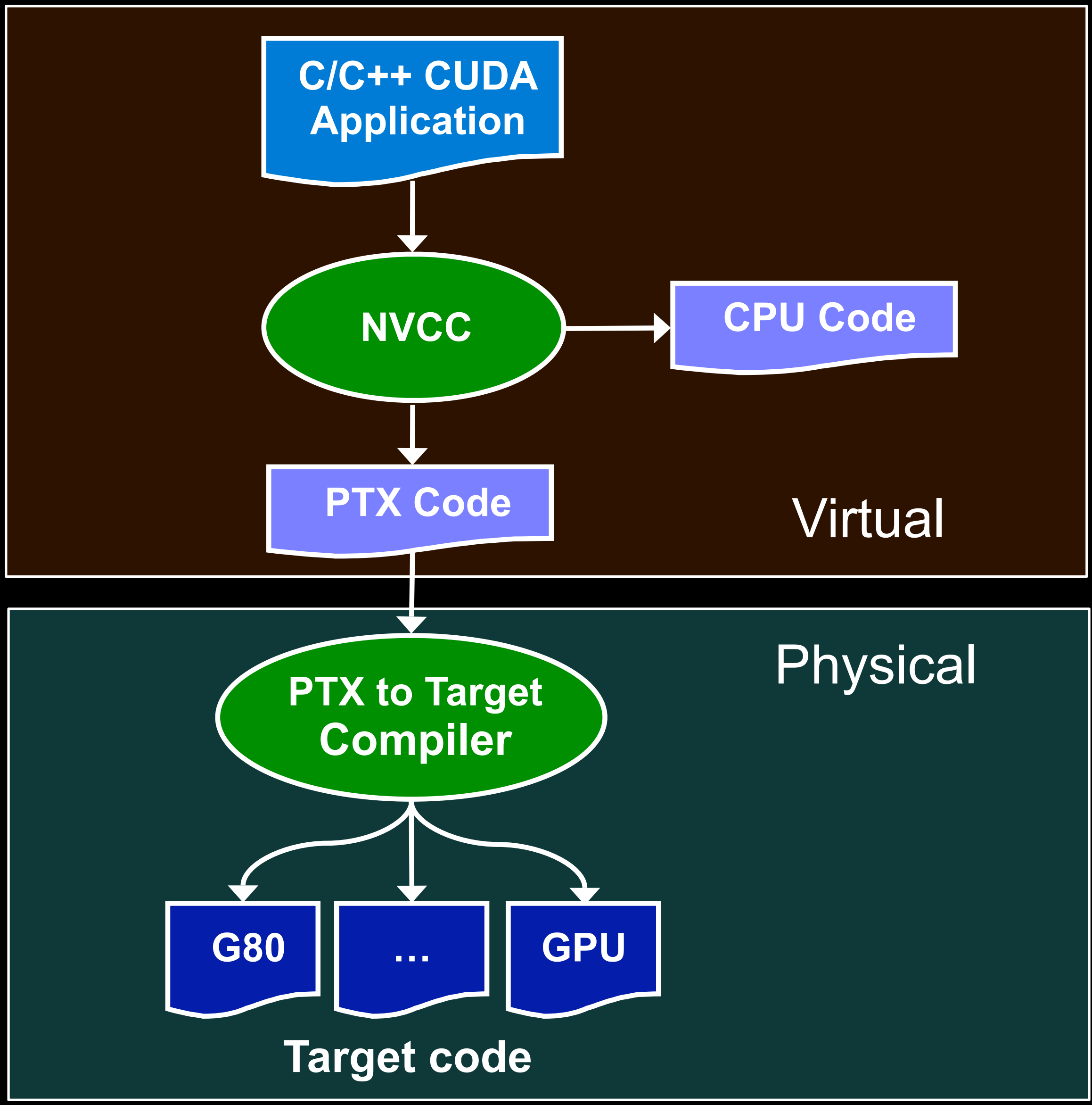


NVIDIA.

# CUDA Software Development



# Compiling CUDA Code



# Build Configurations

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

- **Builds release mode**

nvcc -g <filename>.cu

- **Builds debug mode**
- **Can debug host code but not device code**

nvcc -deviceemu <filename>.cu

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

nvcc -deviceemu -g <filename>.cu

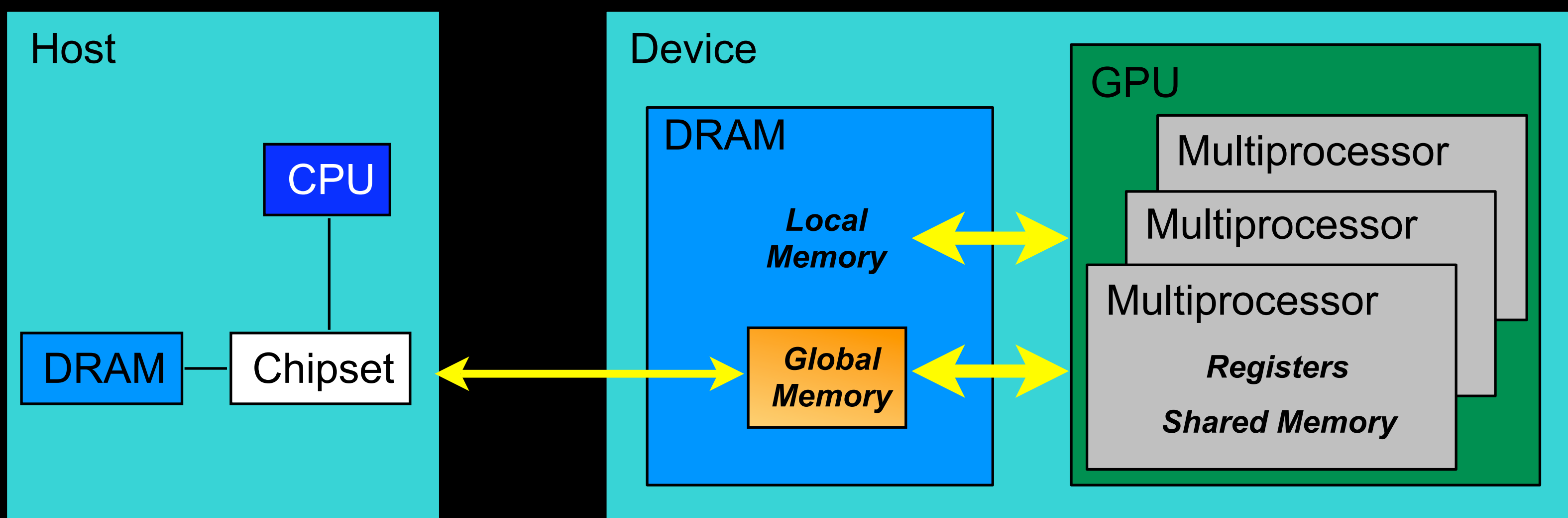
- **Builds debug device emulation mode**
- **All code runs on CPU, with debug symbols**

# Outline

- CUDA programming model
- *Basics of CUDA programming*
  - Software stack
  - *Data management*
  - Executing code on the GPU
- CUDA libraries
  - BLAS
  - FFT

# 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 *a_d = 0;
cudaMalloc( (void**)&a_d,  nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

NVISION 08
THE WORLD OF VISUAL COMPUTING



NVIDIA

# 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

NVISION 08 THE WORLD OF VISUAL COMPUTING



NVIDIA

# Data Movement Example

```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```

# Data Movement Example

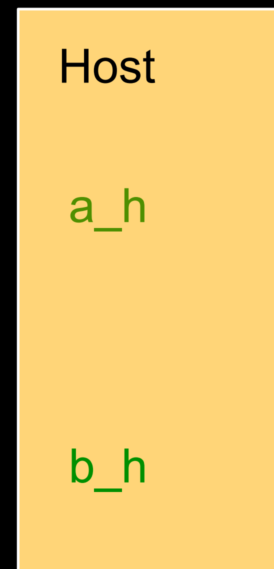
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



# Data Movement Example

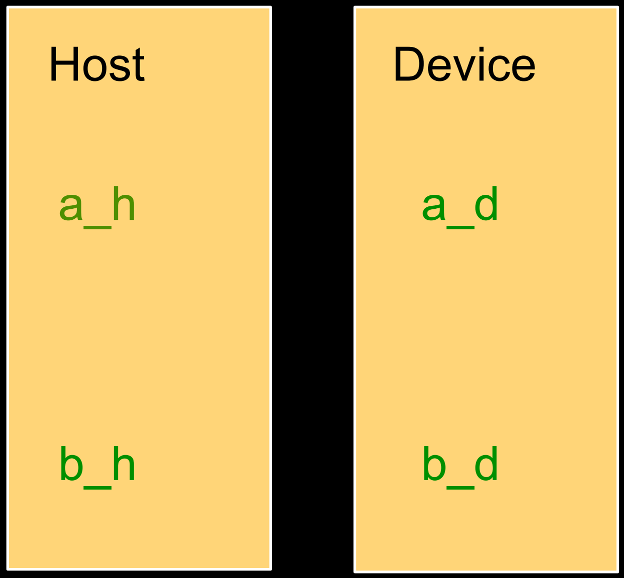
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



# Data Movement Example

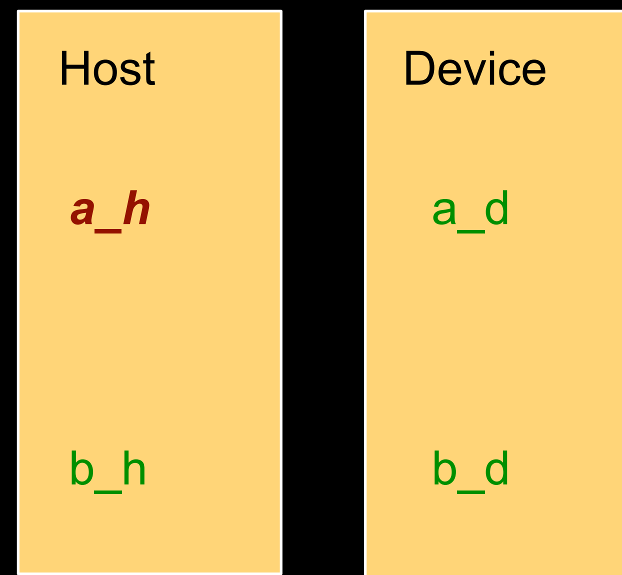
```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```



# Data Movement Example

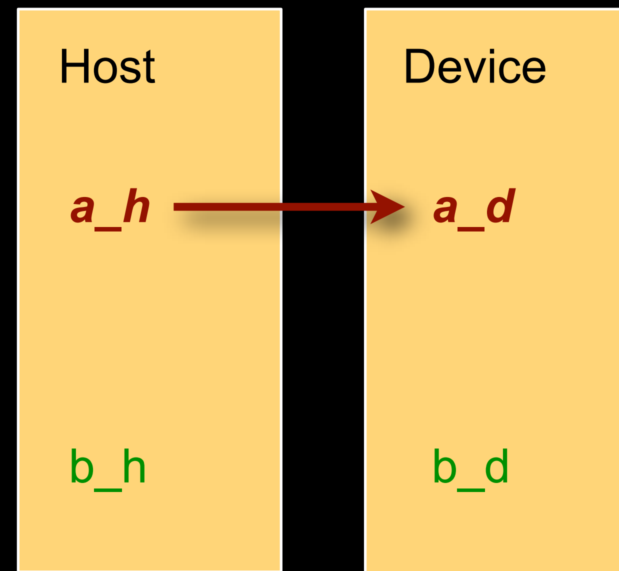
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



# Data Movement Example

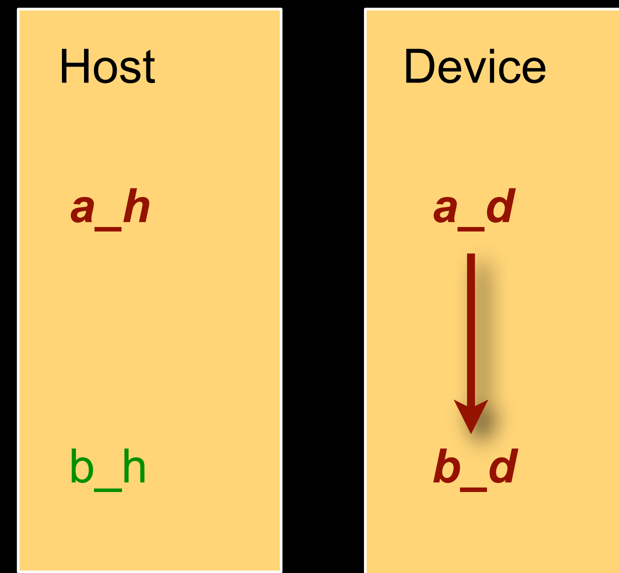
```
int main(void)
{
   float *a_h, *b_h;  // host data
   float *a_d, *b_d;  // device data
   int N = 14, nBytes, i ;

   nBytes = N*sizeof(float);
   a_h = (float *)malloc(nBytes);
   b_h = (float *)malloc(nBytes);
   cudaMalloc((void **) &a_d, nBytes);
   cudaMalloc((void **) &b_d, nBytes);

   for (i=0, i<N; i++) a_h[i] = 100.f + i;

   cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
   cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
   cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

   for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
   free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
   return 0;
}
```



# Data Movement Example

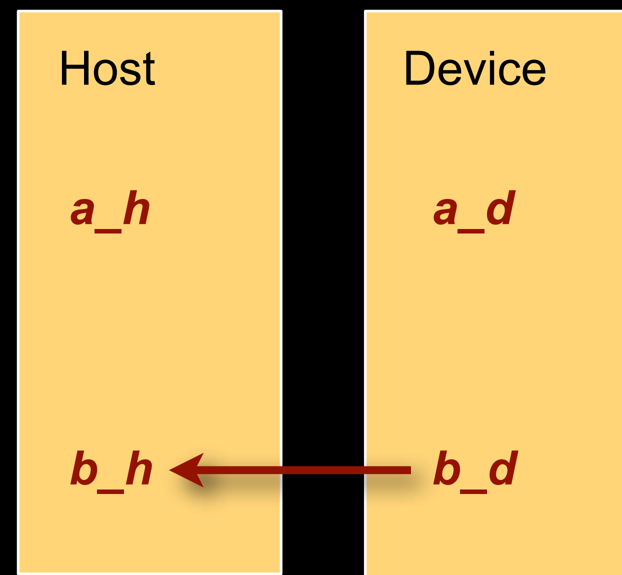
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



# Data Movement Example

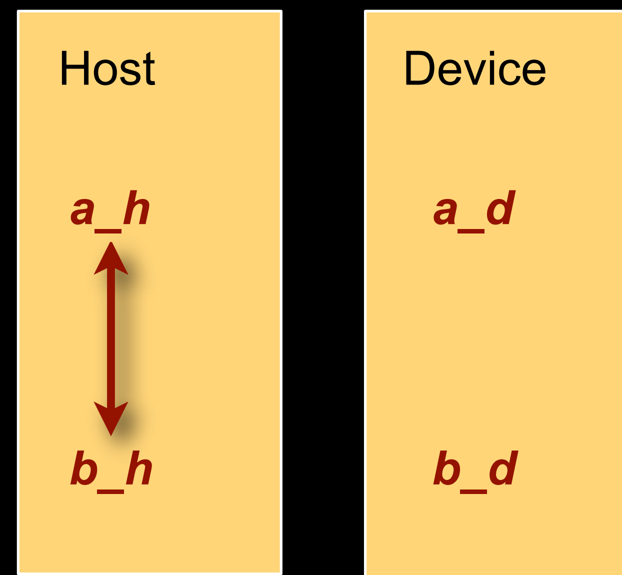
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```




nVISION 08
THE WORLD OF VISUAL COMPUTING

# Data Movement Example

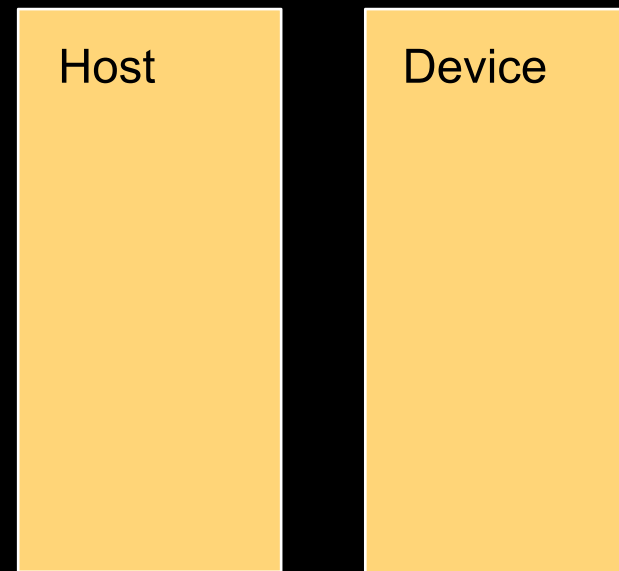
```
int main(void)
{
  float *a_h, *b_h;  // host data
  float *a_d, *b_d;  // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```



# Outline

- CUDA programming model
- *Basics of CUDA programming*
  - Software stack
  - Data management
  - *Executing code on the GPU*
- CUDA libraries
  - FFT
  - BLAS

# Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Cannot access host memory
  - Must have void return type
  - No variable number of arguments (“varargs”)
  - Not recursive
  - No static variables
- Function arguments automatically copied from host to device

# Function Qualifiers

- Kernels designated by function qualifier:
  - __global__
    - Function called from host and executed on device
    - Must return void
- Other CUDA function qualifiers
  - __device__
    - Function called from device and run on device
    - Cannot be called from host code
  - __host__
    - Function called from host and executed on host (default)
    - __host__ and __device__ qualifiers can be combined to generate both CPU and GPU code

# Launching Kernels

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(…)
```

- Execution Configuration (“`<<< >>>`”)
  - dG - dimension and size of grid in blocks
    - Two-dimensional: x and y
    - Blocks launched in the grid: dG.x * dG.y
  - dB - dimension and size of blocks in threads:
    - Three-dimensional: x, y, and z
    - Threads per block: dB.x * dB.y * dB.z
  - Unspecified dim3 fields initialize to 1

# Execution Configuration Examples

```
dim3 grid, block;
grid.x = 2; grid.y = 4;
block.x = 8; block.y = 16;

kernel<<<grid, block>>>(...);
```

```
dim3 grid(2, 4), block(8,16);

kernel<<<grid, block>>>(...);
```

Equivalent assignment using constructor functions

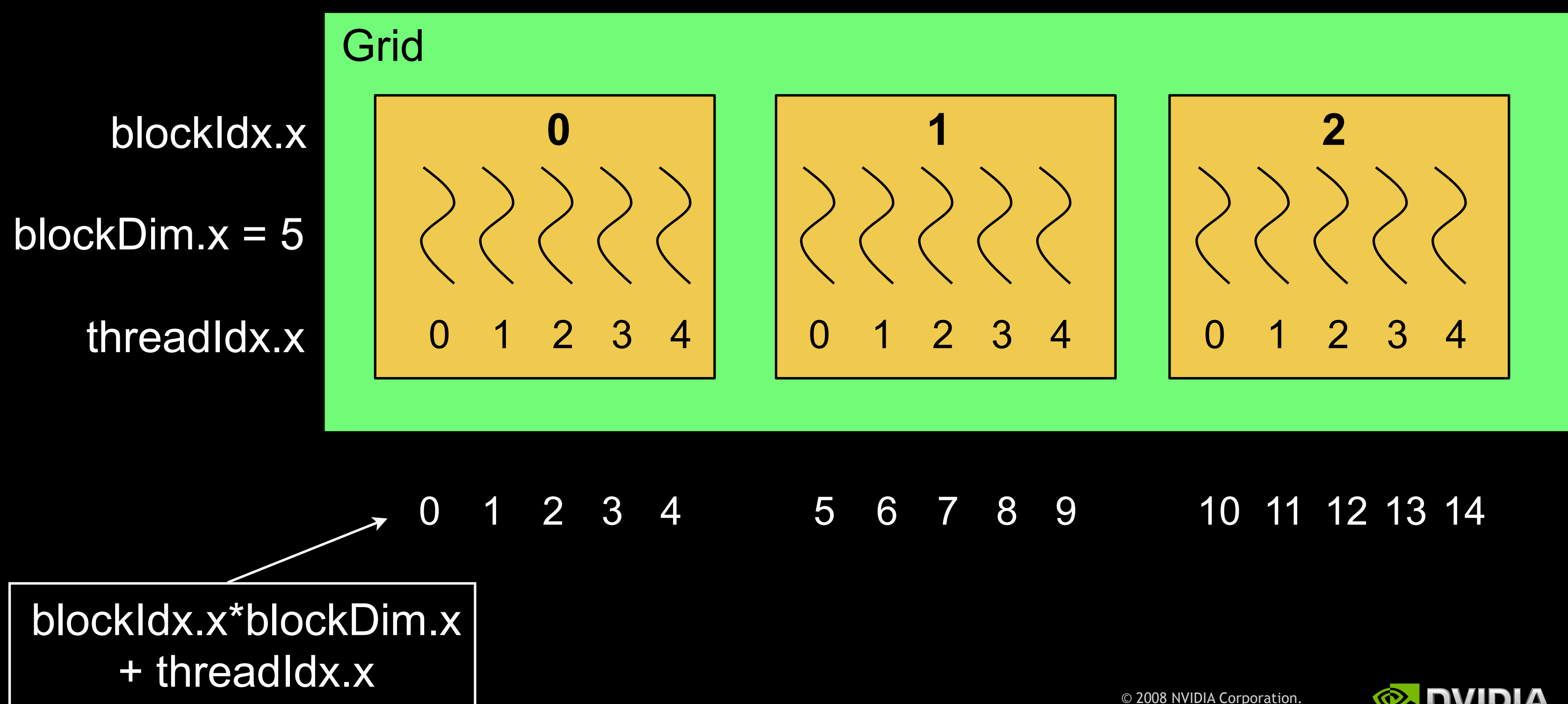
```
kernel<<<32,512>>>(...);
```

# 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

# Unique Thread IDs

- **Built-in variables are used to determine unique thread IDs**
  - **Map from local thread ID (threadIdx) to a global ID which can be used as array indices**



# Minimal Kernels

```
__global__ void minimal( int* a_d, int value)
{
    *a_d = value;
}

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

# Increment Array Example

## CPU program

```
void inc_cpu(int *a, int N)
{
   int idx;

   for (idx = 0; idx<N; idx++)
      a[idx] = a[idx] + 1;
}

int main()
{
    ...
    inc_cpu(a, N);
}
```

## CUDA program

```
__global__ void inc_gpu(int *a, int N)
{
  int idx = blockIdx.x * blockDim.x
               + threadIdx.x;
  if (idx < N)
     a[idx] = a[idx] + 1;
}

int main()
{
  ...
  dim3 dimBlock (blocksize);
  dim3 dimGrid( ceil( N / (float)blocksize) );
  inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```

# 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

nVISION 08 THE WORLD OF VISUAL COMPUTING



NVIDIA

# Host Synchronization Example

```
// copy data from host to device
cudaMemcpy(a_d, a_h, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
inc_gpu<<<ceil(N/(float)blocksize), blocksize>>>(a_d, N);

// run independent CPU code
run_cpu_stuff();

// copy data from device back to host
cudaMemcpy(a_h, a_d, numBytes, cudaMemcpyDeviceToHost);
```

# Variable Qualifiers (GPU code)

- __device__
  - Stored in global 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)
  - Specified by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: thread block
- Unqualified variables:
  - Scalars and built-in vector types are stored in registers
  - What doesn't fit in registers spills to "local" 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>>>(…);
   …
}
```

# 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

NVISION 08 THE WORLD OF VISUAL COMPUTING



NVIDIA

# GPU Atomic Operations

- Associative operations
  - add, sub, increment, decrement, min, max, ...
  - and, or, xor
  - exchange, compare, swap
- Atomic operations on 32-bit words in *global* memory
  - Requires compute capability 1.1 or higher (G84/G86/G92)
- Atomic operations on 32-bit words in *shared* memory and 64-bit words in global memory
  - Requires compute capability 1.2 or higher

# 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]`, `double[1..2]`
  - 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)

# 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() ) );
```

# CUDA Programming Resources

- We only covered basic features
  - See Programming Guide for more of the API
  - Additional features covered in the Optimization session
- CUDA SDK examples
- CUDA Zone - http://www.nvidia.com/cuda
  - CUDA U
  - Forums

NVISION 08
THE WORLD OF VISUAL COMPUTING



NVIDIA

# Outline

- CUDA programming model
- Basics of CUDA programming
  - Software stack
  - Data management
  - Executing code on the GPU
- *CUDA libraries*
  - *BLAS*
  - FFT

# 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 CUBLAS functions
  - Retrieve data
- CUBLAS library helper functions
  - Creating and destroying data in GPU space
  - Writing data to and retrieving data from objects

# Supported Features

| | Single Precision | | Double Precision* | |
|---|---|---|---|---|
| | Real | Complex | Real | Complex |
| Level 1 | ✔ | ✔ | ✔ | |
| Level 2 | ✔ | | dgemv, dger, dsyr, dtrsv | |
| Level 3 | ✔ | cgemm | ✔ | zgemm |

**Double-precision functions only supported on GPUs with double-precision hardware*

# Using CUBLAS

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

# CUBLAS Helper Functions

- cublasInit()
  - Initializes CUBLAS library
- cublasShutdown()
  - Releases resources used by CUBLAS library
- cublasGetError()
  - Returns last error from CUBLAS core function (+ resets)
- cublasAlloc()
  - Wrapper around cudaMalloc() to allocate space for array
- cublasFree()
  - destroys object in GPU memory
- cublas[Set|Get][Vector|Matrix]()
  - Copies array elements between CPU and GPU memory
  - Accommodates non-unit strides

NVISION 08
THE WORLD OF VISUAL COMPUTING



NVIDIA

# sgemmExample.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "cublas.h"

int main(void)
{
 float *a_h, *b_h, *c_h;
 float *a_d, *b_d, *c_d;
 float alpha = 1.0f, beta = 0.0f;
 int N = 2048, n2 = N*N;
 int nBytes, i;

 nBytes = n2*sizeof(float);

 a_h = (float *)malloc(nBytes);
 b_h = (float *)malloc(nBytes);
 c_h = (float *)malloc(nBytes);

 for (i=0; i < n2; i++) {
   a_h[i] = rand() / (float) RAND_MAX;
   b_h[i] = rand() / (float) RAND_MAX;
 }

 cublasInit();

 cublasAlloc(n2, sizeof(float), (void **)&a_d);
 cublasAlloc(n2, sizeof(float), (void **)&b_d);
 cublasAlloc(n2, sizeof(float), (void **)&c_d);

 cublasSetVector(n2, sizeof(float), a_h, 1, a_d, 1);
 cublasSetVector(n2, sizeof(float), b_h, 1, b_d, 1);

 cublasSgemm('n', 'n', N, N, N, alpha, a_d, N,
                          b_d, N, beta, c_d, N);

 cublasGetVector(n2, sizeof(float), c_d, 1, c_h, 1);

 free(a_h); free(b_h); free(c_h);
 cublasFree(a_d); cublasFree(b_d);
 cublasFree(c_d);

 cublasShutdown();
 return 0;
}
```

# Additional Resources

- CUDA SDK example
  - simpleCUBLAS
- CUBLAS Library documentation
  - in doc folder of CUDA Toolkit or download from CUDA Zone

# Outline

- CUDA programming model
- Basics of CUDA programming
  - Software stack
  - Data management
  - Executing code on the GPU
- *CUDA libraries*
  - BLAS
  - *FFT*

# CUFFT

- CUFFT is the CUDA FFT library
- 1D, 2D, and 3D transforms of complex and real single-precision data
- Batched execution for multiple 1D transforms in parallel
- 1D transforms up to 8 million elements
- 2D and 3D transforms in the range of [2,16384]
- In-place and out-of-place

# More on CUFFT

- For 2D and 3D transforms, CUFFT uses row-major order
- CUFFT performs un-normalized transforms
  - IFFT(FFT(A)) = length(A)*A
- CUFFT modeled after FFTW
  - Based on *plans* used to specify optimal configuration for a particular sized FFT
  - Once a *plan* is created it can be reused (to avoid recomputing the optimal configuration)

# CUFFT Types and Definitions

- cufftHandle
  - Type used to store and access CUFFT plans
- cufftResults
  - Enumeration of API function return values
- cufftReal
  - single-precision, real datatype
- cufftComplex
  - single-precision, complex datatype

- Real and complex transforms
  - CUFFT_C2C, CUFFT_C2R, CUFFT_R2C
- Directions
  - CUFFT_FORWARD, CUFFT_INVERSE

# CUFFT Example

```
#include <stdio.h>
#include <math.h>
#include "cufft.h"

int main(int argc, char *argv[])
{
  cufftComplex *a_h, *a_d;
  cufftHandle plan;
  int N = 1024, batchSize = 10;
  int i, nBytes;
  double maxError;

  nBytes = sizeof(cufftComplex)*N*batchSize;
  a_h = (cufftComplex *)malloc(nBytes);

  for (i=0; i < N*batchSize; i++) {
    a_h[i].x = sinf(i);
    a_h[i].y = cosf(i);
  }

  cudaMalloc((void **)&a_d, nBytes);
  cudaMemcpy(a_d, a_h, nBytes,
                       cudaMemcpyHostToDevice);

  cufftPlan1d(&plan, N, CUFFT_C2C, batchSize);

  cufftExecC2C(plan, a_d, a_d, CUFFT_FORWARD);
  cufftExecC2C(plan, a_d, a_d, CUFFT_INVERSE);

  cudaMemcpy(a_h, a_d, nBytes,
                       cudaMemcpyDeviceToHost);

  // check error - normalize
  for (maxError = 0.0, i=0; i < N*batchSize; i++) {
    maxError = max(fabs(a_h[i].x/N-sinf(i)), maxError);
    maxError = max(fabs(a_h[i].y/N-cosf(i)), maxError);
  }

  printf("Max fft error = %g\n", maxError);

  cufftDestroy(plan);
  free(a_h); cudaFree(a_d);

  return 0;
}
```

# Additional CUFFT Resources

- CUDA SDK examples
  - simpleCUFFT
  - convolutionFFT2D
  - oceanFFT
- CUFFT Library documentation
  - In doc folder of CUDA Toolkit or download from CUDA Zone

nvision 08

THE WORLD OF VISUAL COMPUTING

# Getting Started with CUDA

Greg Ruetsch, Brent Oster



NVIDIA.