NVIDIA®

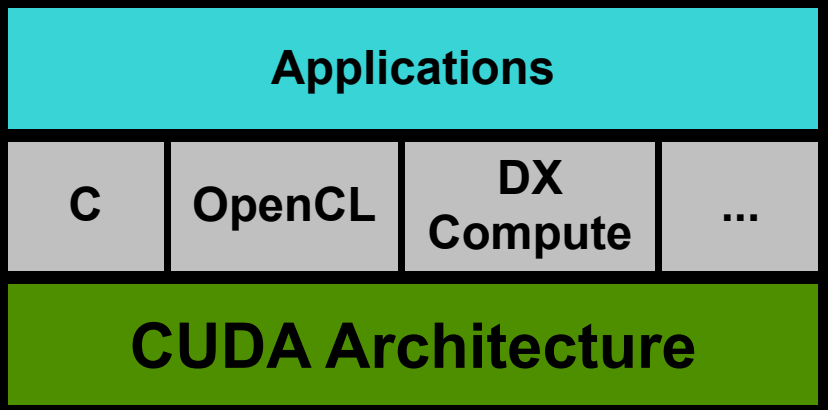
High Performance Computing with CUDA

DoD HPCMP: 2009 Users Group Conference
San Diego, CA
June 15, 2009

CUDA

A Parallel Computing Architecture for NVIDIA GPUs

- Supports standard languages and APIs
  - C
  - OpenCL
  - DX Compute
- Supported on common operating systems
  - Linux
  - MacOS
  - Windows



Slides

- Hostname: corpftp.nvidia.com
- Username: TeslaAE
- Password: $BigBird

- Filename: CUDA06152009.pdf

Schedule

- Programming Model
- Programming Basics
- Libraries
- Fortran Integration
- Performance Analysis and Optimization
- Driver API
- OpenCL
- MultiGPU
- Q&A

NVIDIA®

CUDA Programming Model Overview

CUDA C Programming

- Heterogeneous programming model
  - CPU and GPU are separate devices with separate memory spaces
  - CPU code is standard C/C++
    - Driver API: low-level interface
    - Runtime API: high-level interface (one extension to C)
  - GPU code
    - Subset of C with extensions
- CUDA goals
  - Scale GPU code to 100s of cores, 1000s of parallel threads
  - Facilitate heterogeneous computing

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
    - Fast switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions

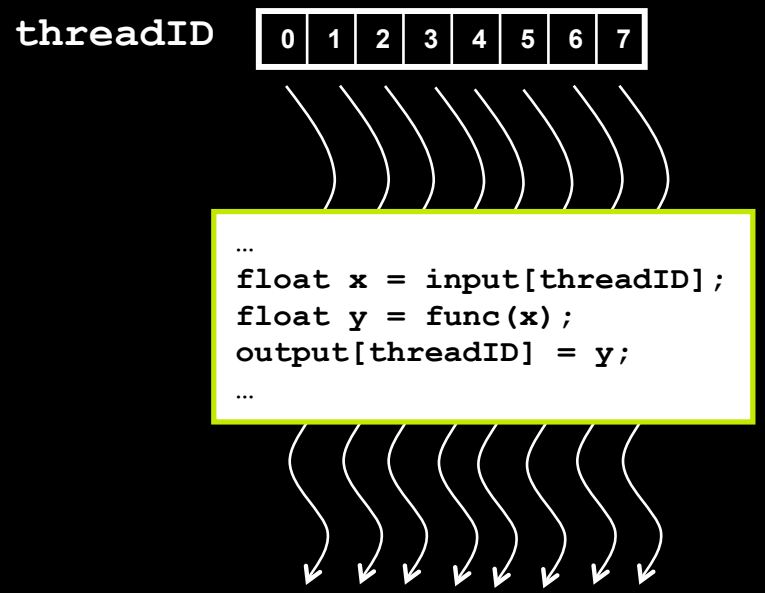
Device = GPU

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

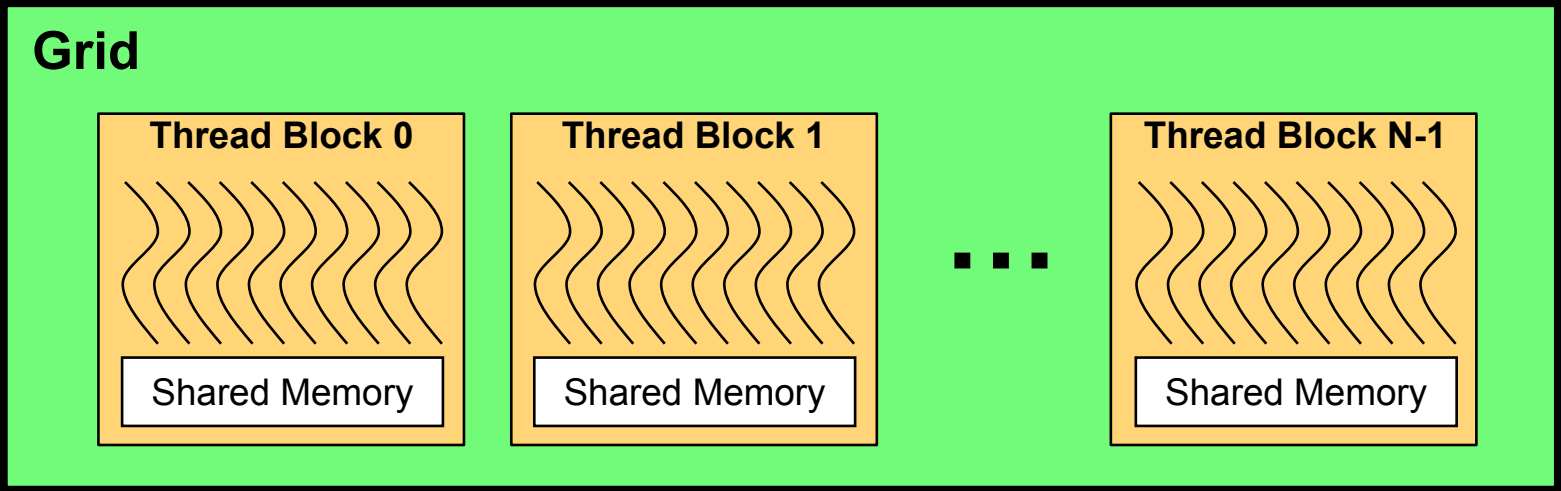


Thread Cooperation

- The Missing Piece: threads may need to cooperate
- Thread cooperation is a powerful feature of CUDA
- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Bandwidth reduction
- Cooperation between a monolithic array of threads is not scalable
  - Cooperation within smaller batches of threads is scalable

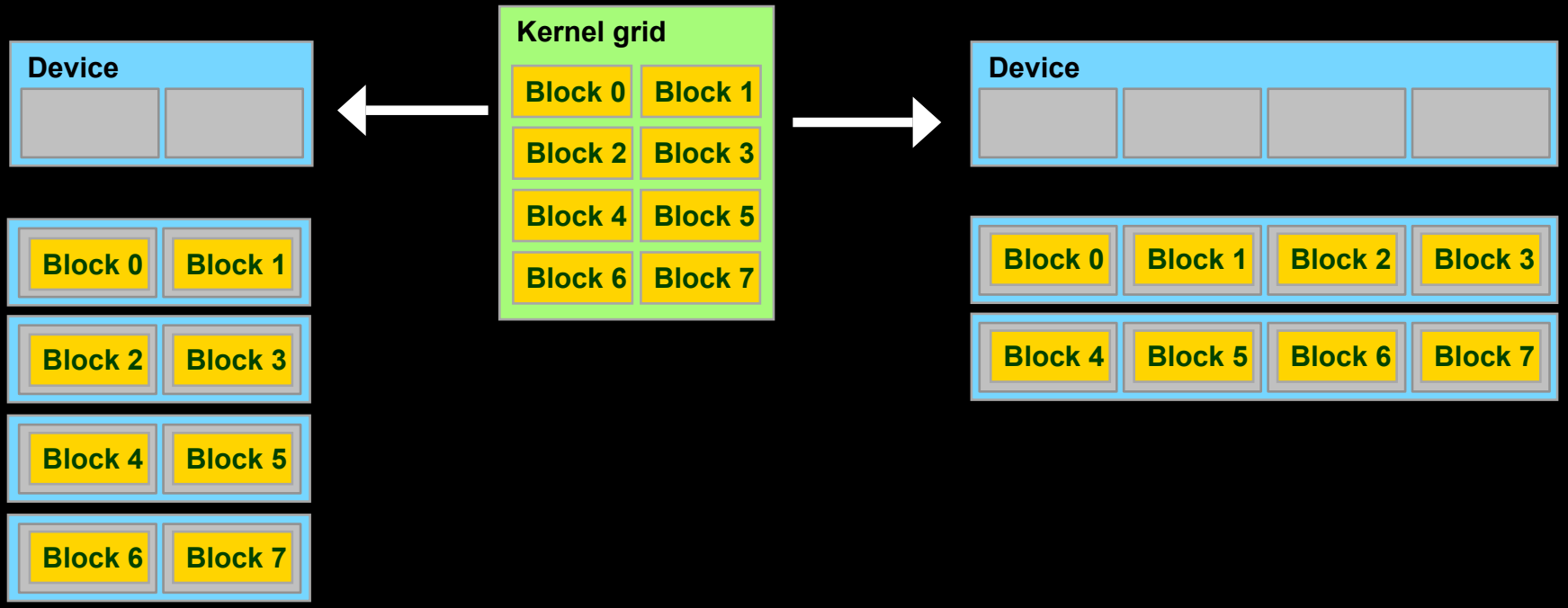
Thread Batching

- 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



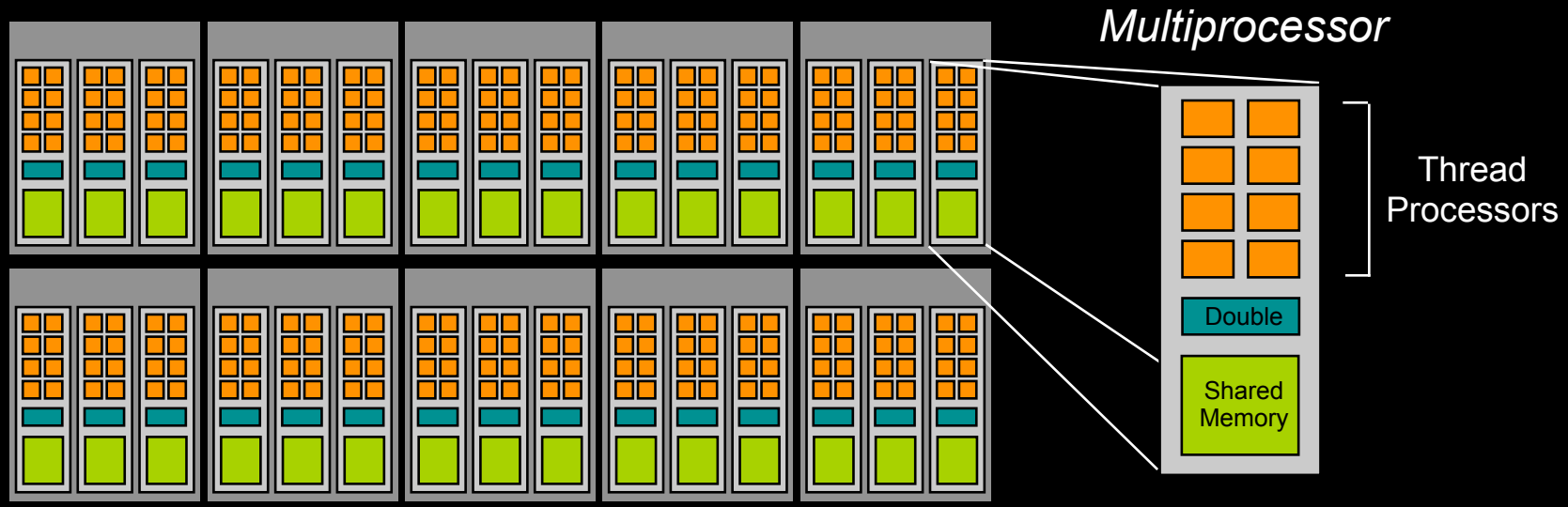
Transparent Scalability

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



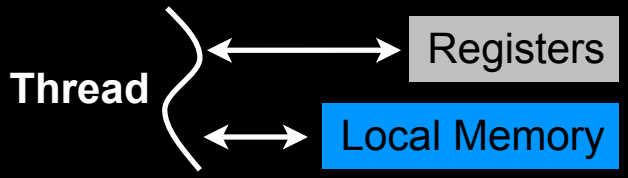
10-Series Architecture

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

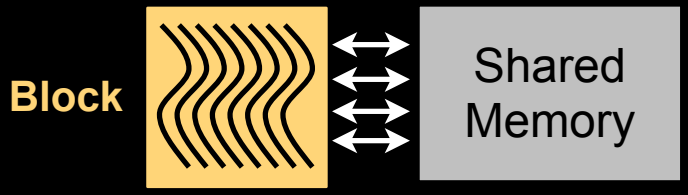


Kernel Memory Access

- **Per-thread**

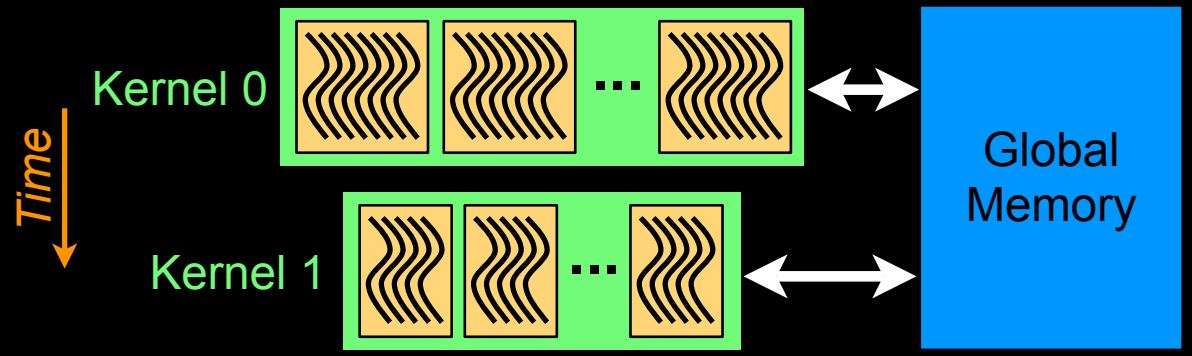


- **Per-block**



- On-chip, small
- Fast

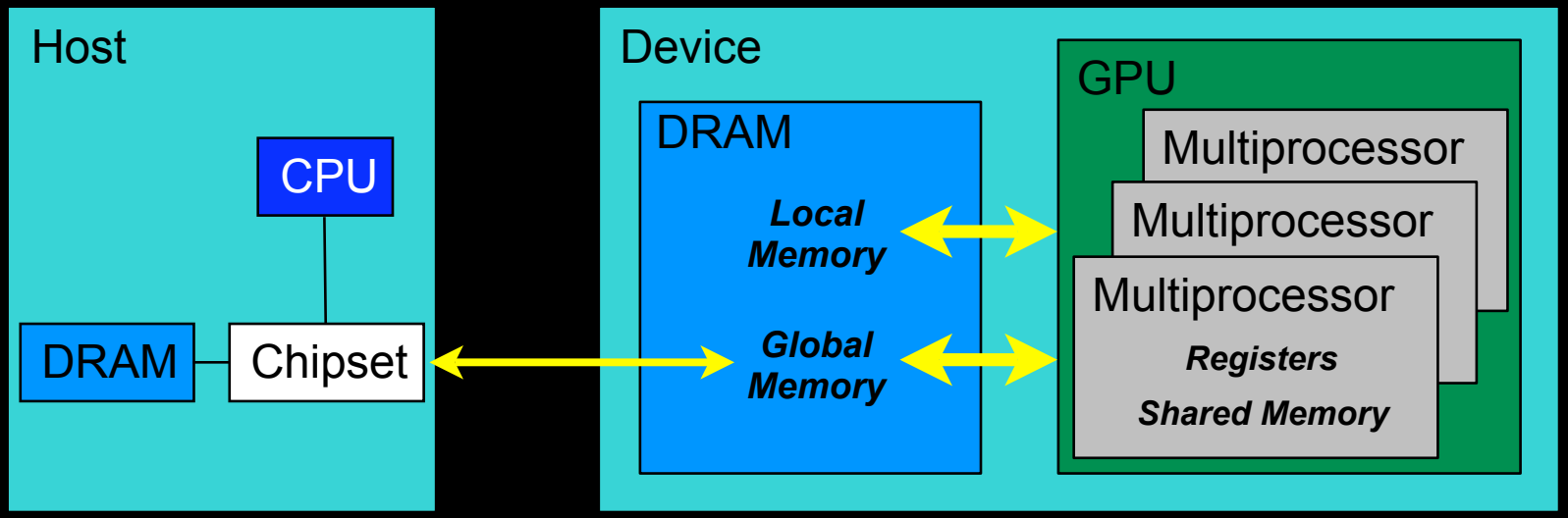
- **Per-device**



- Off-chip, large
- Uncached
- Persistent across kernel launches
- Kernel I/O

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

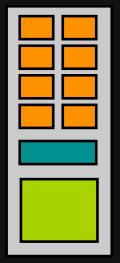
Software **Hardware**

Thread

Thread Processor

Threads are executed by thread processors

Thread Block

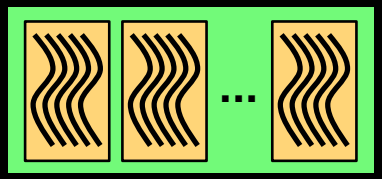


Multiprocessor

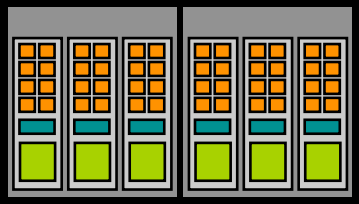
Thread blocks are executed on multiprocessors

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

NVIDIA®

CUDA Programming Basics

Part I - Software Stack and Memory Management

Outline of CUDA Programming Basics

Part I

- CUDA software stack and compilation
- GPU memory management

Part II

- Kernel launches
- Some specifics of GPU code

NOTE: only the basic features are covered

- See the Programming Guide for many more API functions

CUDA Software Development Environment

- Main components
  - Device Driver (part of display driver)
  - Toolkit (compiler, documentation, libraries)
  - SDK (example codes, white papers)
- Consult Quickstart Guides for installation instructions on different platforms
  - http://www.nvidia.com/cuda

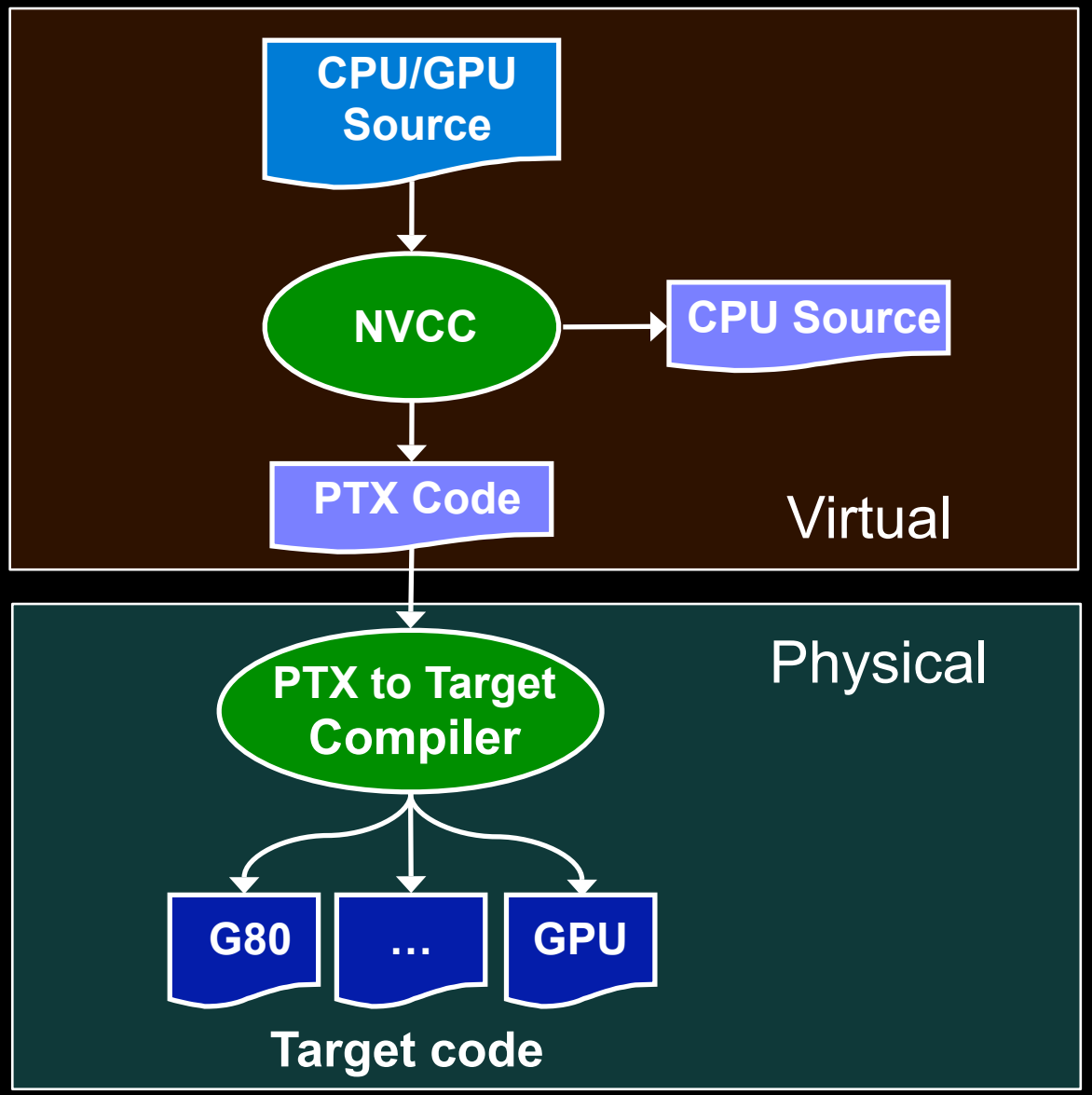
CUDA Software Development Tools

- Profiler
  - Available now for all supported OSs
  - Command-line or GUI
  - Sampling signals on GPU for:
    - Memory access parameters
    - Execution (serialization, divergence)
- Debugger
  - Currently Linux only (gdb)
  - Runs on the GPU
- Emulation mode
  - Compile with `-deviceemu`

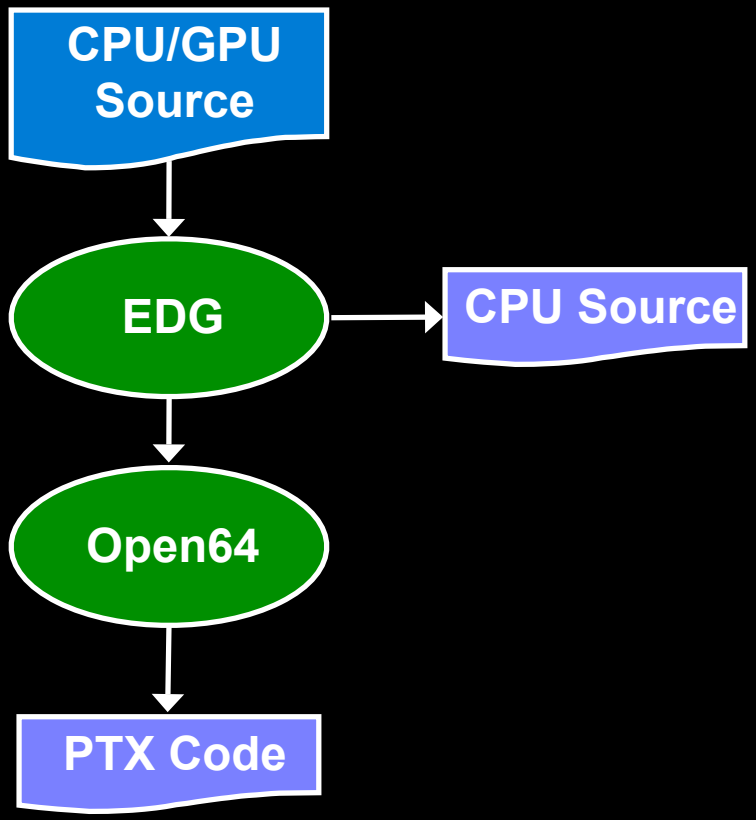
Compiler

- Any source file containing language extensions, like "<<< >>>", must be compiled with `nvcc`
- `nvcc` is a *compiler driver*
  - Invokes 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
  - PTX or object code directly
- An executable requires linking to:
  - Runtime library (`cudart`)
  - Core library (`cuda`)

Compiling



nvcc & PTX Virtual Machine



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

Compiler Flags

- Important flags:
  - `-arch=sm_13` enables double precision on compatible hardware
  - `-G` enables debugging on device code
  - `--ptxas-options=-v` shows register and memory usage
  - `--maxregcount=N` limits the number of registers to `N`
  - `-use_fast_math` uses fast math library

GPU Memory Management

Memory spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C functions
- Pointers are just addresses
  - Can't tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa

GPU Memory Allocation / Release

- Host (CPU) manages device (GPU) memory
  - `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);
```

Data Copies

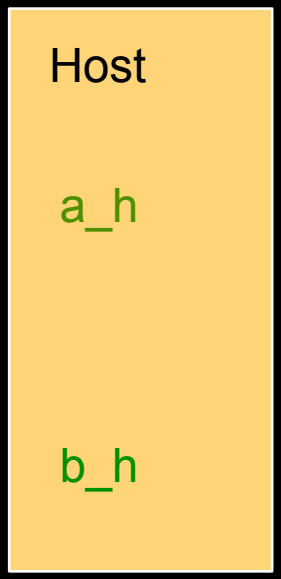
- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - `direction` specifies locations (host or device) of `src` and `dst`
  - Blocks CPU thread: returns after the copy is complete
  - Doesn't start copying until previous CUDA calls complete
- `enum cudaMemcpyKind`
 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

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;
}
```

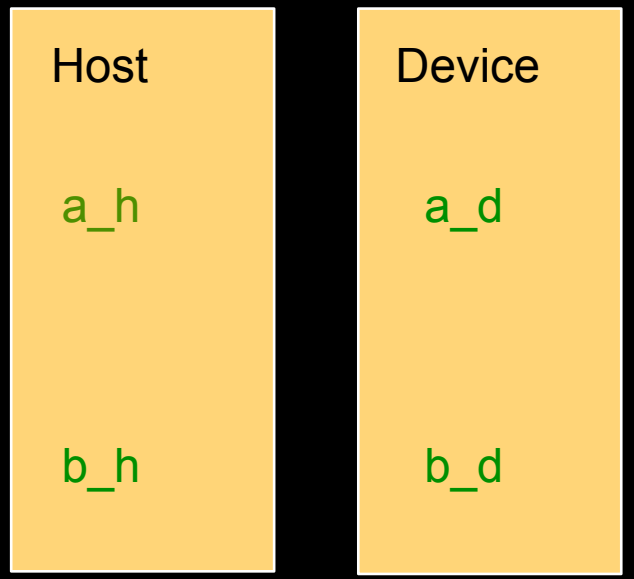
Data Movement Example


```
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;
}
```



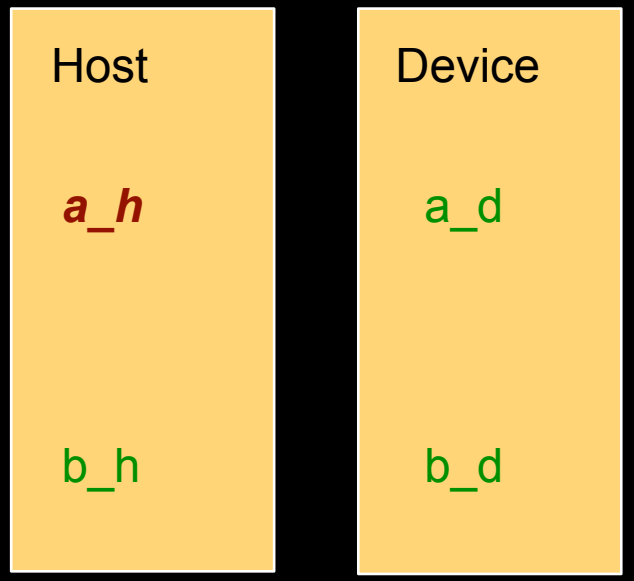
Data Movement Example


```
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;
}
```



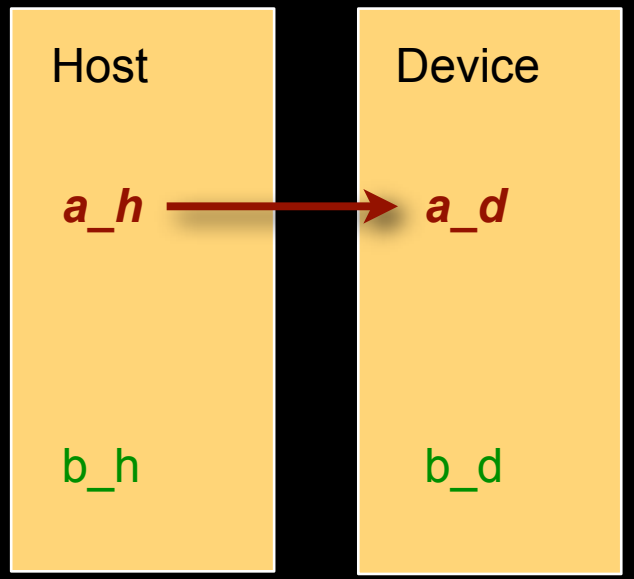
Data Movement Example


```
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;
}
```



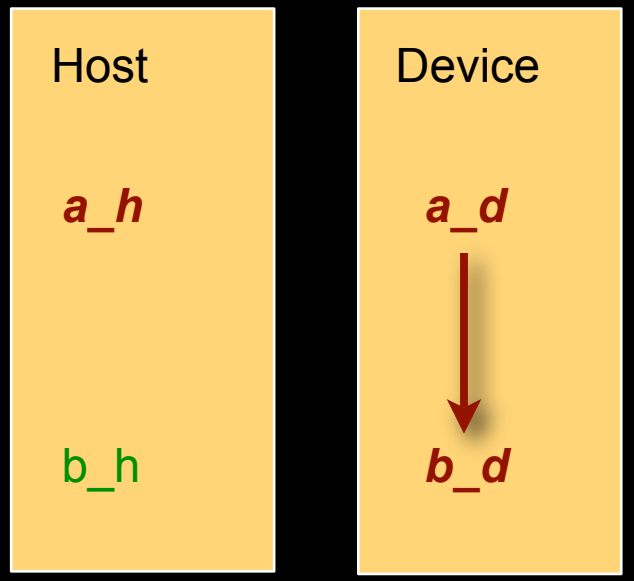
Data Movement Example


```
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;
}
```



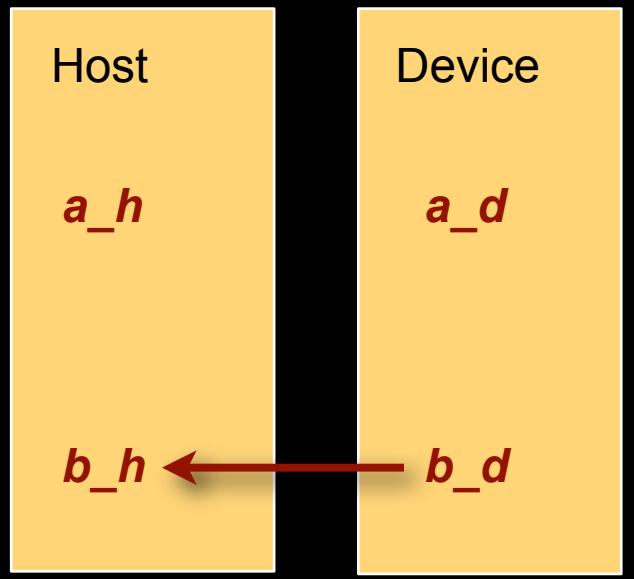
Data Movement Example


```
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;
}
```



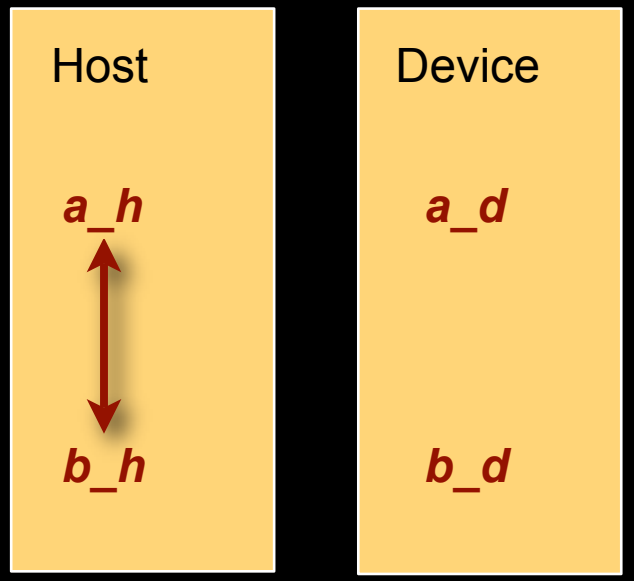
Data Movement Example


```
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;
}
```



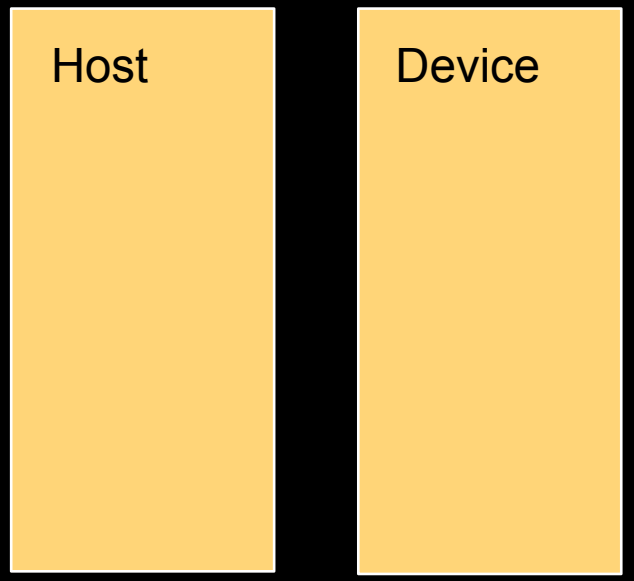
Data Movement Example

```
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;
}
```



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;
}
```



NVIDIA®

CUDA Programming Basics

Part II - Kernels

Outline of CUDA Basics

Part I

- CUDA software stack and compilation
- GPU memory management

Part II

- Kernel launches
- Some specifics of GPU code

NOTE: only the basic features are covered

See the Programming Guide for many more API functions

CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

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

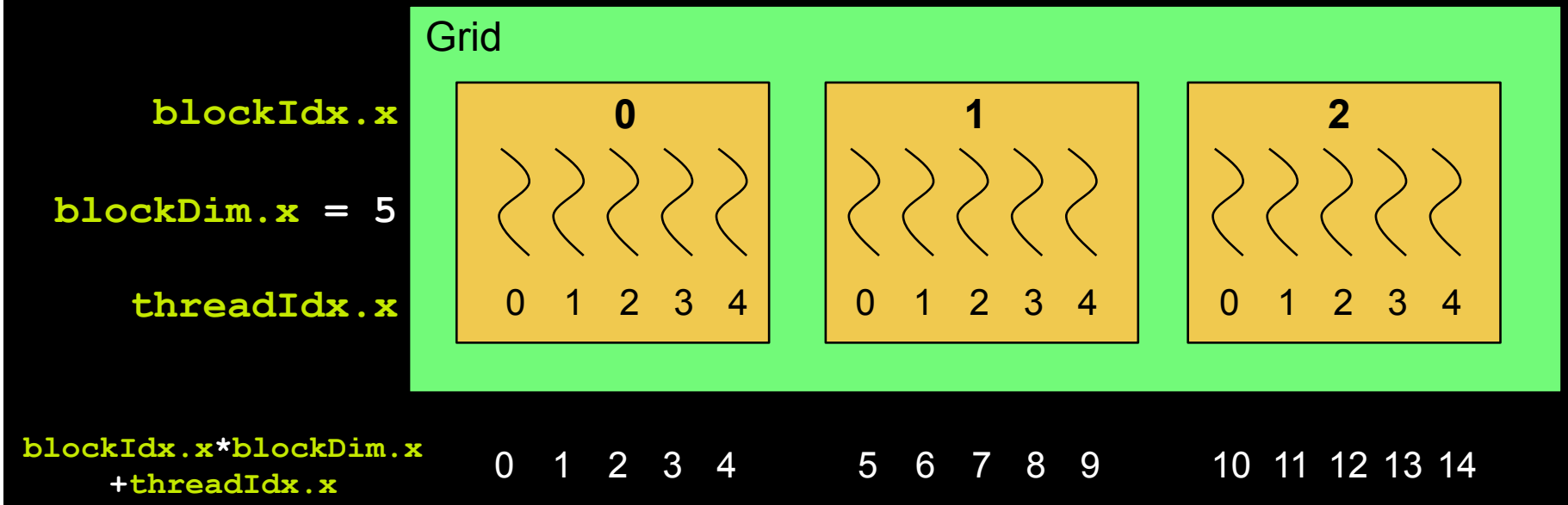
```
kernel<<<32,512>>>(...);
```

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 kernel( int *a )
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 0 1 1 1 1 2 2 2 2 2 2 2

```
__global__ void kernel( int *a )
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0

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;
}

void main()
{
  …
  inc_cpu(a, N);
  …
}
```

CUDA program

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

void main()
{
  …
  dim3 dimBlock (blocksize);
  dim3 dimGrid(ceil(N/(float)blocksize));
  inc_gpu<<<dimGrid, dimBlock>>>(a_d, 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

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
  - Arrays may be in registers or 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

GPU Atomic Integer Operations

- Requires hardware with compute capability >= 1.1
  - G80 = Compute capability 1.0
  - G84/G86/G92 = Compute capability 1.1
  - GT200 = Compute capability 1.3
- 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
- Atomic operations on integers in shared memory
  - Requires compute capability >= 1.2

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- Independence requirement provides scalability

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 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);
```

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()));
```

NVIDIA®

Libraries

Libraries

- Two widely used libraries included in the toolkit
  - CUBLAS: BLAS implementation
  - CUFFT: FFT implementation

- CUDPP (Data Parallel Primitives), available from http://www.gpgpu.org/developer/cudpp/:
  - Reduction
  - Scan
  - Sort
  - Sparse matrix vector multiplication

CUBLAS

- Implementation of BLAS (Basic Linear Algebra Subprograms)
  - Self-contained at the API level, no direct interaction with 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
- BLAS is split into 3 levels:
  - Level 1 (vector-vector operations, O(n))
  - Level 2 (matrix-vector operations, $O(n^2)$)
  - Level 3 (matrix-matrix operations, $O(n^3)$)

Supported Features

- BLAS functions
  - Single precision data:
    - Level 1
    - Level 2
    - Level 3
  - Complex single precision data:
    - Level 1
    - CGEMM
  - Double precision data:
    - Level 1: DASUM, DAXPY, DCOPY, DDOT, DNRM2, DROT, DROTM, DSCAL, DSWAP, ISAMAX, IDAMIN
    - Level 2: DGEMV, DGER, DSYR, DTRSV
    - Level 3: DGEMM, DTRSM, DTRMM, DSYMM, DSYRK, DSYR2K
  - Complex double precision data:
    - ZGEMM

Using CUBLAS

- Following BLAS convention, CUBLAS uses column-major storage
- Interface to CUBLAS library is in cublas.h
- Function naming convention:
  - cublas + BLAS name
  - e.g., cublasSGEMM
- Error handling
  - CUBLAS core functions do not return error
    - CUBLAS provides function to retrieve last error recorded
  - CUBLAS helper functions do return error
- Helper functions:
  - Memory allocation, data transfer

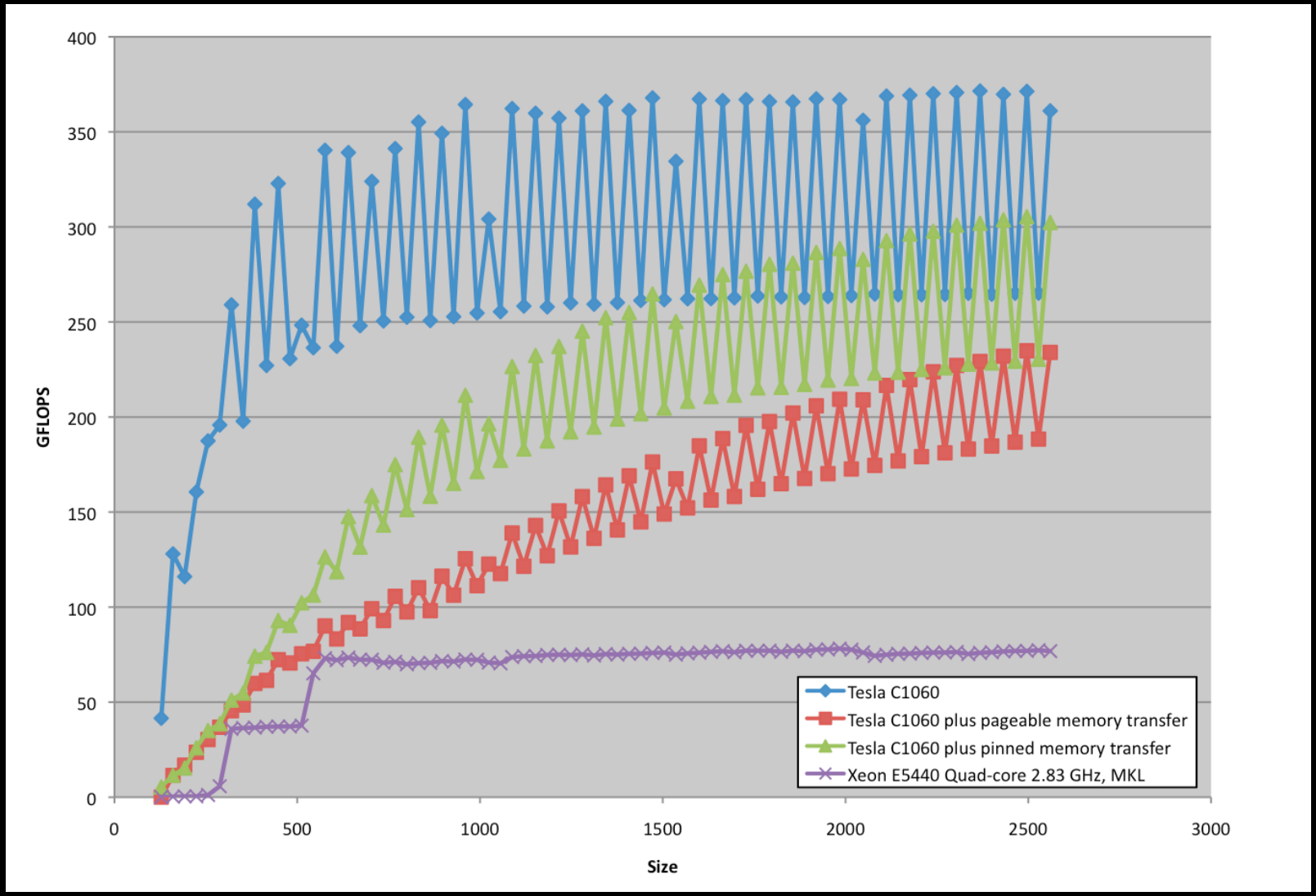
CUBLAS Helper Functions

- 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

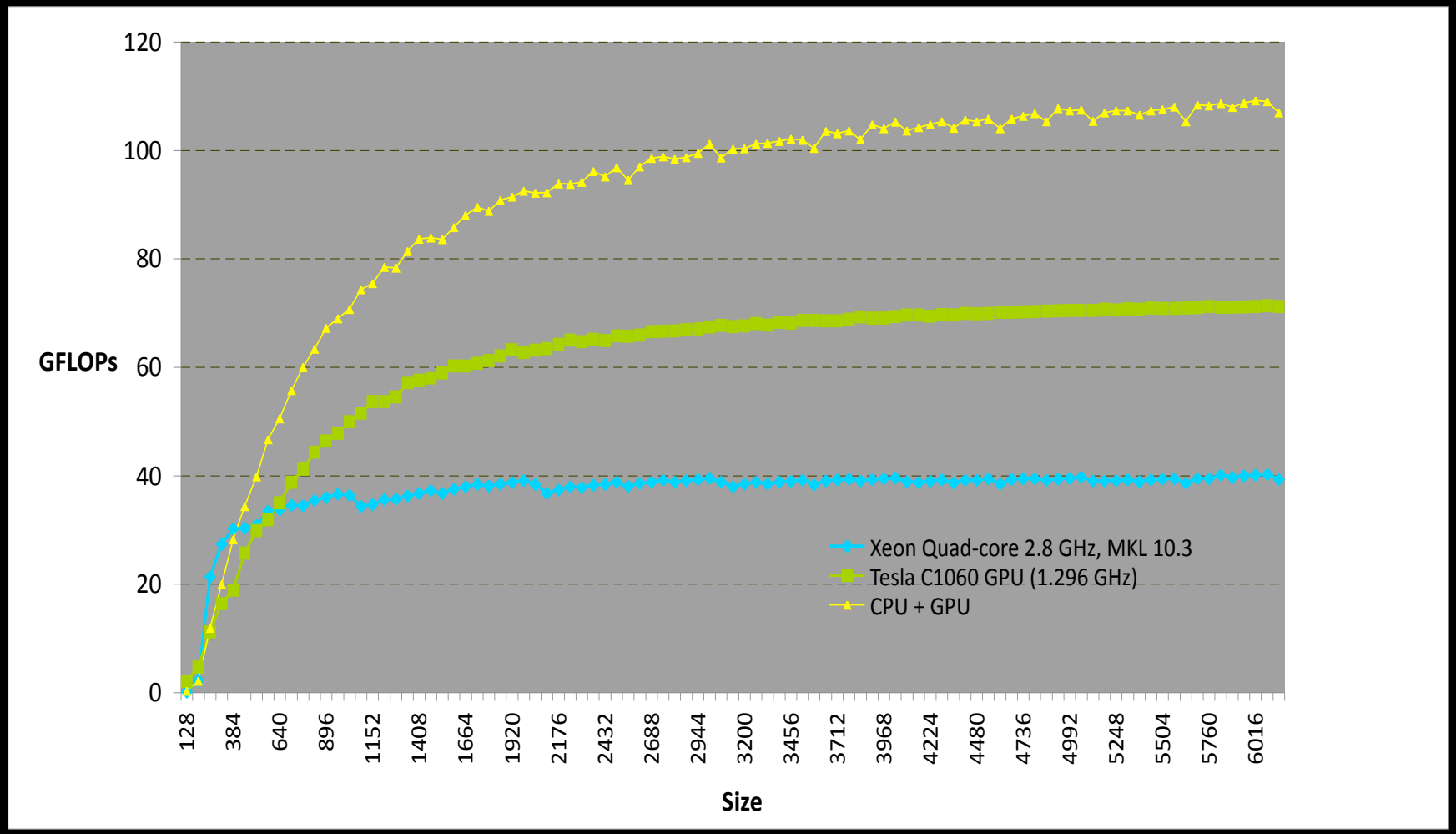
CUBLAS code example


```
float *a, *b, *c;             // host pointers
float *a_d, *b_d, *c_d;       // device pointers

stat = cublasAlloc(n*n, sizeof(float), (void **)&a_d);
assert(stat == CUBLAS_STATUS_SUCCESS);
. . .

stat = cublasSetMatrix(n,n,sizeof(float),(void *)a,n,
                       (void *)a_d,n);
assert(stat == CUBLAS_STATUS_SUCCESS);
. . .

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

stat = cublasGetError();
assert(stat == CUBLAS_STATUS_SUCCESS);

stat = cublasGetMatrix(n,n,sizeof(float),c_d,n,c,n);
assert(stat == CUBLAS_STATUS_SUCCESS);
```

SGEMM Performance



DGEMM Performance



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
  - 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.

Transform Types

- Library supports real and complex transforms
  - CUFFT_C2C, CUFFT_C2R, CUFFT_R2C
- Directions
  - CUFFT_FORWARD (-1) and CUFFT_INVERSE (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

CUFFT Types and Definitions

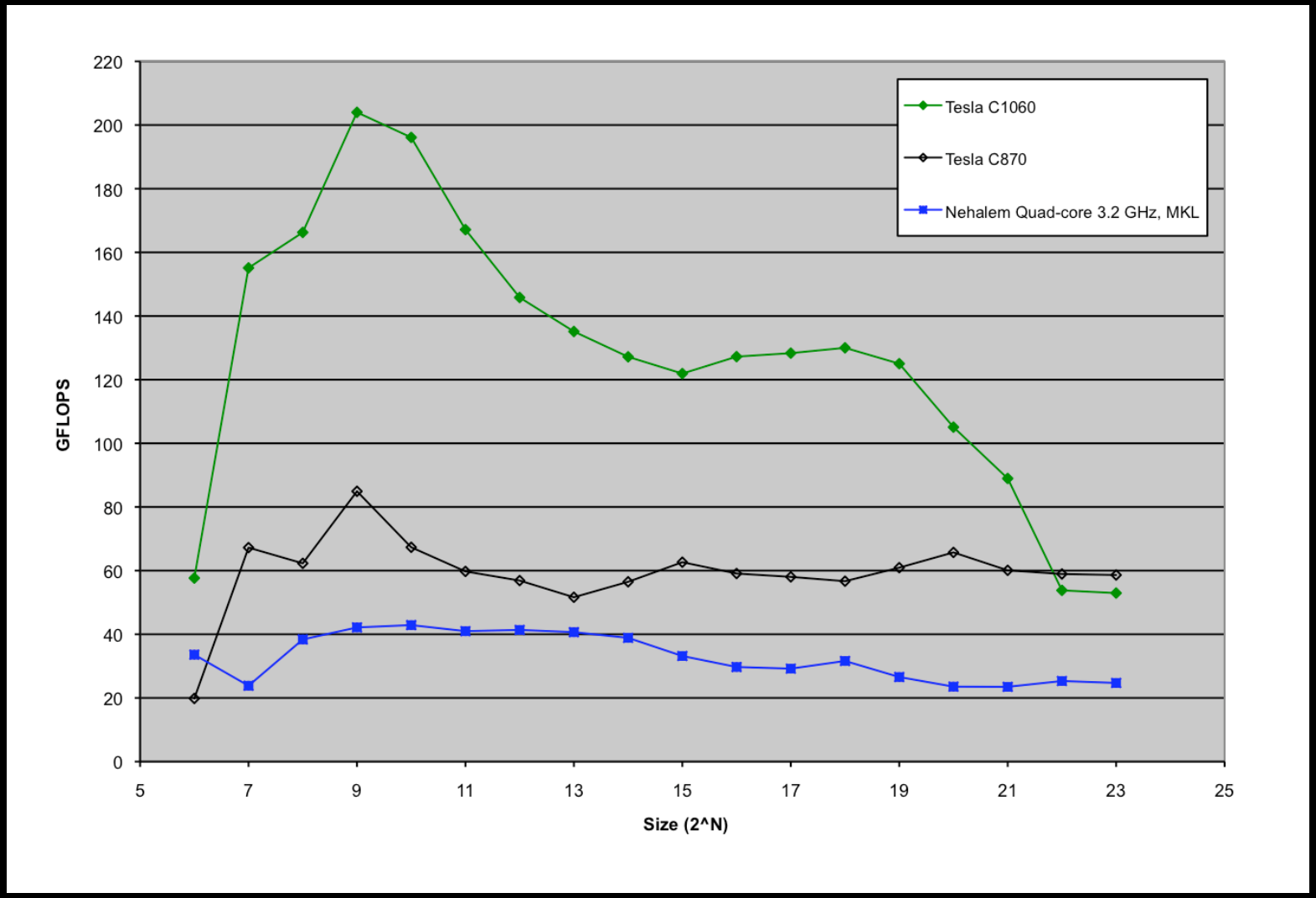
- cufftHandle
  - Type used to store and access CUFFT plans
- cufftResults
  - Enumeration of API function return values
- cufftReal
  - single-precision, real datatype
- cufftComplex
  - single-precision, complex datatype

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

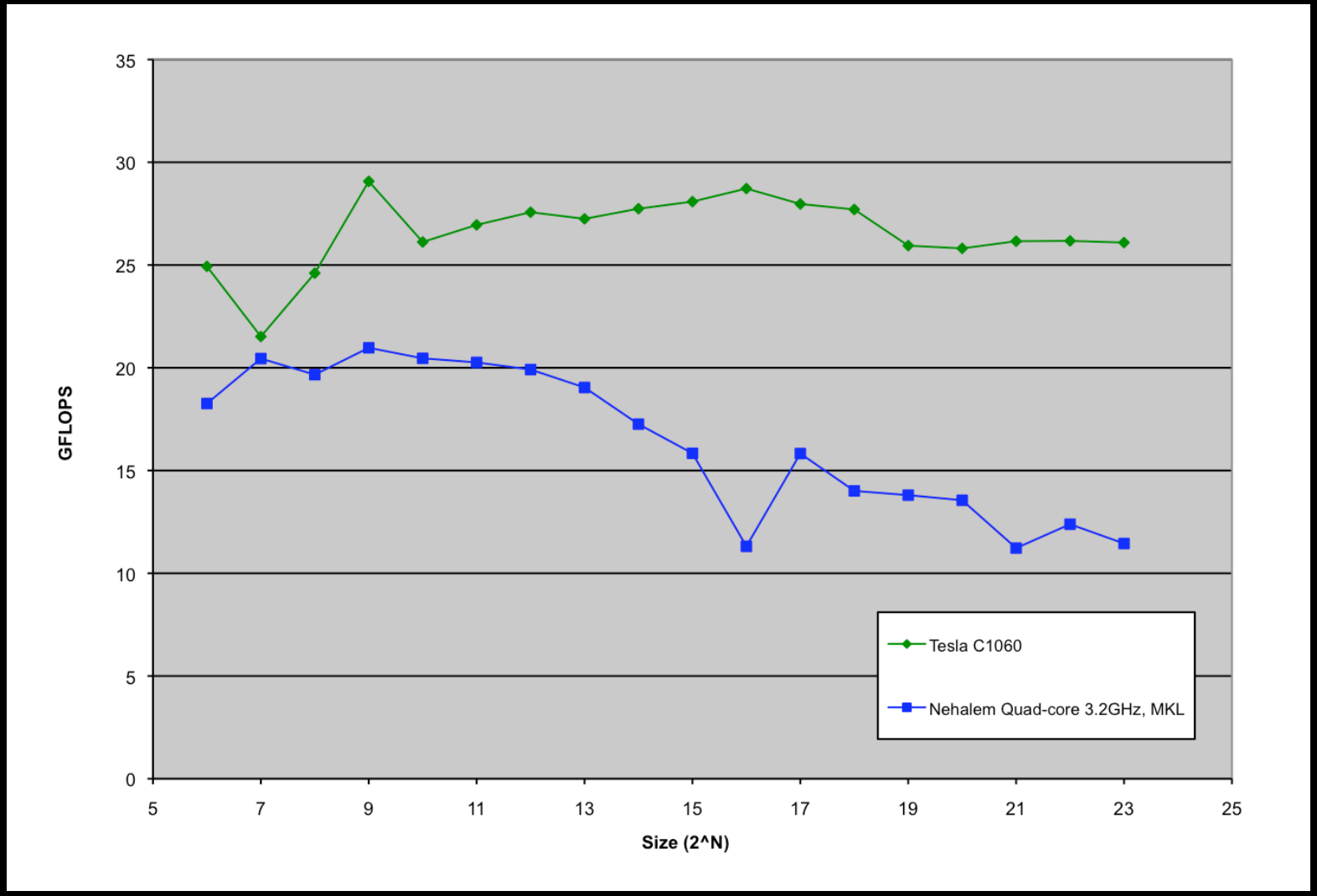
CUFFT code example: 2D complex to complex transform


```
#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 2D 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);

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

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

Single Precision CUFFT Performance



Double Precision CUFFT Performance



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 performs 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.

NVPP

- NVIDIA Performance Primitives
- Focuses on image and video processing
- Currently in Beta

Face Detection

NVIDIA®

Fortran Integration

Fortran Integration

- Fortran-to-C calling conventions are not standardized and differ by platform and toolchain.

- Differences may include:
  - symbol names (capitalization, name mangling)
  - 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)

Fortran-to-C example

Fortran

```
call vectorAdd(a, b, c, n, stat)
```

C

```
__global__void vectorAddKernel(double *a, double *b, double *c,
                                int *n)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) c[i] = a[i] + b[i];
}

extern “C” vectoradd_(double *a, double *b, double *c,
                      int *n, int *stat)
{
  double *a_d, *b_d, *c_d;
  . . .

  vectorAddKernel<<< , >>>(a_d, b_d, c_d, *n);
  . . .

  *stat = cudaGetLastError();
}
```

Calling CUBLAS from FORTRAN

- 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.

CUBLAS 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 GPU memory
  - Intended for light testing due to data transfer and allocation/deallocation 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 GPU memory space (using CUBLAS_ALLOC and CUBLAS_FREE)
    - Copy data between GPU and CPU memory spaces (using CUBLAS_SET_VECTOR, CUBLAS_GET_VECTOR, CUBLAS_SET_MATRIX, and CUBLAS_GET_MATRIX)

SGEMM example (Thunking)


```
! 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 cublasSGEMM('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
```

SGEMM example (Non-Thunking)


```
! Define 3 single precision matrices A, B, C
real , dimension(m1,m1):: A, B, C
integer :: devPtrA, devPtrB, devPtrC, size_of_real=4
. . .

! Initialize A, B, C

! Allocate matrices on GPU
cublasAlloc(m1*m1, size_of_real, devPtrA)
cublasAlloc(m1*m1, size_of_real, devPtrB)
cublasAlloc(m1*m1, size_of_real, devPtrC)

! Copy data from CPU to GPU
cublasSetMatrix(m1, m1, size_of_real, A, m1, devPtrA, m1)
cublasSetMatrix(m1, m1, size_of_real, B, m1, devPtrB, m1)
cublasSetMatrix(m1, m1, size_of_real, C, m1, devPtrC, m1)

! Call SGEMM in CUBLAS library using Non-Thunking interface
! (library is expecting data in GPU memory)
call cublasSGEMM ('n','n',m1,m1,m1,alpha,devPtrA,m1,devPtrB,m1,beta,devPtrC,m1)

! Copy data from GPU to CPU
cublasGetMatrix(m1, m1, size_of_real, devPtrC, m1, C, m1)

! Free memory on device
cublasFree(devPtrA)
. . .
```

Pinned memory for Fortran

- Pinned memory provides a fast PCI-e transfer speed and enables overlapping data transfer and kernel execution
- Allocation needs to be done with `cudaMallocHost`
- Use 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

! Allocate 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
```

http://www.nvidia.com/object/cuda_programming_tools.html

NVIDIA®

Optimizing CUDA

Outline

- Overview
- 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

Use Parallelism Efficiently

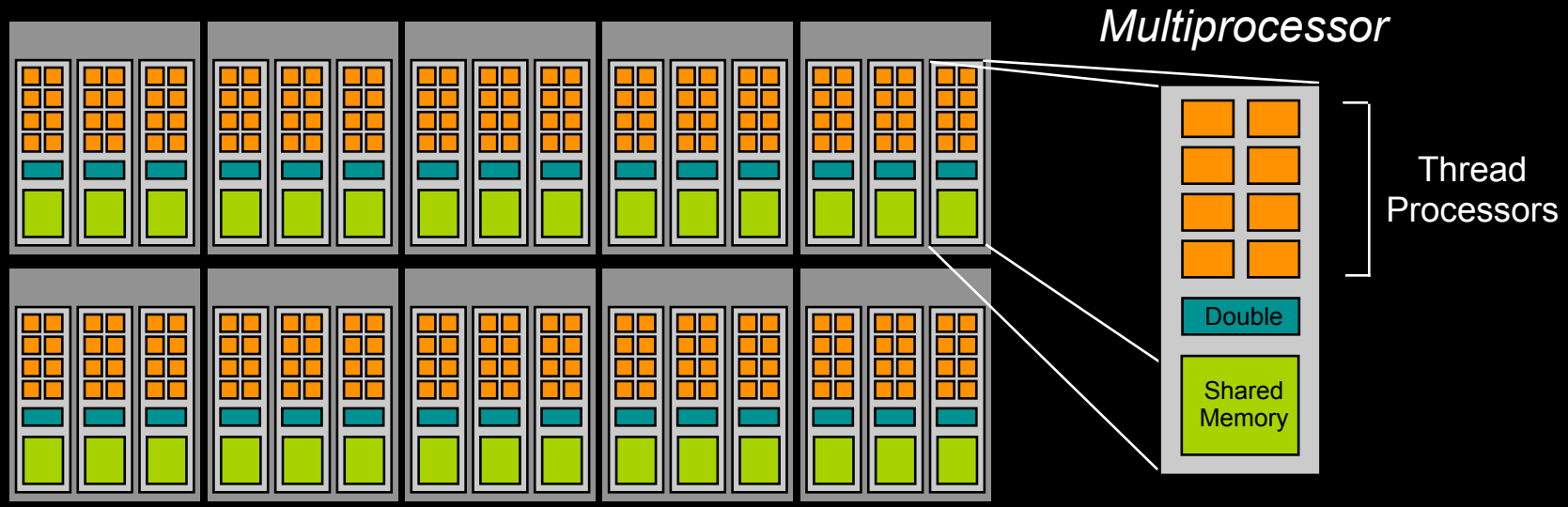
- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

Outline

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

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



Execution Model

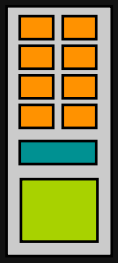
Software **Hardware**

Thread

Thread Processor

Threads are executed by thread processors

Thread Block

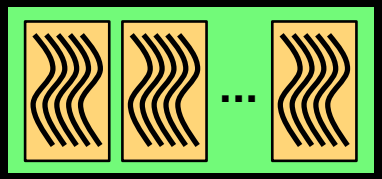


Multiprocessor

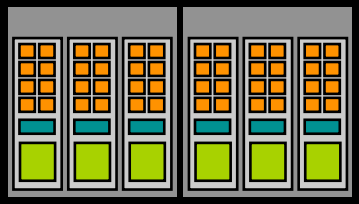
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

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



Grid

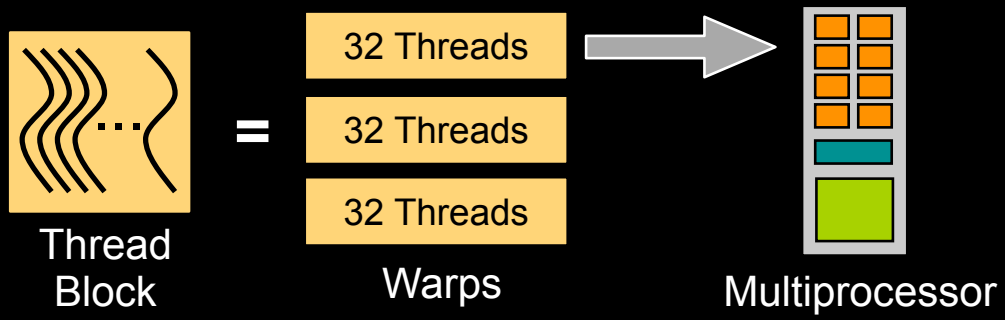


Device

A kernel is launched as a grid of thread blocks

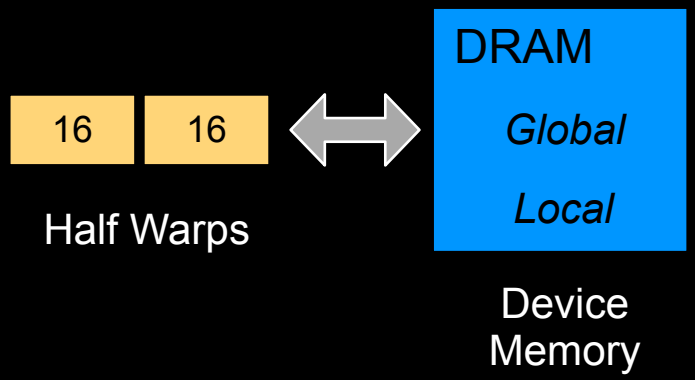
Only one kernel can execute on a device at one time

Warps and Half Warps



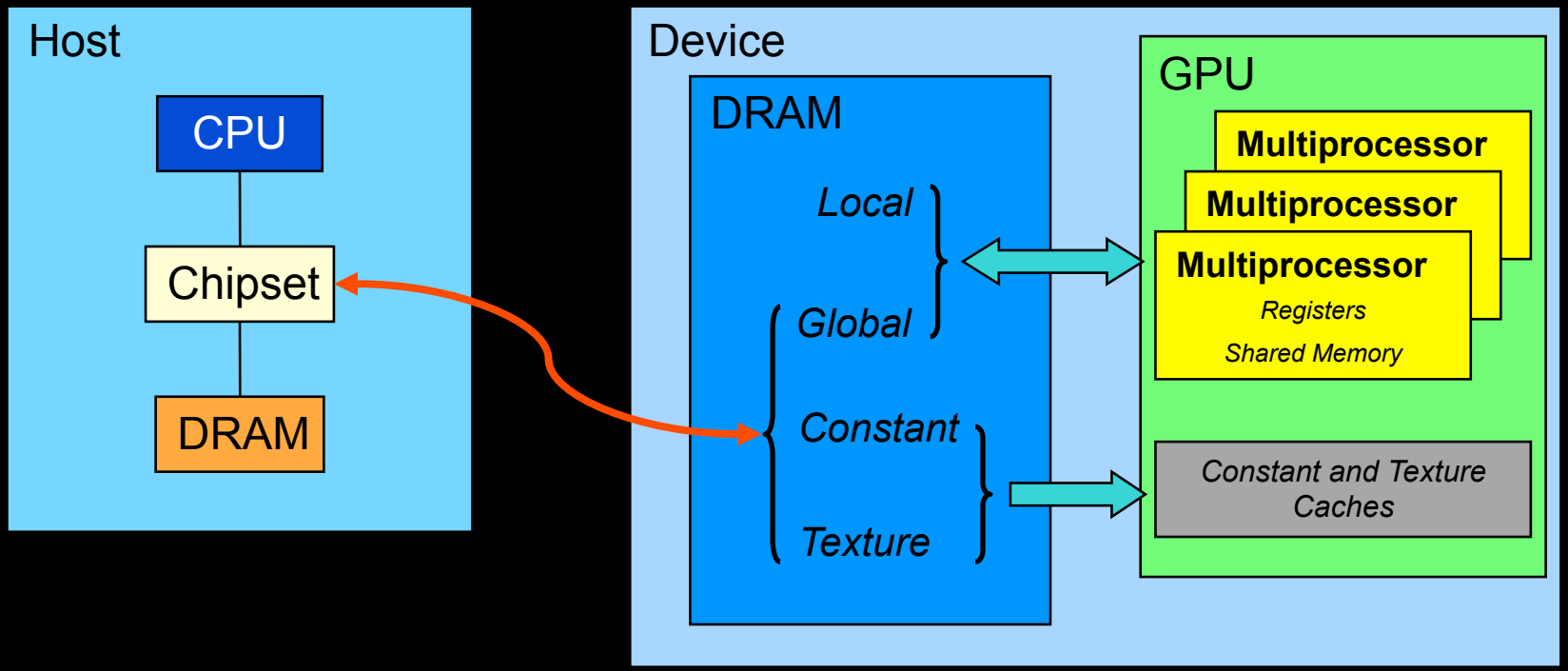
A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor



A half-warp of 16 threads can coordinate global memory accesses into a single transaction

Memory Architecture



Memory Architecture

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

Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Host-Device Data Transfers

- Device to host memory bandwidth much lower than device to device bandwidth
  - 8 GB/s peak (PCI-e x16 Gen 2) vs. 141 GB/s peak (GTX 280)
- Minimize transfers
  - Intermediate data 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

Overlapping Data Transfers and Computation

- Async and Stream APIs allow overlap of H2D or D2H data transfers with computation
  - CPU computation can overlap data transfers on all CUDA capable devices
  - Kernel computation can overlap data transfers on devices with “Concurrent copy and execution” (roughly compute capability >= 1.1)
- Stream = sequence of operations that execute in order on GPU
  - Operations from different streams can be interleaved
  - Stream ID used as argument to async calls and kernel launches

Asynchronous Data Transfers

- Asynchronous host-device memory copy returns control immediately to CPU
  - `cudaMemcpyAsync(dst, src, size, dir, stream);`
  - requires *pinned* host memory (allocated with "`cudaMallocHost`")

- Overlap CPU computation with data transfer
  - `0` = default stream

```
cudaMemcpyAsync(a_d, a_h, size,
                cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

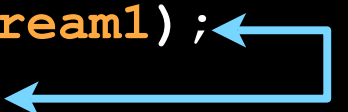
overlapped

Overlapping kernel and data transfer

- **Requires:**
  - "Concurrent copy and execute"
    - `deviceOverlap` field of a `cudaDeviceProp` variable
  - Kernel and transfer use different, *non-zero* streams
    - A CUDA call to stream-0 blocks until all previous calls complete and cannot be overlapped

- **Example:**

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



overlapped

GPU/CPU Synchronization

- Context based
  - `cudaThreadSynchronize()`
    - Blocks until all previously issued CUDA calls from a CPU thread complete
- Stream based
  - `cudaStreamSynchronize(stream)`
    - Blocks until all CUDA calls issued to given stream complete
  - `cudaStreamQuery(stream)`
    - Indicates whether stream is idle
    - Returns `cudaSuccess`, `cudaErrorNotReady, ...`
    - Does not block CPU thread

GPU/CPU Synchronization

- Stream based using events
  - Events can be inserted into streams:

    ```
    cudaEventRecord(event, stream)
    ```

  - Event is recorded then GPU reaches it in a stream
    - Recorded = assigned a timestamp (GPU clocktick)
    - Useful for timing

  - `cudaEventSynchronize(event)`
    - Blocks until given event is recorded

  - `cudaEventQuery(event)`
    - Indicates whether event has recorded
    - Returns `cudaSuccess`, `cudaErrorNotReady, ...`
    - Does not block CPU thread

Zero copy

- Access host memory directly from device code
  - Transfers are implicitly preformed as needed by device code
  - Introduced in CUDA 2.2
  - Check `canMapHostMemory` field of `cudaDeviceProp` variable
- All set-up is done on host using mapped memory

```
cudaSetDeviceFlags(cudaDeviceMapHost);
...
cudaHostAlloc((void **)&a_h, nBytes, cudaHostAllocMapped);
cudaHostGetDevicePointer((void **)&a_d, (void *)a_h, 0);
for (i=0; i<N; i++) a_h[i] = i;
increment<<<grid, block>>>(a_d, N);
```

Zero copy consideration

- Integrated devices (that utilize CPU memory)
  - Zero copy is always a performance win
  - Check `integrated` field in `cudaDeviceProp`
- Discrete devices
  - Data should be read/written from/to global memory only once
    - Data is not cached, each instance results in PCI-e transfer
  - Transactions should be coalesced
  - Potentially easier and faster alternative to using `cudaMemcpyAsync`
    - For example, can both read and write CPU memory from within one kernel
- Note that current devices use pointers that are 32-bit so there is a limit of *4GB per context*

Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance - effective bandwidth
    - Coalescing
    - Shared memory
    - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Theoretical Bandwidth

- Device Bandwidth of GTX 280
  - 1107 * 10^6 * (512 / 8) * 2 / 1024^3 = 131.9 GB/s

DDR ↓

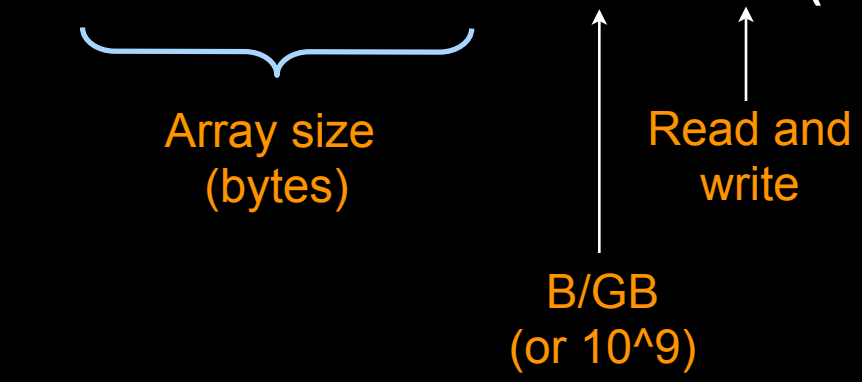
Memory clock (Hz)

Memory interface (bytes)

  - Specs report 141 GB/s
    - Use 10^9 B/GB conversion rather than 1024^3
    - Whichever you use, be consistent

Effective Bandwidth

- Effective Bandwidth (for copying array of N floats)
  - N * 4 B/element / 1024^3 * 2 / (time in secs) = GB/s



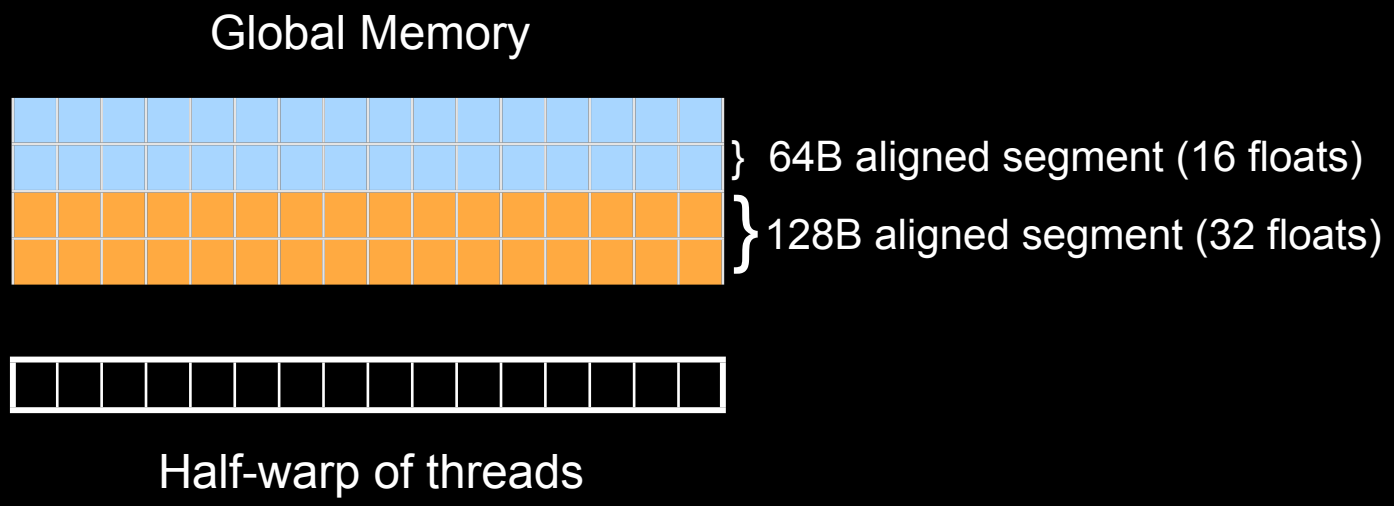
Outline

- Overview
- Hardware
- Memory Optimizations
  - Data transfers between host and device
  - Device memory optimizations
    - Measuring performance - effective bandwidth
    - Coalescing
    - Shared memory
    - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Coalescing

- Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met
- Depends on compute capability
  - 1.0 and 1.1 have stricter access requirements

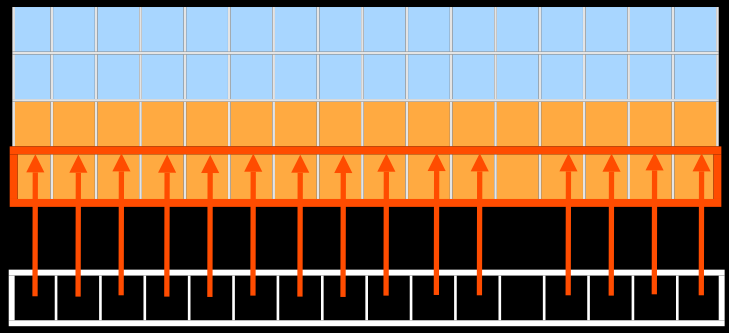
Examples – float (32-bit) data



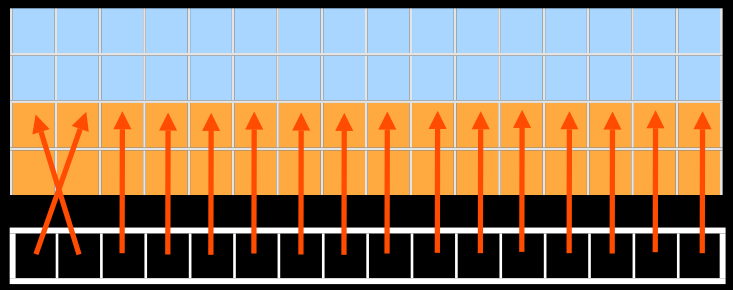
Coalescing
Compute capability 1.0 and 1.1

- **K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate**

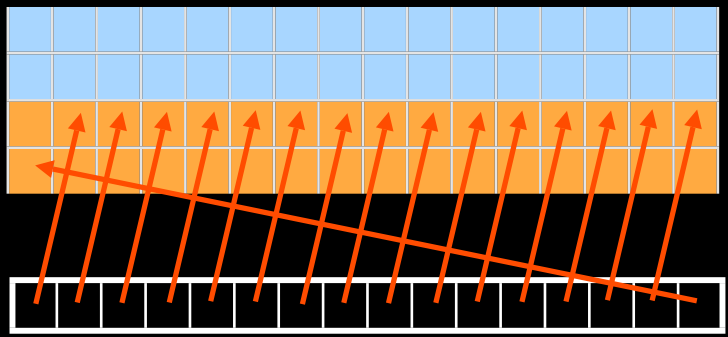
Coalesces – 1 transaction



Out of sequence – 16 transactions

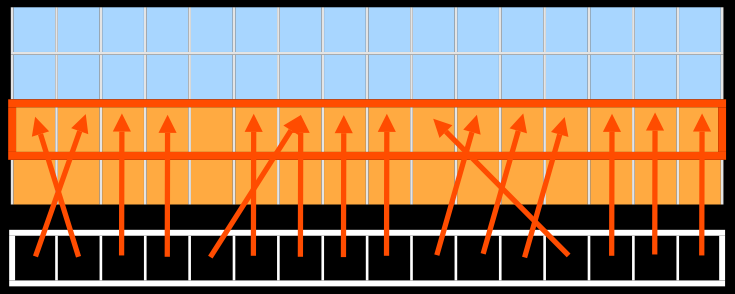


Misaligned – 16 transactions



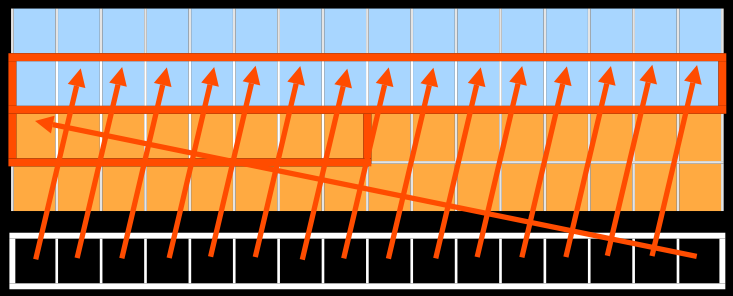
Coalescing
Compute capability 1.2 and higher

- Coalescing is achieved for any pattern of addresses that fits into a segment of size: 32B for 8-bit words, 64B for 16-bit words, 128B for 32- and 64-bit words
- Smaller transactions may be issued to avoid wasted bandwidth due to unused words

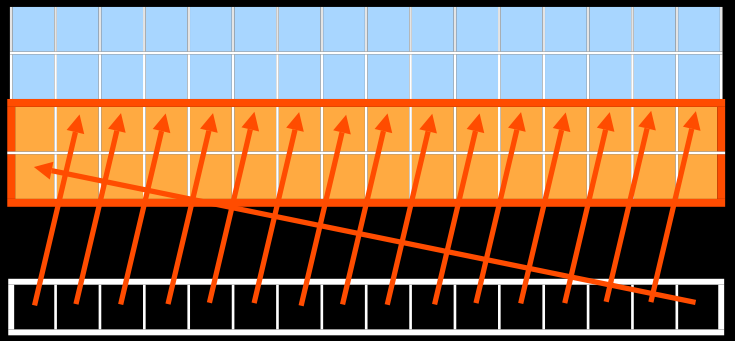


1 transaction - 64B segment

2 transactions - 64B and 32B segments



1 transaction - 128B segment



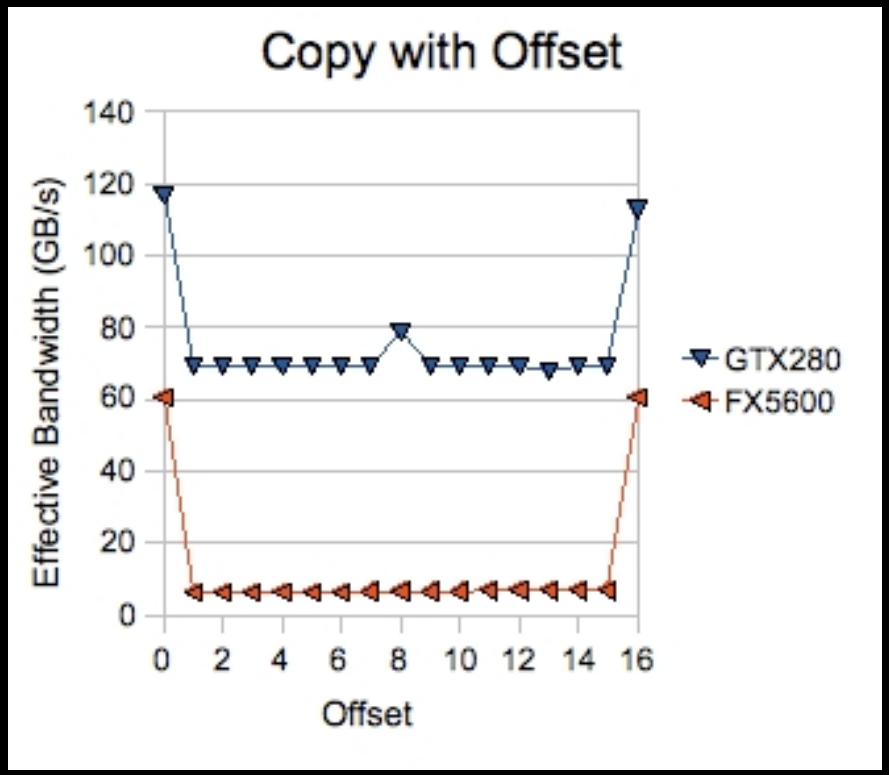
Coalescing Examples

- Effective bandwidth of small kernels that copy data
  - Effects of offset and stride on performance

- Two GPUs
  - GTX 280
    - Compute capability 1.3
    - Peak bandwidth of 141 GB/s
  - FX 5600
    - Compute capability 1.0
    - Peak bandwidth of 77 GB/s

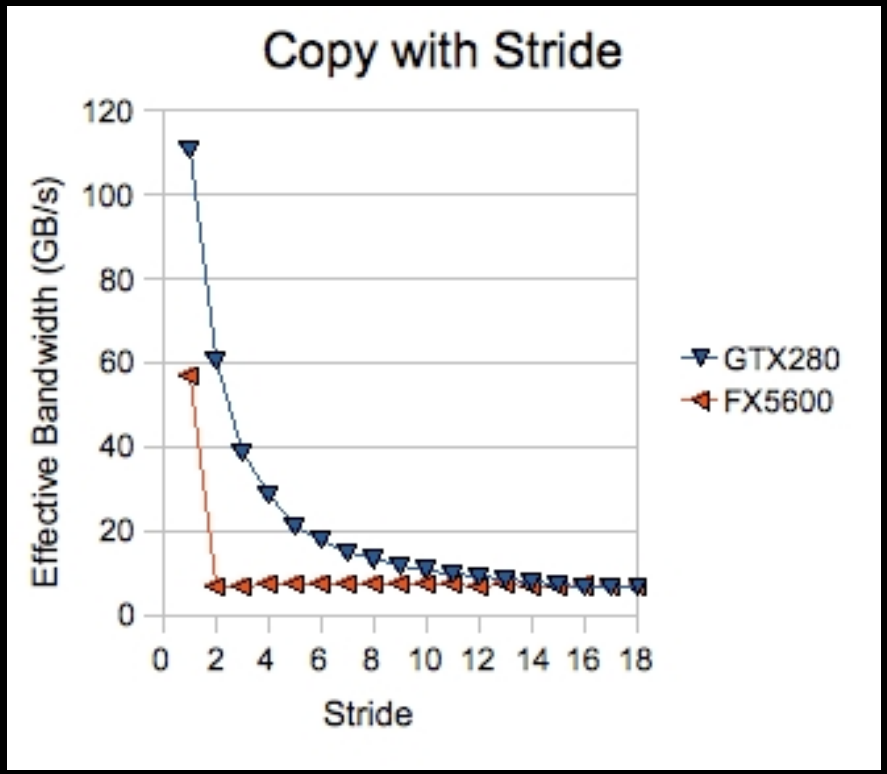
Coalescing Examples


```
__global__ void offsetCopy(float *odata, float* idata,
                           int offset)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
  odata[xid] = idata[xid];
}
```



Coalescing Examples


```
__global__ void strideCopy(float *odata, float* idata,
                           int stride)
{
  int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
  odata[xid] = idata[xid];
}
```

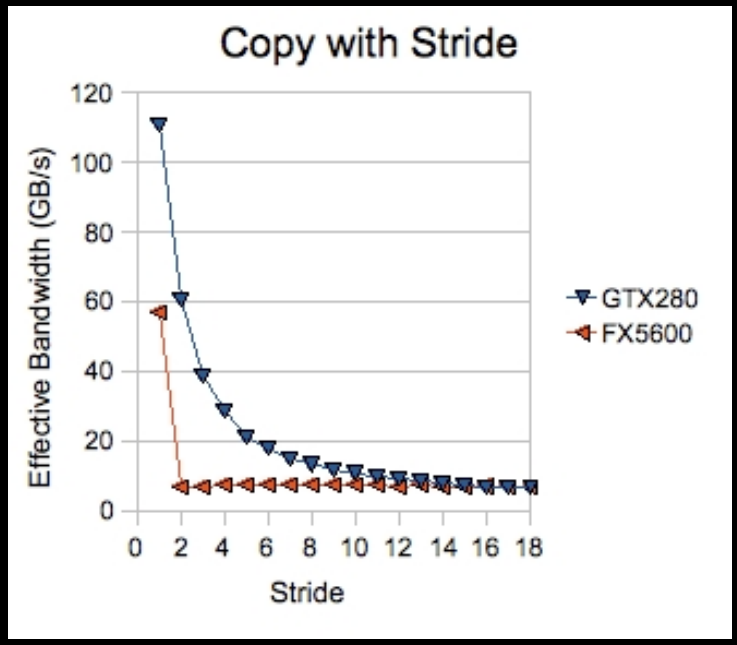


Coalescing Examples

- Strided memory access is inherent in many multidimensional problems
  - Stride is generally large (>>18)

  However ...

- Strided access to *global memory* can be avoided using shared memory



Outline

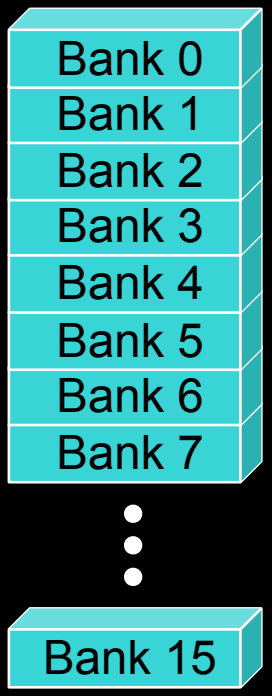
- Overview
- Hardware
- Memory Optimizations
  - Data Transfers between host and device
  - Device memory optimizations
    - Measuring performance - effective bandwidth
    - Coalescing
    - Shared memory
    - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Shared Memory

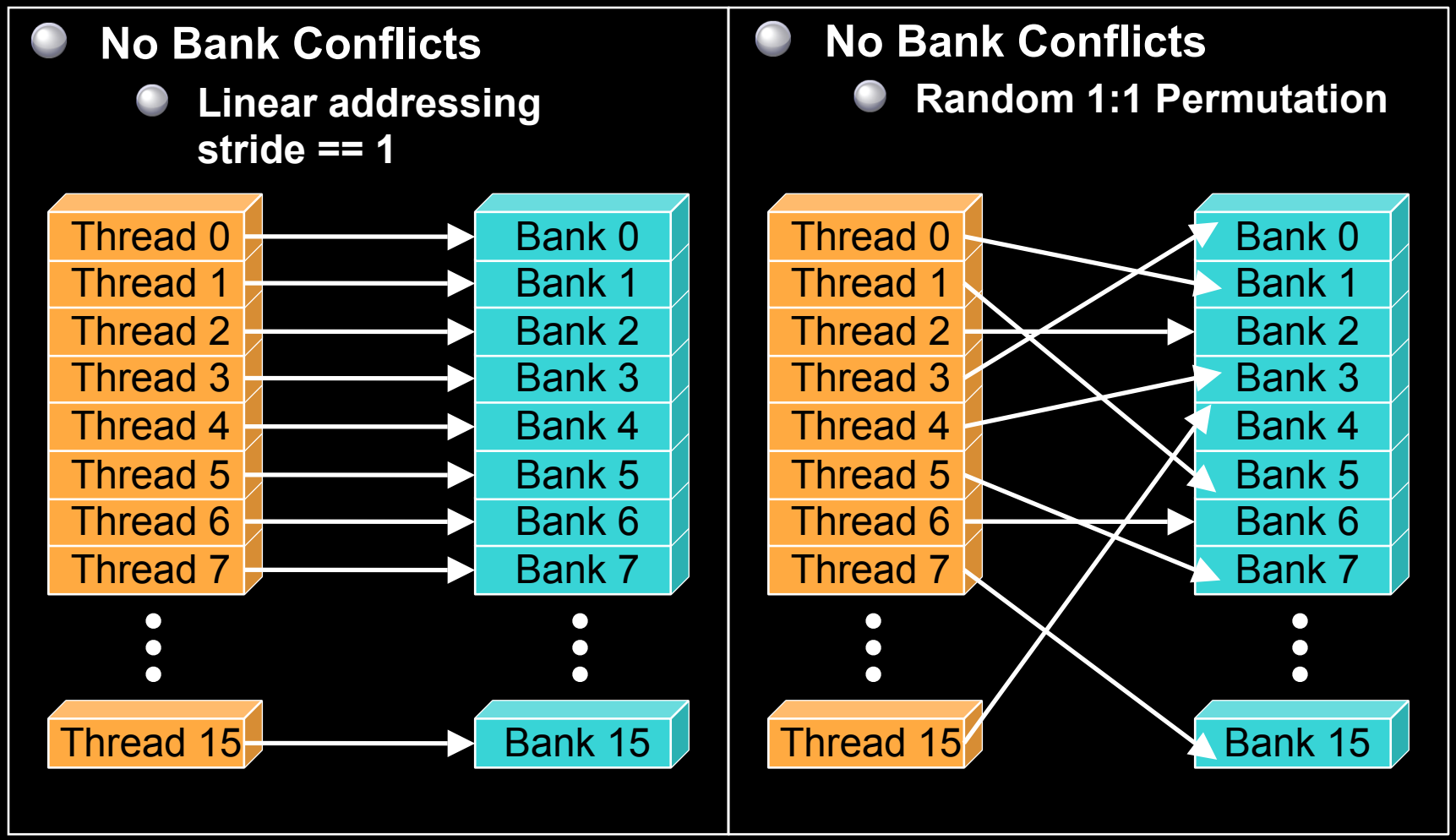
- ~Hundred times faster than global memory
- Cache data to reduce global memory accesses
- Threads can cooperate via shared memory
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

Shared Memory Architecture

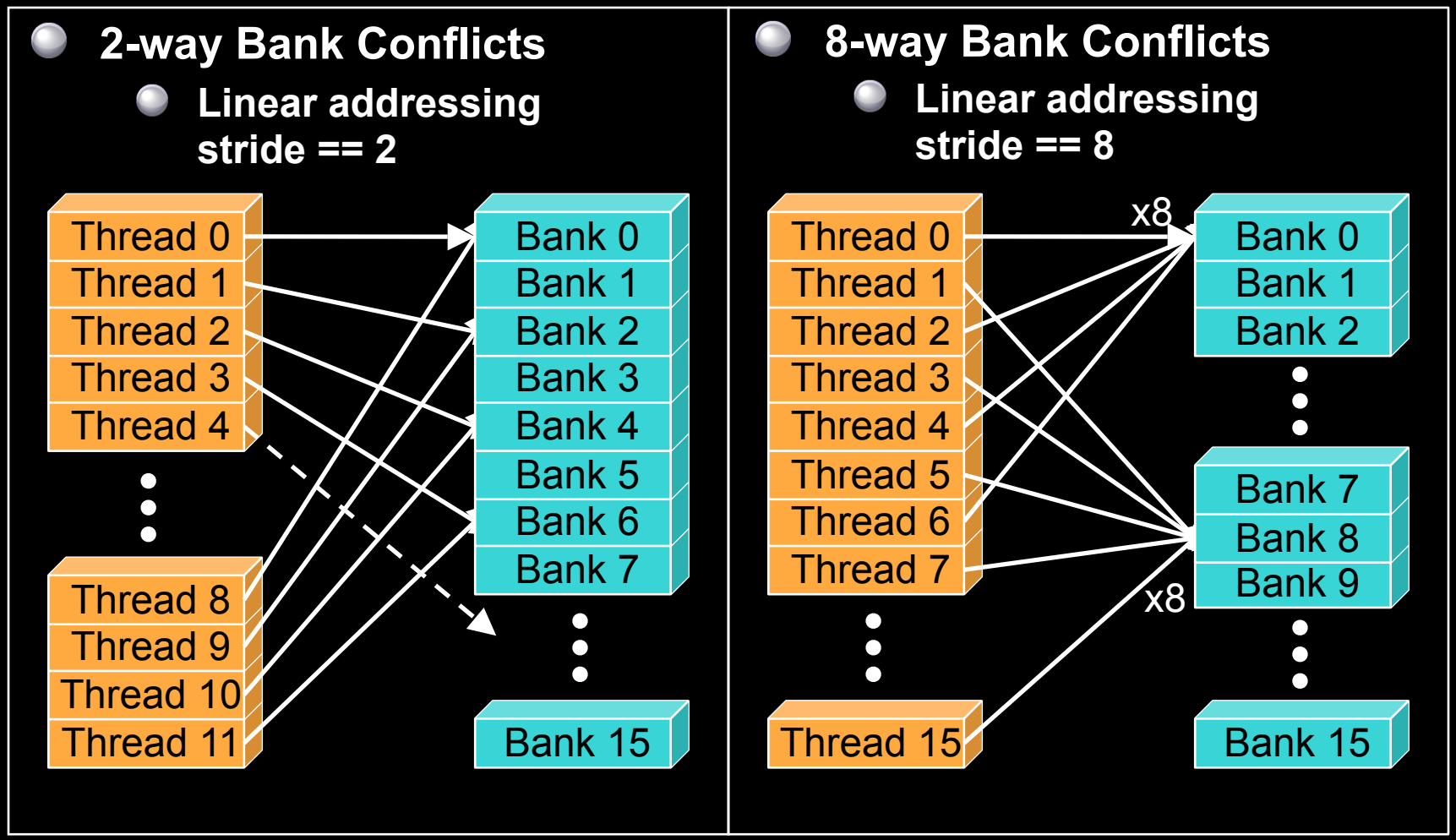
- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Successive 32-bit words assigned to successive banks
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



Bank Addressing Examples



Bank Addressing Examples

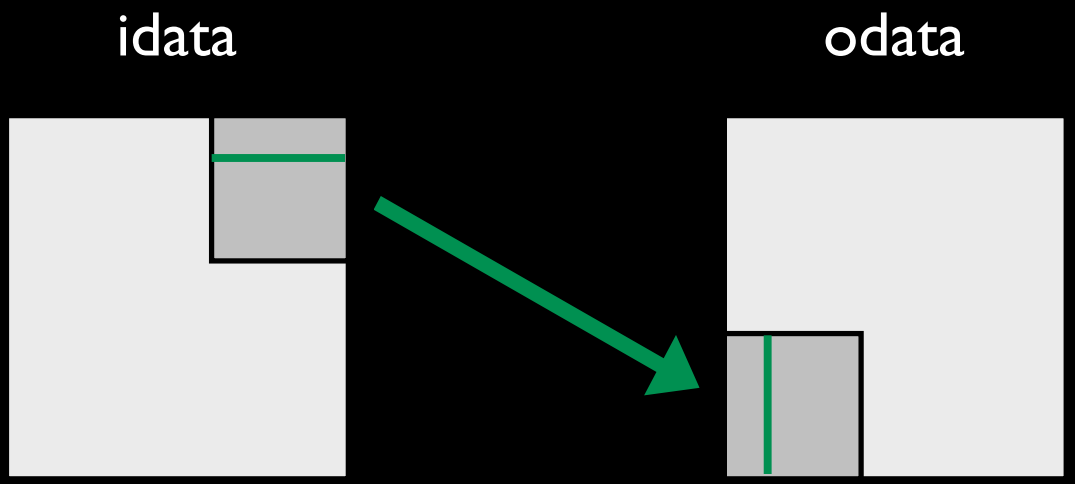


Shared memory bank conflicts

- Shared memory is ~ as fast as registers if there are no bank conflicts
- warp_serialize profiler signal reflects 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

Shared Memory Example: Transpose

- Each thread block works on a tile of the matrix
- Naïve implementation exhibits strided access to global memory

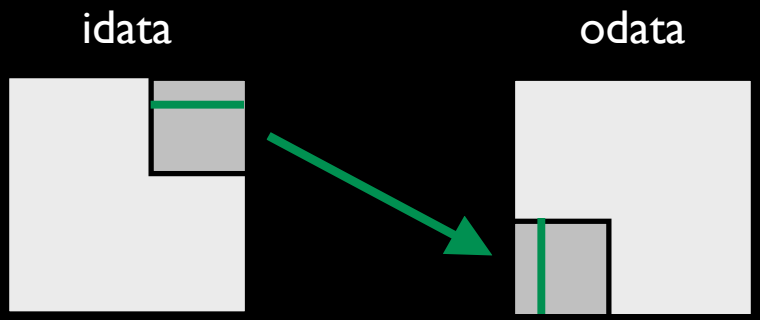


Elements transposed by a half-warp of threads

Naïve Transpose

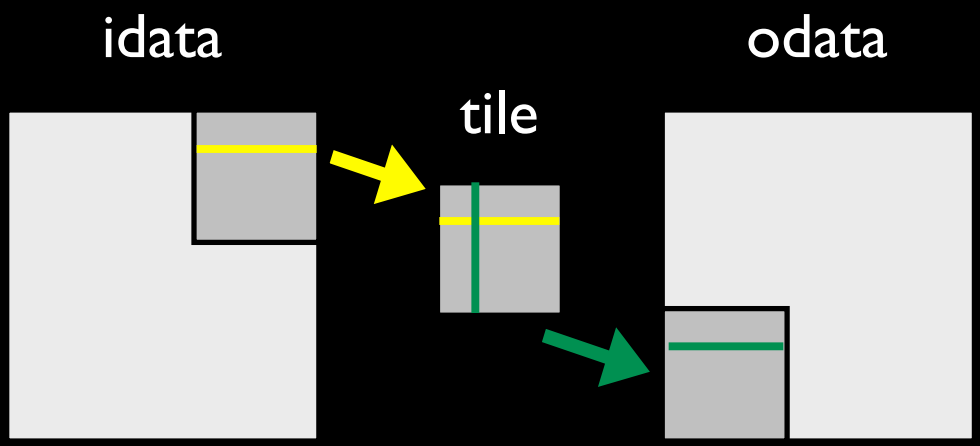
- Loads are coalesced, stores are not (strided by height)

```
__global__ void transposeNaive(float *odata, float* idata,
                               int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;

  odata[index_out] = idata[index_in];
}
```



Coalescing through shared memory

- Access columns of a tile in shared memory to write contiguous data to global memory
- Requires `__syncthreads()` since threads access data in shared memory stored by other threads



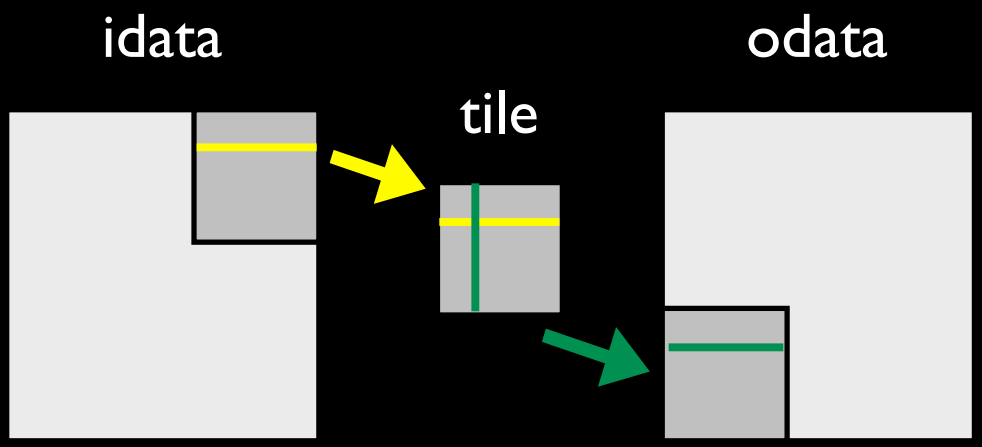
Elements transposed by a half-warp of threads

Coalescing through shared memory


```
__global__ void transposeCoalesced(float *odata, float *idata,
                                   int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  tile[threadIdx.y][threadIdx.x] = idata[index_in];

  __syncthreads();

  odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

Bank Conflicts in Transpose

- 16x16 shared memory tile of floats
  - Data in columns are in the same bank
  - 16-way bank conflict reading columns in tile
- Solution - pad shared memory array
  - `__shared__ float tile[TILE_DIM][TILE_DIM+1];`
  - Data in anti-diagonals are in same bank



Elements transposed by a half-warp of threads

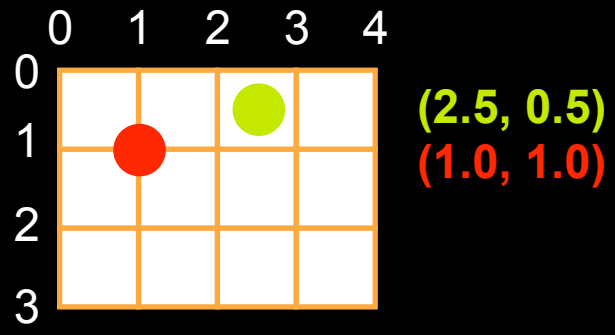
Outline

- Overview
- Hardware
- Memory Optimizations
  - Data Transfers between host and device
  - Device memory optimizations
    - Measuring performance - effective bandwidth
    - Coalescing
    - Shared memory
    - Textures
- Execution Configuration Optimizations
- Instruction Optimizations
- Summary

Textures in CUDA

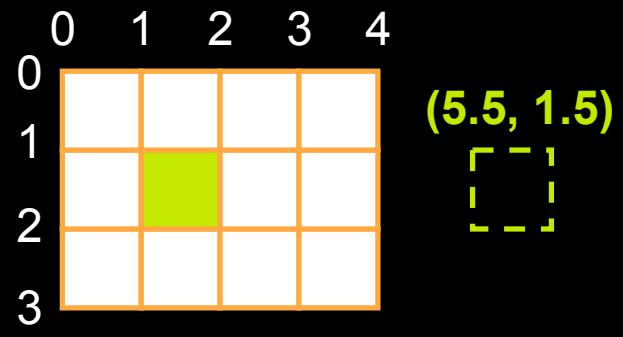
- Texture is an object for *reading* data
- Benefits:
  - Data is cached (optimized for 2D locality)
    - Helpful when coalescing is a problem
  - Filtering
    - Linear / bilinear / trilinear
    - Dedicated hardware
  - Wrap modes (for “out-of-bounds” addresses)
    - Clamp to edge / repeat
  - Addressable 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



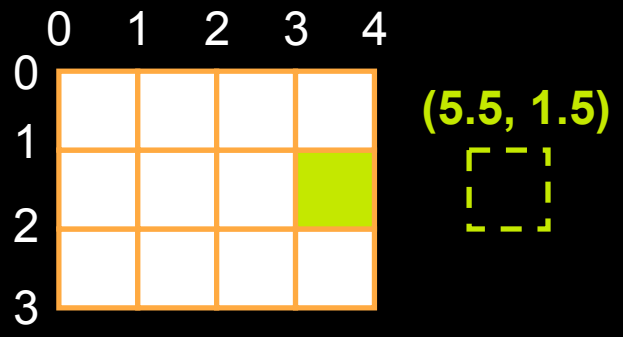
Wrap

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



Clamp

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



CUDA Texture Types

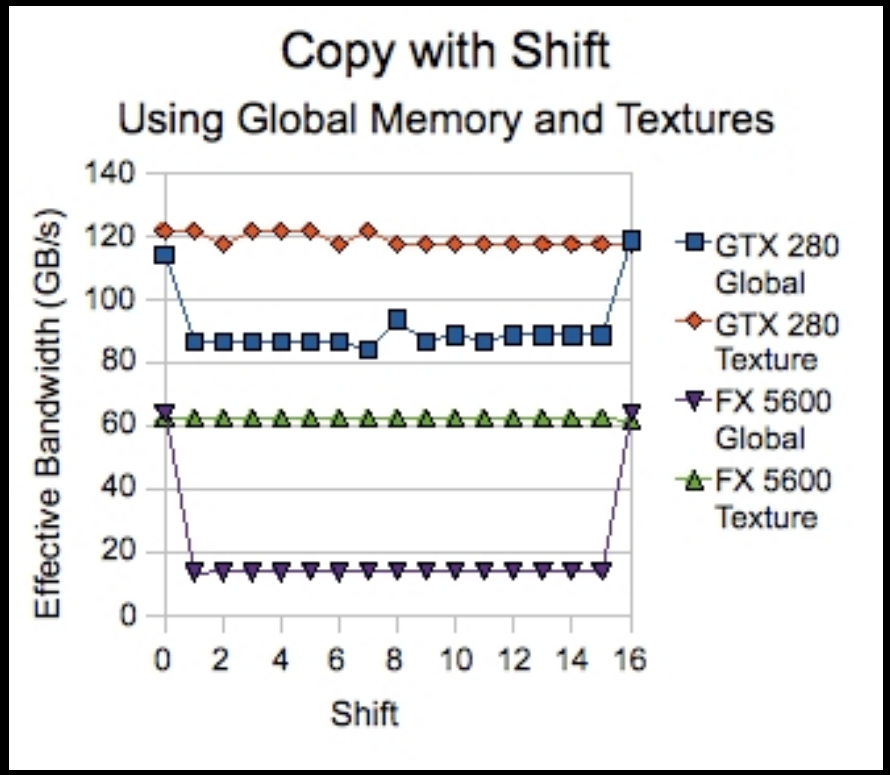
- **Bound to linear**
  - Global memory address is bound to a texture
  - Only 1D
  - Integer addressing
  - No filtering or addressing modes
- **Bound to CUDA arrays**
  - CUDA array is bound to a texture
  - 1D, 2D, or 3D
  - Float addressing (size-based or normalized)
  - Filtering
  - Addressing modes (clamping, repeat)
- **Bound to pitch linear (CUDA 2.2)**
  - Global memory address is bound to a texture
  - 2D
  - Float/integer addressing, filtering, and clamp/repeat addressing modes similar to CUDA arrays

CUDA Texturing Steps

- Host (CPU) code:
  - Allocate/obtain memory (global linear/pitch 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()
  - Pitch linear: tex2D()

Texture Example

```
__global__ void
shiftCopy(float *odata,
          float *idata,
          int shift)
{
  int xid = blockIdx.x * blockDim.x
          + threadIdx.x;
  odata[xid] = idata[xid+shift];
}
```




```
texture <float> texRef;

__global__ void
textureShiftCopy(float *odata,
                 float *idata,
                 int shift)
{
  int xid = blockIdx.x * blockDim.x
          + threadIdx.x;
  odata[xid] = tex1Dfetch(texRef, xid+shift);
}
```

Outline

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

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

Blocks per Grid 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 ~24 cycles later
  - Scenarios:

CUDA:

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

PTX:

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

CUDA:

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

PTX:

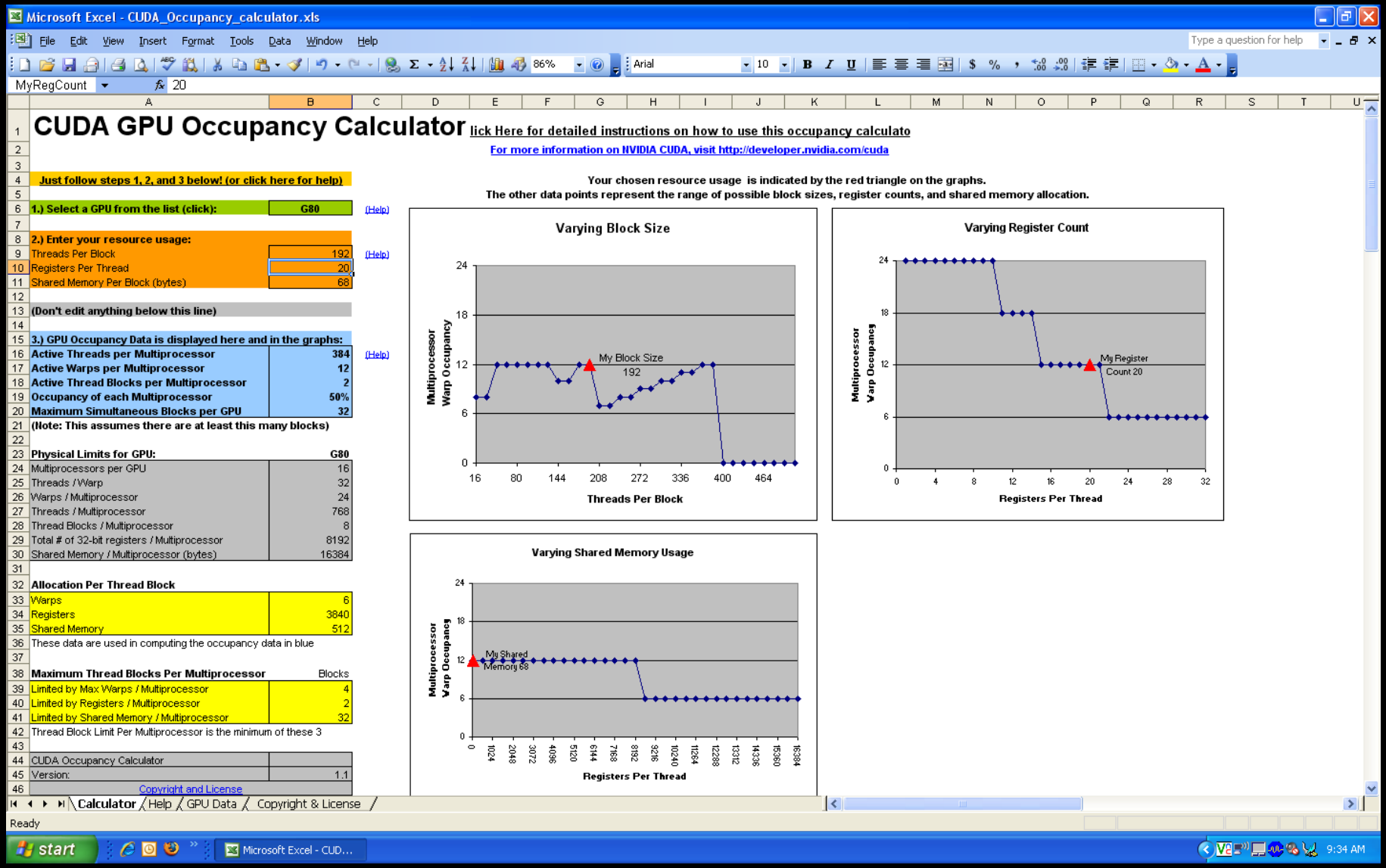
```
ld.shared.f32  $f3, [$r31+0]
add.f32        $f3, $f3, $f4
```

- **To completely hide the latency:**
  - Run at least **192** threads (6 warps) per multiprocessor
    - At least **25%** occupancy (1.0/1.1), **18.75%** (1.2/1.3)
  - Threads do not have to belong to the same thread block

Register Pressure

- Hide latency by using more threads per multiprocessor
- Limiting Factors:
  - Number of registers per kernel
    - 8K/16K per multiprocessor, partitioned among concurrent threads
  - Amount of shared memory
    - 16KB per multiprocessor, 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 local memory may occur
    - Reduces performance – local memory is slow

Occupancy Calculator



Optimizing threads per block

- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
  - Facilitates coalescing
- More threads per block != higher occupancy
  - Granularity of allocation
  - Eg. compute capability 1.1 (max 768 threads/multiprocessor)
    - 512 threads/block => 66% occupancy
    - 256 threads/block can have 100% occupancy
- 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

Outline

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

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
  - 1.3 GHz on the Tesla C1060, 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

Runtime Math Library

- There are two types of runtime math operations in single precision
  - `__funcf()`: direct mapping to hardware ISA
    - Fast but lower accuracy (see prog. guide for details)
    - Examples: `__sinf(x)`, `__expf(x)`, `__powf(x,y)`
  - `funcf()` : compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: `sinf(x)`, `expf(x)`, `powf(x,y)`
- The `-use_fast_math` compiler option forces every `funcf()` to compile to `__funcf()`

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

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

NVIDIA®

Driver API

Driver API

- Up to this point the host code we've seen has been from the *runtime API*
  - Runtime API: `cuda*()` functions
  - Driver API: `cu*()` functions
- Advantages:
  - No dependency on runtime library
  - More control over devices
    - One CPU thread can control multiple GPUs
  - No C extensions in host code, so you can use something other than the default host CPU compiler (e.g. icc, etc.)
  - PTX Just-In-Time (JIT) compilation
    - Parallel Thread eXecution (PTX) is our virtual ISA (more on this later)
- Disadvantages:
  - No device emulation
  - More verbose code
- *Device code is identical whether you use the runtime or driver API*

Initialization and Device Management

- Must initialize with `cuInit()` before any other call
- Device management:

```
int deviceCount;
cuDeviceGetCount(&deviceCount);
int device;
for (int device = 0; device < deviceCount; ++device)
{
  CUdevice cuDevice;
  cuDeviceGet(&cuDevice, device);
  int major, minor;
  cuDeviceComputeCapability(&major, &minor, cuDevice);
}
```

Context Management

- CUDA context analogous to CPU process
  - Each context has its own address space
- Context created with `cuCtxCreate()`
- A host CPU thread can only have one context current at a time
- Each host CPU thread can have a stack of current contexts
- `cuCtxPopCurrent()` and `cuCtxPushCurrent()` can be used to detach and push a context to a new thread
- `cuCtxAttach()` and `cuCtxDetach()` increment and decrement the usage count and allow for interoperability of code in the same context (e.g. libraries)

Module Management

- Modules are dynamically loadable pieces of device code, analogous to DLLs or shared libraries
- For example to load a module and get a handle to a kernel:

```
CUmodule cuModule;
cuModuleLoad(&cuModule, “myModule.cubin”);
CUfunction cuFunction;
cuModuleGetFunction(&cuFunction, cuModule, “myKernel”);
```

- A module may contain binaries (a .cubin file) or PTX code that will be Just-In-Time (JIT) compiled

Execution control


```
cuFuncSetBlockShape(cuFunction, blockWidth, blockHeight, 1);
int offset = 0;
int i;
cuParamSeti(cuFunction, offset, i);
offset += sizeof(i);
float f;
cuParamSetf(cuFunction, offset, f);
offset += sizeof(f);
char data[32];
cuParamSetv(cuFunction, offset, (void*)data, sizeof(data));
offset += sizeof(data);
cuParamSetSize(cuFunction, offset);
cuFuncSetSharedSize(cuFunction, numElements * sizeof(float));
cuLaunchGrid(cuFunction, gridWidth, gridHeight);
```

Memory management

- Linear memory is managed using `cuMemAlloc()` and `cuMemFree()`

```
CUdeviceptr devPtr;
cuMemAlloc(&devPtr, 256 * sizeof(float));
```

- Synchronous copies between host and device

```
cuMemcpyHtoD(devPtr, hostPtr, bytes);
cuMemcpyDtoH(hostPtr, devPtr, bytes);
```

Summary of Runtime and Driver API

- The runtime API is probably the best place to start for virtually all developers
- Easy to migrate to driver API if/when it is needed
- Anything which can be done in the runtime API can also be done in the driver API, but not vice versa (e.g. migrate a context)
- Much, much more information on both APIs in the *CUDA Reference Manual*

NVIDIA®

OpenCL

OpenCL Objectives

- Open, royalty-free standard for heterogeneous parallel computing
  - Applicable to CPUs, GPUs, Cell, etc.
- Cross-vendor software portability to a wide range of hardware
- Support for a wide diversity of applications
  - From embedded and mobile software through consumer applications to HPC solutions

Timeline

- Six months from proposal to released specification
- Apple's Mac OS X Snow Leopard will include OpenCL
- Multiple OpenCL implementations expected in the next 12 months

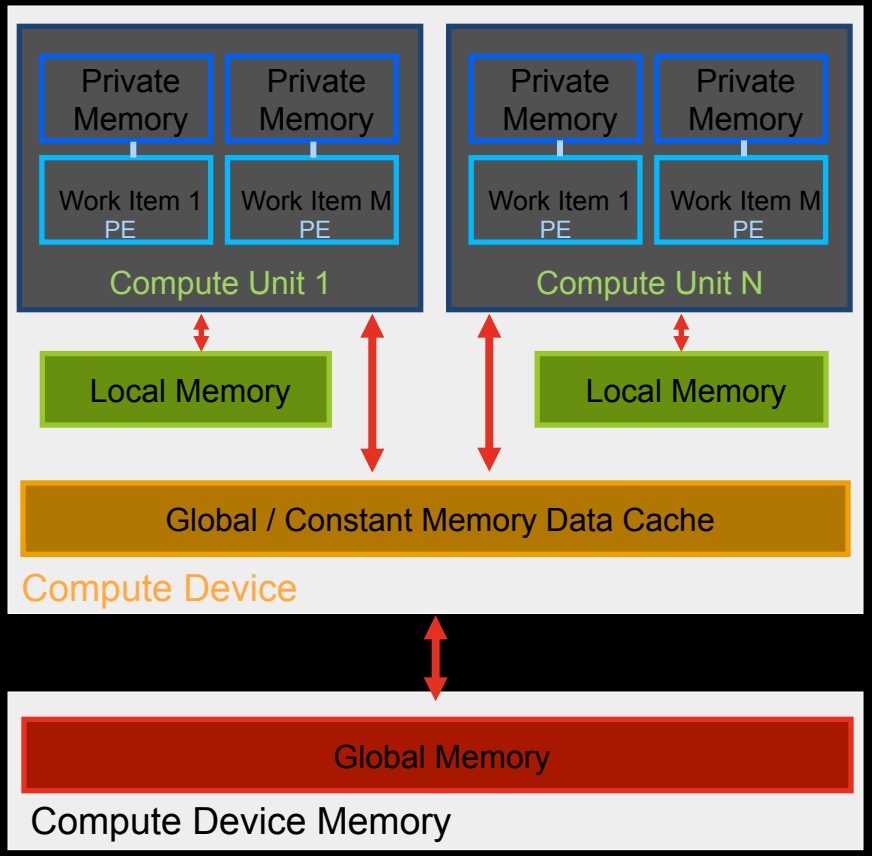
Design Requirements

- Use all computational resources in system
- Efficient C-based parallel programming model
  - Abstract the specifics of underlying hardware
- Abstraction is low-level and high-performance but device-portable
  - Approachable – but primarily targeted at expert developers
  - Ecosystem foundation – no middleware or "convenience" functions
- Drive future hardware requirements
  - Floating point precision requirements
  - Applicable to both consumer and HPC applications

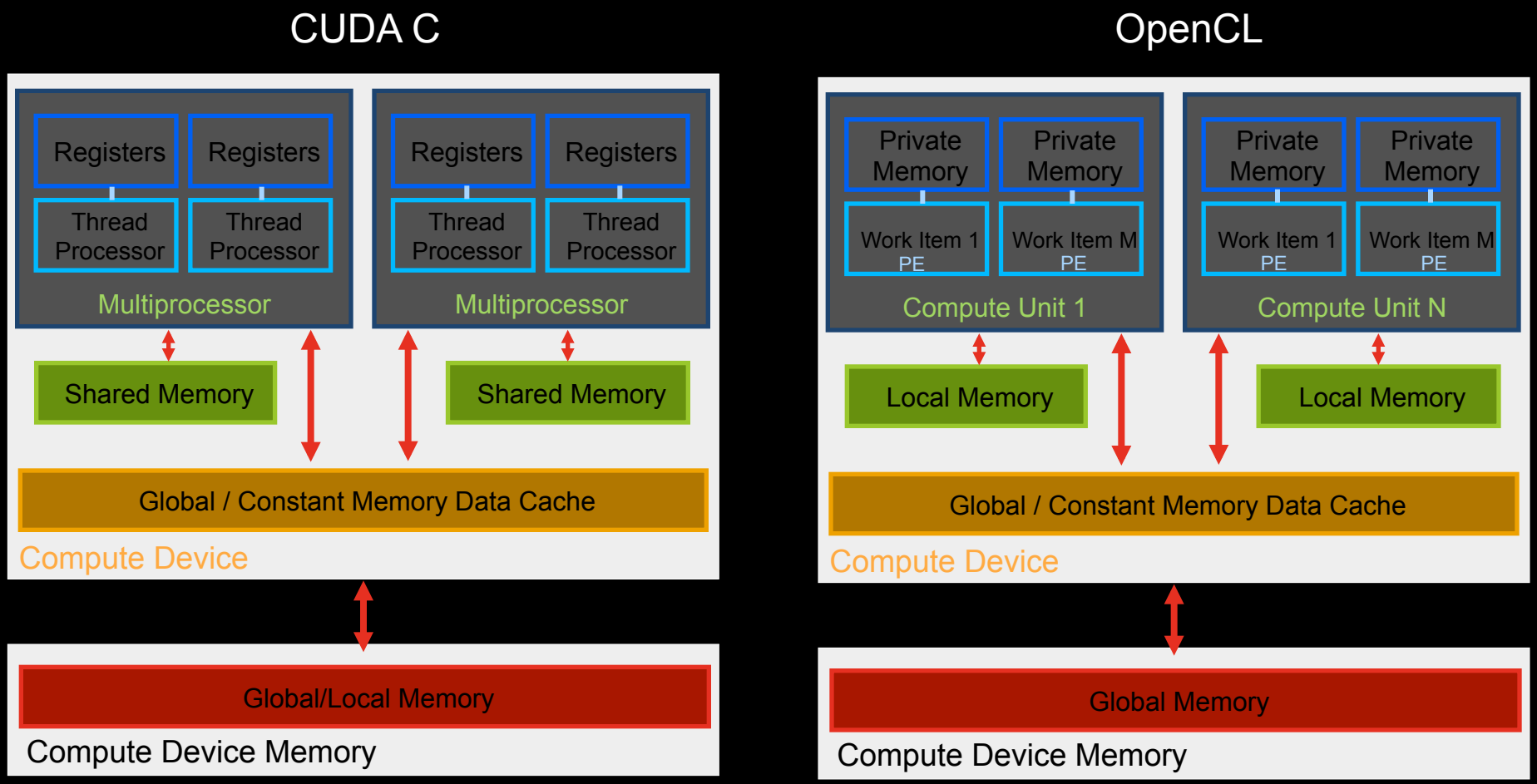
Anatomy

- Language Specification
  - C-based cross-platform programming interface
  - Subset of ISO C99 with language extensions
  - Well-defined numerical accuracy
  - JIT/Online or offline compilation and build of compute kernel executables
  - Includes a rich set of built-in functions
- Platform Layer API
  - A hardware abstraction layer over diverse computational resources
  - Query, select and initialize *compute devices*
  - Create *compute contexts* and *work-queues*
- Runtime API
  - Execute *compute kernels*
  - Manage scheduling, compute, and memory resources

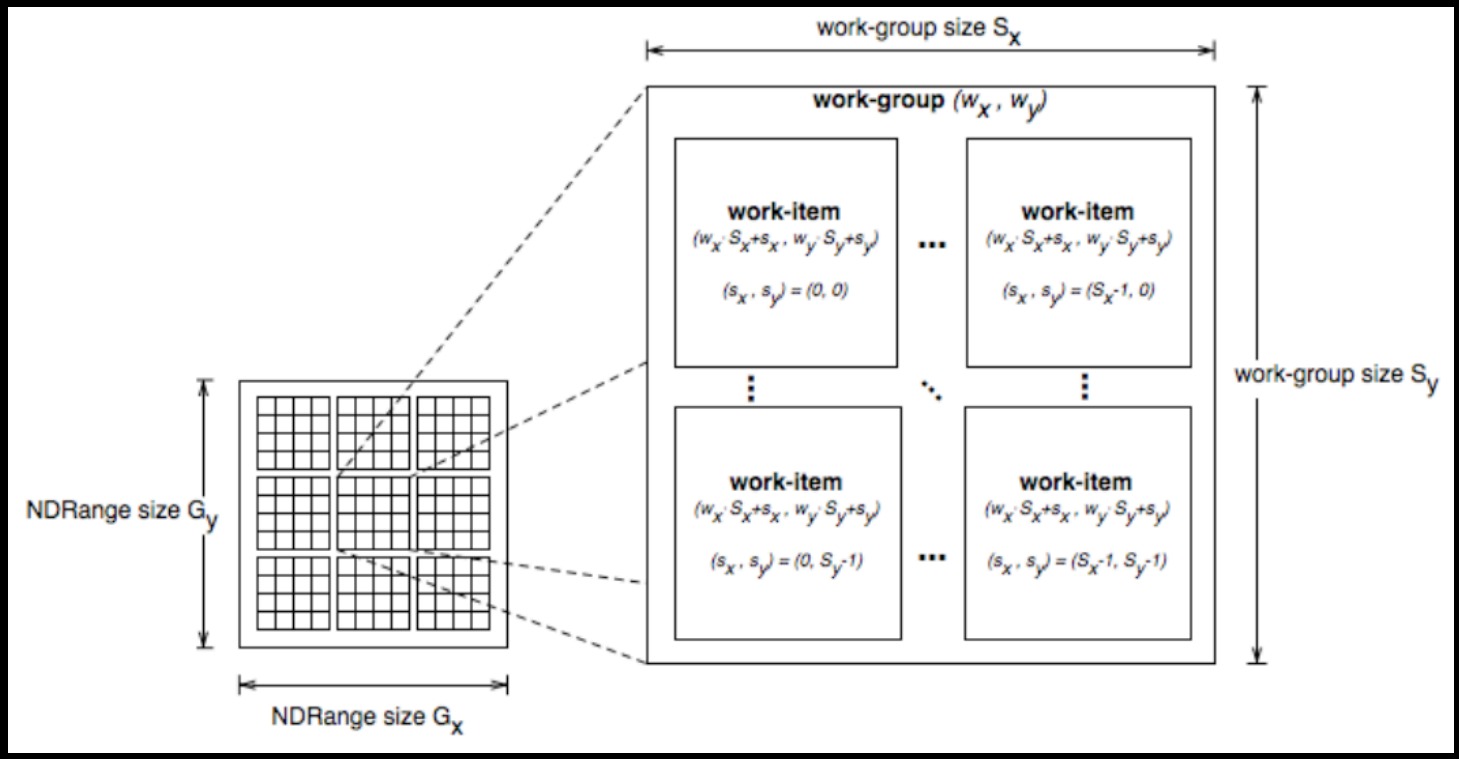
Memory Model



Decoder Ring



Kernel Execution



- Total number of work-items = G_x * G_y
- Size of each work-group = S_x * S_y
- Global ID can be computed from work-group ID and local ID

Basic Program Structure

- **Host program**
  - **Query compute devices**
  - **Create contexts**

Platform Layer

  - **Create memory objects associated to contexts**
  - **Compile and create kernel program objects**
  - **Issue commands to command-queue**
  - **Synchronization of commands**
  - **Clean up OpenCL resources**

Runtime

- **Kernels**
  - **C code with some restrictions and extensions**

Language

Memory Objects

- Buffer objects
  - 1D collection of objects (like C arrays)
  - Scalar types & Vector types, as well as user-defined Structures
  - Buffer objects accessed via pointers in the kernel
- Image objects
  - 2D or 3D texture, frame-buffer, or images
  - Must be addressed through built-in functions
- Sampler objects
  - Describe how to sample an image in the kernel
    - Addressing modes
    - Filtering modes

Language Highlights

- Function qualifiers
  - “__kernel” qualifier declares a function as a kernel
- Address space qualifiers
  - __global, __local, __constant, __private
- Work-item functions
  - Query work-item identifiers
  - get_work_dim()
  - get_global_id(), get_local_id(), get_group_id()
- Image functions
  - Images must be accessed through built-in functions
  - Reads/writes performed through sampler objects from host or defined in source
- Synchronization functions
  - Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue

Optional Extensions

- Extensions are optional features exposed through OpenCL
- The OpenCL working group has already approved many extensions that are supported by the OpenCL specification:
  - Double precision floating-point types
  - Built-in functions to support doubles
  - Atomic functions
  - 3D Image writes
  - Byte addressable stores (write to pointers with types < 32-bits)
  - Built-in functions to support half types

Vector Addition Example

- Query compute devices
- Create Context and Queue
- Create memory objects associated to contexts
- Compile and create kernel program objects
- Issue commands to command-queue
- Synchronization of commands
- Clean up OpenCL resources

Kernel Code


```
const char* cVectorAdd[ ] =
{
    "__kernel void VectorAdd(",
    "    __global const float* a,",
    "    __global const float* b,",
    "    __global float* c)",
    "{",
    "        int iGID = get_global_id(0);",
    "        c[iGID] = a[iGID] + b[iGID];",
    "}"
};
const int SOURCE_NUM_LINES = sizeof(cVectorAdd)/sizeof(cVectorAdd[0]);
```

Declarations


```
cl_context cxMainContext;          // OpenCL context
cl_command_queue cqCommandQue;     // OpenCL command que
cl_device_id* cdDevices;           // OpenCL device list
cl_program cpProgram;              // OpenCL program
cl_kernel ckKernel;                // OpenCL kernel
cl_mem cmMemObjs[3];               // OpenCL memory buffer objects
cl_int ciErrNum = 0;               //  Error code var
size_t szGlobalWorkSize[1];        //  Global # of work items
size_t szLocalWorkSize[1];         //  # of Work Items in Work Group
size_t szParmDataBytes;            //  byte length of parameter storage
size_t szKernelLength;             //  byte Length of kernel code
int iTestN = 10000;                //  Length of demo test vectors
```

Contexts and Queues


```
// create the OpenCL context on a GPU device
cxMainContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU,
                                        NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(cxMainContext, CL_CONTEXT_DEVICES, 0, NULL,
                 &szParmDataBytes);
cdDevices = (cl_device_id*) malloc(szParmDataBytes);
clGetContextInfo(cxMainContext, CL_CONTEXT_DEVICES, szParmDataBytes,
                 cdDevices, NULL);

// create a command-queue
cqCommandQue = clCreateCommandQueue (cxMainContext, cdDevices[0],
                                     0, NULL);
```

Create Memory Objects


```
// allocate the first source buffer memory object
cmMemObjs[0] = clCreateBuffer (cxMainContext,
                    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    sizeof(cl_float) * iTestN, srcA, NULL);

// allocate the second source buffer memory object
cmMemObjs[1] = clCreateBuffer (cxMainContext,
                    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    sizeof(cl_float) * iTestN, srcB, NULL);

// allocate the destination buffer memory object
cmMemObjs[2] = clCreateBuffer (cxMainContext, CL_MEM_WRITE_ONLY,
                    sizeof(cl_float) * iTestN, NULL, NULL);
```

Create Program and Kernel

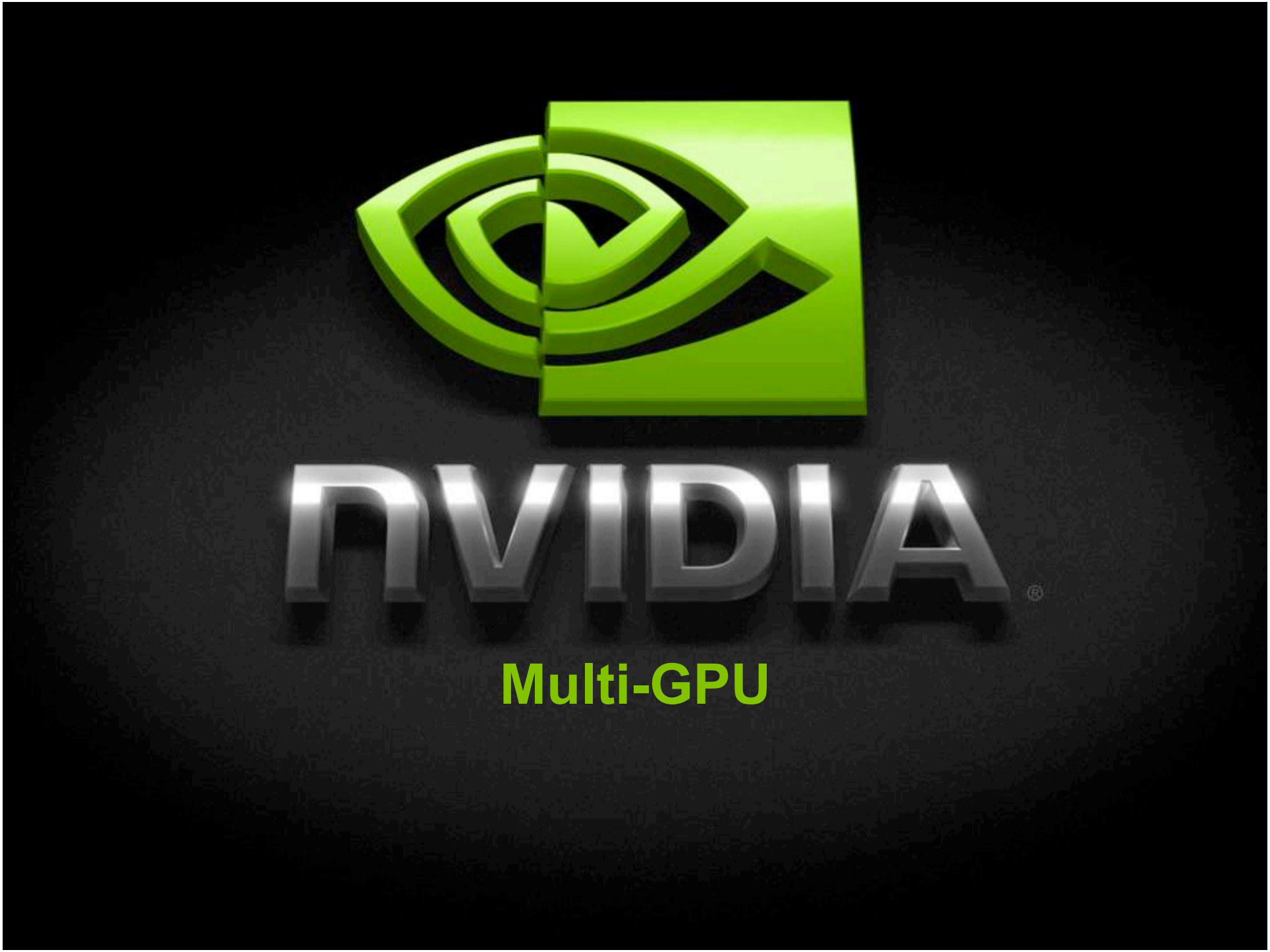

```
// create the program from OpenCL C source string array
cpProgram = clCreateProgramWithSource (cxMainContext, SOURCE_NUM_LINES,
                                       cVectorAdd, NULL, &ciErrNum);

// build the program
ciErrNum = clBuildProgram (cpProgram, 0, NULL, NULL, NULL, NULL);

// create the kernel
ckKernel = clCreateKernel (cpProgram, "VectorAdd", &ciErrNum);

// set the kernel argument values
ciErrNum = clSetKernelArg (ckKernel, 0, sizeof(cl_mem),
                           (void*)&cmMemObjs[0] );
ciErrNum |= clSetKernelArg (ckKernel, 1, sizeof(cl_mem),
                            (void*)&cmMemObjs[1] );
ciErrNum |= clSetKernelArg (ckKernel, 2, sizeof(cl_mem),
                            (void*)&cmMemObjs[2] );
```

Launch Kernel and Read Results


```
// set work-item dimensions
szGlobalWorkSize[0] = iTestN;
szLocalWorkSize[0]= 1;

// execute kernel
ciErrNum = clEnqueueNDRangeKernel(cqCommandQue, ckKernel, 1, NULL,
                                  szGlobalWorkSize, szLocalWorkSize,
                                   0, NULL, NULL);

// read output
ciErrNum = clEnqueueReadBuffer(cqCommandQue, cmMemObjs[2], CL_TRUE,
                               0, iTestN * sizeof(cl_float), dst,
                               0, NULL, NULL);
```

Cleanup


```
// release kernel, program, and memory objects
DeleteMemobjs (cmMemObjs, 3);
free (cdDevices);
clReleaseKernel (ckKernel);
clReleaseProgram (cpProgram);
clReleaseCommandQueue (cqCommandQue);
clReleaseContext (cxMainContext);
```


NVIDIA
Multi-GPU

Why Multi-GPU Programming?

- Many systems contain multiple GPUs:
  - Servers (Tesla/Quadro servers and desksides)
  - Desktops (2- and 3-way SLI desktops, GX2 boards)
  - Laptops (hybrid SLI)
- Additional processing power
  - Increasing processing throughput
- Additional memory
  - Some problems do not fit within a single GPU memory

Multi-GPU Memory

- GPUs do not share global memory
  - One GPU cannot access another GPUs memory directly
- Inter-GPU communication
  - Application code is responsible for moving data between GPUs
  - Data travels across the PCIe bus
    - Even when GPUs are connected to the same PCIe switch

CPU-GPU Context

- A CPU-GPU context must be established before calls are issued to the GPU
- CUDA resources are allocated per context
- A context is established by:
  - The first CUDA call that changes state
    - cudaMalloc, cudaMemcpy, kernel launch, ...
  - On device 0, unless there is a cudaSetDevice(...) call
- A context is destroyed by one of:
  - Explicit cudaThreadExit() call
  - Host thread terminating

Run-Time API Device Management:

- A host thread can maintain one context at a time
  - GPU is part of the context and cannot be changed once a context is established
  - Need as many host threads as GPUs
  - Note that multiple host threads can establish contexts with the same GPU
    - Driver handles time-sharing and resource partitioning
- GPUs have consecutive integer IDs, starting with 0
- Device management calls:
  - cudaGetDeviceCount(int *num_devices)
  - cudaSetDevice(int device_id)
  - cudaGetDevice(int *current_device_id)
  - cudaThreadExit( )

Choosing a Device

- Properties for a given device can be queried
  - No context is necessary or is created
  - cudaGetDeviceProperties(cudaDeviceProp *properties, int device_id)
  - This is useful when a system contains different GPUs
- Default behavior:
  - Device 0 is chosen when no explicit cudaSetDevice is called
    - Note this will cause multiple contexts with the same GPU
    - Except when driver is in the *exclusive mode* (details later)

Ensuring One Context Per GPU

- Two ways to achieve:
  - Application-control
  - Driver-control
- Application-control:
  - Host threads negotiate which GPUs to use
  - For example, OpenMP threads set device based on OpenMPI thread ID
  - Pitfall: different applications are not aware of each other's GPU usage

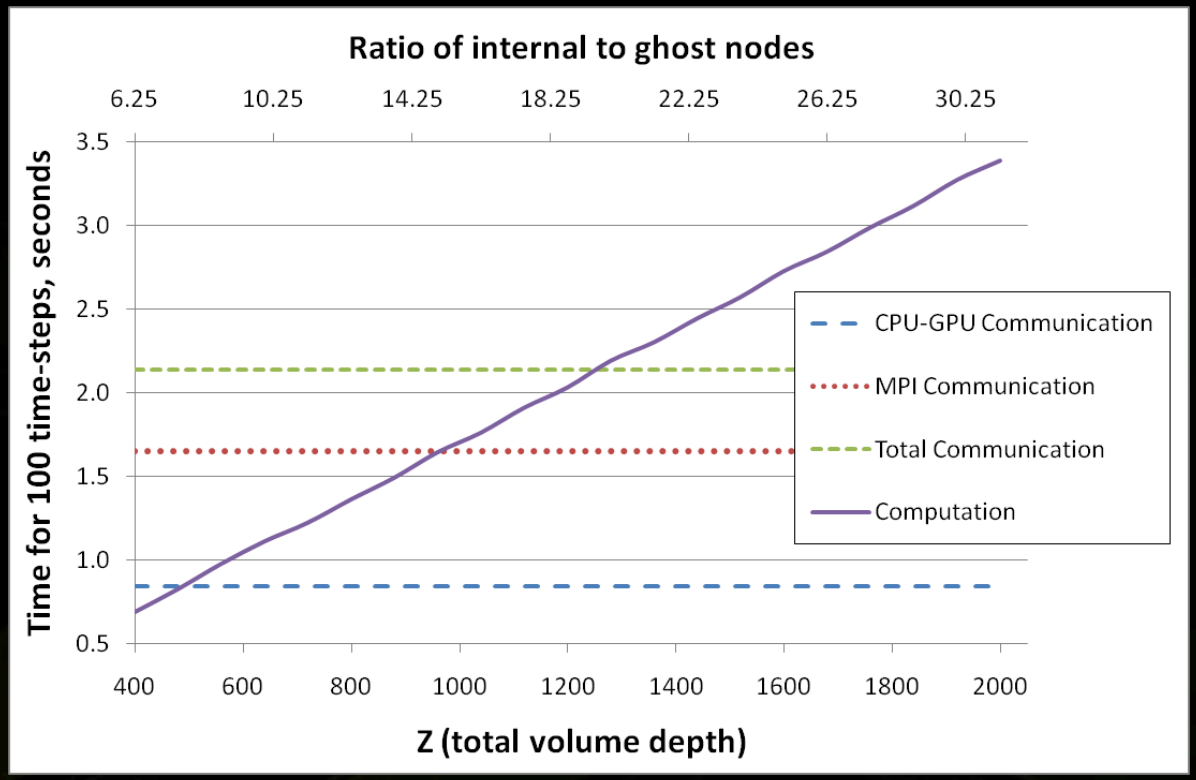
Driver-control (Exclusive Mode)

- To use exclusive mode:
  - Driver: set the GPU to exclusive mode using SMI
    - SMI (System Management Tool) is provided with Linux drivers
  - Application: do not explicitly set the GPU in the application
- Behavior:
  - Driver will implicitly set a GPU with no contexts
  - Implicit context creation will fail if all GPUs have contexts
    - The first state-changing CUDA call will fail and return an error
- Device mode can be checked by querying its properties

Inter-GPU Communication

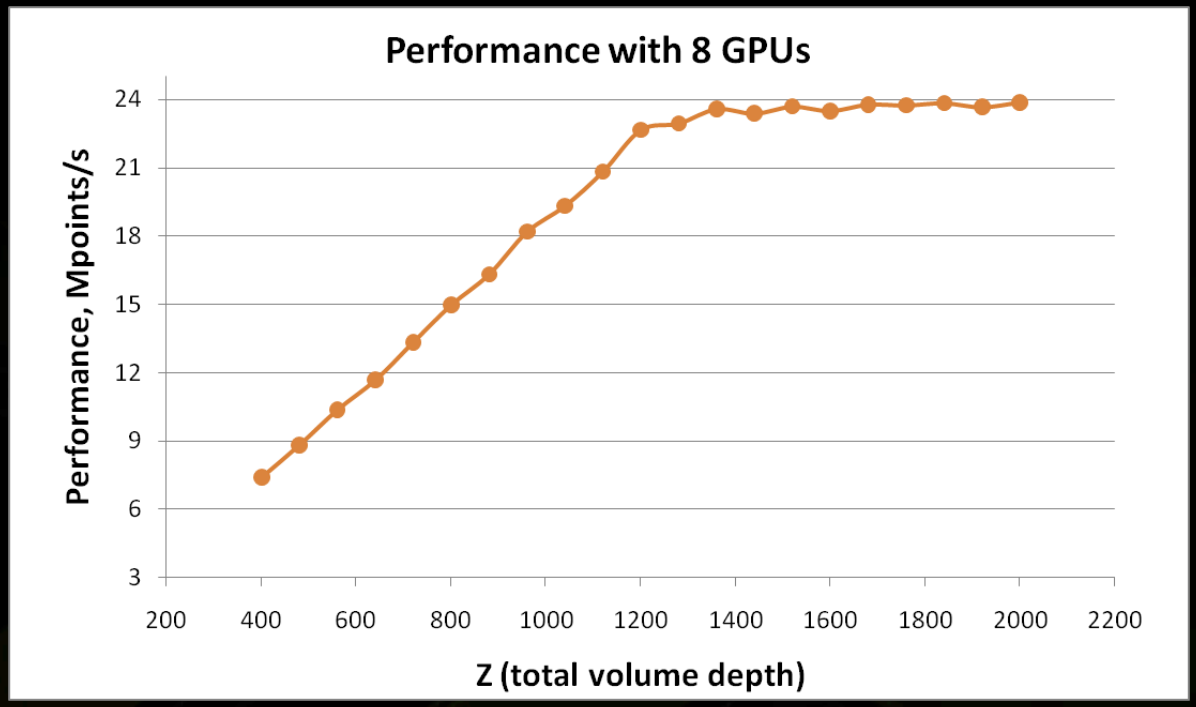
- Application is responsible for moving data between GPUs:
  - Copy data from GPU to host thread A
  - Copy data from host thread A to host thread B
    - Use any CPU library (MPI, ...)
  - Copy data from host thread B to its GPU
- Use asynchronous memcopies to overlap kernel execution with data copies
- Lightweight host threads (OpenMP, pthreads) can reduce host-side copies by sharing pinned memory
  - Allocate with cudaHostAlloc(...)

Performance Example: 3DFD



- Fixed *x* and *y* dimensions, varying *z*
- Data is partitioned among GPUs along *z*
  - Computation increases with *z*, communication stays constant

Performance Example: 3DFD



- Single GPU performance is 3,000 MPoints/s
- Note that 8x scaling is sustained at z > 1,300
  - Exactly where computation exceeds communication