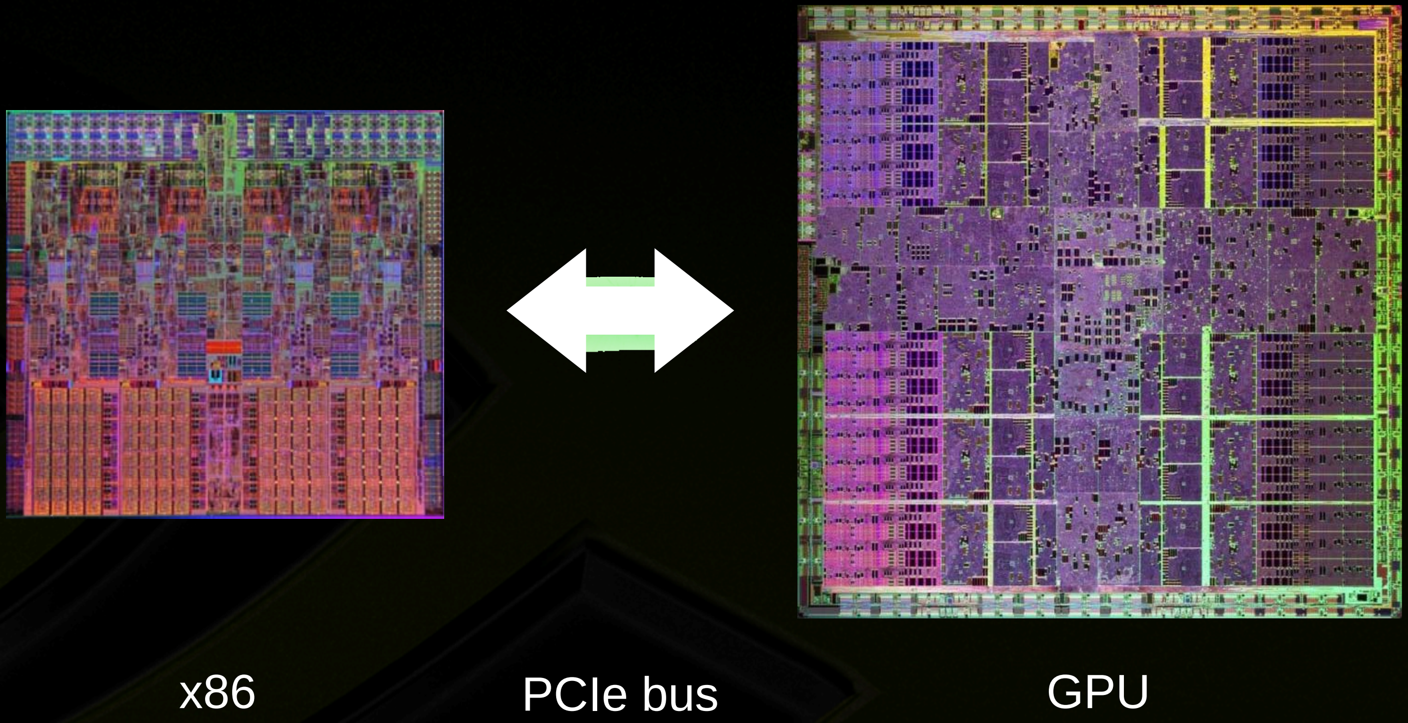
High-Performance Computing with CUDA and Tesla GPUs

Timothy Lanfear, NVIDIA

What is GPU Computing?

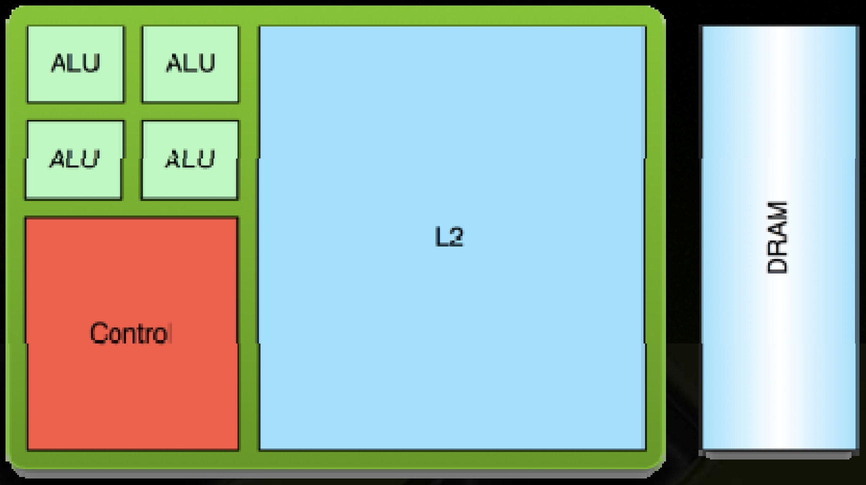
What is GPU Computing?



Computing with CPU + GPU

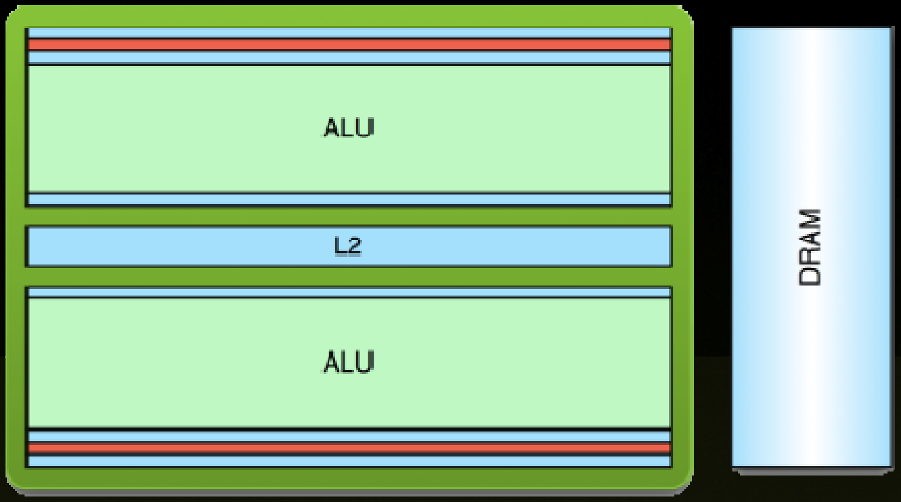
Heterogeneous Computing

Low Latency or High Throughput?



CPU

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



GPU

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

Fermi: The Computational GPU

Performance

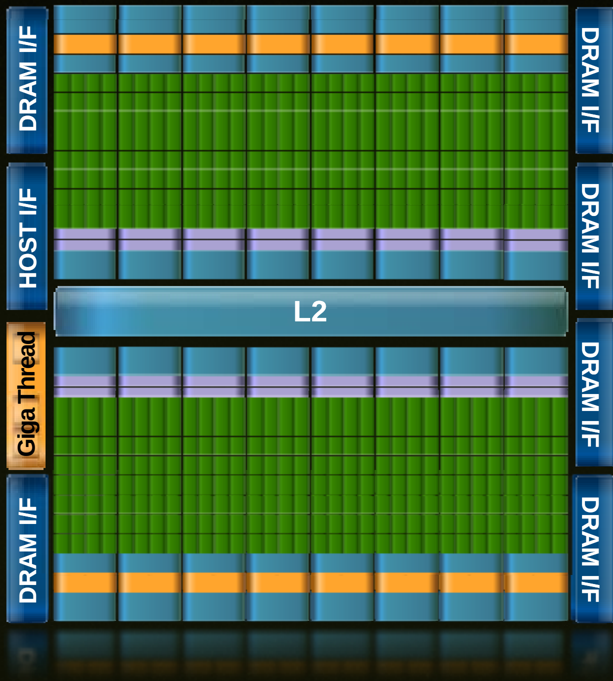
- 13× Double Precision of CPUs
- IEEE 754-2008 SP & DP Floating Point

Flexibility

- Increased Shared Memory from 16 KB to 64 KB
- Added L1 and L2 Caches
- ECC on all Internal and External Memories
- Enable up to 1 TeraByte of GPU Memories
- High Speed GDDR5 Memory Interface

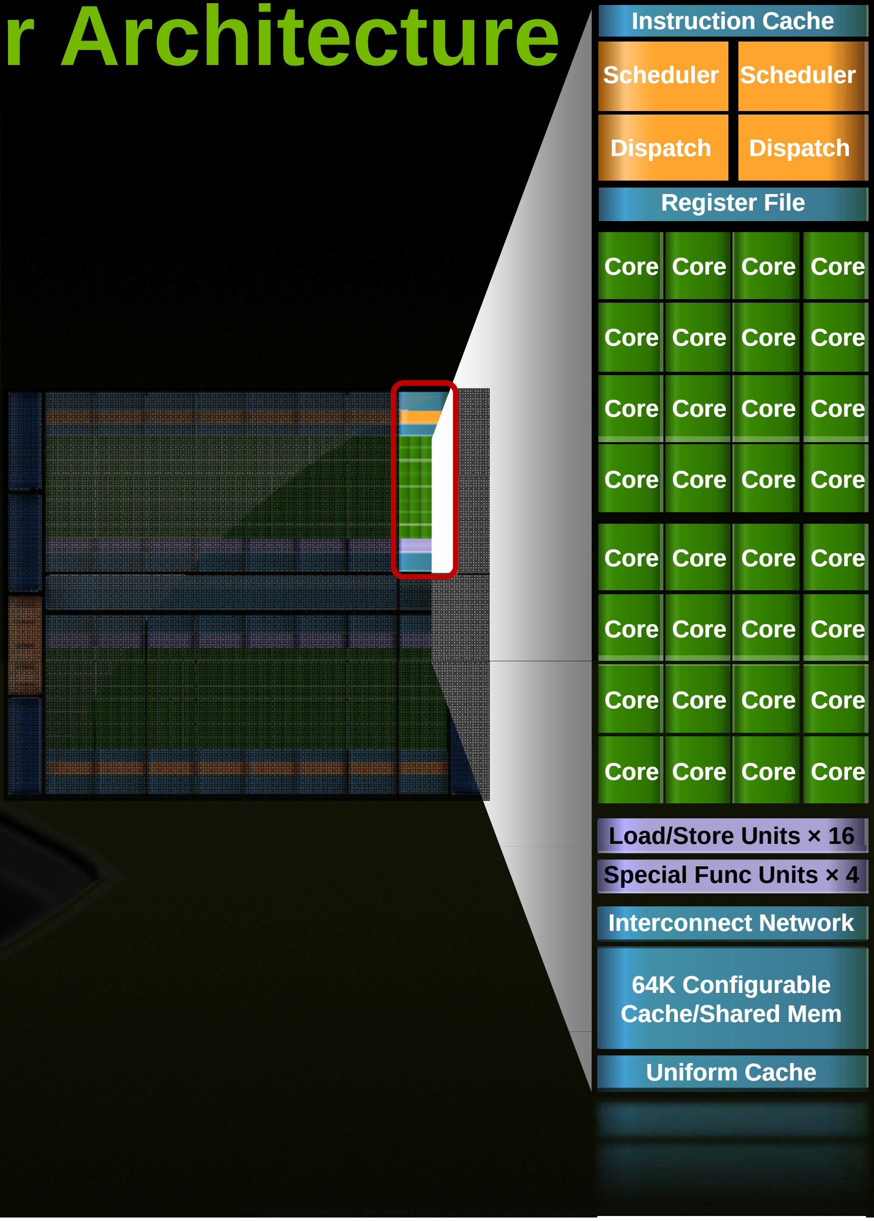
Usability

- Multiple Simultaneous Tasks on GPU
- 10× Faster Atomic Operations
- C++ Support
- System Calls, printf support



Streaming Multiprocessor Architecture

- 32 CUDA cores per SM (512 total)
- 2:1 ratio SP:DP floating-point performance
- Dual Thread Scheduler
- 64 KB of RAM for shared memory and L1 cache (configurable)



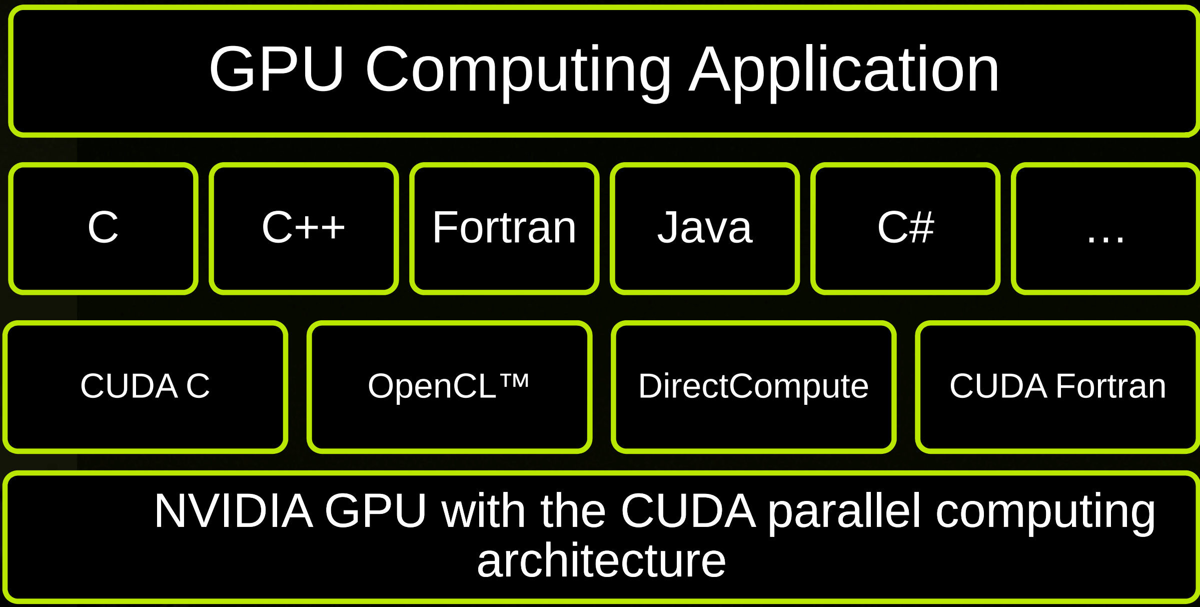
Tesla C-Series Workstation GPUs

	Tesla C1060	Tesla C2050	Tesla C2070
Architecture	Tesla 10-series GPU	Tesla 20-series GPU	
Number of Cores	240	448	
Caches	16 KB Shared Memory / 8 cores	64 KB L1 cache + Shared Memory / 32 cores, 768 KB L2 cache	
Floating Point Peak Performance	933 Gigaflops (single) 78 Gigaflops (double)	1030 Gigaflops (single) 515 Gigaflops (double)	
GPU Memory	4 GB	3 GB 2.625 GB with ECC on	6 GB 5.25 GB with ECC on
Memory Bandwith	102 GB/s (GDDR3)	144 GB/s (GDDR5)	
System I/O	PCIe x16 Gen2	PCIe x16 Gen2	
Power	188 W (max)	247 W (max)	225 W (max)
Available	Available now	Shipping in May	Q3 2010

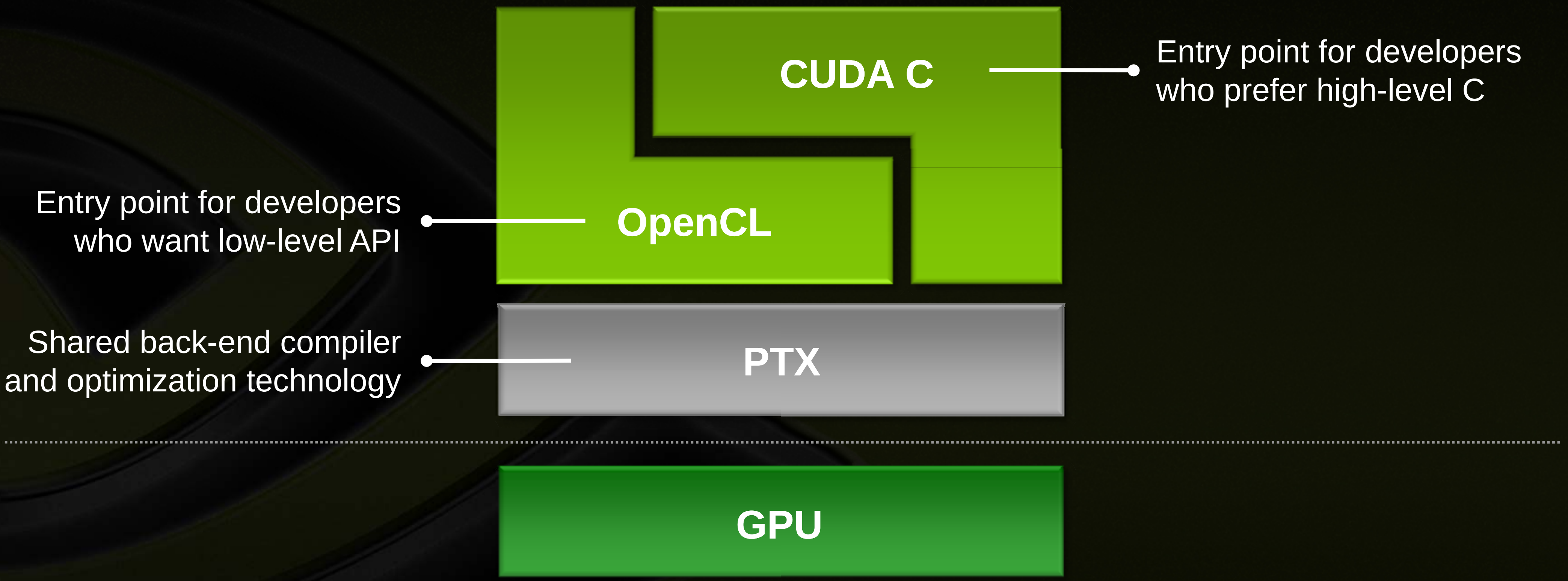
CUDA ARCHITECTURE

CUDA Parallel Computing Architecture

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

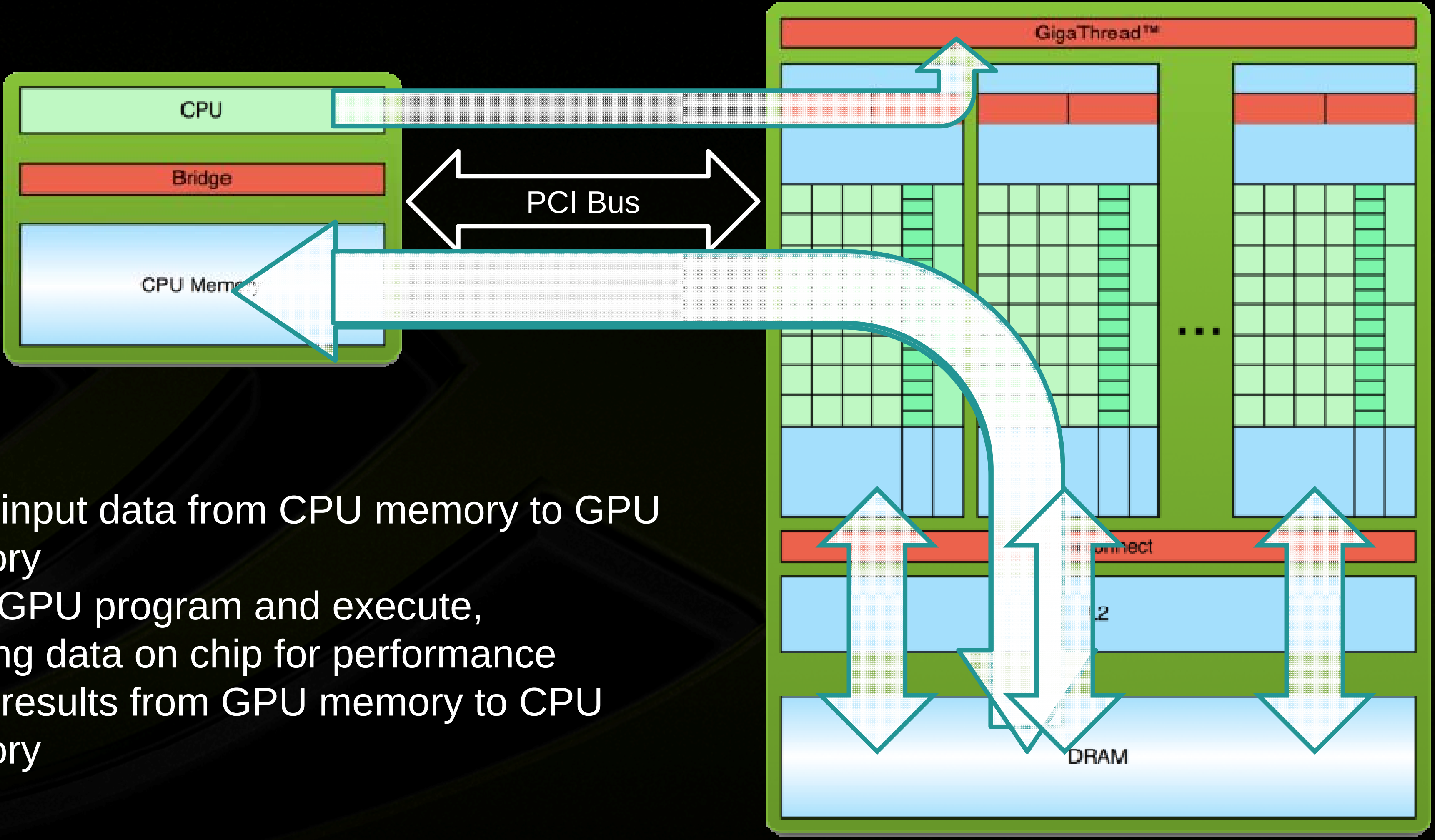

GPU Computing Application
C
C++
Fortran
Java
C#
…
CUDA C
OpenCL™
DirectCompute
CUDA Fortran
NVIDIA GPU with the CUDA parallel computing architecture

NVIDIA CUDA C and OpenCL


CUDA C
Entry point for developers who prefer high-level C
OpenCL
Entry point for developers who want low-level API
PTX
Shared back-end compiler and optimization technology
GPU

CUDA Programming Model

Processing Flow



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

CUDA Kernels

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

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

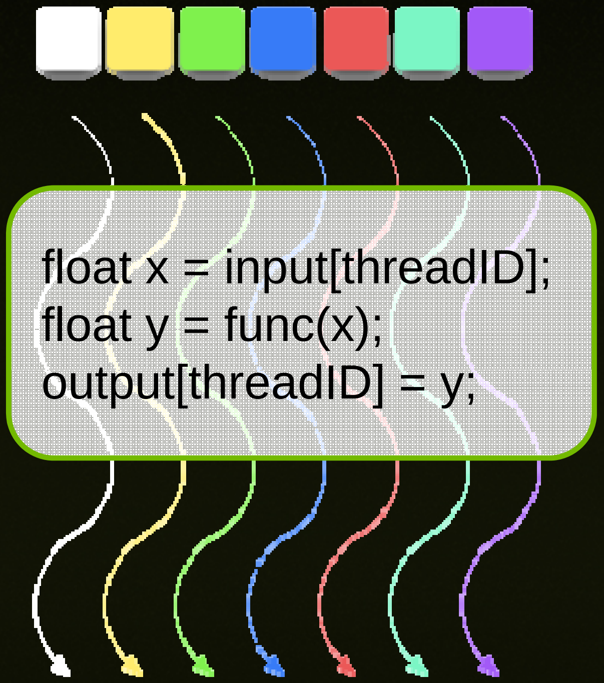
CPU	Host	Executes functions
GPU	Device	Executes kernels

CUDA Kernels: Parallel Threads

- A kernel is an array of threads, executed in parallel

- All threads execute the same code

- Each thread has an ID
  - Select input/output data
  - Control decisions



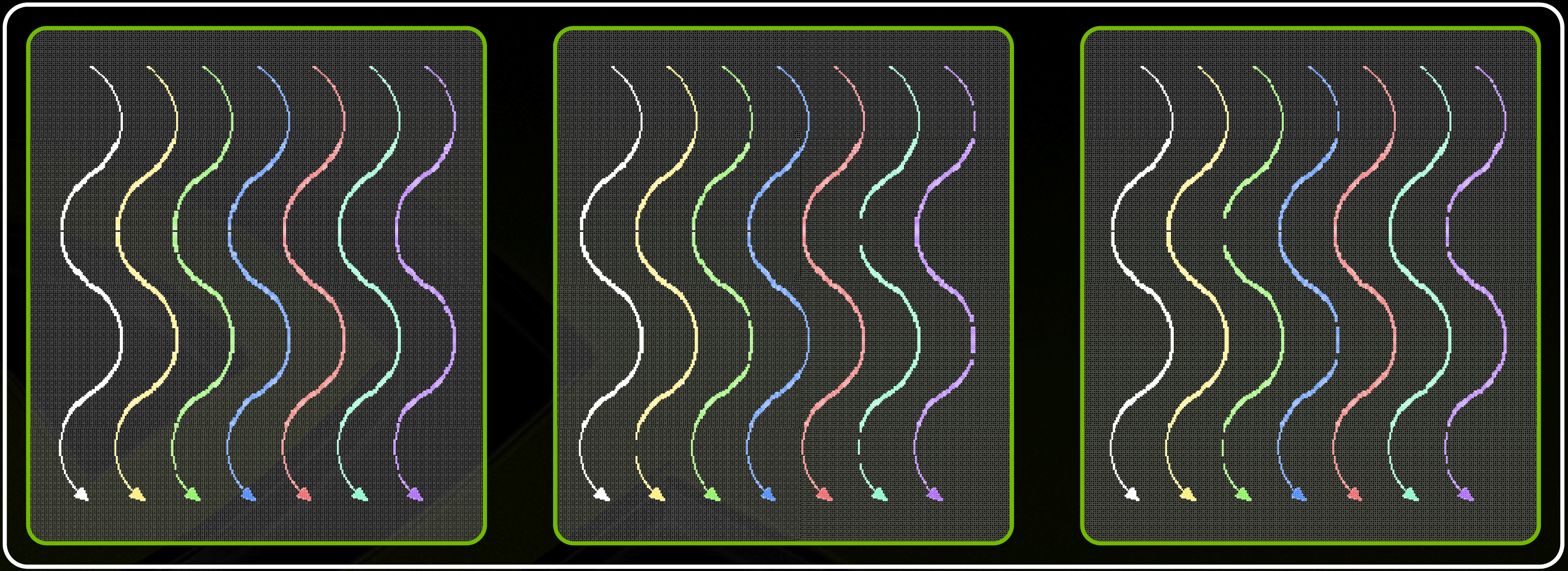
```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

CUDA Kernels: Subdivide into Blocks

CUDA Kernels: Subdivide into Blocks

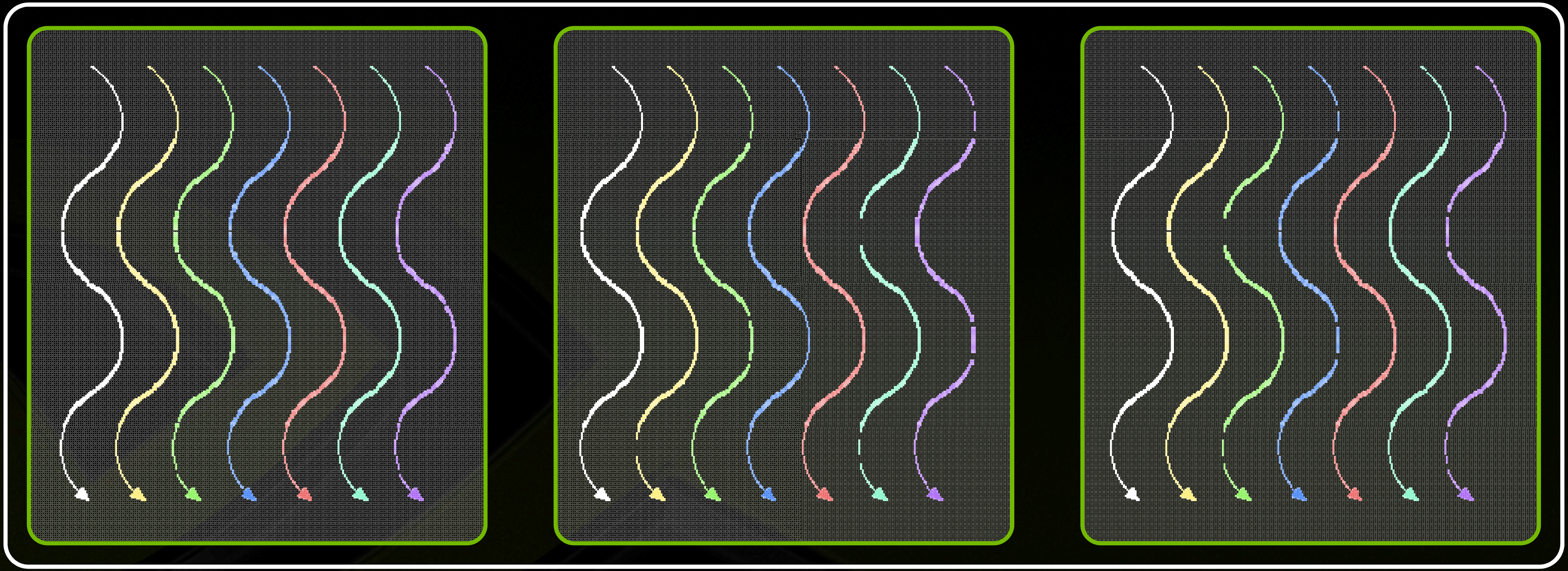
- Threads are grouped into **blocks**

CUDA Kernels: Subdivide into Blocks



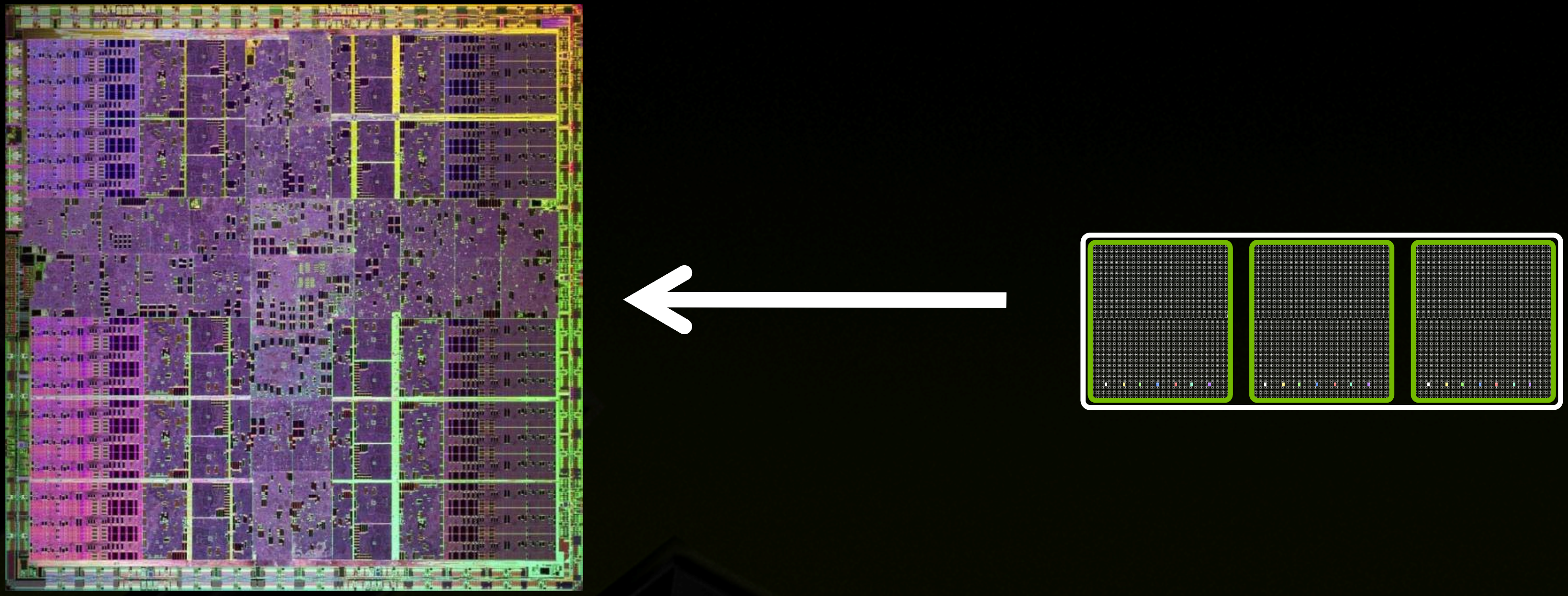
- Threads are grouped into blocks
- Blocks are grouped into a grid

CUDA Kernels: Subdivide into Blocks



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

CUDA Kernels: Subdivide into Blocks

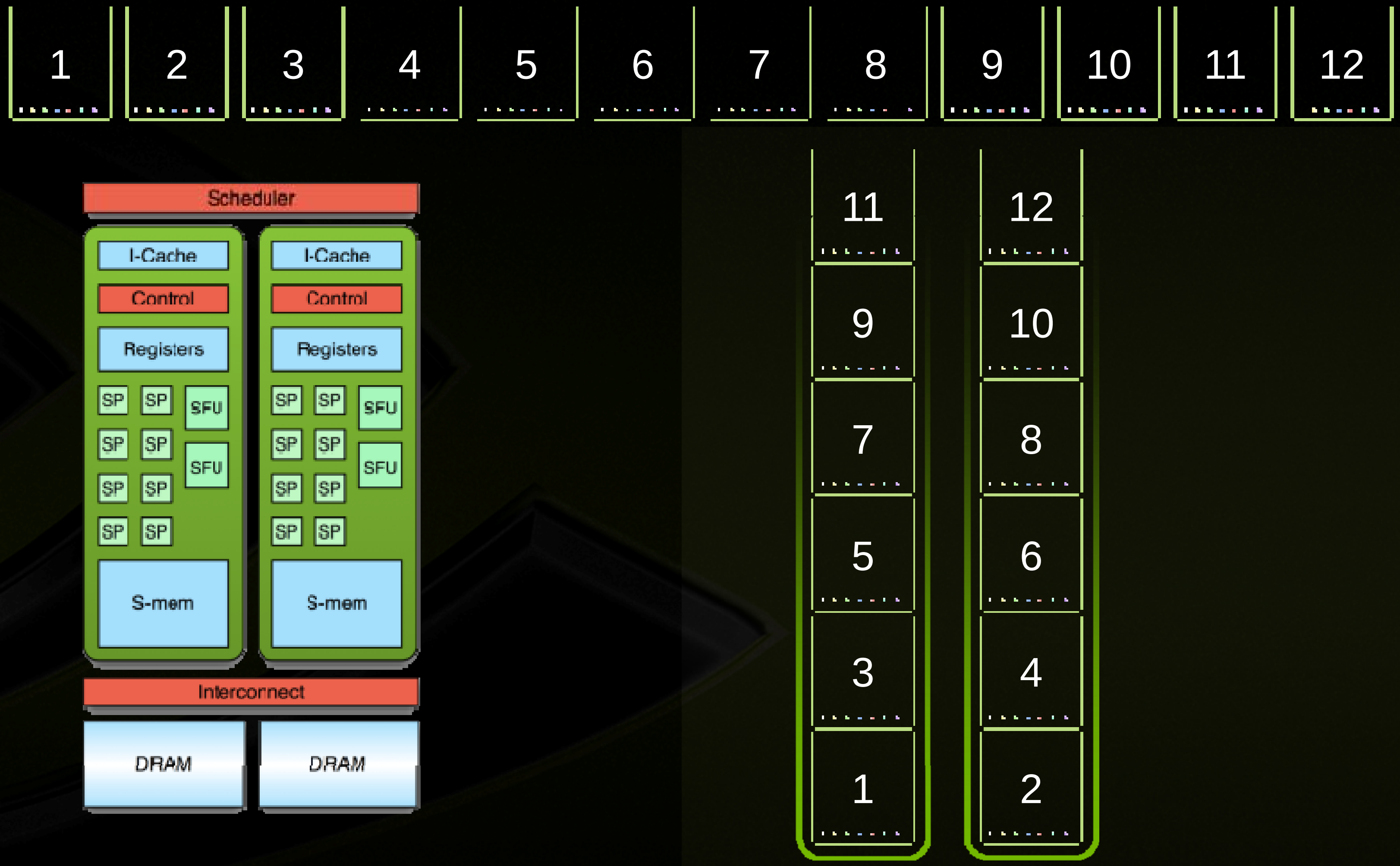


- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

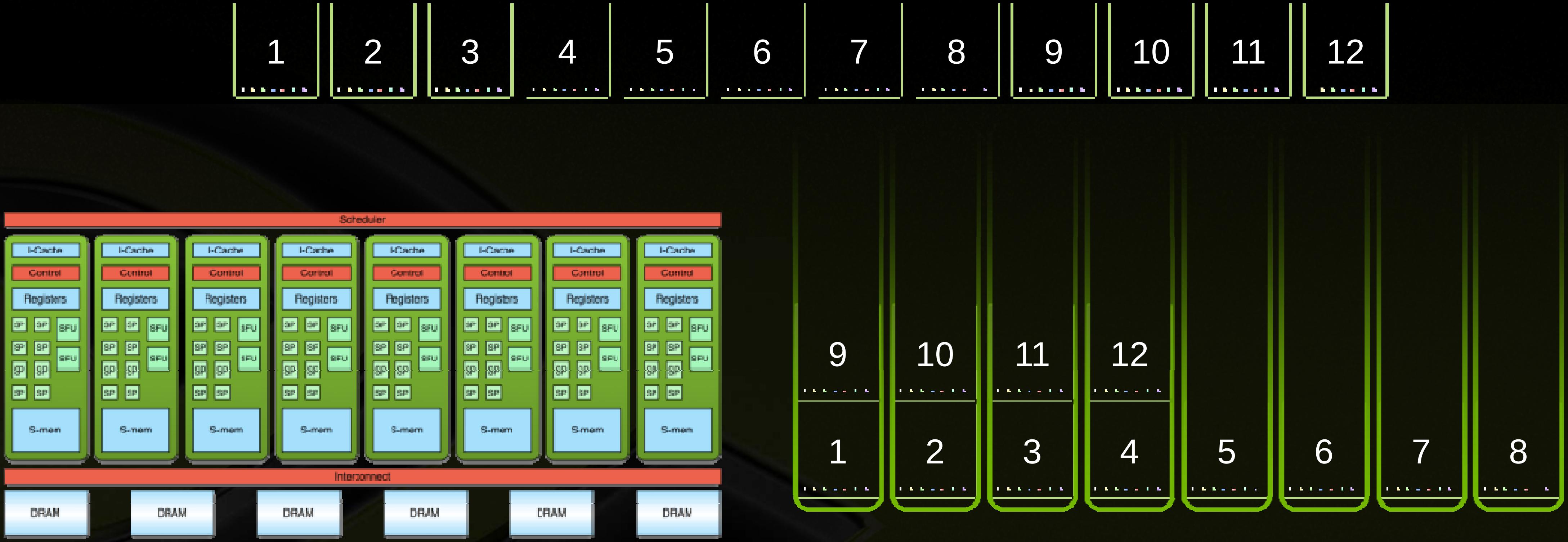
Communication Within a Block

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

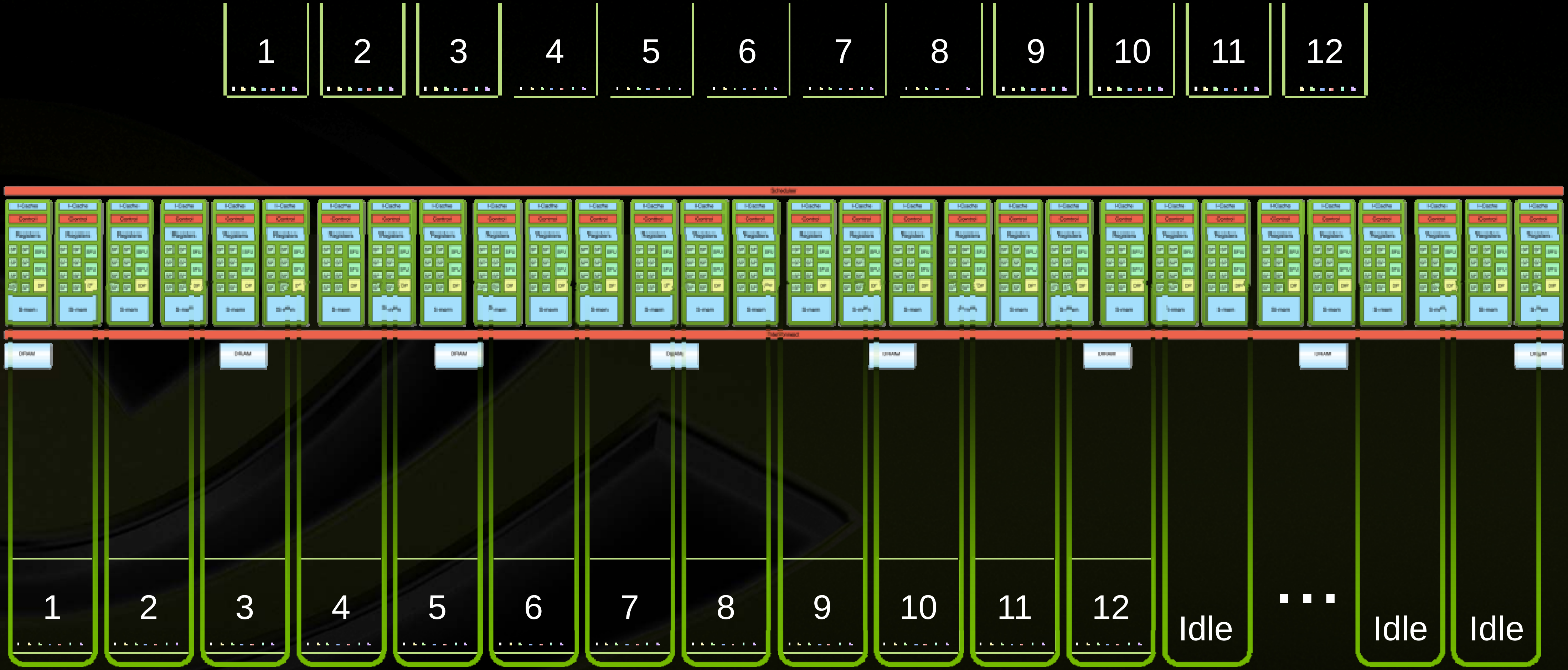
Transparent Scalability – G84



Transparent Scalability – G80

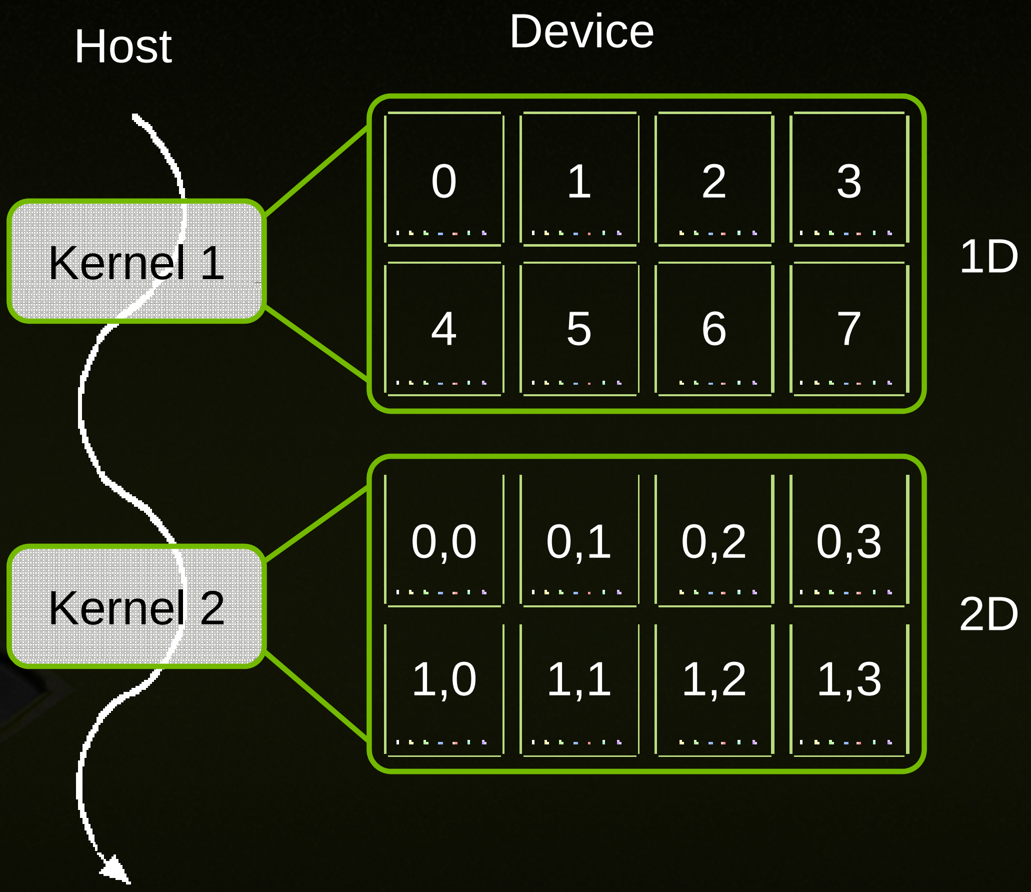


Transparent Scalability – GT200



CUDA Programming Model - Summary

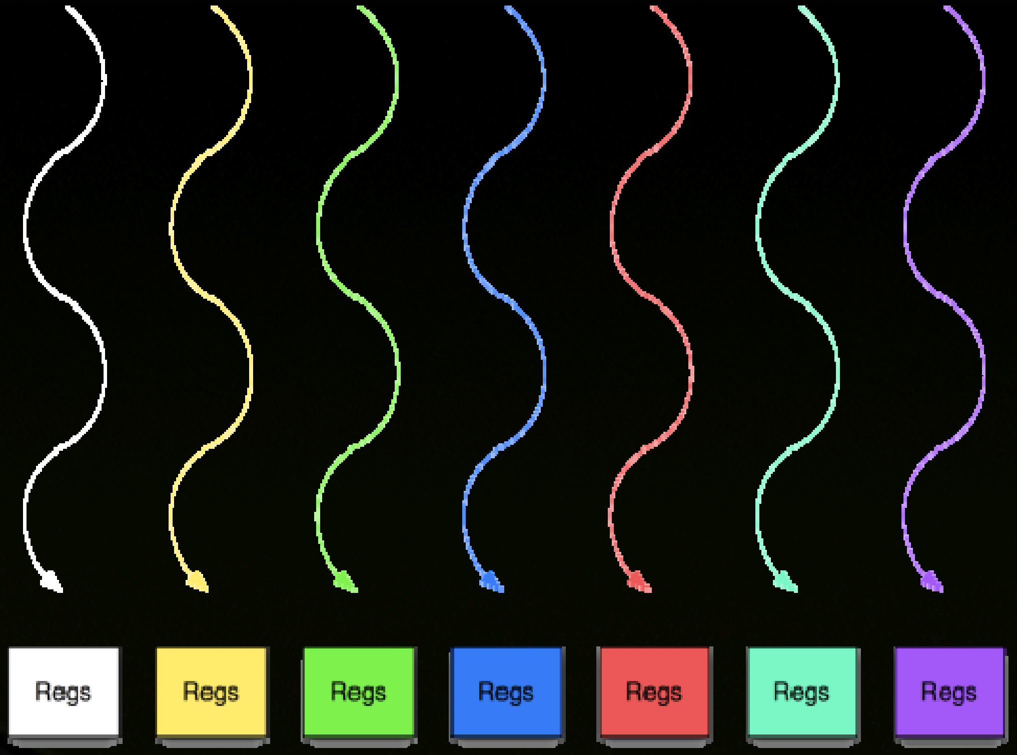
- A kernel executes as a grid of thread blocks
- A block is a batch of threads
  - Communicate through shared memory
- Each block has a block ID
- Each thread has a thread ID



MEMORY MODEL

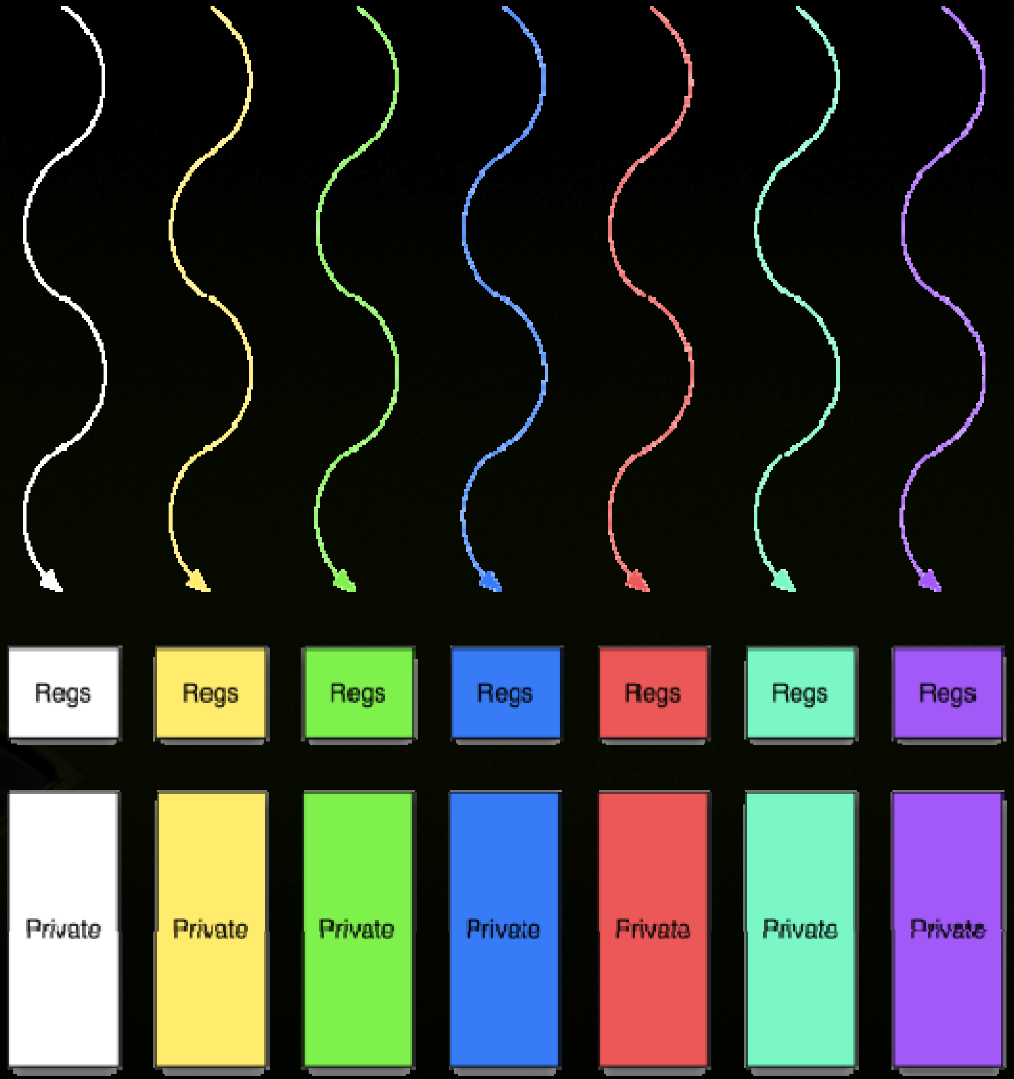
Memory hierarchy

- **Thread:**
  - Registers



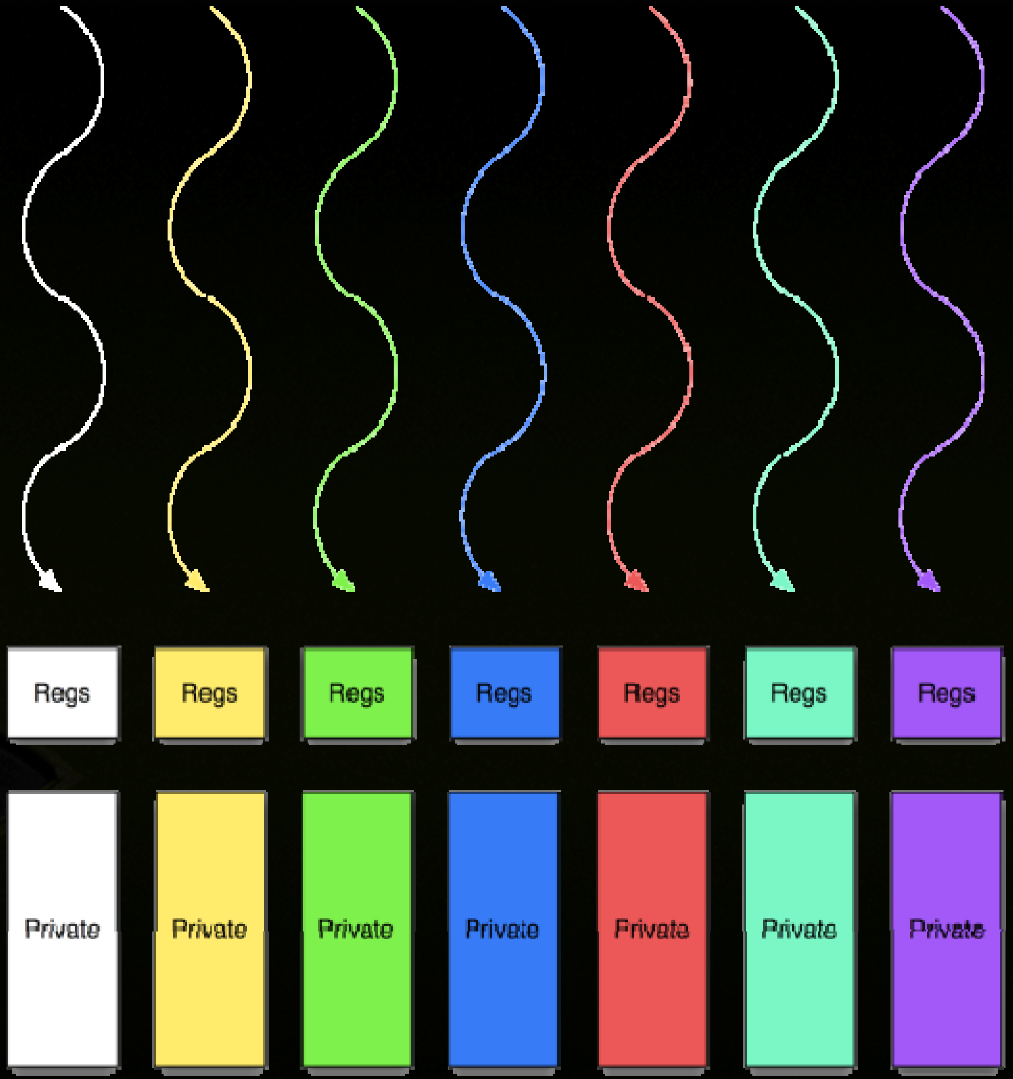
Memory hierarchy

- **Thread:**
  - **Registers**
- **Thread:**
  - **Private memory**



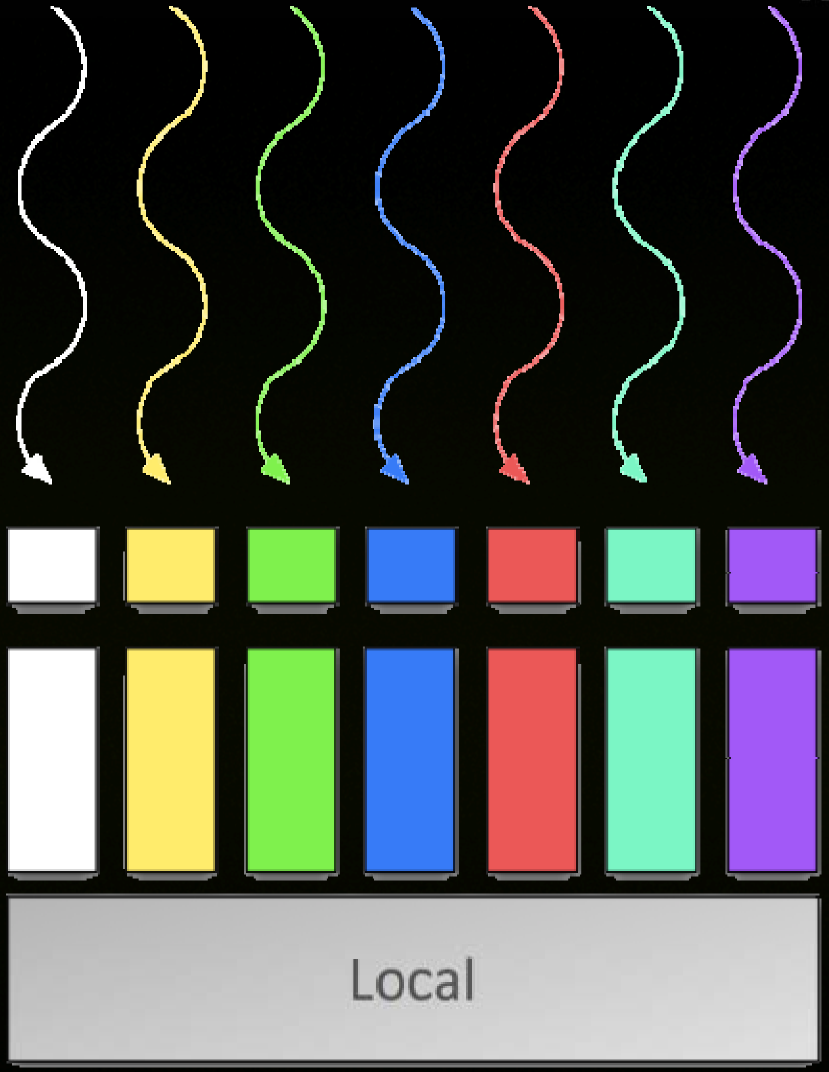
Memory hierarchy

- **Thread:**
  - **Registers**
- **Thread:**
  - **Private memory**
- **Block of threads (work group):**
  - **Local memory**



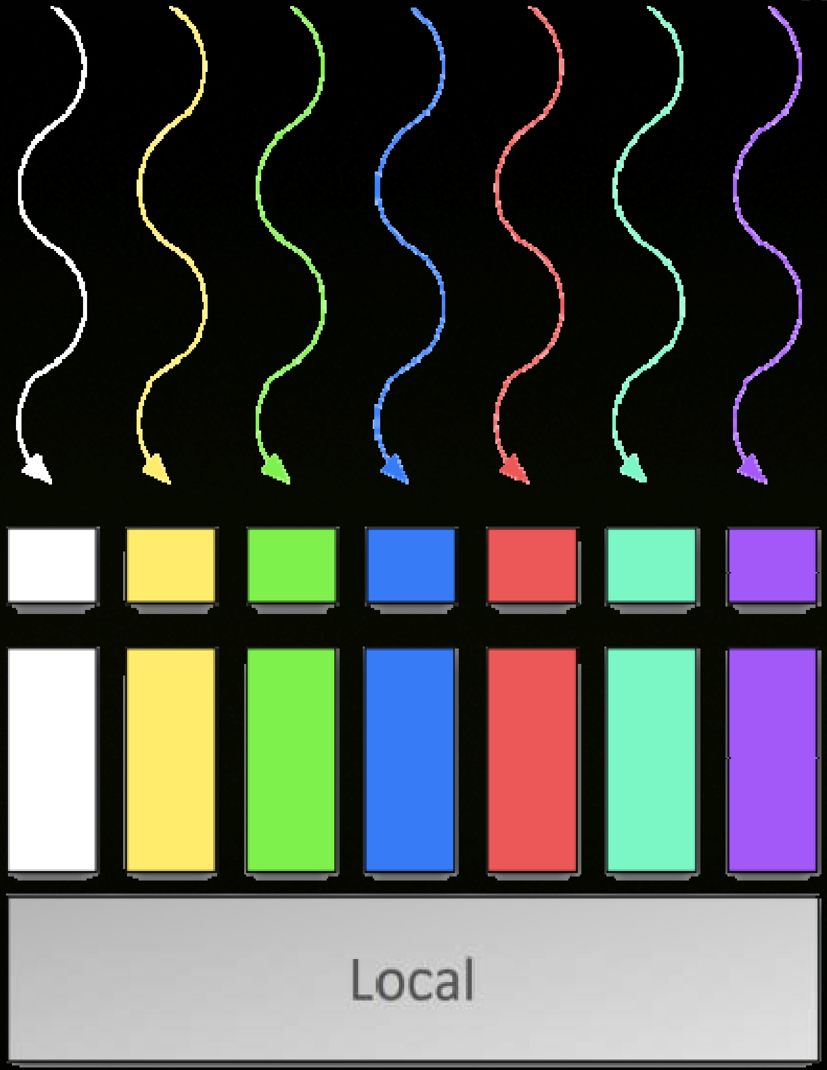
Memory hierarchy

- **Thread:**
  - **Registers**
- **Thread:**
  - **Private memory**
- **Block of threads (work group):**
  - **Local memory**



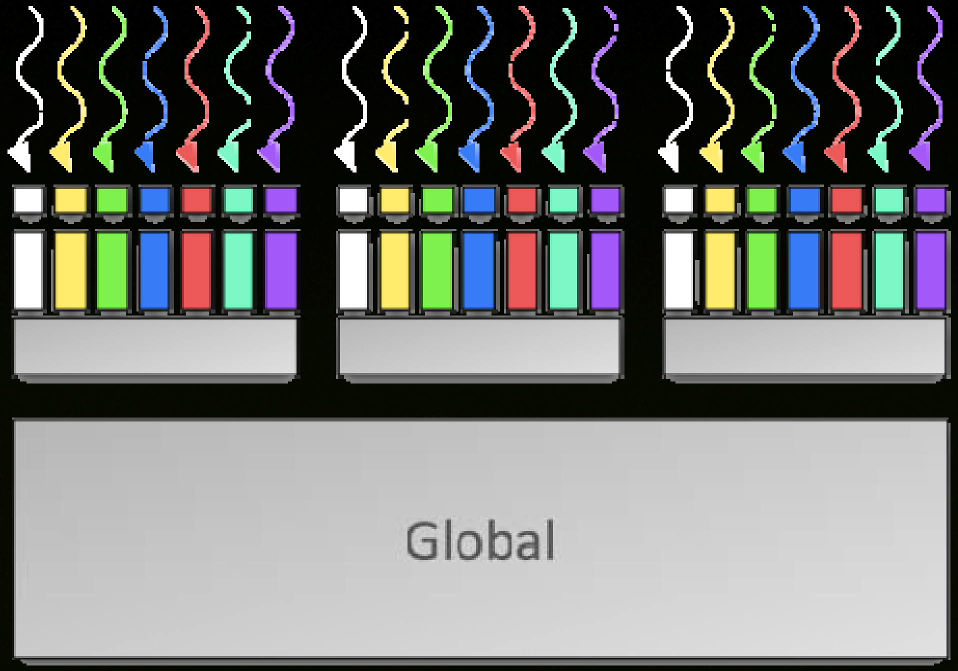
Memory hierarchy

- Thread:
  - Registers
- Thread:
  - Private memory
- Block of threads (work group):
  - Local memory
- All blocks:
  - Global memory



Memory hierarchy

- Thread:
  - Registers
- Thread:
  - Private memory
- Block of threads (work group):
  - Local memory
- All blocks:
  - Global memory



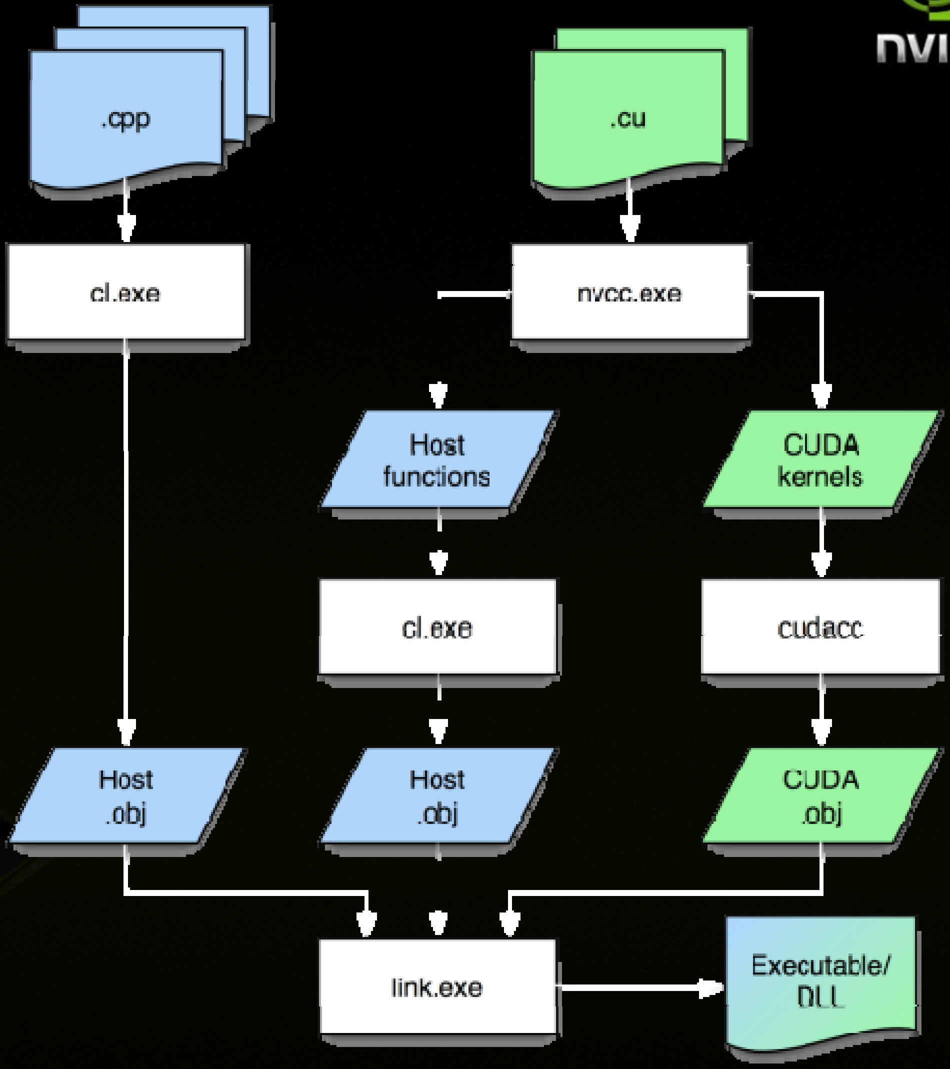
Memory Spaces

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

COMPILATION

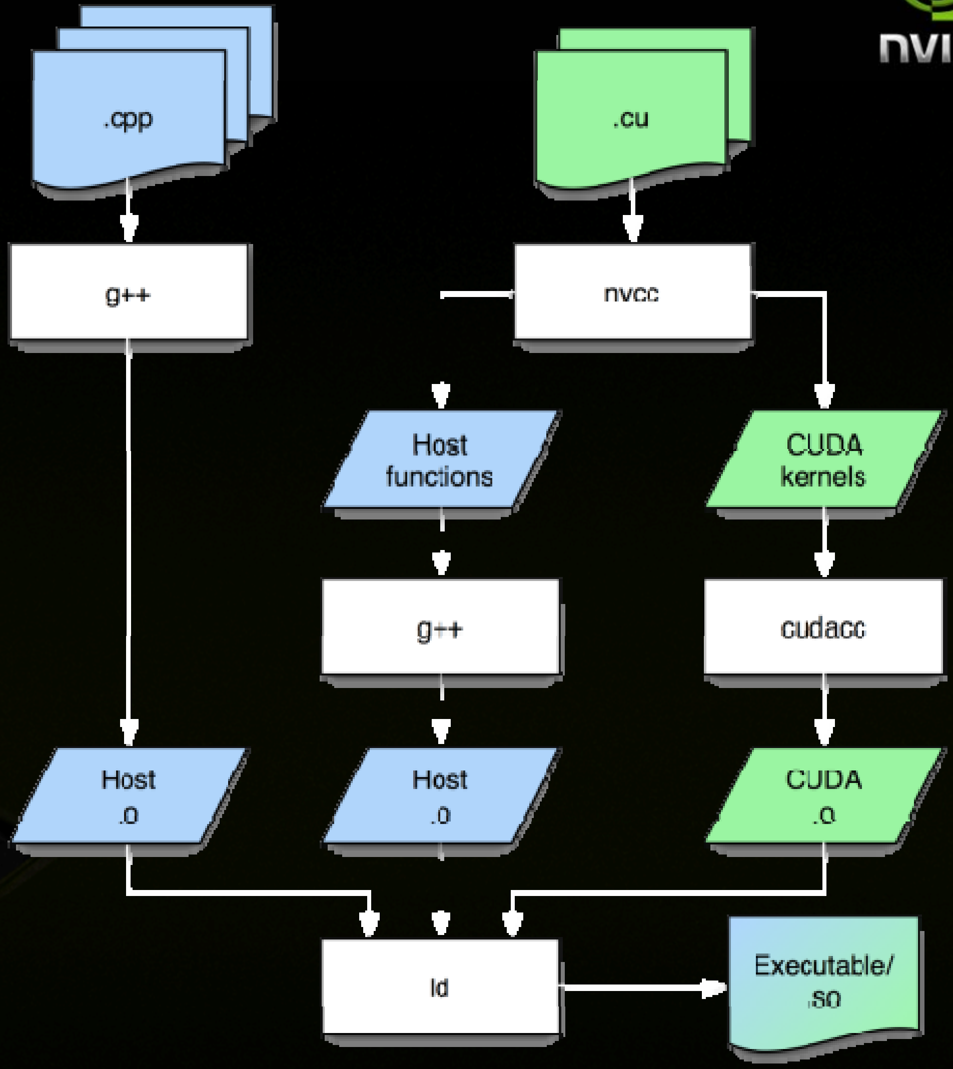
Visual Studio

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



Linux

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



Compilation Commands

- nvcc <filename>.cu [-o <executable>]
  - Builds release mode
- nvcc –g <filename>.cu
  - Builds debug (device) mode
  - Can debug host code but not device code (runs on GPU)
- nvcc –deviceemu <filename>.cu
  - Builds device emulation mode
  - All code runs on CPU, but no debug symbols
- nvcc –deviceemu –g <filename>.cu
  - Builds debug device emulation mode
  - All code runs on CPU, with debug symbols
  - Debug using gdb or other linux debugger

Exercise 0: Run a Simple Program

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

```
 There is 1 device supporting CUDA

 Device 0: "Quadro FX 570M"
   Major revision number:                         1
   Minor revision number:                         1
   Total amount of global memory:                 268107776 bytes
   Number of multiprocessors:                     4
   Number of cores:                               32
   Total amount of constant memory:               65536 bytes
   Total amount of shared memory per block:       16384 bytes
   Total number of registers available per block: 8192
   Warp size:                                     32
   Maximum number of threads per block:           512
   Maximum sizes of each dimension of a block:    512 x 512 x 64
   Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
   Maximum memory pitch:                          262144 bytes
   Texture alignment:                             256 bytes
   Clock rate:                                    0.95 GHz
   Concurrent copy and execution:                 Yes

 Test PASSED

 Press ENTER to exit...
```

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

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

CUDA C PROGRAMMING LANGUAGE

CUDA C — C with Runtime Extensions

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

CUDA C — C with Language Extensions

- Function qualifiers

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

- Variable qualifiers

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

- Built-in vector types

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

CUDA C — C with Language Extensions

- Execution configuration

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

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

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

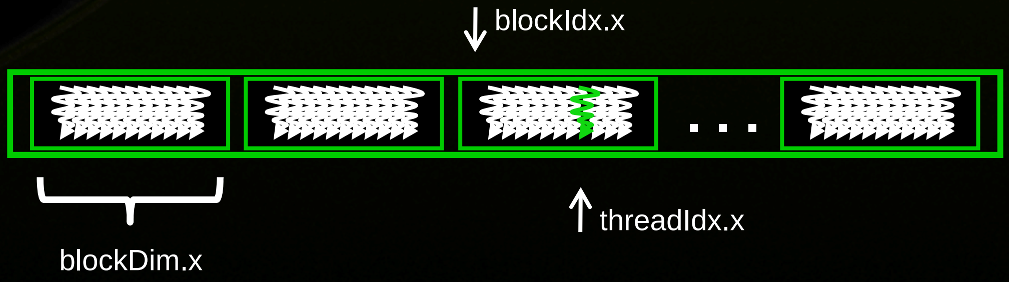
SAXPY: Device Code

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

Standard C Code

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

Parallel C Code



SAXPY: Host Code

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

// Initialize them...

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

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

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

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

Exercise 1: Move Data between Host and GPU

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

Launching a Kernel

- **Call a kernel with**

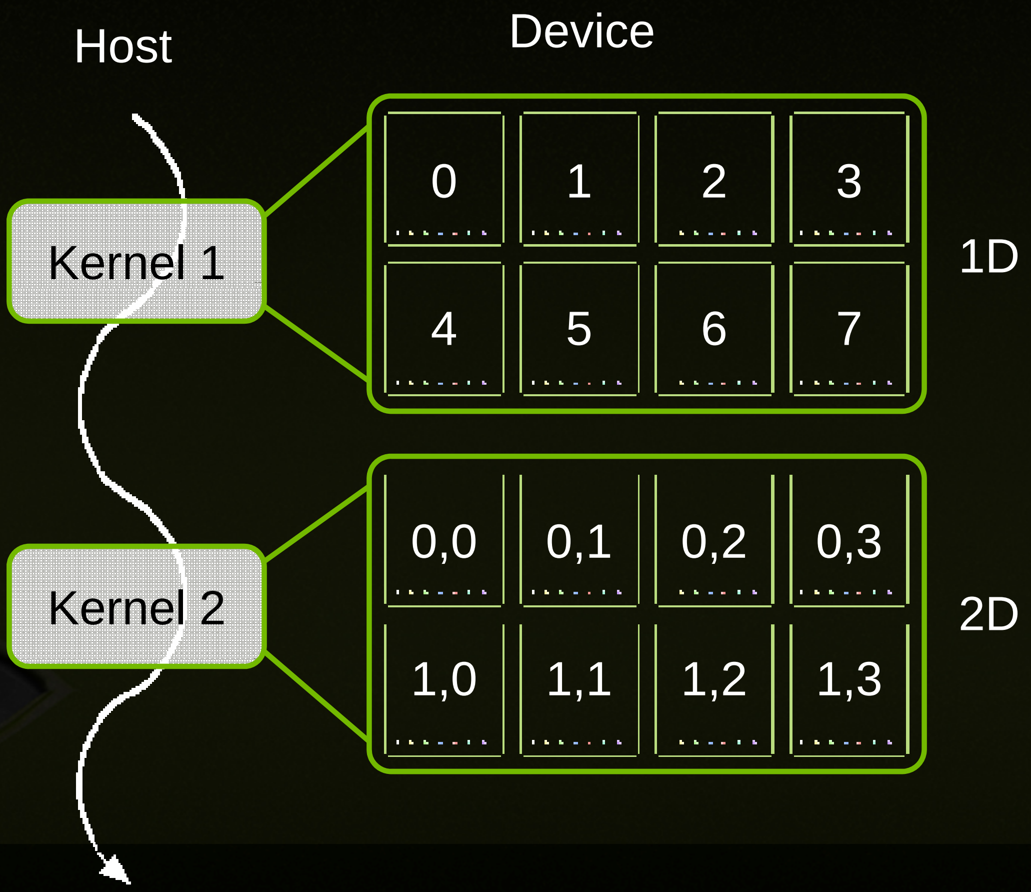
```
Func <<<Dg,Db,Ns,S>>> (params);
dim3 Dg(mx,my,1); // grid spec
dim3 Db(nx,ny,nz); // block spec
size_t Ns; // shared memory
cudaStream_t S; // CUDA stream
```

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

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

- **Extract components with**

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



Exercise 2: Launching Kernels

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

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

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

Exercise 3: Reverse Array, Single Block

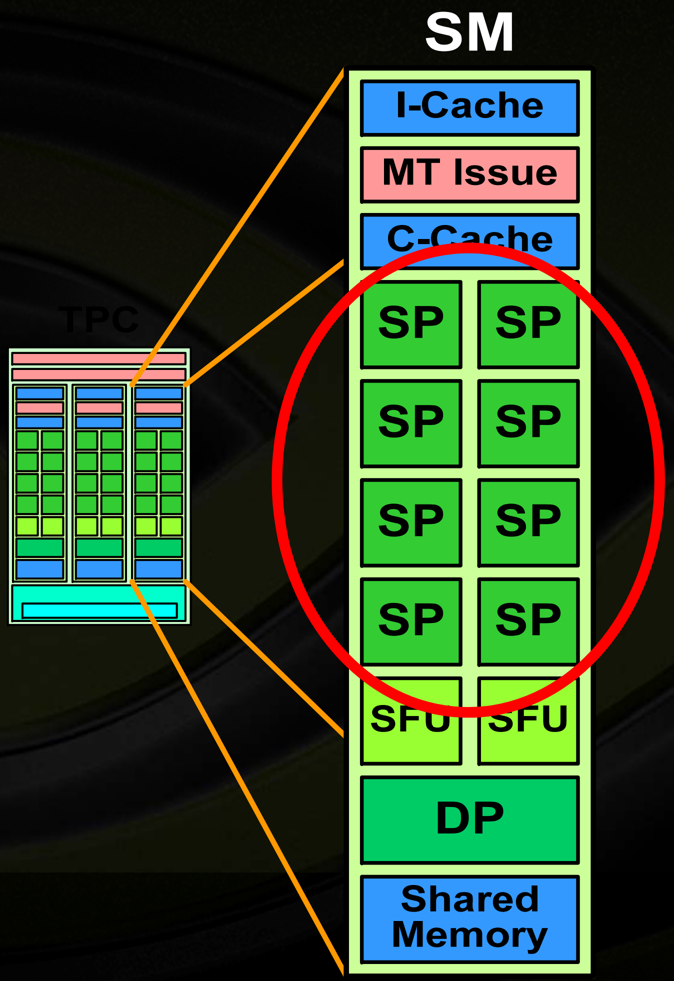
- Given an input array $\{a_0, a_1, \ldots, a_{n-1}\}$ in pointer *d_a,* store the reversed array $\{a_{n-1}, a_{n-2}, \ldots, a_0\}$ in pointer *d_b*
- Start from the "reverseArray_singleblock" template
- Only one thread block launched, to reverse an array of size N = numThreads = 256 elements
- Part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

Exercise 4: Reverse Array, Multi-Block

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

Performance Considerations

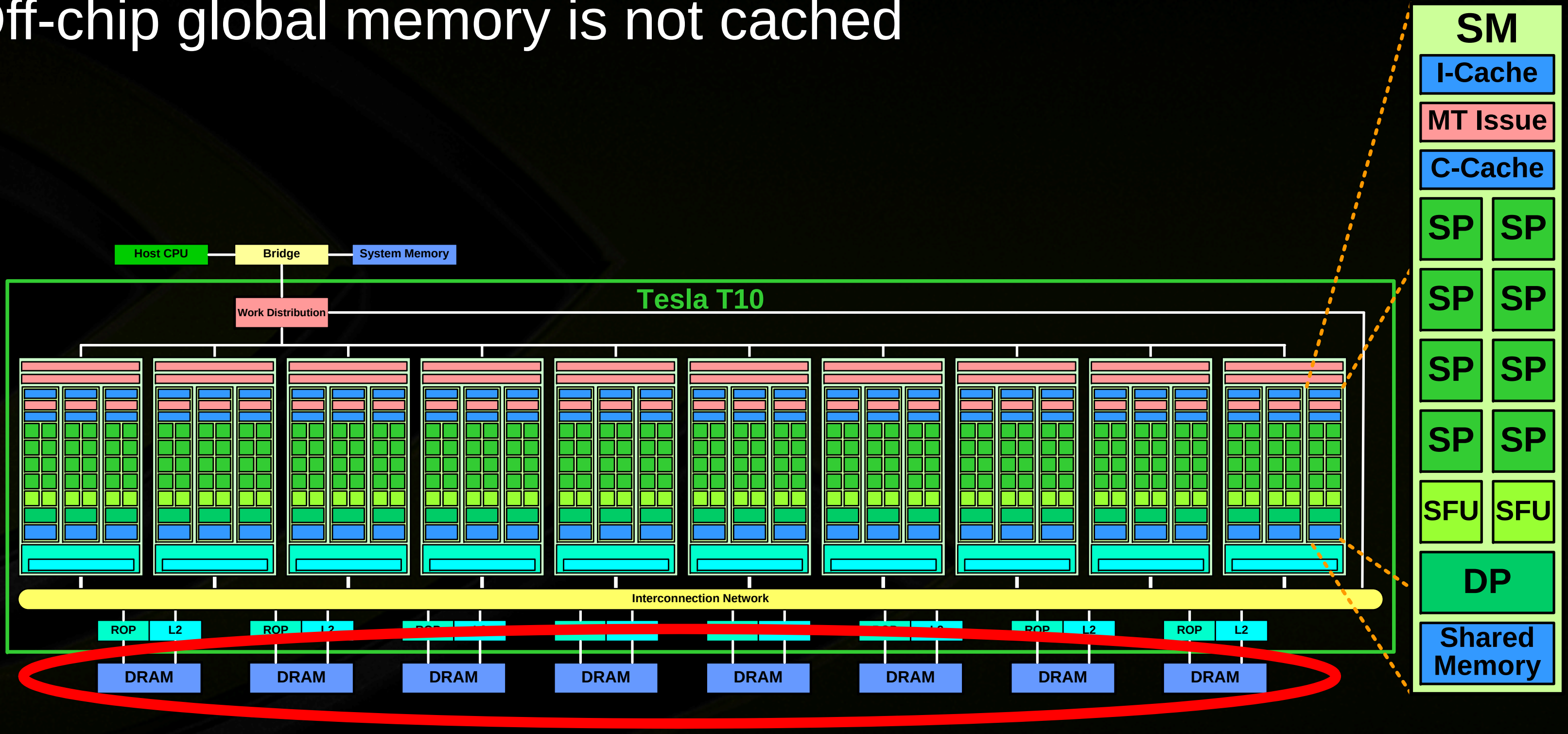
Single-Instruction, Multiple-Thread Execution



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

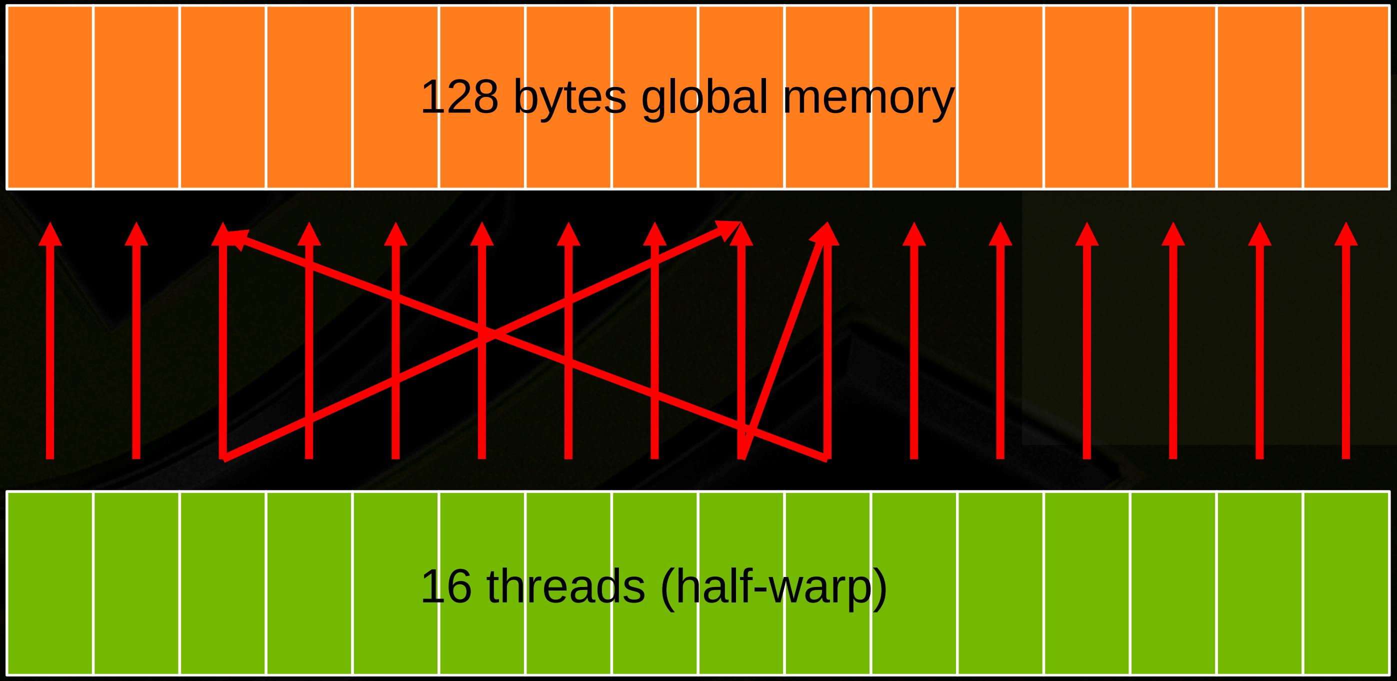
Global Memory

Off-chip global memory is not cached



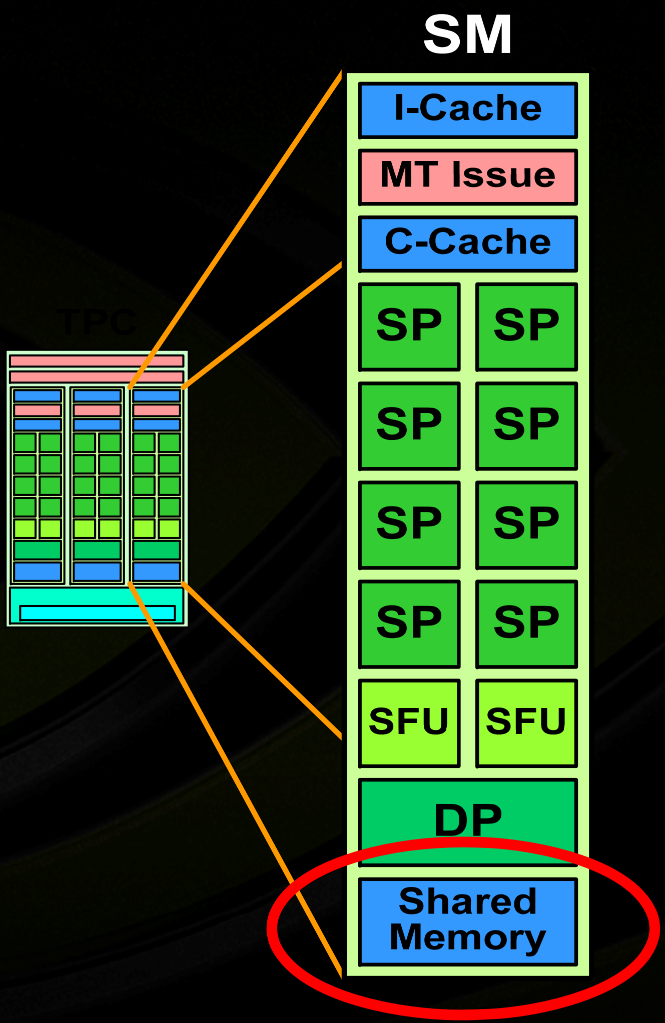
Efficient Access to Global Memory

Single memory transaction (coalescing) for some memory addressing patterns



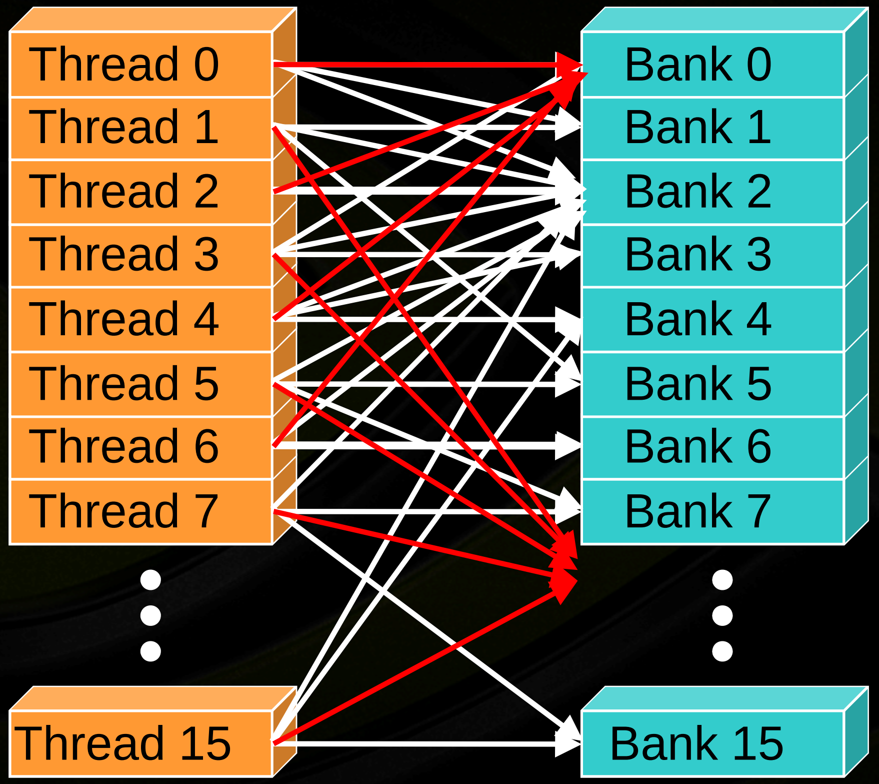
- Linear pattern
- Not all need participate
- Anywhere in block OK

Shared Memory



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

Shared Memory Bank Conflicts



- **Successive 32-bit words assigned to different banks**
- **Simultaneous access to the same bank by threads in a half-warp causes conflict and serializes access**
- **Linear access pattern**
- **Permutation**
- **Broadcast (from one address)**
- **Conflict, stride 8**

Exercise 5: Optimize Reverse Array

- Array reversal has a performance problem
- Use the CUDA visual profiler to run the code
- Look at
  - GLD_INCOHERENT
  - GST_INCOHERENT
  - WARP_SERIALIZE
- Take a note of GPU Time

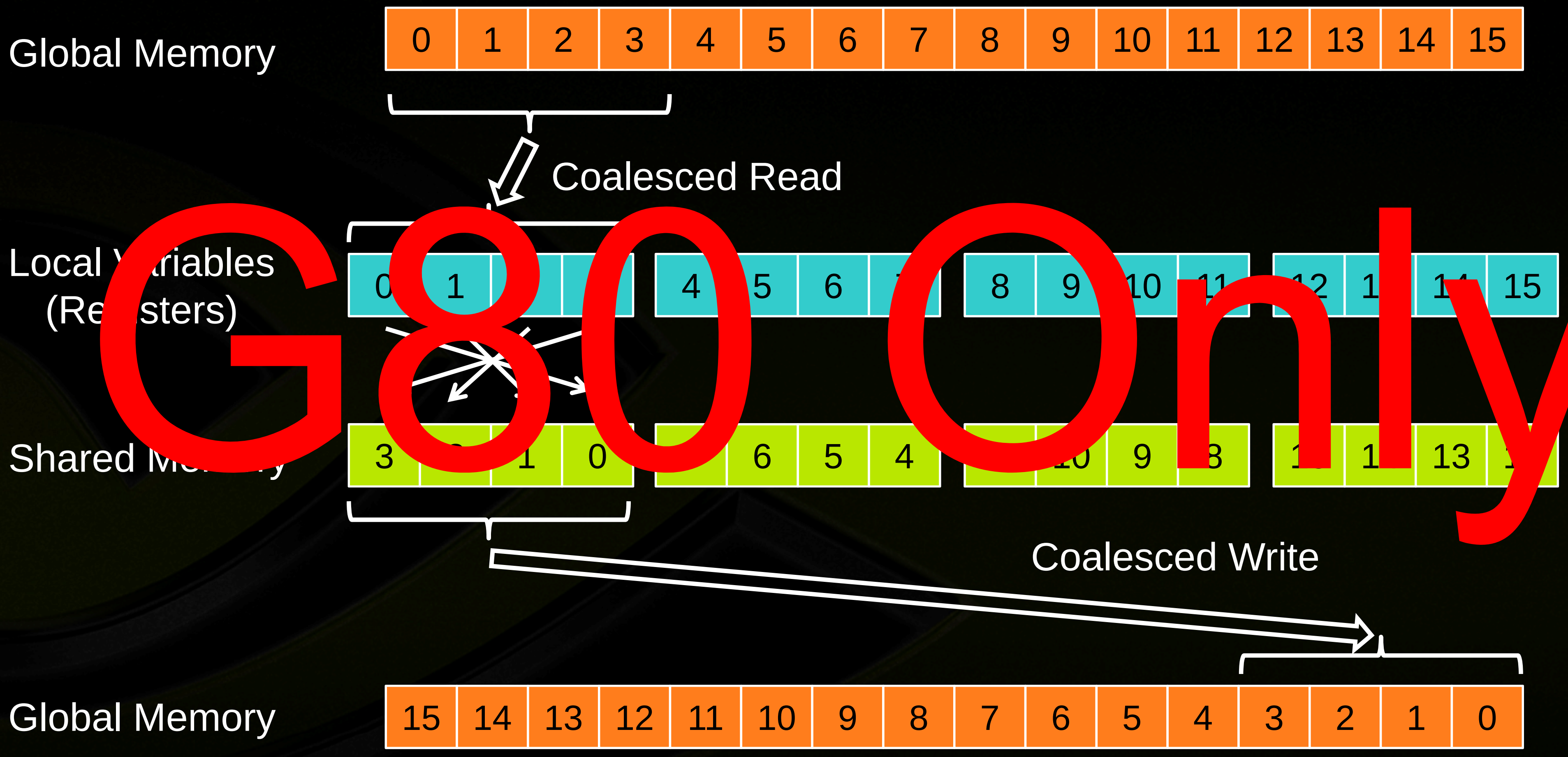
G80 Only

Exercise 5: Optimize Reverse Array

G80 Only

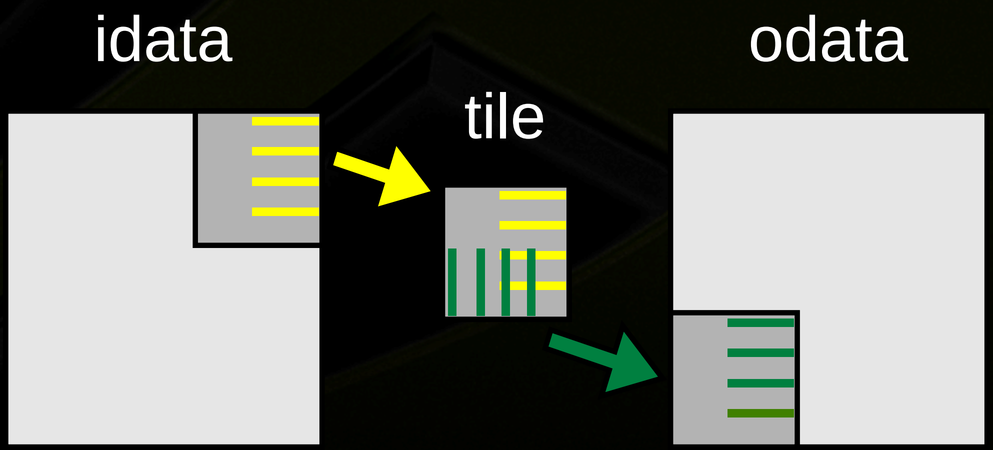
- Goal: Get rid of incoherent loads/stores and improve performance
- Use shared memory to reverse each block
- Part 1: compute the number of bytes of shared memory
  - One element per thread
- Part 2: implement the kernel
  - Comments should help
  - Don’t forget to compute the correct block offset!
- Part 3: Profile the working code
  - Compare value of GLD/GST_INCOHERENT to previous
  - Compare GPU Time to previous

Reverse in Shared Memory



Matrix Transpose

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

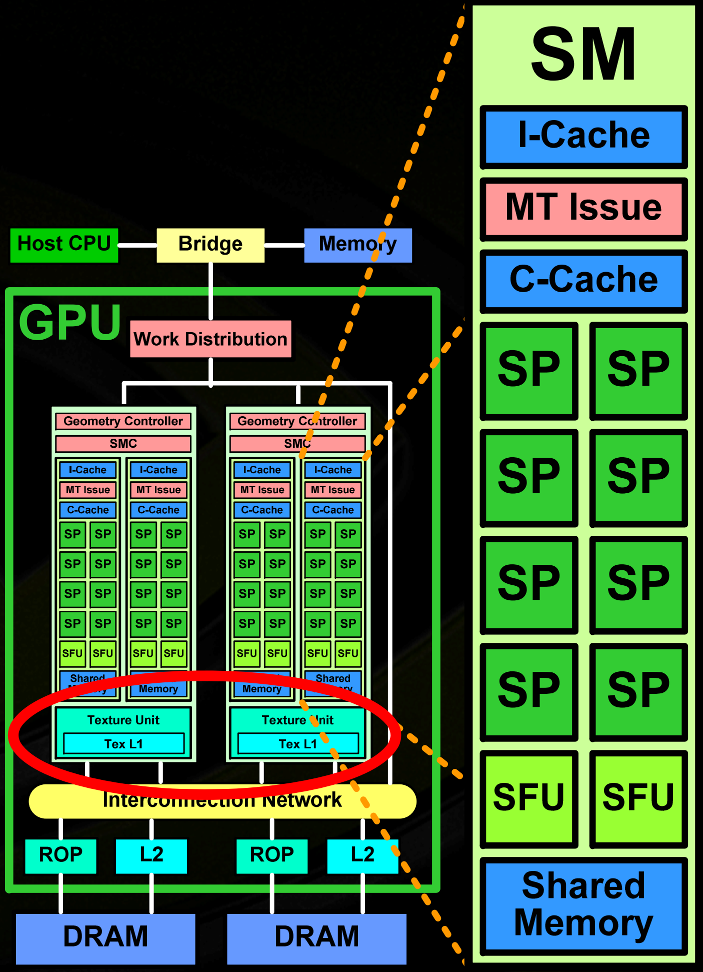


Matrix Transpose

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

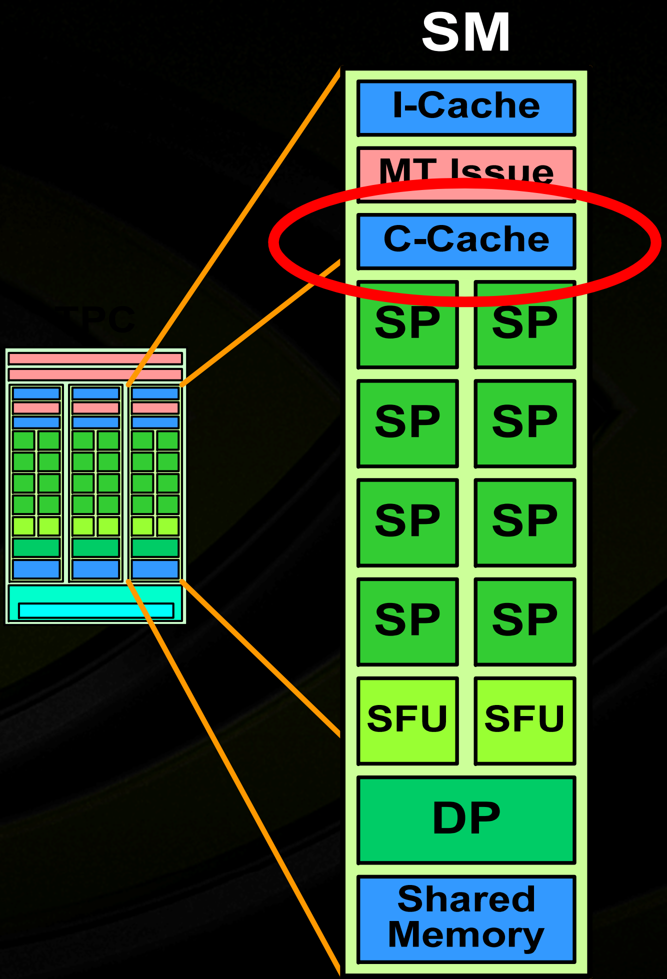
OTHER GPU MEMORIES

Texture Memory



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

Constant Memory



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

EXECUTION CONFIGURATION

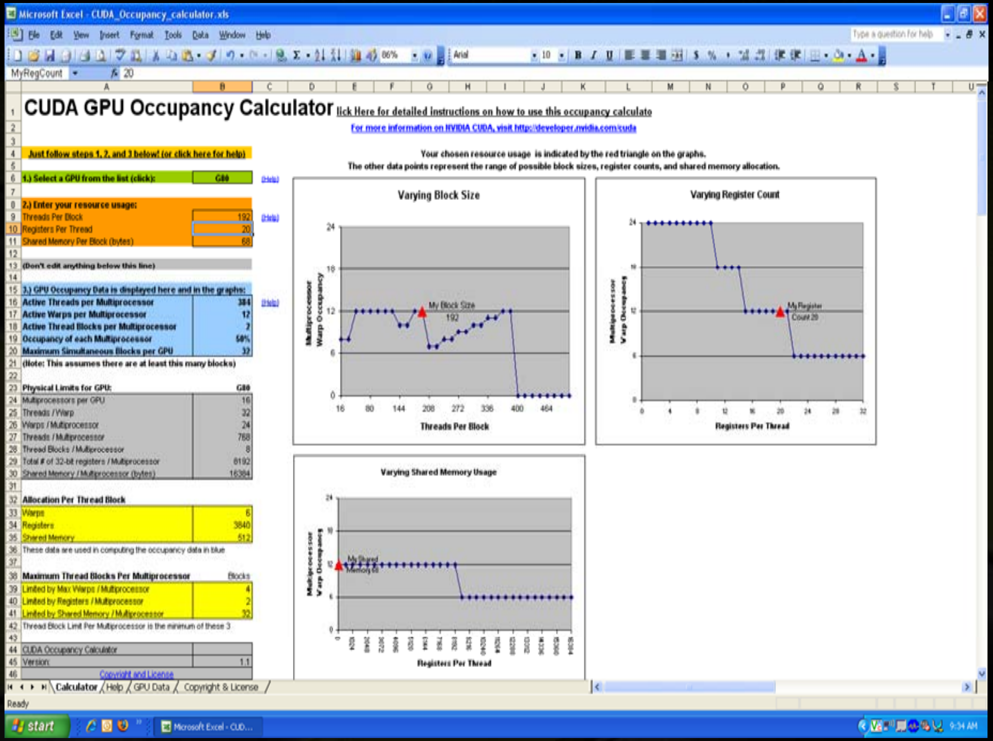
Execution Configuration

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

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

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

Occupancy Calculator



Microsoft Excel - CUDA_Occupancy_calculator.xls

MyRegCount 20

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

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

1.) Select a GPU from the list (click):	G80
2.) Enter your resource usage:	
Threads Per Block	192
Registers Per Thread	20
Shared Memory Per Block (bytes)	68
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	384
Active Warps per Multiprocessor	12
Active Thread Blocks per Multiprocessor	2
Occupancy of each Multiprocessor	50%
Maximum Simultaneous Blocks per GPU	32
(Note: This assumes there are at least this many blocks)	
Physical Limits for GPU:	G80
Multiprocessors per GPU	16
Threads / Warp	32
Warps / Multiprocessor	24
Threads / Multiprocessor	768
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	8192
Shared Memory / Multiprocessor (bytes)	16384
Allocation Per Thread Block	
Warps	6
Registers	3840
Shared Memory	512
These data are used in computing the occupancy data in blue	
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	2
Limited by Shared Memory / Multiprocessor	32
Thread Block Limit Per Multiprocessor is the minimum of these 3	
CUDA Occupancy Calculator	
Version:	1.1
Copyright and License	

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

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

DEBUGGING AND PROFILING

Debugging

- nvcc flags
  - –debug (-g)

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

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

  ```
  cuda-gdb a.out
  ```
- Usual gdb commands available

Debugging

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

Parallel Nsight 1.0

Nsight Parallel Debugger

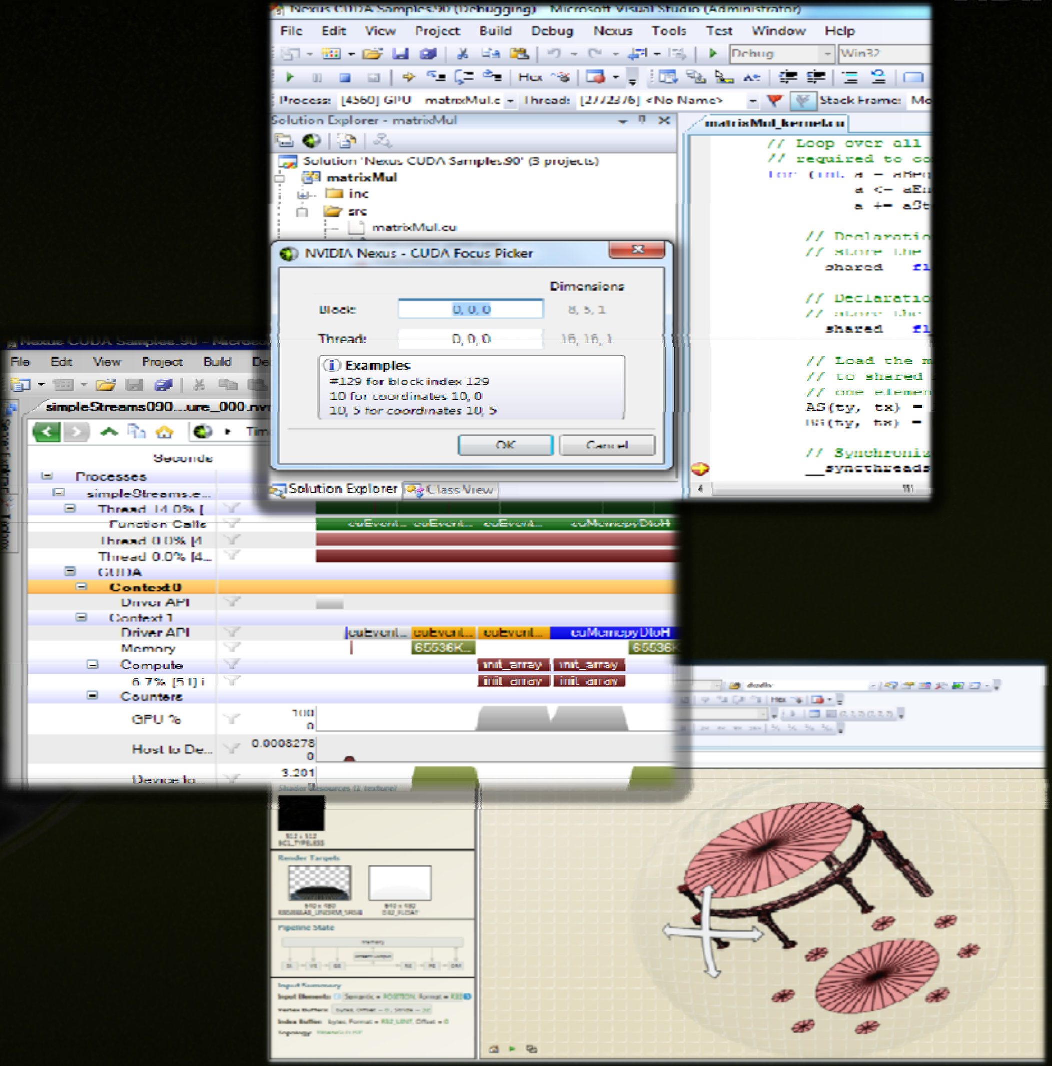
- GPU source code debugging
- Variable & memory inspection

Nsight Analyzer

- Platform-level Analysis
- For the CPU and GPU

Nsight Graphics Inspector

- Visualize and debug graphics content



Nexus CUDA Samples90 (Debugging) - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Nexus Tools Test Window Help
Solution Explorer - matrixMul
Solution 'Nexus CUDA Samples90' (3 projects)
matrixMul
inc
src
matrixMul.cu
NVIDIA Nexus - CUDA Focus Picker
Dimensions
Block: 0, 0, 0 8, 5, 1
Thread: 0, 0, 0 16, 16, 1
Examples
#129 for block index 129
10 for coordinates 10, 0
10, 5 for coordinates 10, 5
OK
Cancel
Solution Explorer
Class View
matrixMul_kernel.cu
simpleStreams090...ure_000.nvreport
Processes
Thread
Function Calls
CUDA
Context 0
Driver API
Context 1
Memory
Compute
Counters
GPU %
Host to De...
Device to...
cuEvent...
cuMemcpyDtoH
init_array

Allinea DDT

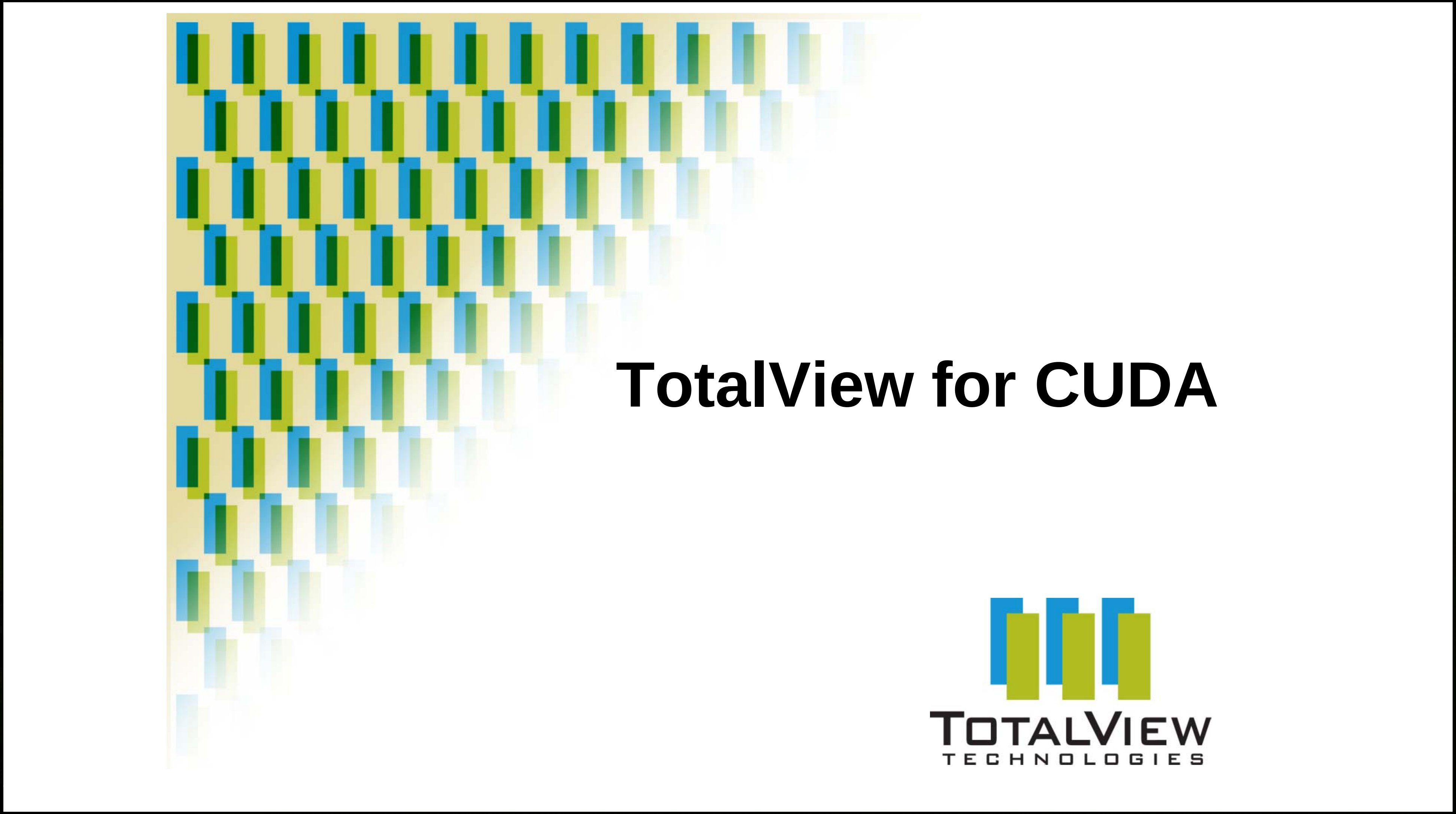
SCALE TO NEW HEIGHTS

GPU Debugging

Making it easy

Allinea DDT — CUDA Enabled

TotalView for CUDA

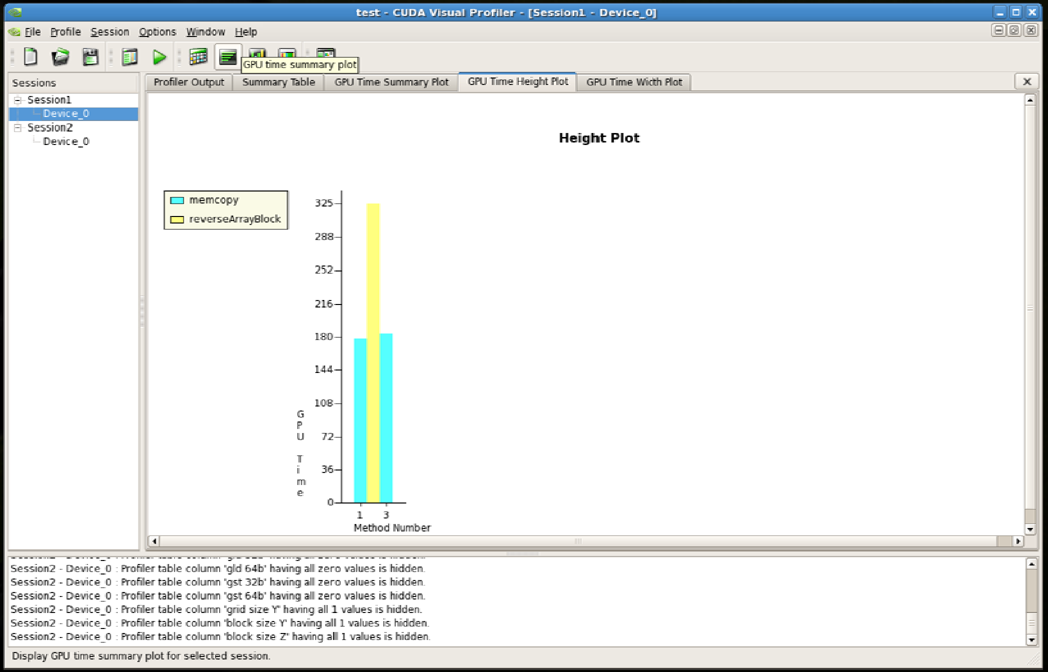


TotalView for CUDA

TOTALVIEW
TECHNOLOGIES

CUDA Visual Profiler

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



```
Session2 - Device_0 : Profiler table column 'gld 64b' having all zero values is hidden.
Session2 - Device_0 : Profiler table column 'gst 32b' having all zero values is hidden.
Session2 - Device_0 : Profiler table column 'gst 64b' having all zero values is hidden.
Session2 - Device_0 : Profiler table column 'grid size Y' having all 1 values is hidden.
Session2 - Device_0 : Profiler table column 'block size Y' having all 1 values is hidden.
Session2 - Device_0 : Profiler table column 'block size Z' having all 1 values is hidden.
Display GPU time summary plot for selected session.
```

CUDA Visual Profiler

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

MISCELLANEOUS TOPICS

Expensive Operations

- 32-bit multiply; __mul24() and __umul24() are fast 24-bit multiplies
- sin(), exp() etc.; faster, less accurate versions are __sin(), __exp() etc.
- Integer division and modulo; avoid if possible; replace with bit shift operations for powers of 2
- Branching where threads of warp take differing paths of control flow

Host to GPU Data Transfers

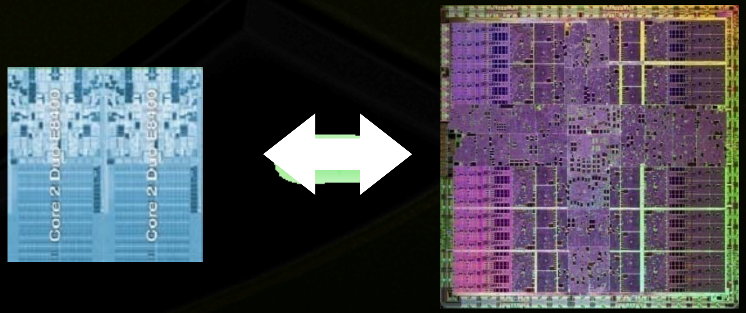
- PCI Express Gen2, 8 Gbytes/sec peak
- Use page-locked (pinned) memory for maximum bandwidth between GPU and host
- Data transfer host-GPU and GPU-host can overlap with computation both on host and GPU

Application Software (written in C)

CUDA Libraries

cuFFT | cuBLAS | cuDPP

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



4 cores

240 cores

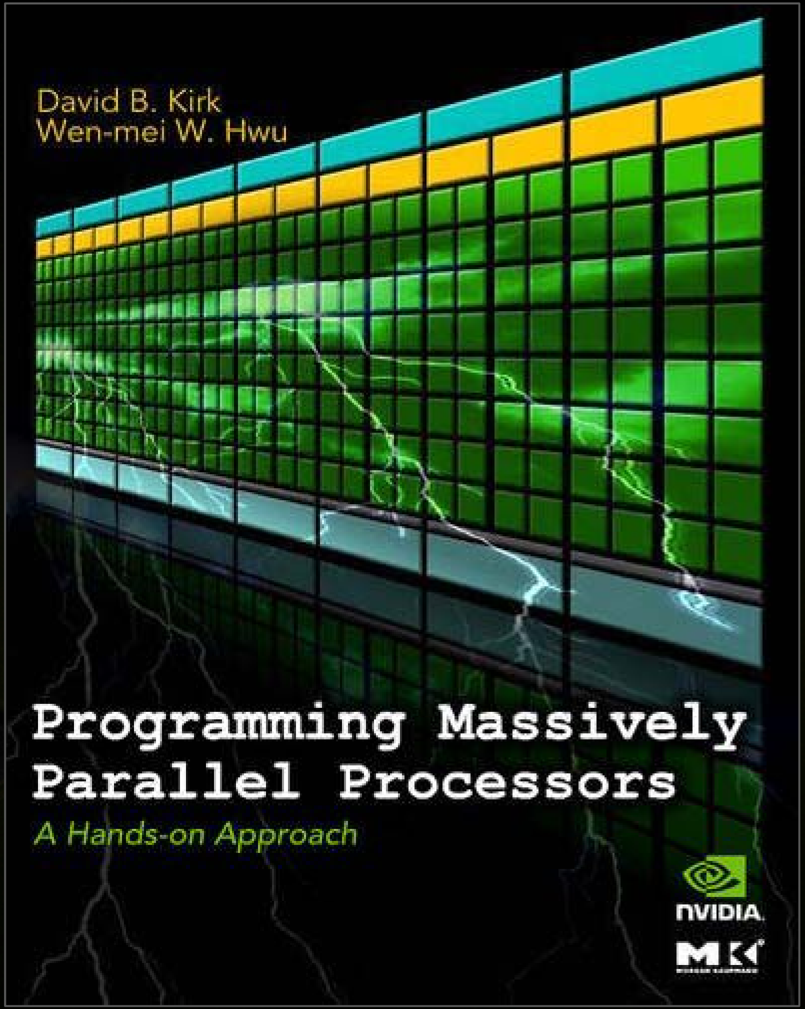
On-line Course

- Programming Massively Parallel Processors, Wen-Mei Hwu, University of Illinois at Urbana-Champaign http://courses.ece.illinois.edu/ece498/al/
- PowerPoint slides, MP3 recordings of lectures, draft of textbook by Wen-Mei Hwu and David Kirk (NVIDIA)

GPU Programming Text Book

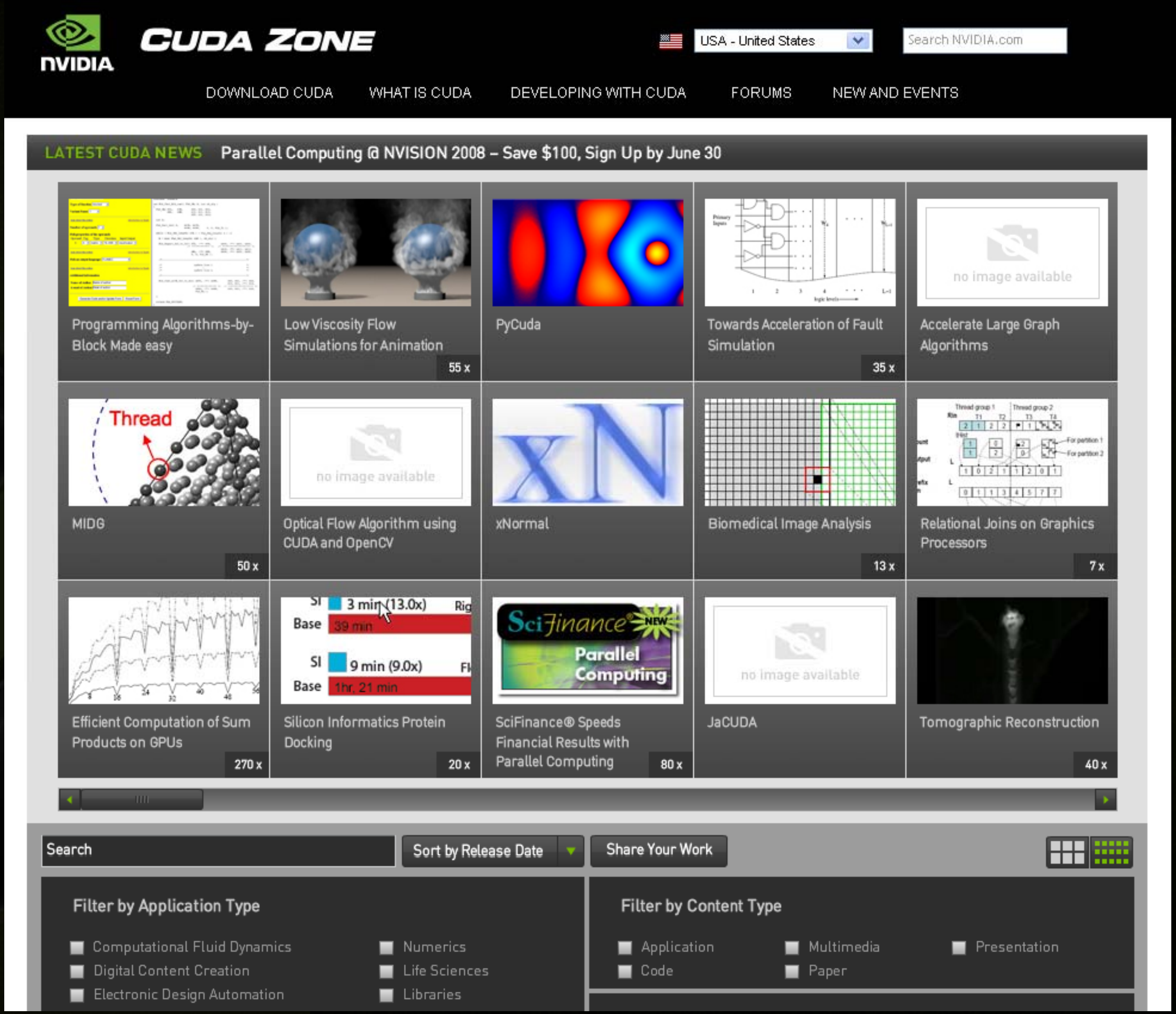
David Kirk (NVIDIA)
Wen-mei Hwu (UIUC)

Chapter 1: Introduction
Chapter 2: History of GPU Computing
Chapter 3: Introduction to CUDA
Chapter 4: CUDA Threads
Chapter 5: CUDA Memories
Chapter 6: Performance Considerations
Chapter 7: Floating-Point Considerations
Chapter 8: Application Case Study I - Advanced MRI Reconstruction
Chapter 9: Application Case Study II – Molecular Visualization and Analysis
Chapter 10: Parallel Programming and Computational Thinking
Chapter 11: A Brief Introduction to OpenCL
Chapter 12: Conclusion and Future Outlook
Appendix A: Matrix Multiplication Example Code
Appendix B: Speeds and feeds of current generation CUDA devices



CUDA Zone: www.nvidia.com/CUDA

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



LATEST CUDA NEWS Parallel Computing @ NVISION 2008 – Save $100, Sign Up by June 30

Programming Algorithms-by-Block Made easy	Low Viscosity Flow Simulations for Animation 55 x	PyCuda	Towards Acceleration of Fault Simulation 35 x	Accelerate Large Graph Algorithms
MIDG 50 x	Optical Flow Algorithm using CUDA and OpenCV	xNormal	Biomedical Image Analysis 13 x	Relational Joins on Graphics Processors 7 x
Efficient Computation of Sum Products on GPUs 270 x	Silicon Informatics Protein Docking 20 x	SciFinance® Speeds Financial Results with Parallel Computing 80 x	JaCUDA	Tomographic Reconstruction 40 x

Filter by Application Type
- Computational Fluid Dynamics
- Digital Content Creation
- Electronic Design Automation
- Numerics
- Life Sciences
- Libraries

Filter by Content Type
- Application
- Code
- Multimedia
- Paper
- Presentation