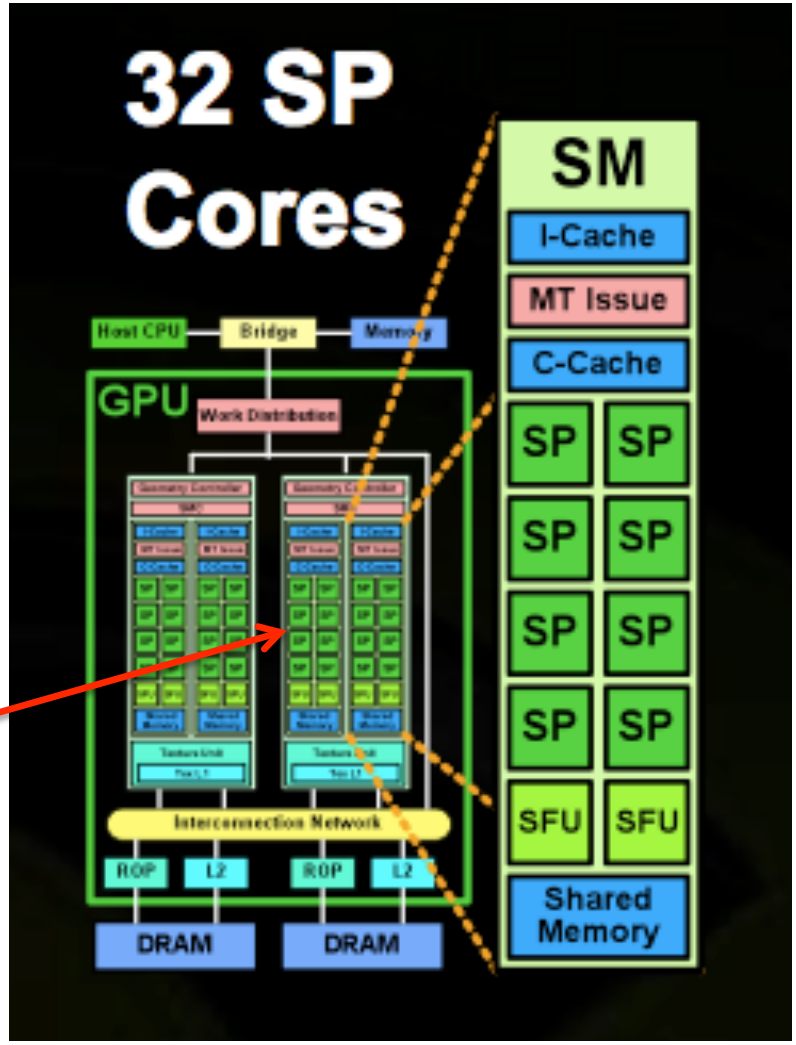
Parallel & Distributed Processing II: *parallel processing on manycore chips*

Nvidia GPUs and CUDA: Basics

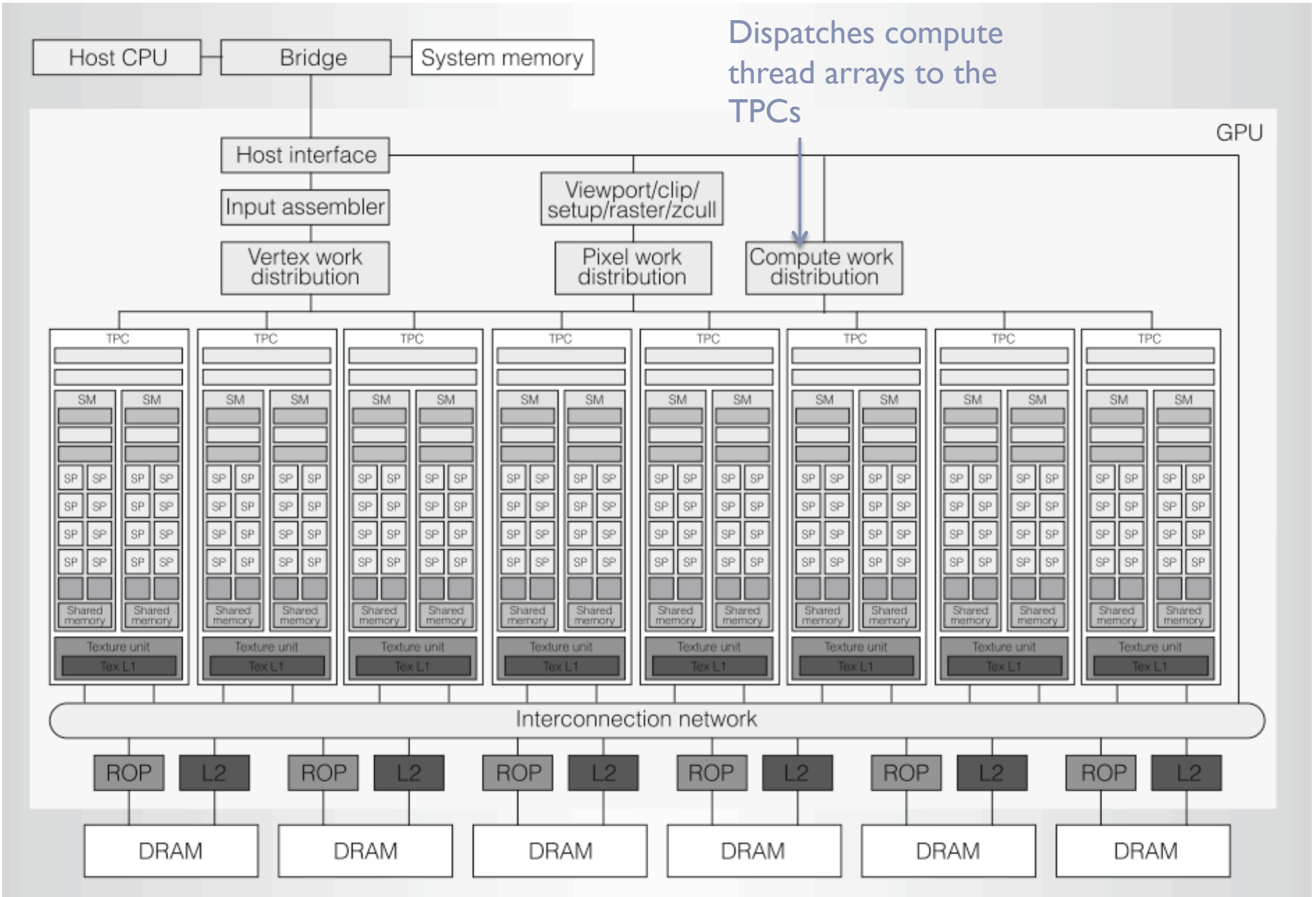
Eric Aubanel
Winter 2010, UNB Fredericton

NVIDIA Tesla Architecture

- Introduced Nov. 2006 with GeForce 8800
- Based on scalable processor array
  - Streaming processor (SP) cores grouped into streaming multiprocessors (SM)
  - SMs grouped in pairs of independent processing units, called texture/processor clusters (TPC)
  - GPUs range from 2 to 15 TPCs



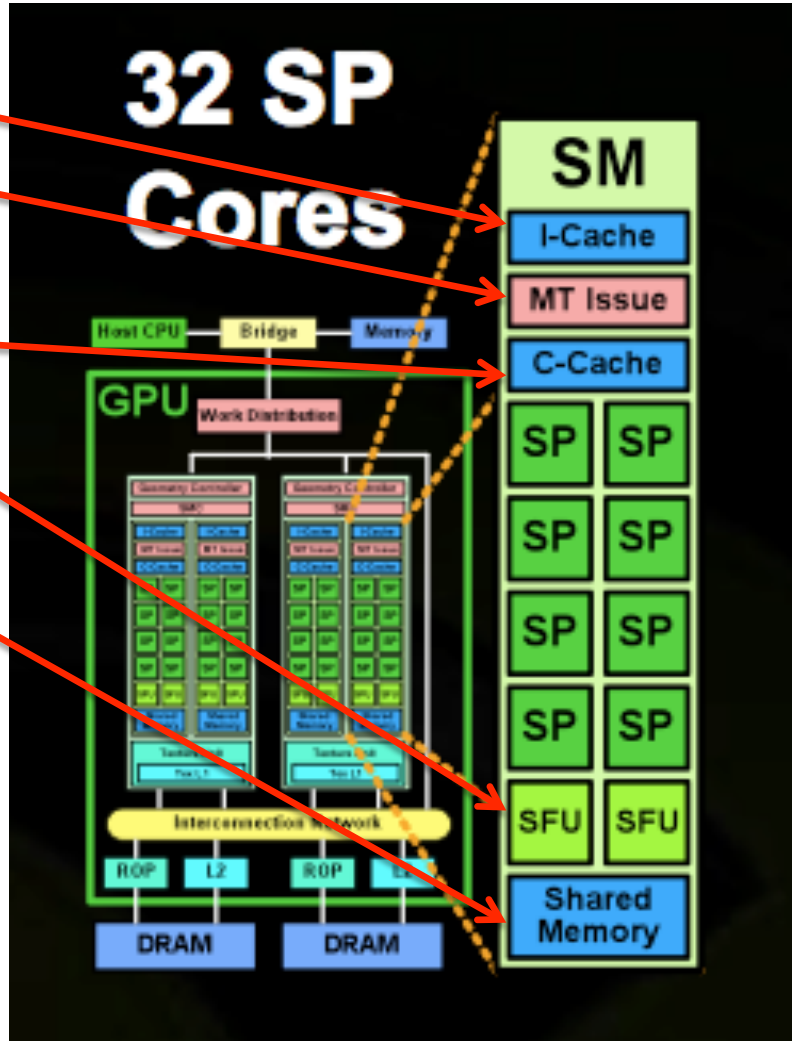
Source: Patrick LeGresley, NVIDIA



Source: IEEE Micro, March-April 2008

Streaming Multiprocessor

- Instruction cache
- Multithreaded instruction fetch & issue
- Read-only constant cache
- Special function units (SFU)
- 16KB read-write shared memory (low latency)
- SP: scalar multiply-add (MAD) & multiply floating point units
- SFU: transcendental functions, 4 floating point multipliers



Source: Patrick LeGresley, NVIDIA

SM flops

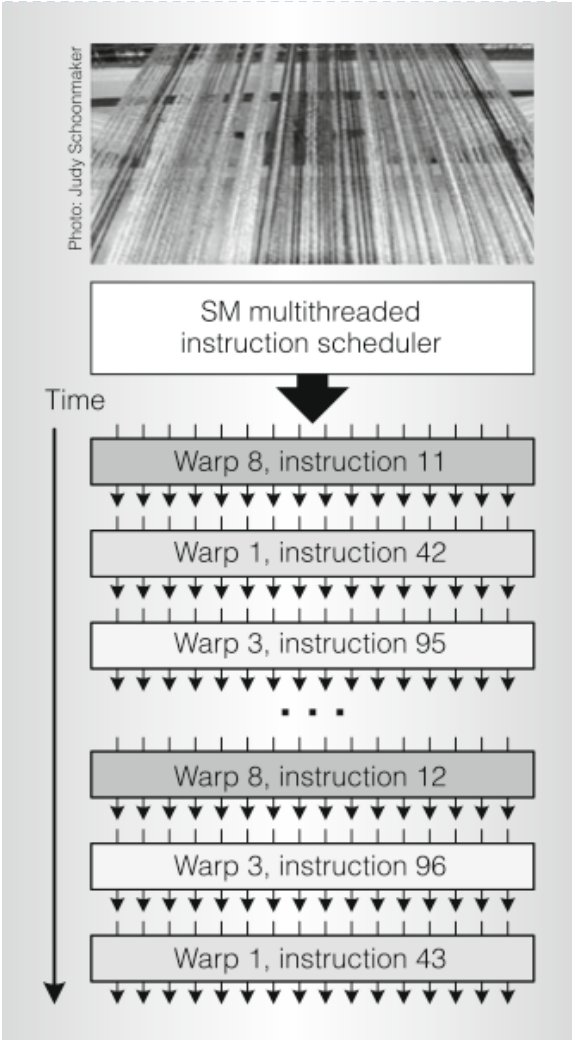
- 24 flop : 8 SPs, 1 MAD (2 flop) and 1 Mul each
- Performance in flops: clock speed X flop per cycle
  - GeForce GTX 285: 1.5 GHz, therefore 36 Gflops per SM, and 1 Tflop in total (30 SMs)

SM Multithreading

- Hardware multithreaded: up to 768 concurrent threads in hardware
- Each SM has own thread execution state and can execute independent code path
- Fine-grained parallelism within SM
  - Lightweight thread creation
  - Zero-overhead thread scheduling
  - Fast barrier synchronization

Warp Scheduling

- Threads scheduled in warps: groups of 32 threads
- SM manages pool of 24 warps
  - Total of 768
- Instruction is broadcast synchronously to warp's active parallel threads
  - Individual threads can be inactive, e.g. due to branching
- SM maps warp threads to SP cores
- Warp instruction issued as 2 sets of 16 threads over 4 cycles
- Scheduler prioritizes ready warps and selects one with highest priority



Source: IEEE Micro, March-April 2008

SM Instructions

- SM executes scalar instructions
- ISA
  - Floating-point, integer, bit, conversion, transcendental, flow control, memory load/store, texture operations
- Three memory spaces:
  - **Local** memory for temporary data
    - External DRAM
  - **Shared** memory for low-latency access for threads in same SM
  - **Global** memory for access by all threads of an application
    - External DRAM
- Local and global memory accesses from same warp coalesced into fewer memory block accesses

Compute Unified Device Architecture

- Software platform for parallel computing
- C plus a few simple extensions
  - write a program for one thread
  - instantiate for many parallel threads
  - familiar language; simple data-parallel extensions
- CUDA is a scalable parallel programming model
  - runs on any number of processors without recompiling
- Hides GPU hardware from developers
  - Still need to know low-level details intimately to get good performance, however
- Enables GPU architecture to change completely, transparently
  - Preserve investment in CUDA programs

CUDA Design Goals

- Support heterogeneous parallel programming (CPU + GPU)
- Scale to hundreds of cores, thousands of parallel threads
- Enable programmer to focus on parallel algorithms
  - not GPU characteristics, programming language, work scheduling ...

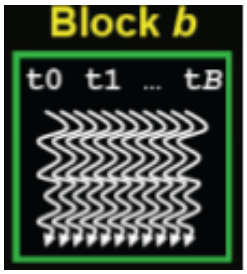
Source: John Mellor Crummey, Rice University

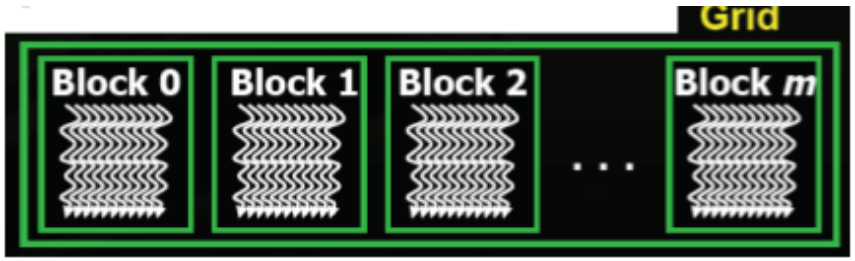
Key CUDA Abstractions

- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads

Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute same sequential program
  - use parallel threads rather than sequential loops
- Threads are grouped into thread blocks
  - threads in block can sync and share memory
- Blocks are grouped into grids
  - threads and blocks have unique IDs
    - threadIdx: 1D, 2D, or 3D
    - blockIdx: 1D or 2D
  - simplifies addressing when processing multidimensional data

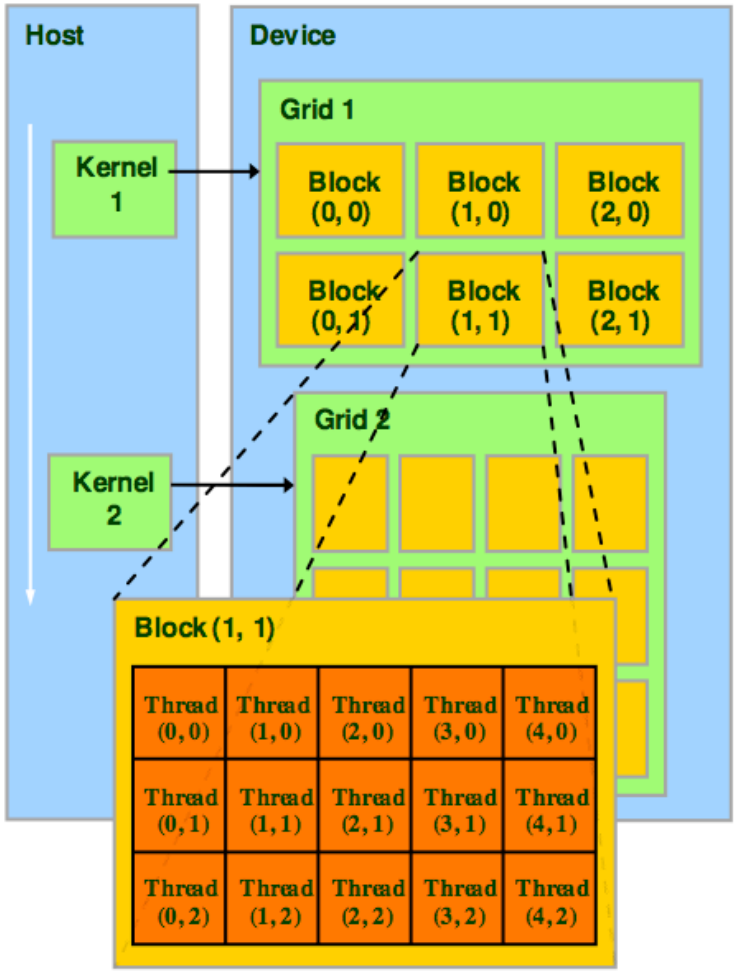




Source: Patrick LeGresley, NVIDIA

Thread Hierarchy: another view

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Source: Wen-Mei Hwu, U of Illinois and David Kirk, Nvidia

CUDA Programming Example

Computing y = ax + y with a serial loop

```
void saxpy_serial(int n, float alpha, float *x, float *y){
    for (int i = 0; i< n; i++)
        y[i] = alpha * x[i] + y[i];
}
// invoke serial saxpy kernel
saxpy_serial(n, 2.0, x_h, y_h)
```

Host code

Computing y = ax + y in parallel using CUDA

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

Device code

```
// invoke parallel saxpy kernel (256 threads per block)
int nblocks = (n + 255)/256;
saxpy_parallel<<<nblocks,256>>>(n,2.0,x_d,y_d);Host code
```

Synchronization and Coordination

- Threads within a block may synchronize with barriers

  .. step 1 ...

  _syncthreads();

  ... step 2 ...

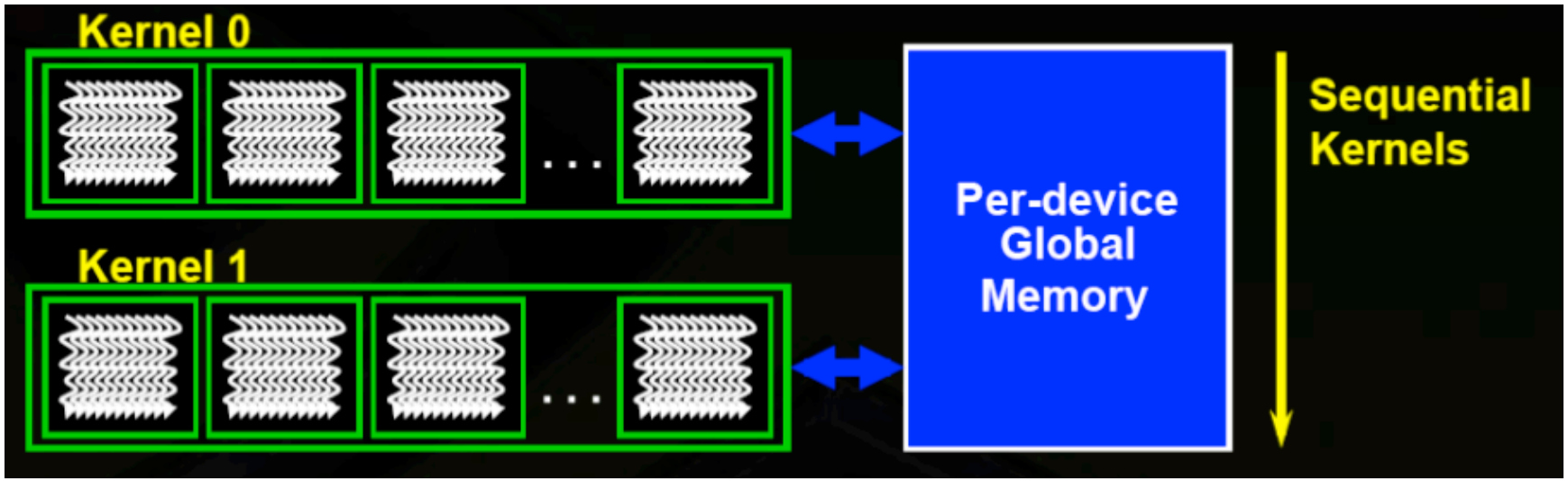
- Blocks can coordinate via atomic memory operations
  - e.g. increment shared queue pointer with atomicInc()
- Implicit barrier between kernels launched by host

  vec_minus<<<nblocks, blksize>>>(a, b, c)

  vec_dot<<<nblocks, blksize>>>(c, c)

Source: Patrick LeGresley, NVIDIA

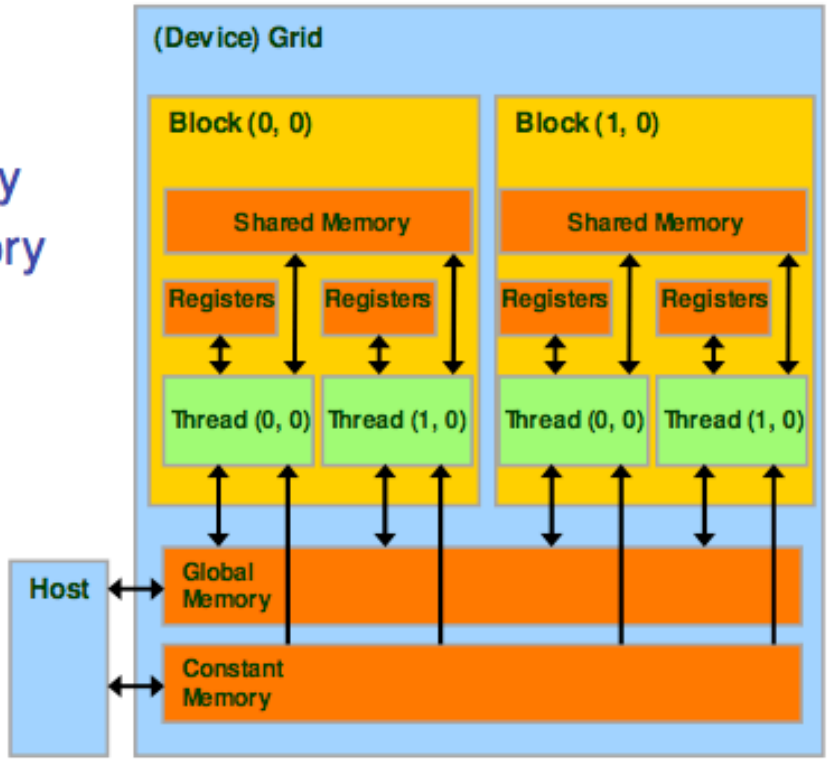
CUDA Memory Model



Source: Patrick LeGresley, NVIDIA

CUDA Memory Model

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - R/W per grid global and constant memories



Source: Wen-Mei Hwu, U of Illinois and David Kirk, Nvidia

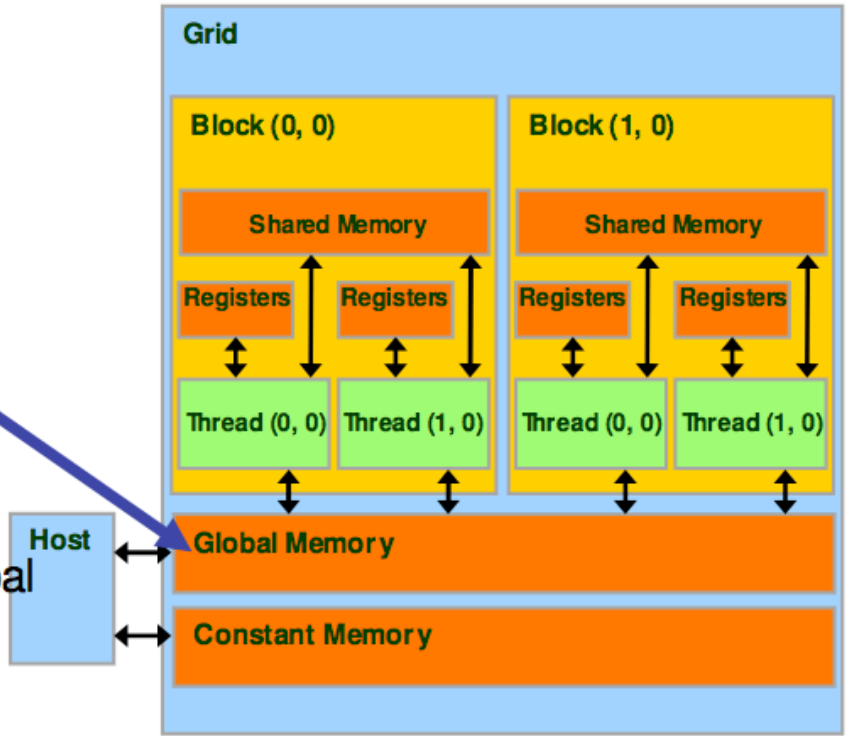
Memory Access Latencies

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles,
  - depends on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles
  - depends on cache locality

Source: John Mellor Crummey, Rice University

Global Memory Management

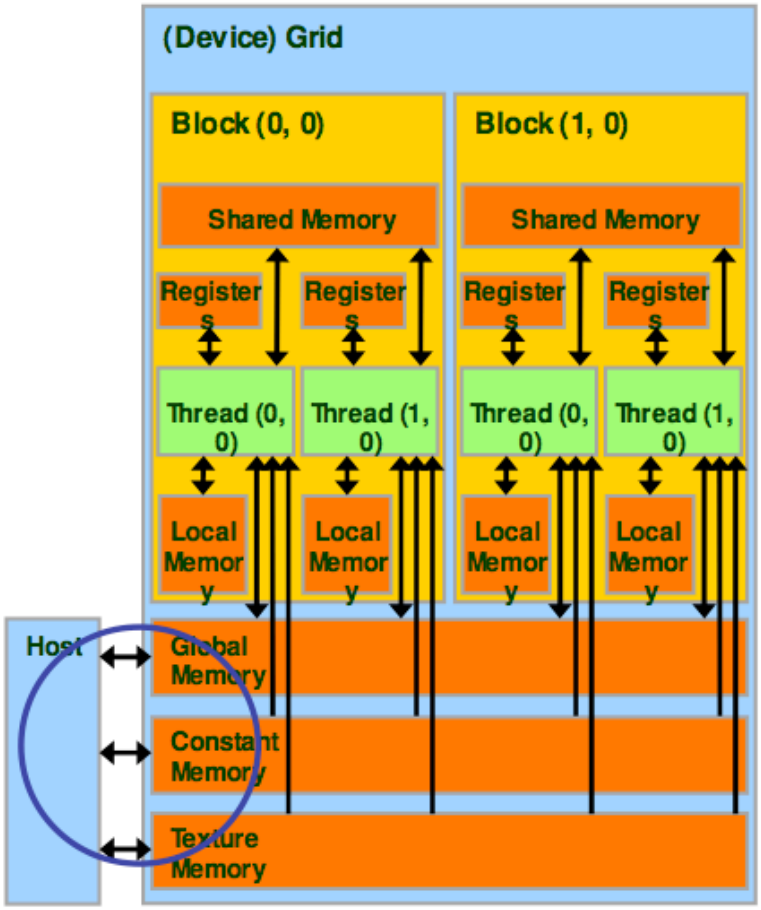
- cudaMalloc()
  - Allocates object in the device Global Memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - **Pointer** to freed object



Source: Wen-Mei Hwu, U of Illinois and David Kirk, Nvidia

cudaMemcpy

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Transfer is asynchronous



Source: Wen-Mei Hwu, U of Illinois and David Kirk, Nvidia

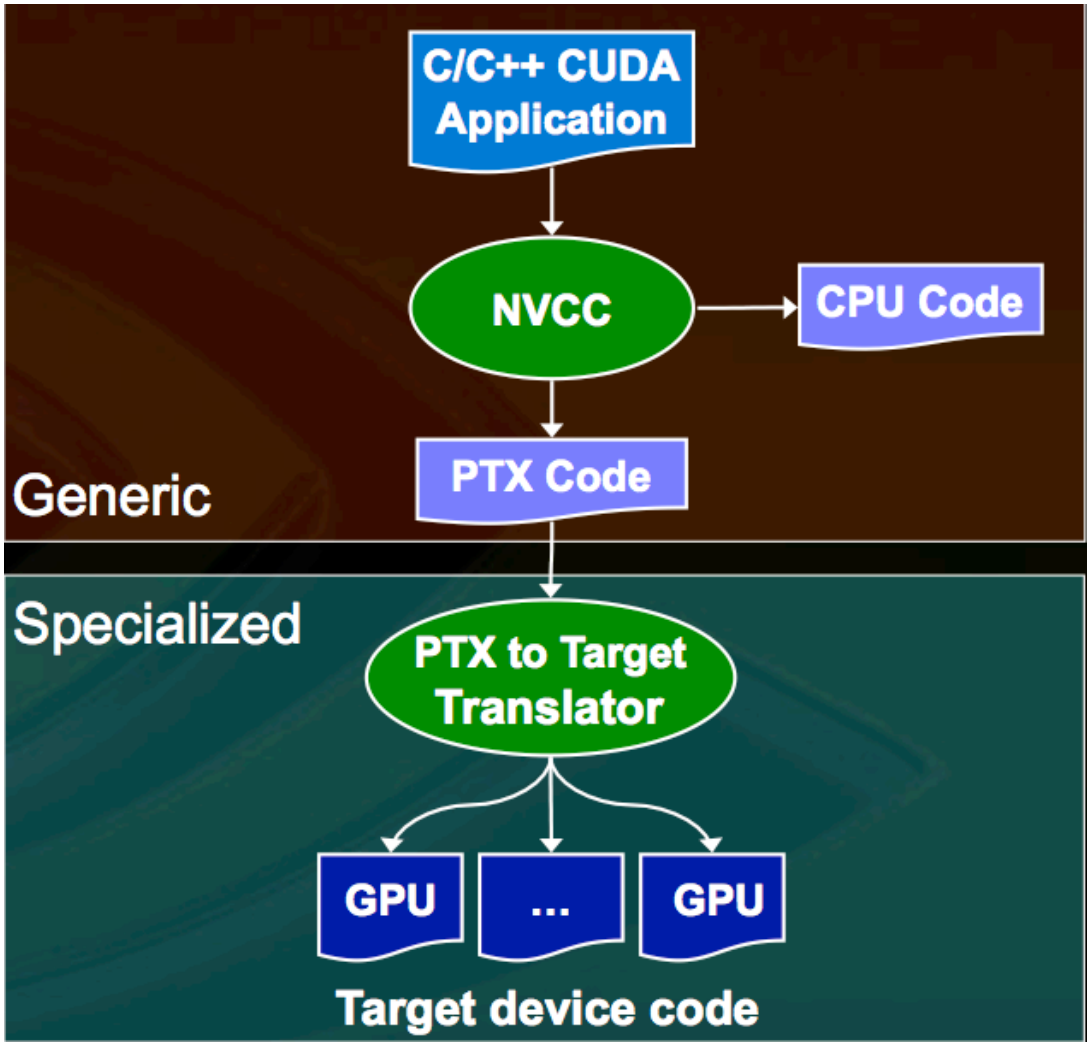
Saxpy example, host code

```
size_t size = n*sizeof(float);
// allocate memory on host
x_h = (float *)malloc(size); // also y_h, z_h
// allocate memory on device
cudaMalloc((void **) &x_d, size);
cudaMalloc((void **) &y_d, size);
// initialize vector x_h & y_h ...
// copy vectors to device
cudaMemcpy(x_d, x_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(y_d, y_h, size, cudaMemcpyHostToDevice);
// invoke parallel saxpy kernel (256 threads per block)
int nblocks = (n + 255)/256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x_d, y_d);
// copy solution vector to host
cudaMemcpy(z_h, y_d, size, cudaMemcpyDeviceToHost);
```

Host stub

1. Allocate device memory
   - `cudaMalloc(...)`
2. Copy host data to device
   - `cudaMemcpy(..., cudaMemcpyHostToDevice)`
3. Invoke device kernel
   - `function_name<<<nblocks, nthreads_per_block>>>(arg1, arg2, ...)`
4. Copy device data to host
   - `cudaMemcpy(..., cudaMemcpyDeviceToHost)`

Compiling CUDA



Source: Patrick LeGresley, NVIDIA