UiT

THE ARCTIC UNIVERSITY OF NORWAY

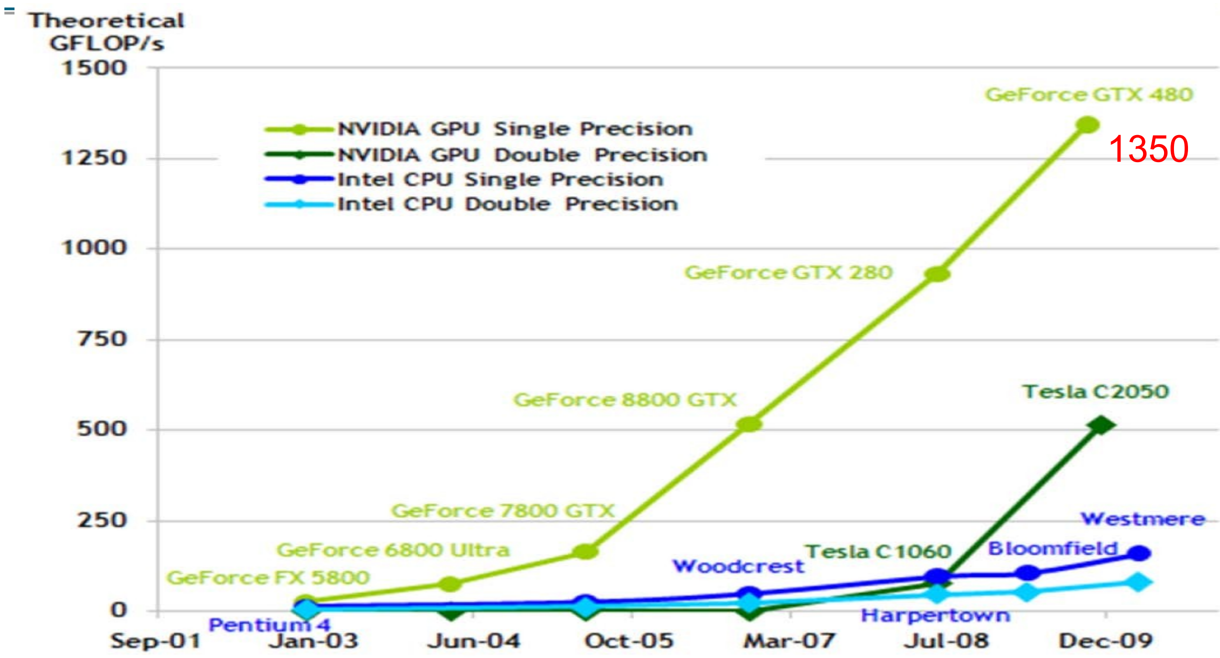
Lecture 9: Introduction to CUDA and OpenCL

Concurrent and Parallell Programming (INF-3201)

Autumn 2012

John Markus Bjørndalen (with slides by Phuong Ha & Lars Tiede)

GPUs: powerful, cheap and ubiquitous

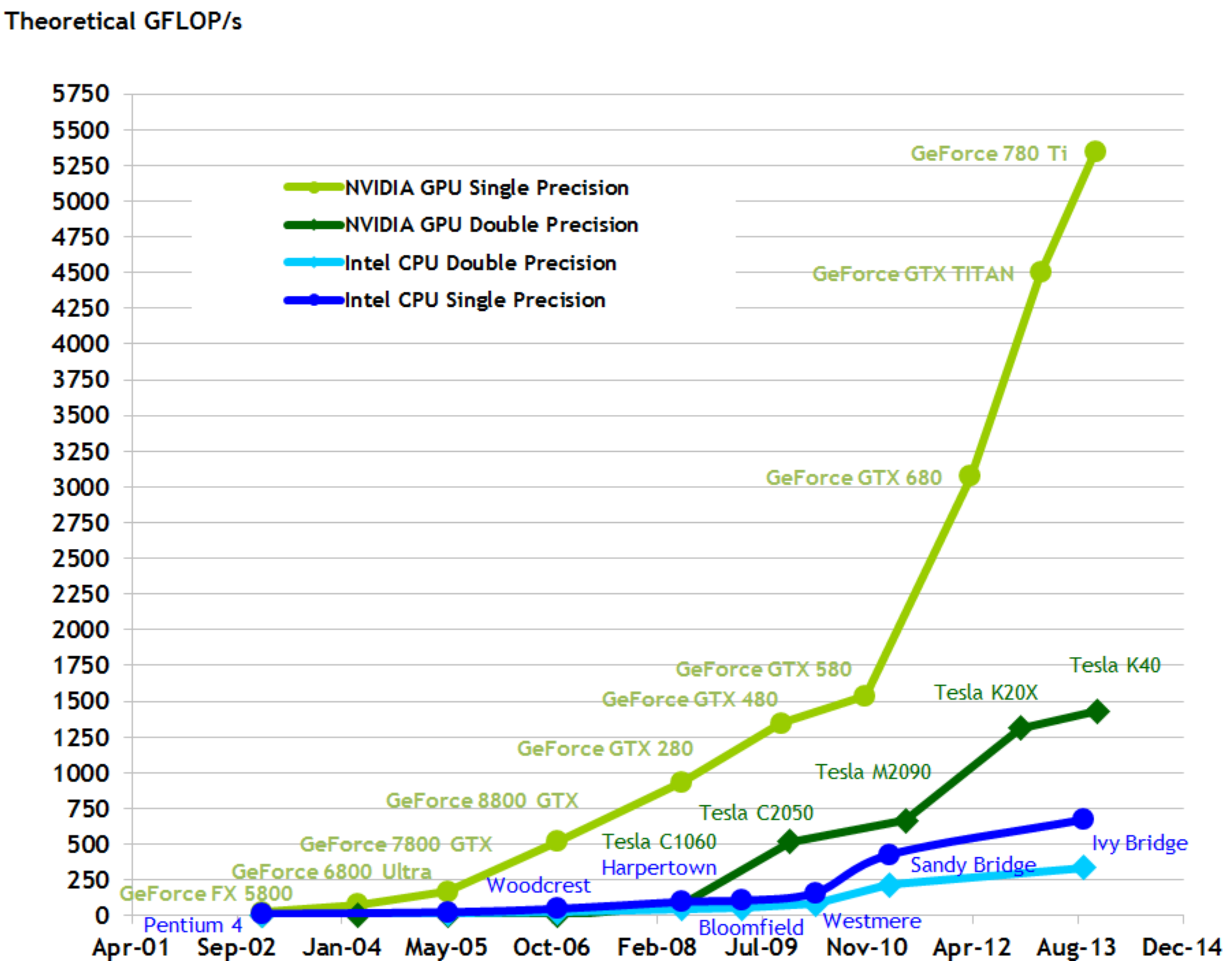


Transistor distribution

- GF GTX 480 in 2010
  - 1350 Gflops
  - ~ $500
  - 480 processors
  - 23k concurrent threads
- The top supercomputer in 1994 [top500.org]
  - 184 Gflops
- 180 million CUDA GPUs sold

Figures from NVIDIA CUDA Programming Guide, version 3.1

GPUs: powerful, cheap and ubiquitous

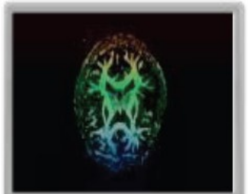


Figures from NVIDIA CUDA Programming Guide, version 7.5
https://docs.nvidia.com/cuda/cuda-c-programming-guide/

A few years later.

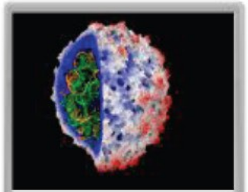
- GF GTX 480 in 2010
  - 448 CUDA cores
  - 1350 Gflops
  - 133.9 GB/sec memory B/W
- Geforce 780 Ti
  - 2880 CUDA cores
  - ~5Tflops
  - 336 GB/s memory B/W
- The top supercomputer [top500.org]
  - 1994: 184 Gflops/140 cores
  - 2000: 4.9 Tflops/8192 cores
  -

Applications in several fields



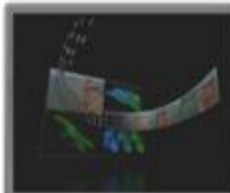
146X

Interactive visualization of volumetric white matter connectivity



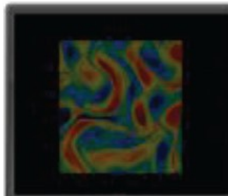
36X

Ionic placement for molecular dynamics simulation on GPU



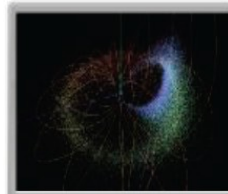
19X

Transcoding HD video stream to H.264



17X

Simulation in Matlab using .mex file CUDA function



100X

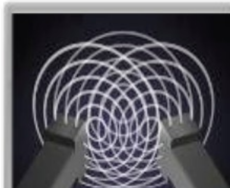
Astrophysics N-body simulation

149X

Financial simulation of LIBOR model with swaptions

47X

GLAME@lab: An M-script API for linear Algebra operations on GPU

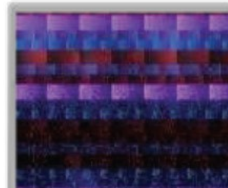


20X

Ultrasound medical imaging for cancer diagnostics

24X

Highly optimized object oriented molecular dynamics



30X

Cmatch exact string matching to find similar proteins and gene sequences

M02: High Performance Computing with CUDA

29

http://www.gpgpu.org/
http://www.nvidia.com/object/cuda_home.html#,

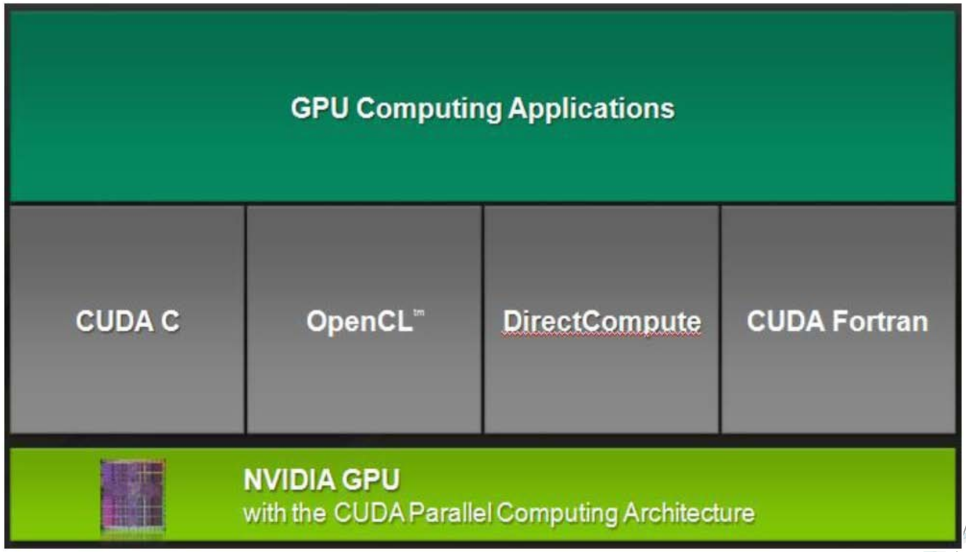
Source: Massimiliano Fatica, 2008

Outline

- Motivations for general-purpose computation on GPU (GPGPU)
- CUDA
  - CUDA programming model overview
  - CUDA C programming basics
- OpenCL

What is CUDA?

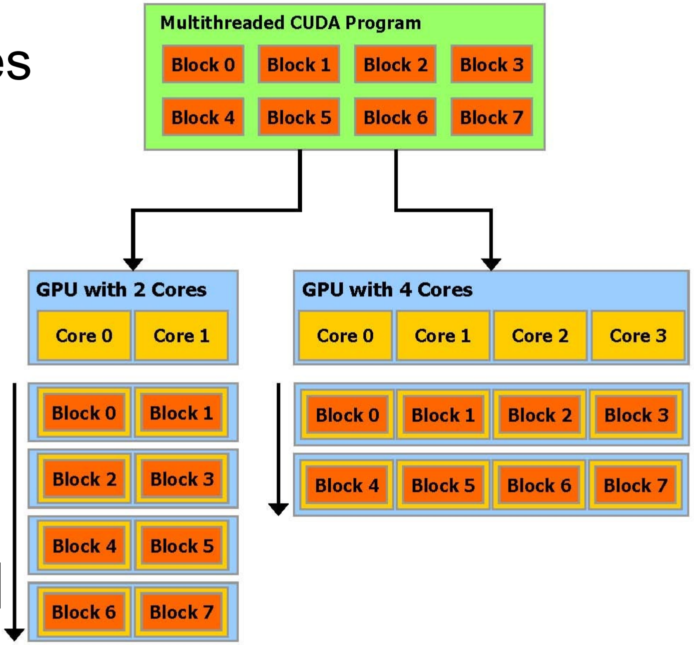
- CUDA = Compute Unified Device Architecture
- A general purpose parallel computing architecture for Nvidia GPUs with
  - A new parallel programming model
  - A software environment supporting standard languages and APIs



Figures from NVIDIA CUDA Programming Guide, version 3.1

Why use CUDA?

- Scalability
  - Transparently scale parallelism to #cores
  - Scale to 100-1000s of cores, 1000s of parallel threads
- Not-so-steep learning curve
  - Small set of extensions to C language
  - Let programmers focus on parallel algorithms
- Heterogeneous serial-parallel model (i.e. CPU + GPU)



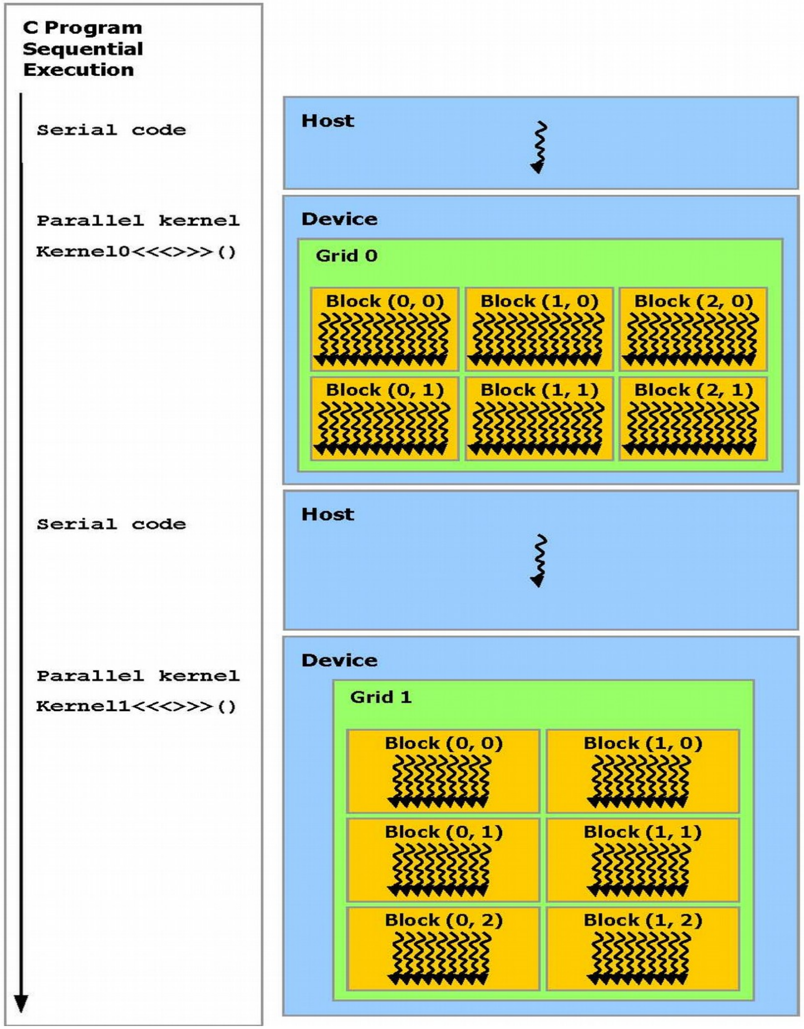
Figures from NVIDIA CUDA Programming Guide, version 3.1

Outline

- Motivations for general-purpose computation on GPU (GPGPU)
- CUDA
  - CUDA programming model overview
  - CUDA programming basics
- OpenCL

CUDA programming model overview

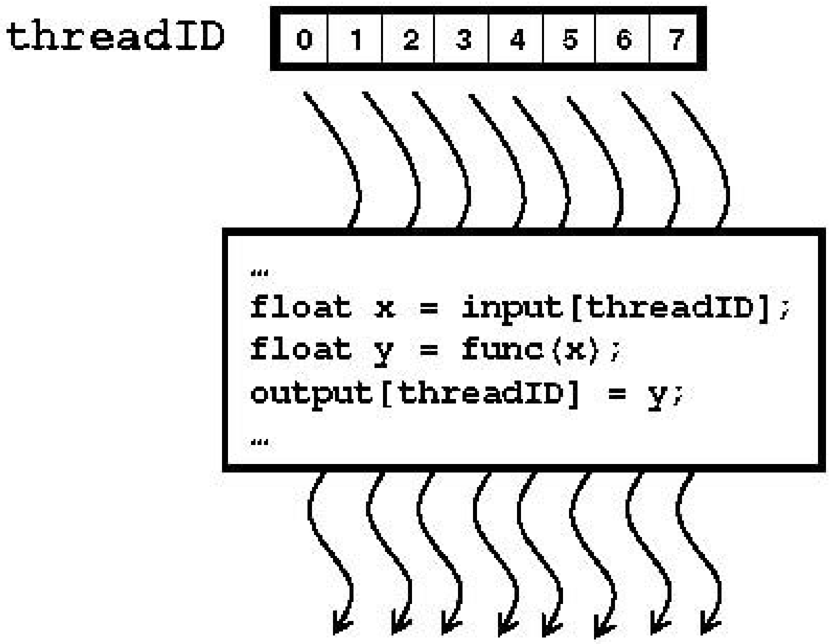
- Some nomenclature
  - Device = GPU
  - Host = CPU
  - Kernel = function running massively parallel on GPU
  - Many threads on the GPU execute each kernel
- CUDA vs. CPU threads
  - extremely lightweight
    - Small overhead for creation & switching
  - 1000s of threads to achieve efficiency



Figures from NVIDIA CUDA Programming Guide, version 3.1

Arrays of parallel threads

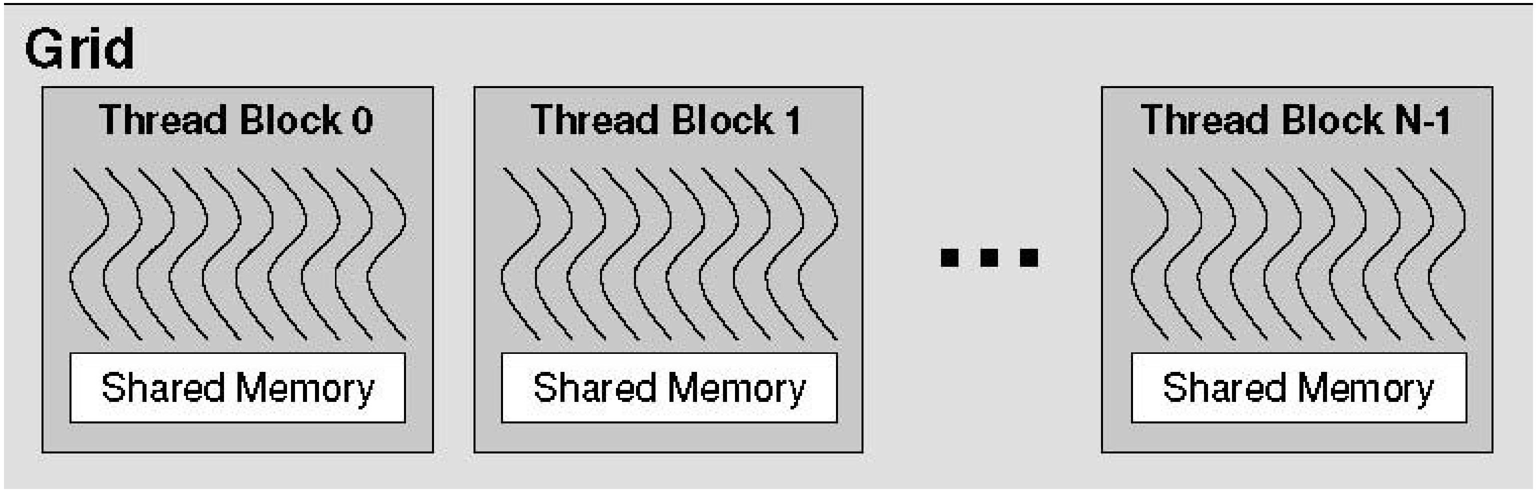
- A CUDA kernel is executed by an array of threads
  - Each thread run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



Figures from NVIDIA Introduction to CUDA Programming

Thread blocks

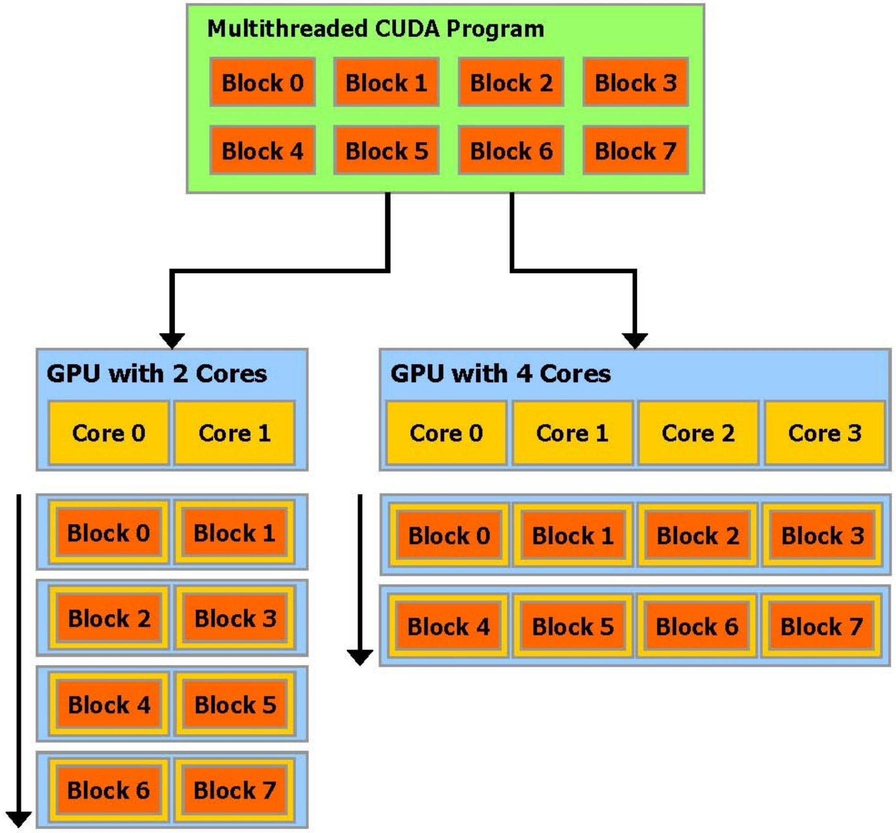
- Divide monolithic array of threads into blocks
  - Threads within a block cooperate efficiently via shared memory and barrier synchronization.
  - Cooperation within small blocks of threads is scalable
- Kernel launches a grid of thread blocks
  - Allows programs to transparently scale to different GPUs



Figures from NVIDIA Introduction to CUDA Programming

Transparent scalability

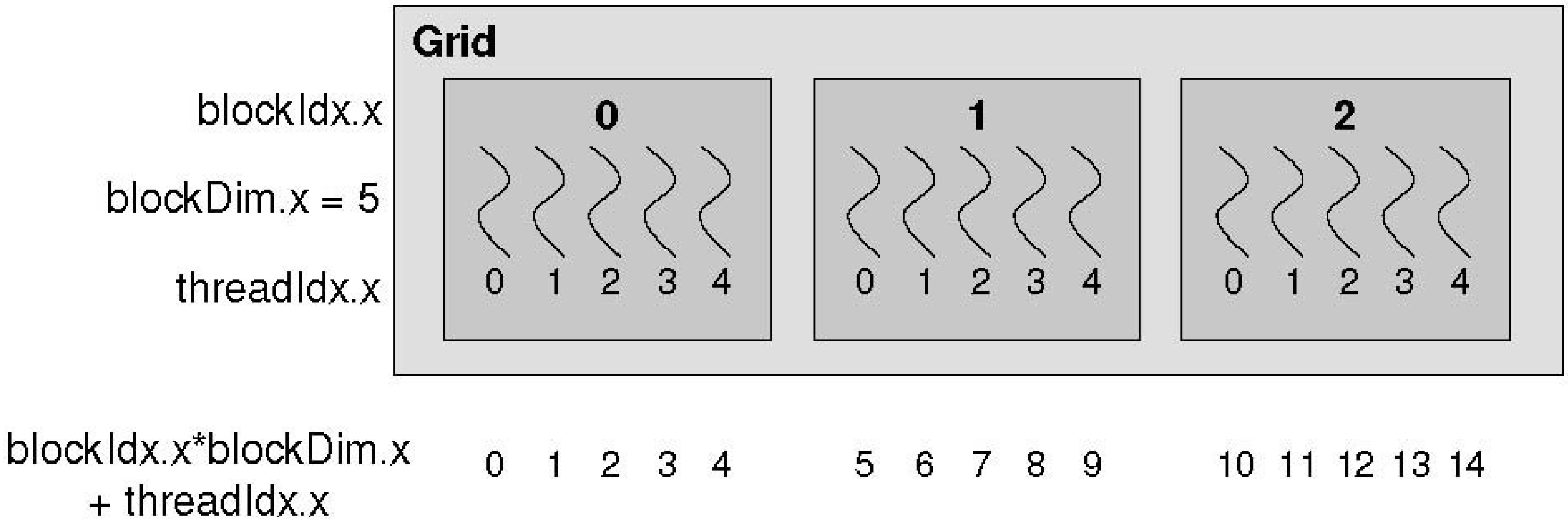
- Hardware is free to schedule thread blocks on any processor
  - Programmer: no assumption on execution order of thread blocks.



Figures from NVIDIA CUDA Programming Guide, version 3.1

Thread hierarchy

- Want each thread to access a different element of an array
- Each thread has access to
  - threadIdx.x: thread ID within block
  - blockIdx.x: block ID within grid
  - blockDim.x: #thread per block

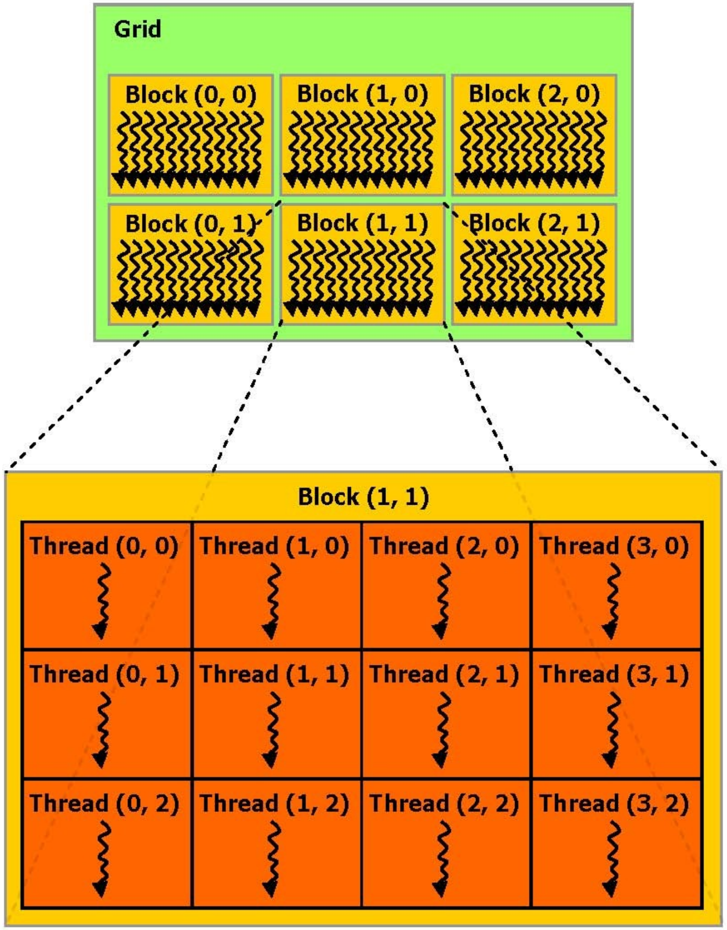


Figures from NVIDIA Introduction to CUDA Programming

Multidimensional IDs

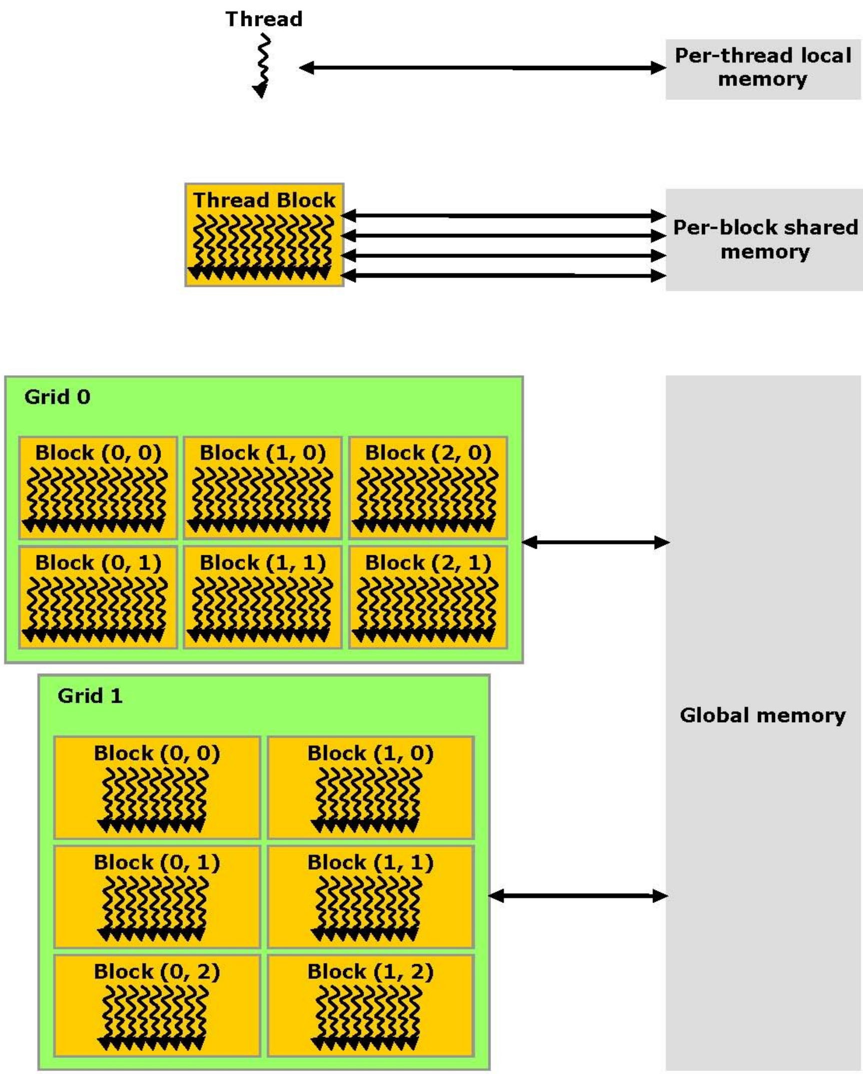
- Blocks
  - 1D or 2D IDs, unique within a grid
- Threads
  - 1D, 2D, or 3D IDs, unique within a block

- Dimensions set at launch time
- Simplifies memory addressing when processing multidimensional data
  - Image processing



Figures from NVIDIA CUDA Programming Guide, version 3.1

Memory hierarchy



- Local memory: Off-chip
- Register: On-chip (Fermi: 128KB)
- Lifetime of thread

- On-chip, small (Fermi:16KB/48KB)
- Fast as registers
- Variables: __shared__, lifetime of block

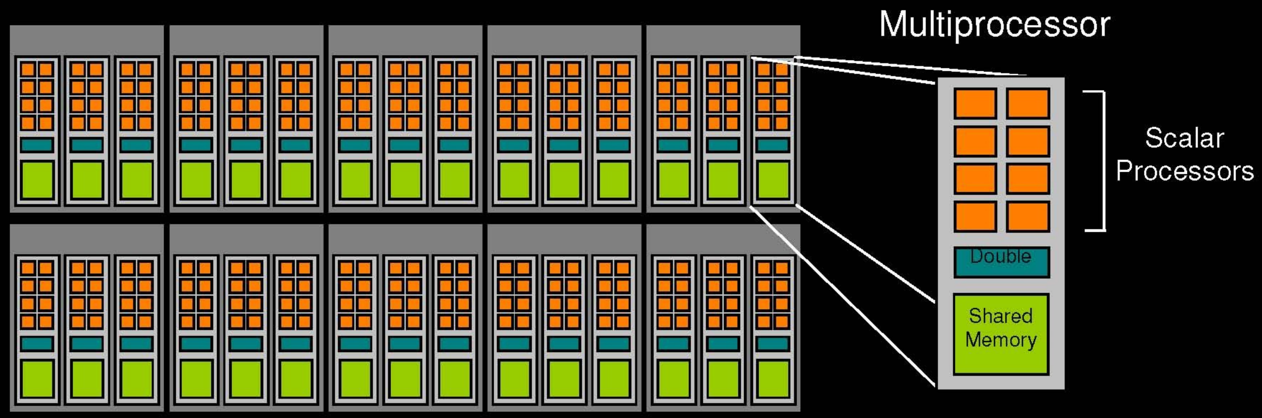
- Off-chip DRAM, large (Fermi: up to 1TB)
- Persistent across kernel launches
- Kernel I/O (host ↔ device)
- Variables: __device__, lifetime of application

Figures from NVIDIA CUDA Programming Guide, version 3.1

CUDA Architecture

10-Series Architecture

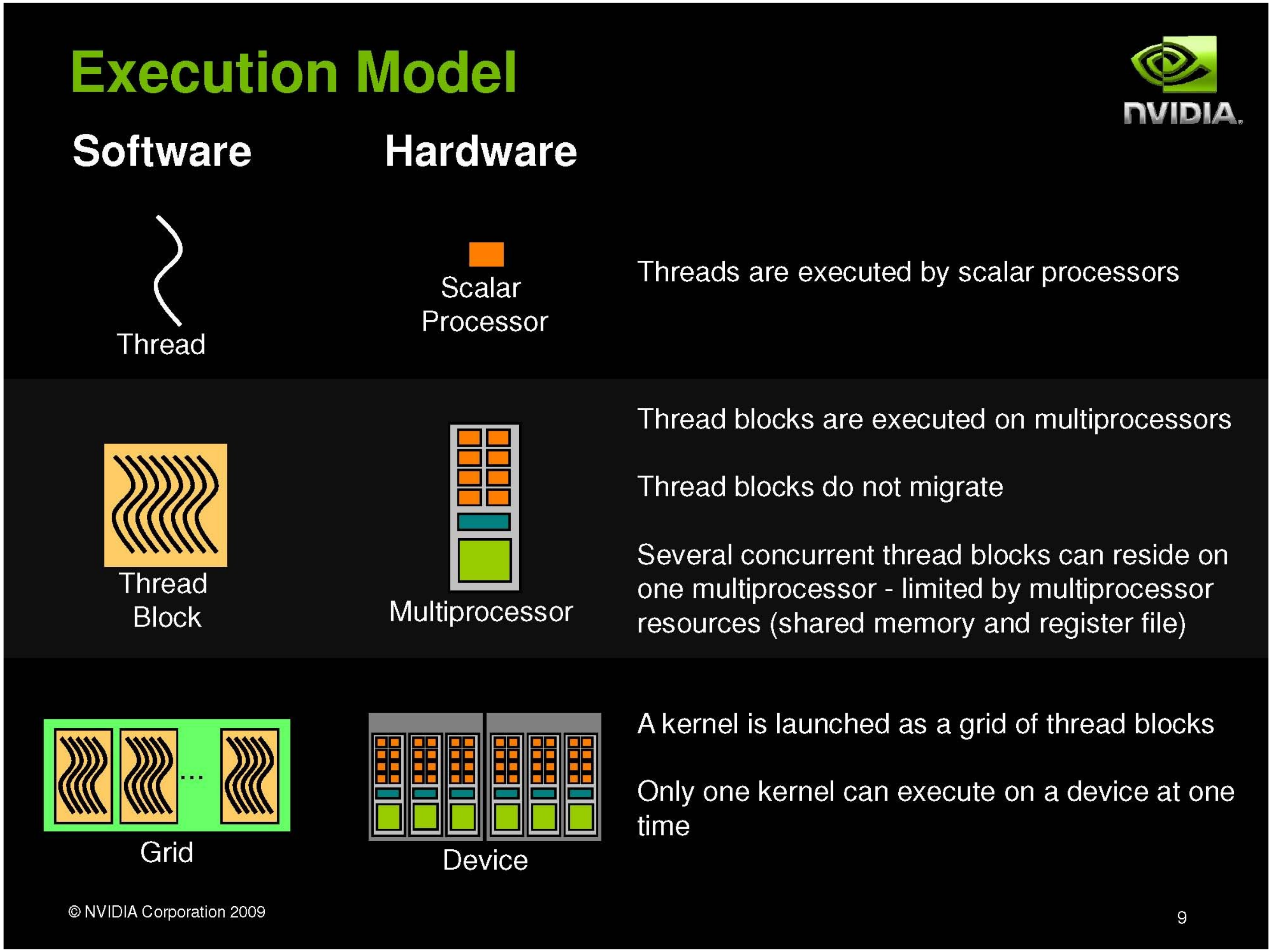
- 240 Scalar Processor (SP) cores execute kernel threads
- 30 Streaming Multiprocessors (SMs) each contain
  - 8 scalar processors
  - 2 Special Function Units (SFUs)
  - 1 double precision unit
  - Shared memory enables thread cooperation



© NVIDIA Corporation 2009

8

Execution model



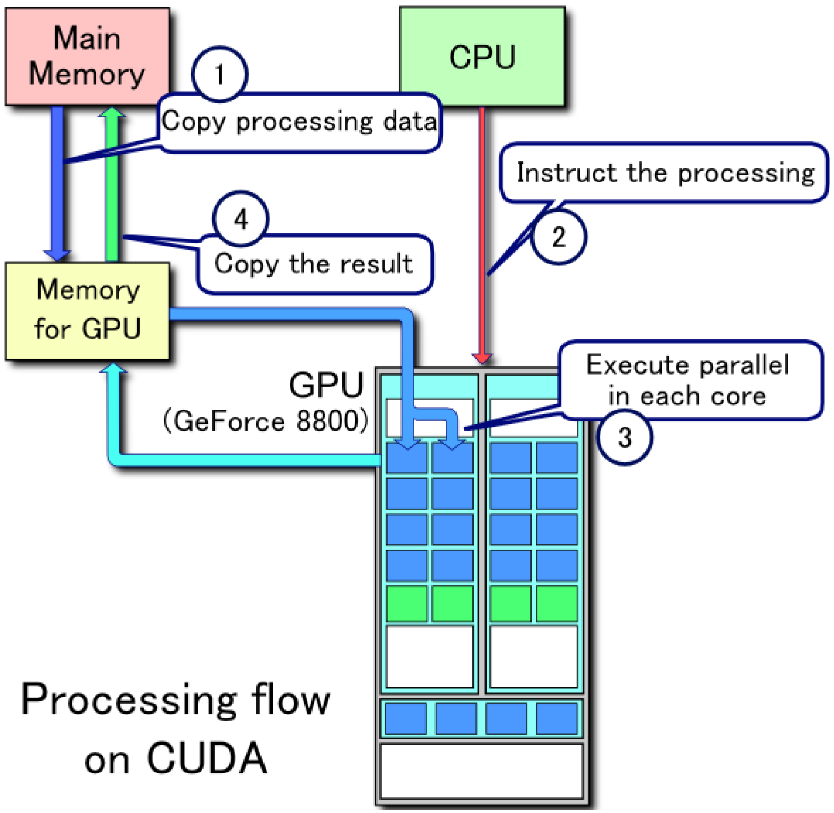
SIMT execution model

- “Single instruction, multiple threads”, and each thread *can* branch differently.
  - However, all threads in a “warp” (a subdivision of a block, defined next week!) are executed in lockstep regardless of divergent branching.
- Example:
  -
    ```
    if threadId is even:
       do A
    else:
       do B
    ```
  - Execution:
    - threads with even threadId execute A, threads with odd threadId idle.
    - Then, threads with odd threadId execute B, and threads with even threadId idle.

Outline

- Motivations for general-purpose computation on GPU (GPGPU)
- CUDA
  - CUDA programming model overview
  - CUDA C programming basics
    - Memory Management
    - Kernels and execution on GPU
    - Variables, data types, synchronization, error report
    - Coordinating GPU and CPU execution
- OpenCL

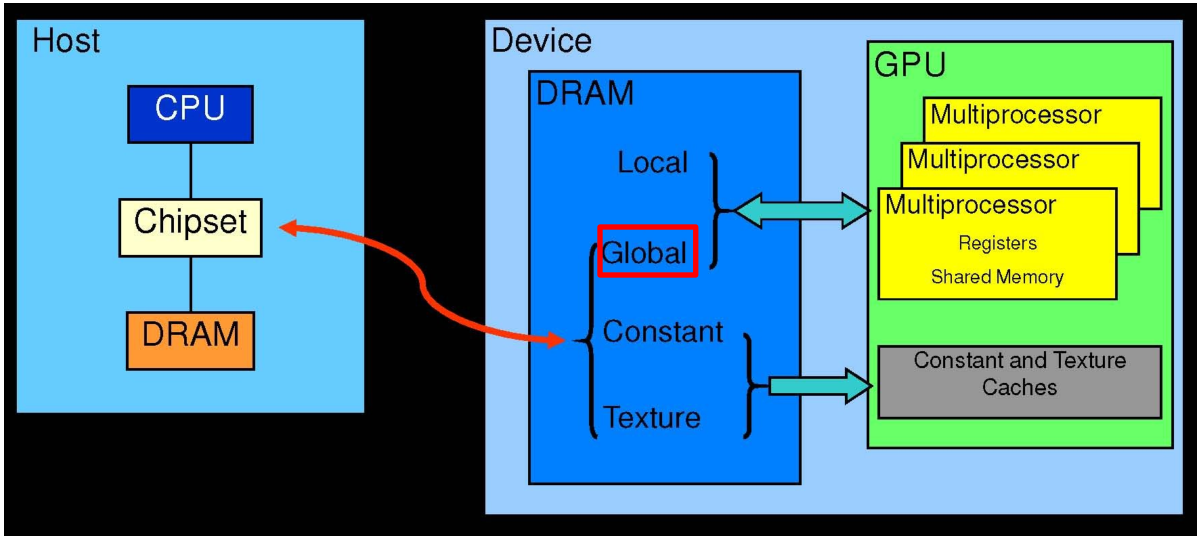
- Memory Management
  - 1 & 4
- Kernels and execution on GPU
  - 2 & 3



Figures from wikipedia.org

Memory management

- CPU and GPU have separate memory spaces
  - Pointers are just addresses
- Host (CPU) code manages device (GPU) memory
  - i.e. global memory in CUDA memory hierarchy
  - Allocate / free / copy memory on GPU



Figures from Nvidia webinars

GPU Memory Allocation / Release

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

- Example:

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

Data copies

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

Example: Data copy in vector addition

```
int main()  //Host code
{
  int N = ...;
  size_t size = N * sizeof(float);

  // Allocate input vectors in host memory
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);
  float* h_C = (float*)malloc(size);

  // Initialize input vectors
  …
  // Allocate vectors in device memory
  float* d_A;
  cudaMalloc(&d_A, size);
  float* d_B;
  cudaMalloc(&d_B, size);
  float* d_C;
  cudaMalloc(&d_C, size);

  // Copy vectors from host to device
    memory
  cudaMemcpy(d_A, h_A, size,
             cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size,
             cudaMemcpyHostToDevice);

  // Invoke kernel to compute C = A + B
  // on device
   … (see later)

  // Copy result from device to host memory
  cudaMemcpy(h_C, d_C, size,
             cudaMemcpyDeviceToHost);

  // Free device memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
  // Free host memory
  ...
}
```

Source code: NVIDIA_GPU_Computing_SDK/C/src/vectorAdd/vectorAdd.cu

Outline

- Motivations for general-purpose computation on GPU (GPGPU)
- CUDA
  - CUDA programming model overview
  - CUDA C programming basics
    - Memory Management
    - Kernels and execution on GPU
    - Variables, data types, synchronization, error report
    - Coordinating GPU and CPU execution
- OpenCL

Executing Code on the GPU

- Kernels are C functions with some restrictions
  - Can only dereference GPU pointer
  - Must have void return type
  - No static variables
  - Some additional restrictions for older GPUs
    - Not recursive
    - No variable number of arguments (“varargs”)
- Function arguments automatically copied from CPU to GPU memory

Function qualifier

- `__global__`
  - Function called from host and executed on device
  - Must return void
- `__device__`
  - Function called from device and executed 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
  - Compiler will generate both CPU and GPU code

Launching kernels

- Modified C function call syntax:
  - `kernel<<<dim3 grid, dim3 block>>>(…)`

- Execution Configuration (“<<< >>>”):
  - grid dimensions: x and y
  - thread-block dimensions: x, y, and z
  - Unspecified dim3 fields initialized to 1

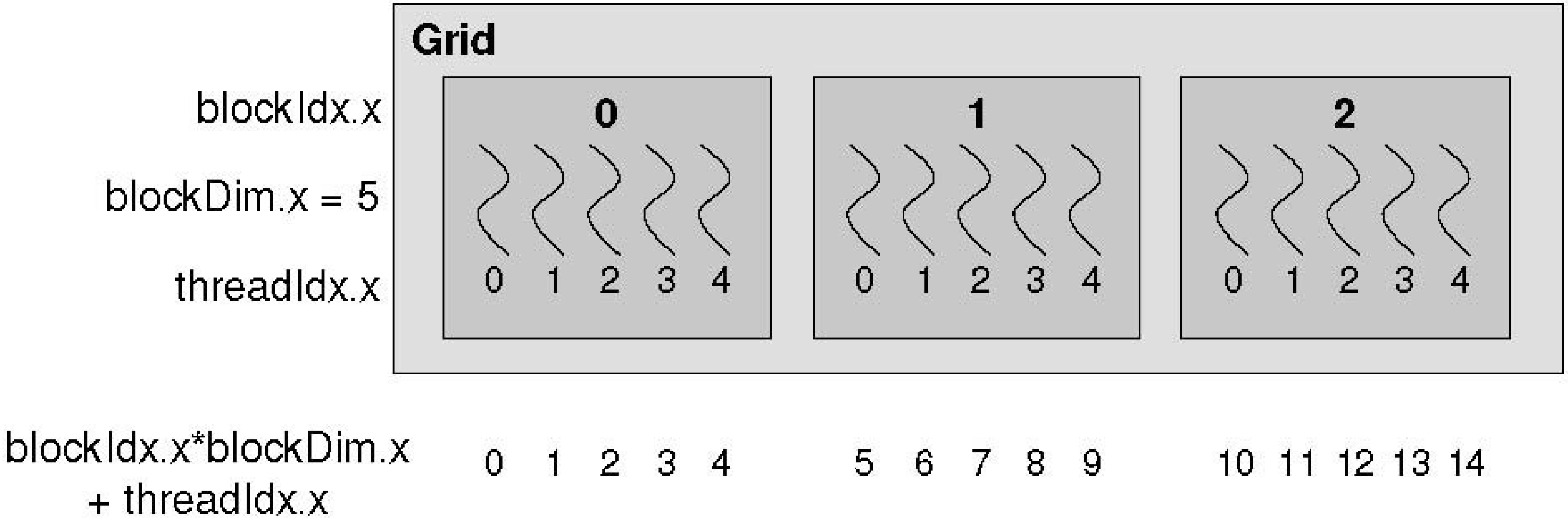
```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

CUDA Built-in Device Variables

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

Global thread IDs

- Determined using built-in variables
  - `threadIdx` (local thread ID), `blockIdx` and `blockDim`
- Used to access different elements of an array



Figures from NVIDIA Introduction to CUDA Programming

Example: vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B,
                       float* C, int N)
{
  int i = blockDim.x * blockIdx.x +
          threadIdx.x;
  if (i < N)
    C[i] = A[i] + B[i];
}
```

```
// Host code
int main()
{
  // Allocate input vectors in host memory
  …
  // Initialize input vectors
  …
  // Allocate vectors d_A, d_B, d_C
  // in device memory
  …
  // Copy vectors from host to device memory
  …
  // Invoke kernel
  int threadsPerBlock = 256;
  int blocksPerGrid =
      (N + threadsPerBlock – 1) /
       threadsPerBlock;
  VecAdd<<<blocksPerGrid,
      threadsPerBlock>>>(d_A, d_B, d_C, N);

  // Copy result from device to host memory
  …
}
```

CPU program vs. CUDA program

Increment array elements: a[idx] += b

CPU program

```
void inc_cpu(float *a, float b, int N)
{

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

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

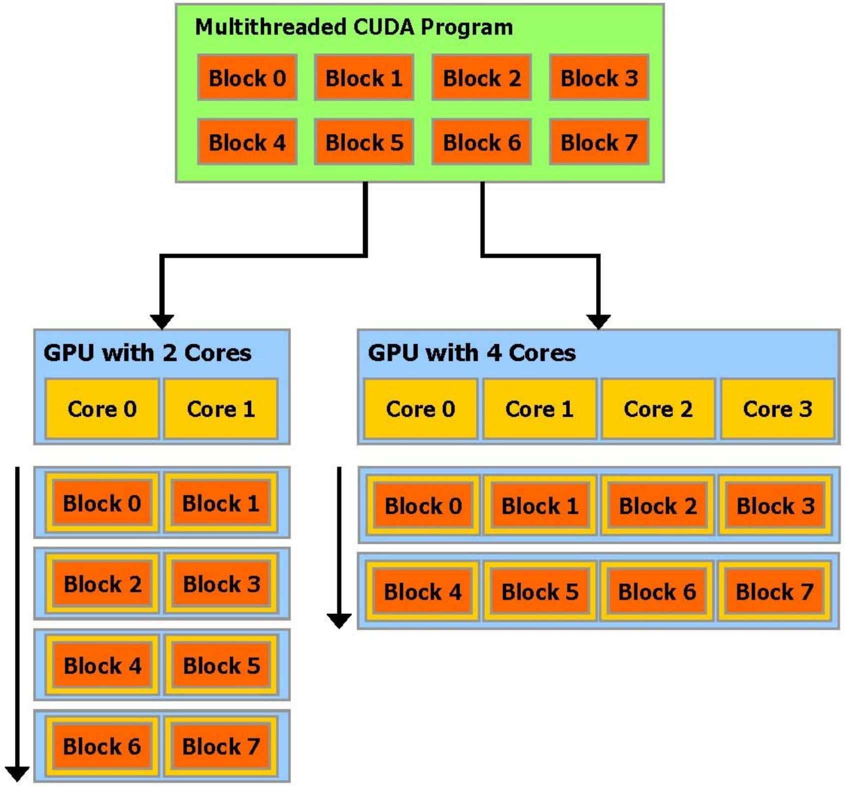
CUDA program

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

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

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - Run to completion without pre-emption
  - Can run in any order
  - Can run concurrently or sequentially
- Independence gives scalability
  - Facilitates scaling of the same code across many devices



Figures from NVIDIA CUDA Programming Guide, version 3.1

Outline

- Motivations for general-purpose computation on GPU (GPGPU)
- CUDA
  - CUDA programming model overview
  - CUDA C programming basics
    - Memory Management
    - Kernels and execution on GPU
    - Variables, data types, synchronization, error report
    - Coordinating GPU and CPU execution
- OpenCL

Variable qualifiers (GPU code)

- `__device__`
  - Stored in global memory (DRAM: large, high latency)
  - Allocated with `cudaMalloc` (`__device__` qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- `__shared__`
  - Stored in on-chip shared memory (very low latency)
  - Allocated by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: thread block
- Unqualified variables:
  - Scalars and built-in vector types are stored in registers
  - Arrays or large structures stored in “local” memory (DRAM)
  - Lifetime: thread

Using shared memory

Size known at compile time

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

int main(void)
{
    …

    kernel<<<nBlocks,blockSize>>>(…)
    ;
    …
}
```

Size known at kernel launch

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

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

Built-in Vector Types

Can be used in GPU and CPU code

- `[u]char[1..4]`, `[u]short[1..4]`, `[u]int[1..4]`, `[u]long[1..4]`, `float[1..4]`
  - Structures accessed with x, y, z, w fields:
    - `uint4 param;`
    - `int y = param.y;`
- `dim3`
  - Based on `uint3`
  - Used to specify dimensions
  - Default value `(1,1,1)`

GPU Thread Synchronization

- `void __synchthreads();`
- 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 condition is uniform across the entire thread block
  - Ex:

```
if( blockId.x == c) {
        …
        __synchthread();
}
```

GPU Atomic Integer Operations

- Exist for all associative operations
  - Associative op: `(x*y)*z = x*(y*z)`
  - atomic(Add, Sub, Increment, Decrement, Min, Max, ...)
  - atomic(And, Or, Xor)
  - atomic(Exchange, Compare, Swap)
- Atomic operations on 32-bit words in global memory
  - Require compute capability 1.1 or higher
- Atomic operations on 32-bit words in shared memory and 64-bit words in global memory
  - Require compute capability 1.2 or higher

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

Outline

- Motivations for general-purpose computation on GPU (GPGPU)
- CUDA
  - CUDA programming model overview
  - CUDA C programming basics
    - Memory Management
    - Kernels and execution on GPU
    - Variables, data types, synchronization, error report
    - Coordinating GPU and CPU execution
- OpenCL

Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls complete
- `cudaMemcpy()` is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls complete
- `cudaThreadSynchronize()`
  - blocks until all previous CUDA calls complete
- Asynchronous CUDA calls
  - Provide non-blocking memcopies
  - Overlap memcopies and kernel execution
  - Execute several kernels concurrently

Host synchronization example

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

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

//run CPU code concurrently with GPU code
independent_cpu_code()

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

CUDA Event API

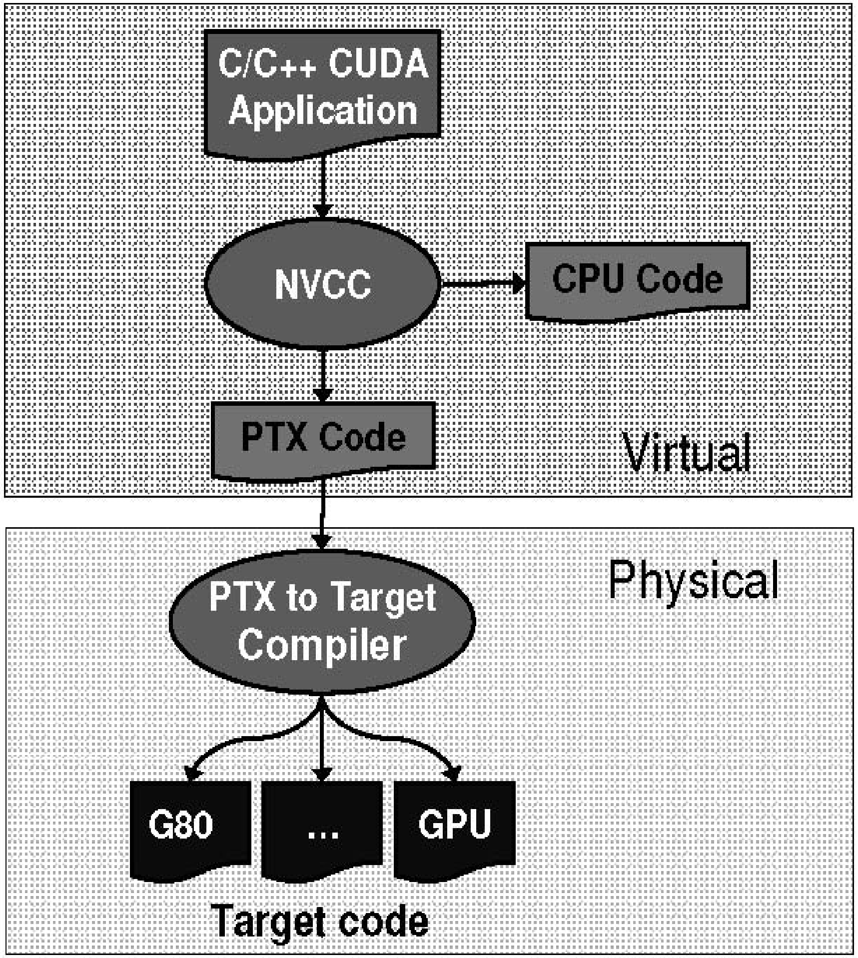
- Events are inserted (recorded) into CUDA call streams
  - An event is recorded when all CUDA tasks preceding the event finished
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to 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); /* in milliseconds, +- 0.5
   microsecond*/
cudaEventDestroy(start); cudaEventDestroy(stop);
```

Device Management

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

Compiling a CUDA program



– PTX : Parallel Thread eXecution

Figures from Introduction to CUDA Programming, Nvidia.

Outline

- Motivations for general-purpose computation on GPU (GPGPU)
- CUDA
  - CUDA programming model overview
  - CUDA C programming basics
- OpenCL
  - Introduction to OpenCL
  - OpenCL vs. CUDA
    - Platform model
    - Execution model
    - Memory model
    - Programming model

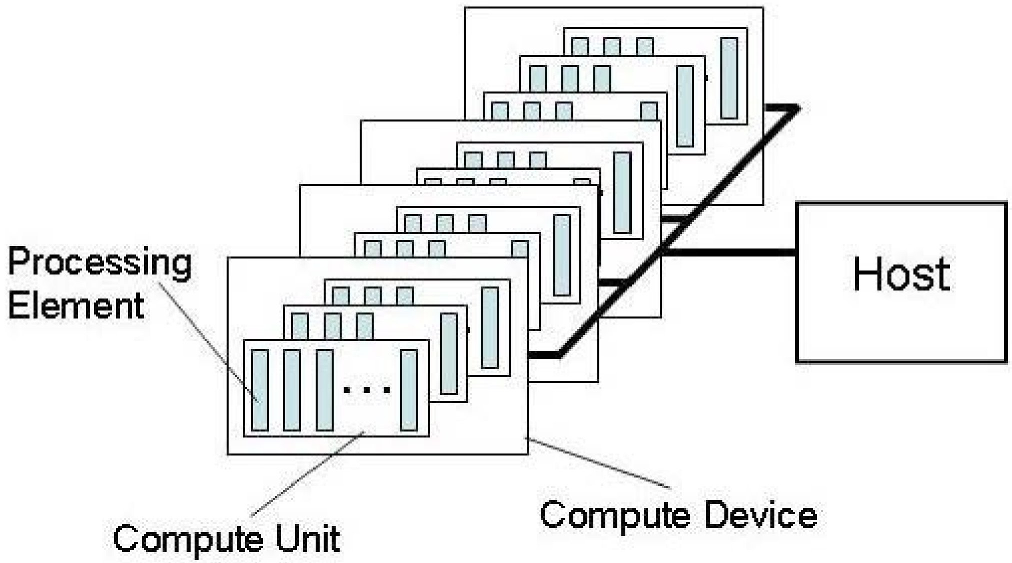
OpenCL

- What is OpenCL?
  - OpenCL = Open Computing Language
  - An open industry standard for programming a heterogeneous collection of CPUs, GPUs and other processors.
    - AMD, Apple, IBM, Intel, Nvidia, …

- Why use OpenCL?
  - Portability
    - CPU, GPU, Cell BE, Xeon Phi, …
  - Efficient parallel programming model
    - Data- and task- parallel computing model
    - Abstract the specifics of underlying hardware

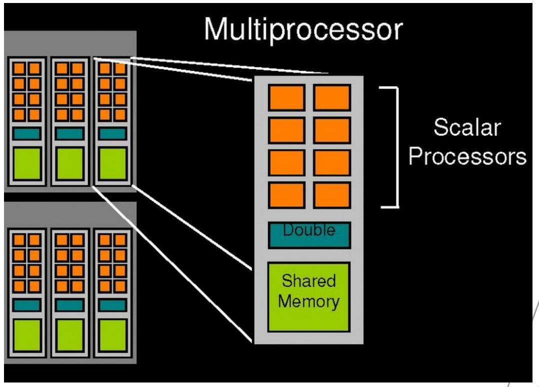
OpenCL platform model

- A host connected to one or more OpenCL devices
- An OpenCL device consists of compute units (CU)
- A compute unit consists of processing elements (PE)
  - PE executes code as SIMD or SPMD



Source: OpenCL Specification 1.1, Khronos group

CUDA GPU



Source: Nvidia webinar

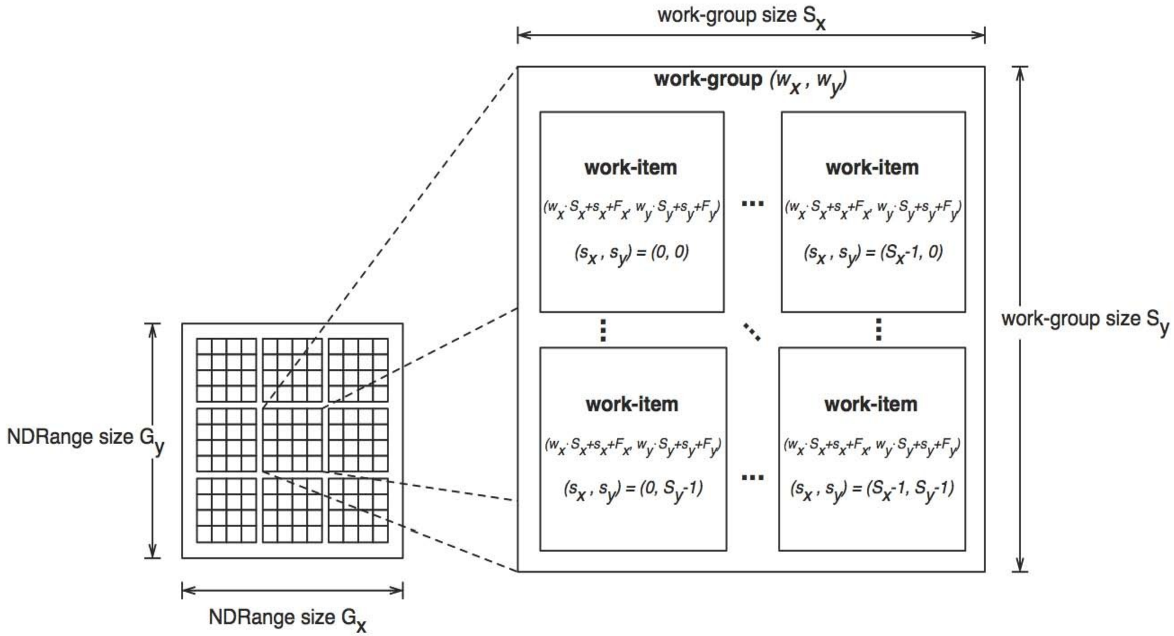
OpenCL execution model

- OpenCL to CUDA data parallelism model mapping

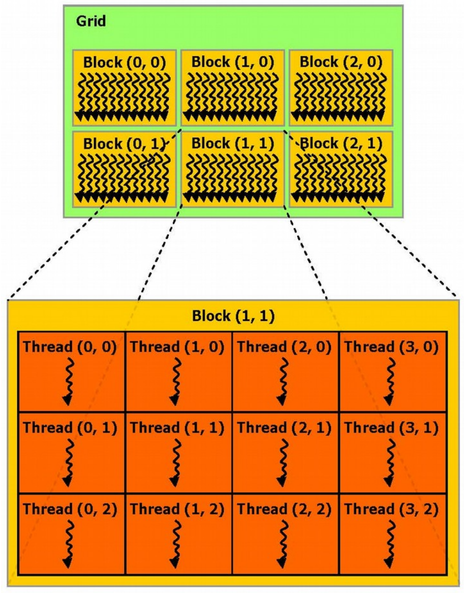
OpenCL parallelism concept	CUDA equivalent
Kernel	Kernel
Host program	Host program
NDRange (index space)	Grid
Work group	Block
Work item	Thread

OpenCL execution model (2)

- Two-dimensional index space:
  - Total number of work items = Gx * Gy
  - #work items in work group = Sx * Sy
  - Global ID of a work item computed from its local ID (sx,sy) and work group ID (wx,wy)



Source: OpenCL Specification 1.1, Khronos group



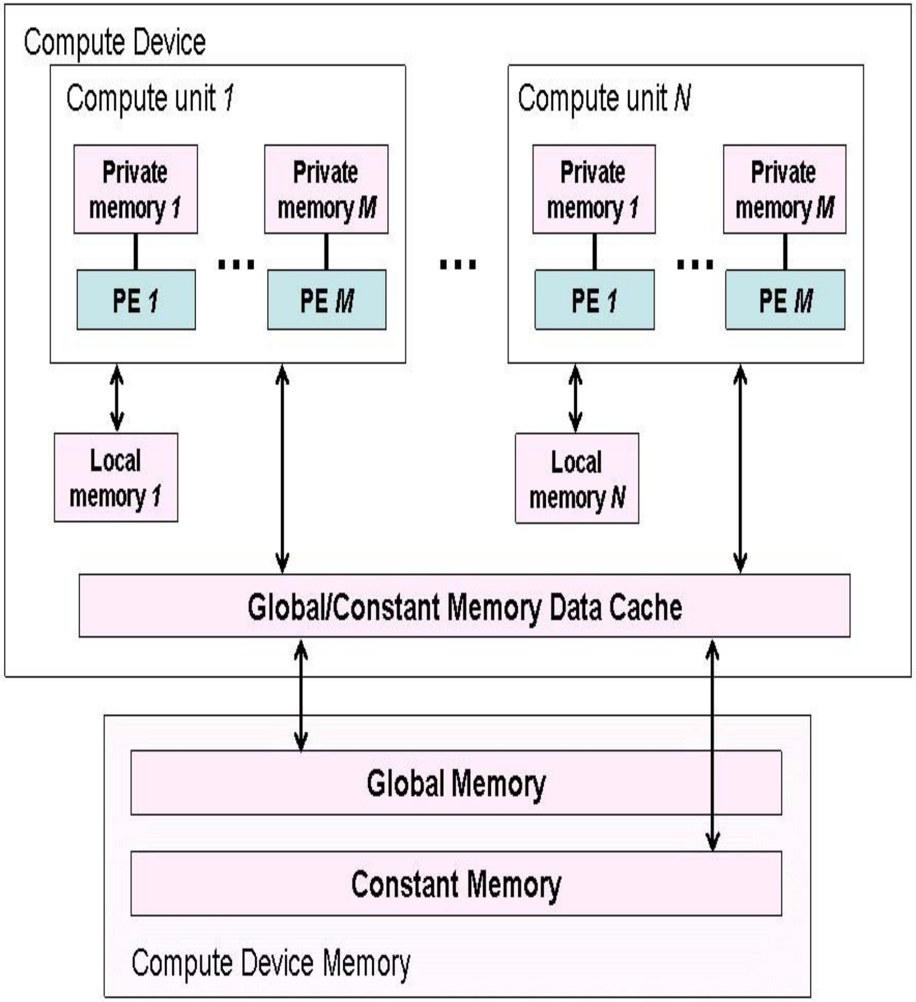
Source: CUDA Programming Guide

OpenCL execution model (3)

- Mapping OpenCL dimensions and indices to CUDA

OpenCL API call	Explanation	CUDA equivalent
get_global_id(0) parameters: 0:x; 1:y; 2:z	global index of the work item in the x dimension	blockIdx.x * blockDim.x + threadIDx.x
get_local_id(0)	local index of the work item in the work group in the x dimension	threadIdx.x
get_global_size(0)	size of NDRange in the x dimension	gridDim.x * blockDim.x
get_local_size(0)	size of each work group in the x dimension	blockDim.x

OpenCL memory model



Private memory:
read/write access for work item

Local memory:
read/write access for entire work groups

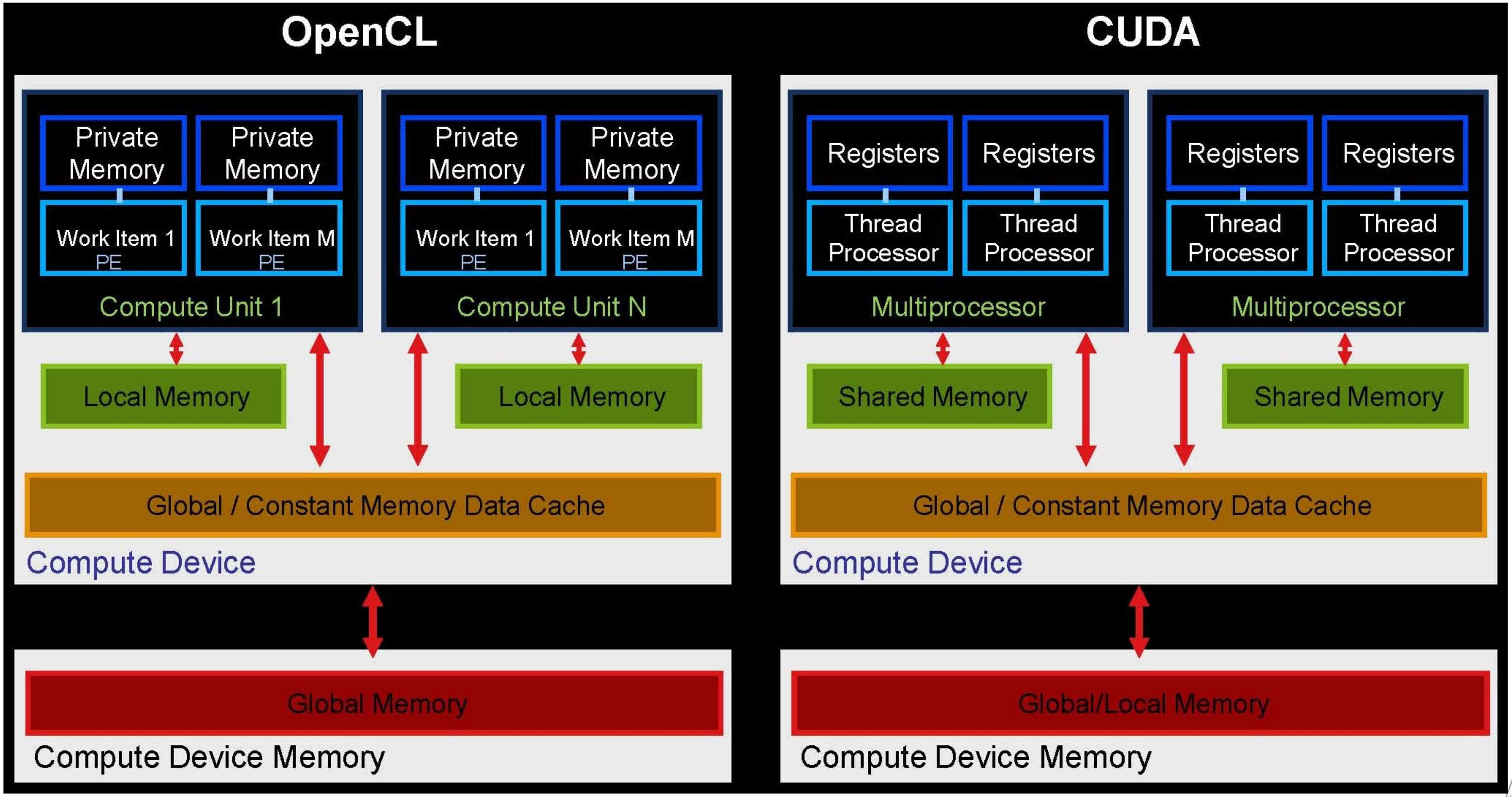
Global memory:
read/write access for entire ND-range
(all work items, all work groups)

Constant memory:
read access for entire ND-range

Source: OpenCL Specification 1.1, Khronos group

OpenCL memory model (2)

- Memory model: OpenCL vs. CUDA



Source: Nvidia webinar

OpenCL Programming model

- Data parallel programming model
  - The primary model driving the design of OpenCL
  - a sequence of instructions applied to multiple elements of a memory object
    - Typically, one-to-one mapping between work-item (thread in CUDA) and the element.

- Task parallel programming model
  - execute a kernel on a compute unit (multiprocessor) with a work-group (threadblock) containing a single work-item (thread)
  - Express parallelism by enqueuing multiple tasks

References

- CUDA Programming Guide 3.1, Nvidia, 2009
- Introduction to CUDA Programming, Nvidia, 2008
- Nvidia CUDA Webinars
- OpenCL Specification 1.1, Khronos group, 2010
- Nvidia OpenCL Webinars
- David B. Kirk & Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach, ISBN-13: 978-0123814722.