GPGPU – A Current Trend in High Performance Computing

Chokchai **Box** Leangsuksun, PhD

SWEPCO Endowed Professor*, Computer Science
Director, High Performance Computing Initiative
Louisiana Tech University
box@latech.edu

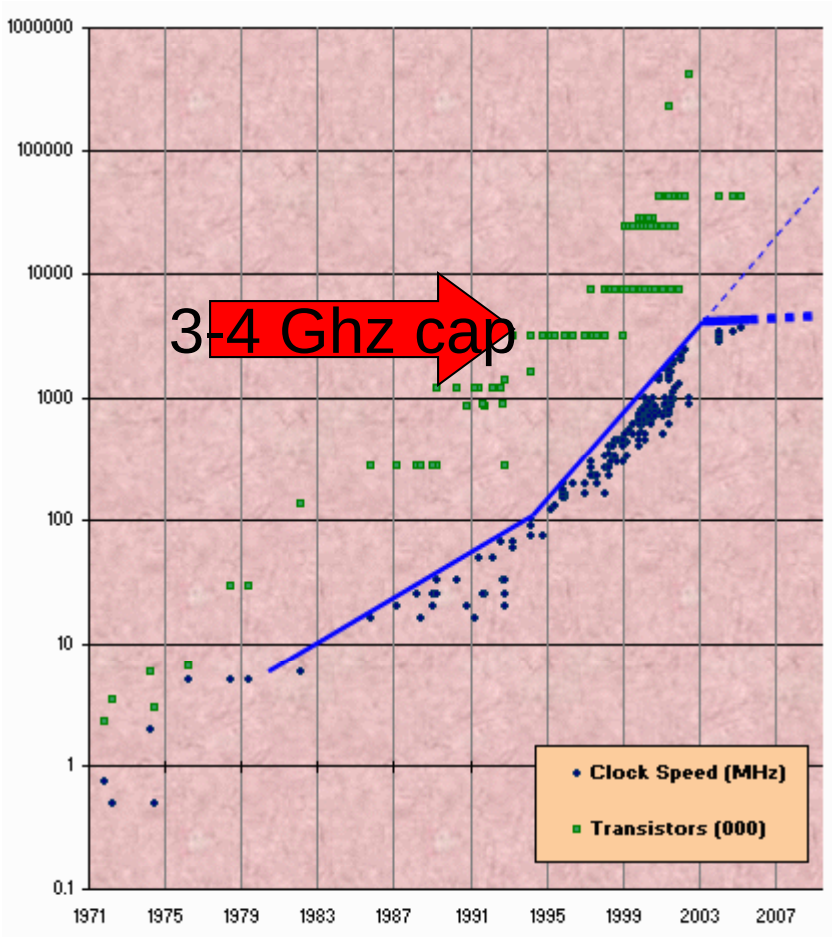
*SWEPCO endowed professorship is made possible by LA Board of Regents

Outline

- Intro to HPC - Box
- GPU Tutorial – Box
- CUDA programming concepts - Box
- Case study: Advanced performance improvement

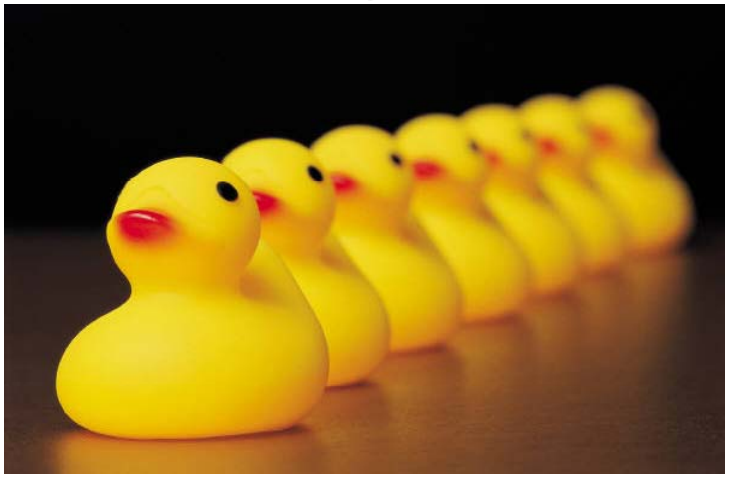
Mainstream CPUs

- CPU speed – plateaus 3-4 Ghz
- More cores in a single chip
  - Dual/Quad core is now
  - Manycore (GPGPU)
- Traditional Applications won't get a free rides
- Conversion to parallel computing (HPC, MT)



This diagram is from "no free lunch article in DDJ

New trends in computing



- Old & current – SMP, Cluster
- Multicore computers
  - Intel Core 2 Duo
  - AMD 2x 64
- Many-core accelerators
  - GPGPU, FPGA, Cell
    - More Many brains in one computer
    - Not to increase CPU frequency
    - Harness many computers – a cluster computing

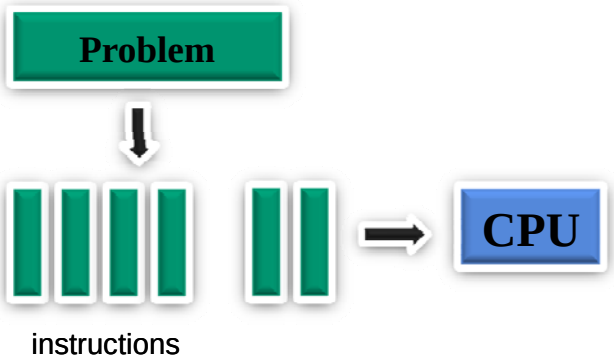
What is HPC?

- High Performance Computing – Parallel , Supercomputing
  - Achieve the fastest possible computing outcome
  - Subdivide a very large job into many pieces
  - Enabled by multiple high speed CPUs, networking, software & programming paradigms – fastest possible solution
  - Technologies that help solving non-trivial tasks including scientific, engineering, medical, business, entertainment and etc.

- Time to insights, Time to discovery, Times to markets

Parallel Programming Concepts

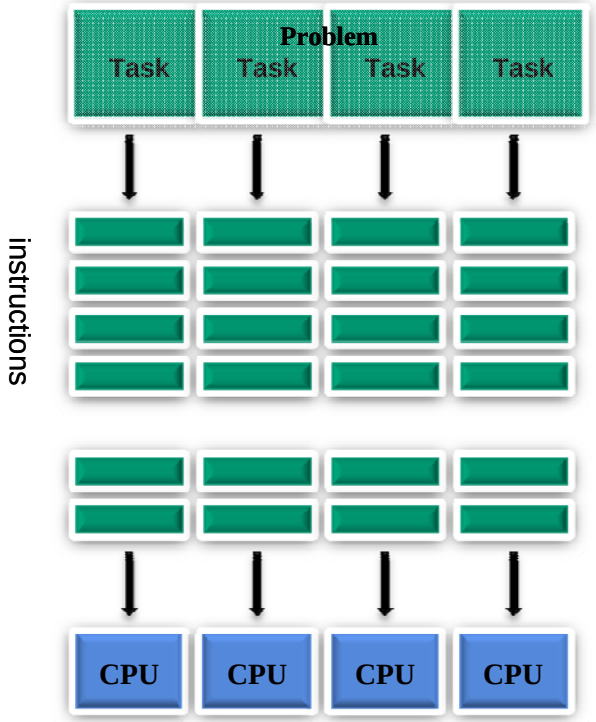
Conventional *serial execution* where the problem is represented as a series of instructions that are executed by the CPU



Parallel computing takes advantage of concurrency to :

- Solve larger problems with less time
- Save on Wall Clock Time
- Overcoming memory constraints
- Utilizing non-local resources

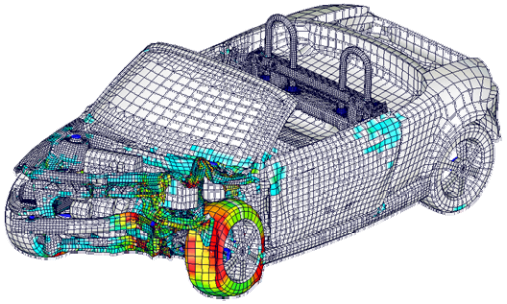
Parallel execution of a problem involves partitioning of the problem into multiple executable parts that are mutually exclusive and collectively exhaustive represented as a partially ordered set exhibiting concurrency.

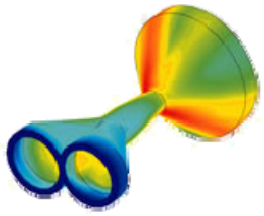


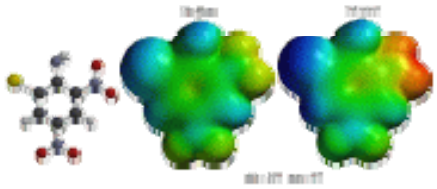
Source from Thomas Sterling's intro to HPC

HPC Applications and Major Industries

- Finite Element Modeling
  - Auto/Aero
- Fluid Dynamics
  - Auto/Aero, Consumer Packaged Goods Mfgs, Process Mfg, Disaster Preparedness (tsunami)
- Imaging
  - Seismic & Medical
- Finance & Business
  - Banks, Brokerage Houses (Regression Analysis, Risk, Options Pricing, What if, …)
  - Wal-mart's HPC in their operations
- Molecular Modeling
  - Biotech and Pharmaceuticals







Complex Problems, Large Datasets, *Long Runs*

This slide is from Intel presentation "**Technologies for Delivering Peak Performance on HPC and Grid Applications**"

The GPGPU Tutorial

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

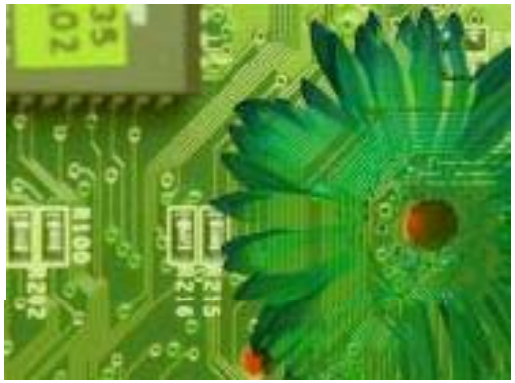
What & Why is GPGPU ?

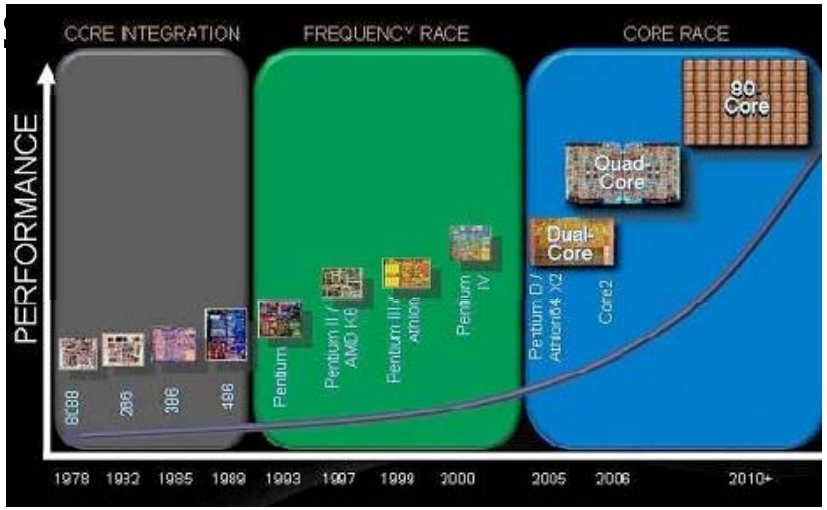
- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- One of the hottest computing trends
  - Heterogeneous computing
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
    - Oil exploration, Realtime MRI-CT-scan,
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

© David Kirk/NVIDIA

Why is GPGPU?

- Large number of cores –
  - 100-1000 cores in a single card
- Low cost – less than $100-$1500
- Green computing
  - Low power consumption – 135 watts/card
  - 135 w vs 30000 w (300 watts * 100)
- 1 card can perform > 100 desktops
  - $750 vs 50000 ($500 * 100)



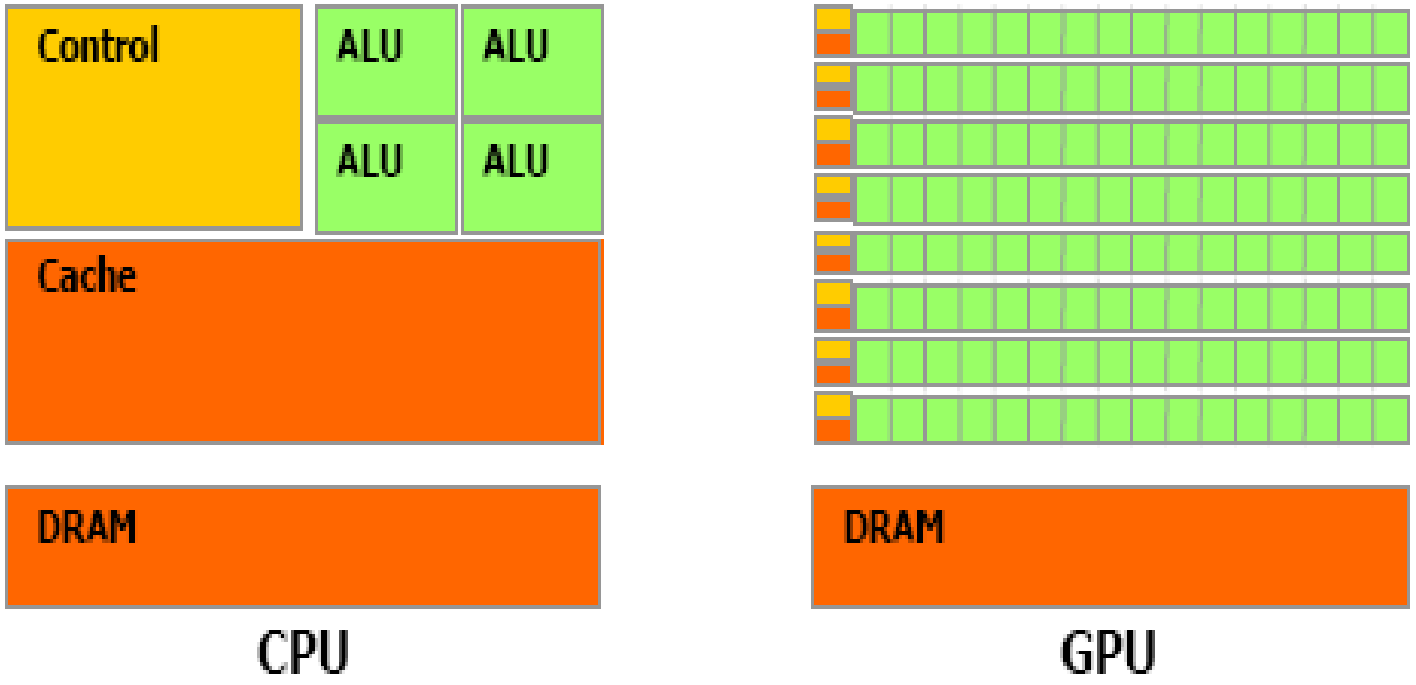


CPU vs. GPU

- CPU
  - Fast caches
  - Branching adaptability
  - High performance
- GPU
  - Multiple ALUs
  - Fast onboard memory
  - High throughput on parallel tasks
    - Executes program on each fragment/vertex

- CPUs are great for ***task*** parallelism
- GPUs are great for ***data*** parallelism

CPU vs. GPU - Hardware



- More transistors devoted to data processing

Two major players

Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture
  - Via a HW device interface
  - In laptops, desktops, workstations, servers
- Tesla T10 1070 from 1-4 TFLOPS
- AMD/ATI 4870 x2 1600 cores
- NVIDIA Tegra is an all-in-one (system-on-a-chip) processor architecture derived from the ARM family
- GPU parallelism is better than Moore's law, more doubling every year
- GPGPU is a GPU that allows user to process both graphics and non-graphics applications.

ATI 4850

GeForce 8800

© David Kirk/NVIDIA

Requirements of a GPU system

- GPGPU is a GPU that allows user to process both graphics and non-graphics applications.

- GPGPU-capable video card
- Power supply
- Cooling
- PCI-Express 16x

GeForce 8800

© David Kirk/NVIDIA

Examples of GPU devices

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

NVIDIA GeForce 8800 (G

- the eighth generation of NVIDIA's GeForce graphic cards.
- High performance CUDA-enabled GPGPU
- 128 cores
- Memory 256-768 MB or 1.5 GB in Tesla
- High-speed memory bandwidth
- Supports Scalable Link Interface (SLI)

NVIDIA Tesla™

- Feature
  - GPU Computing for HPC
  - **No display ports**
  - Dedicate to computation
  - For massively Multi-threaded computing
  - Supercomputing performance

NVIDIA Tesla C

- C-Series(Card) = 1 GPU with 1.5 GB
- D-Series(Deskside unit) = 2 GPUs
- S-Series(1U server) = 4 GPUs

- Note: 1 G80 GPU = 128 cores = ~500 GFLOPs
- 1 T10 = 240 cores = 1 TFLOPs

<< NVIDIA G80

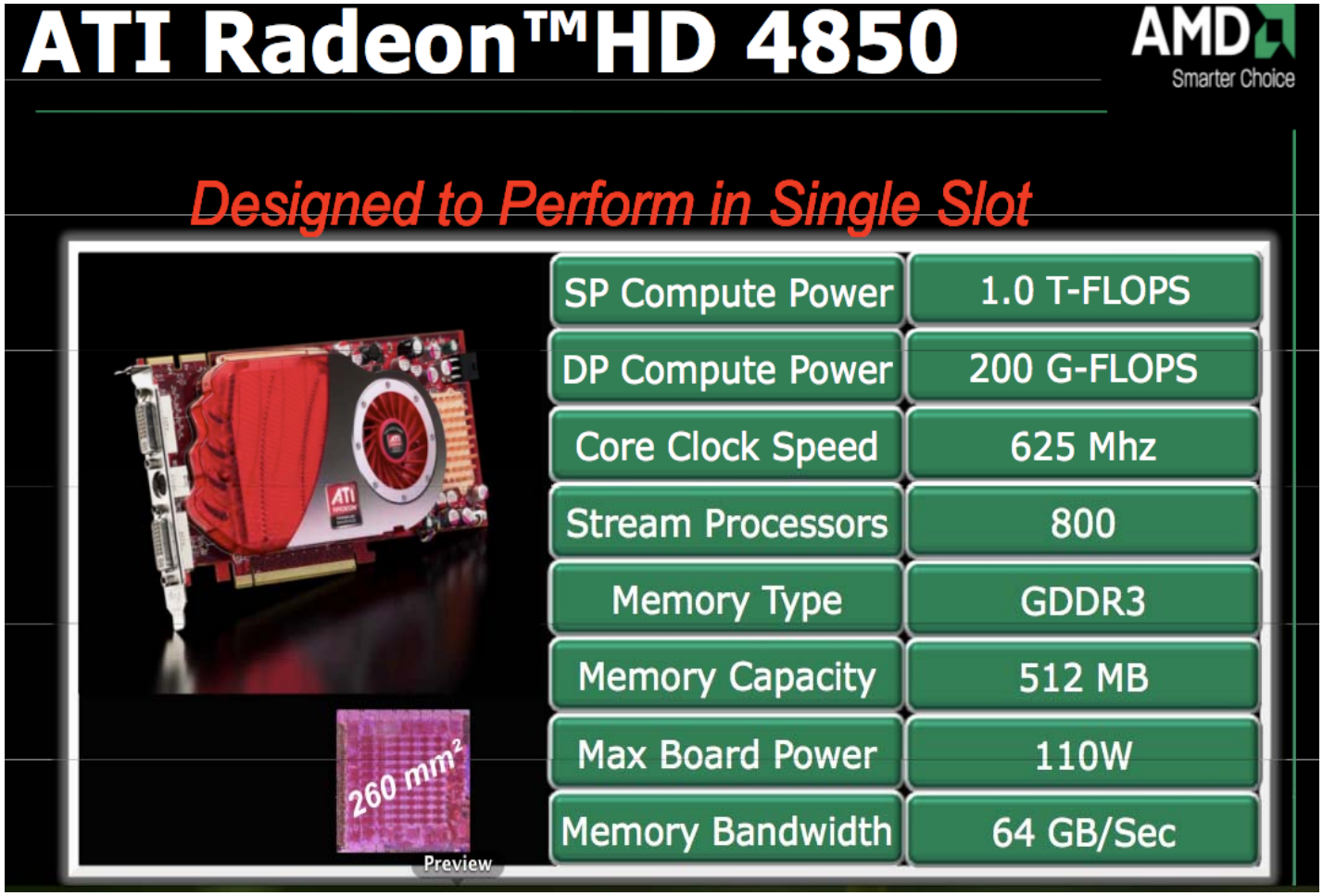
Programming Model: A Massively Parallel Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs 1000's of threads in parallel
- Data-parallel portions of an application execute on the device as kernels which run many cooperative threads in parallel
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
    - Instruction level thread switching
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few
  - Threads are non-persistent
    - Run and exit

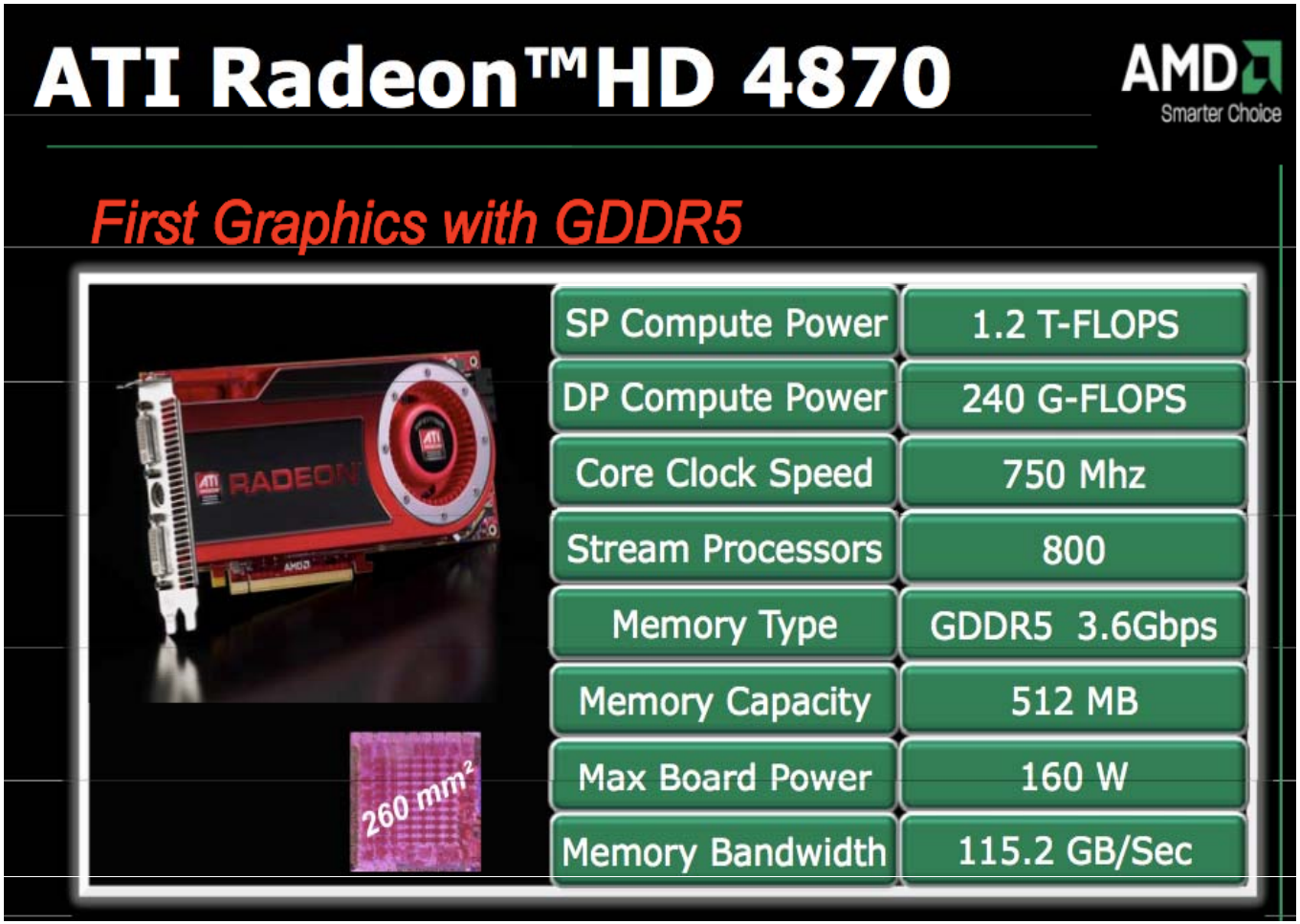
This slide is from NVDIA CUDA tutorial

© David Kirk/NVIDIA

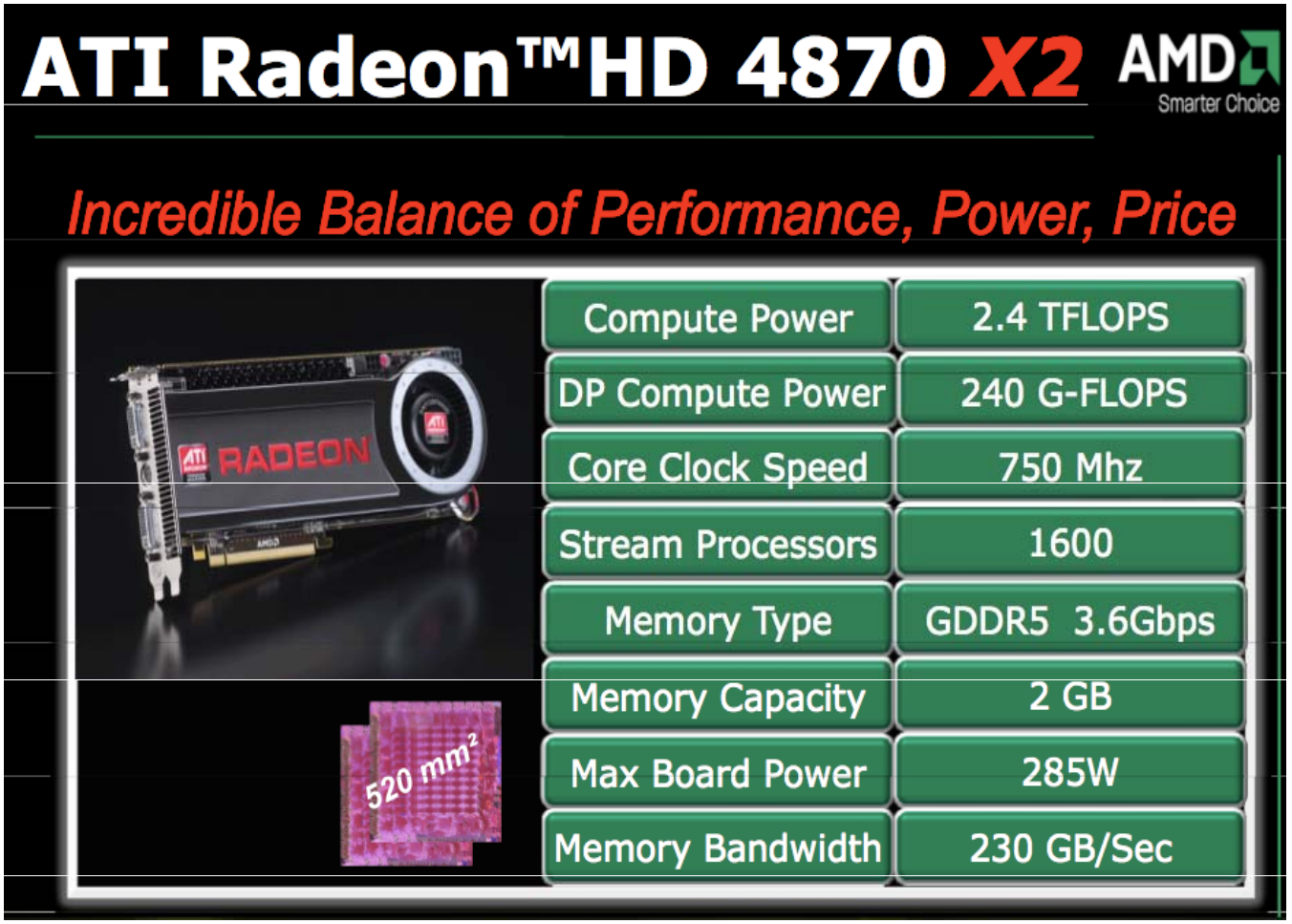
ATI Stream (1)

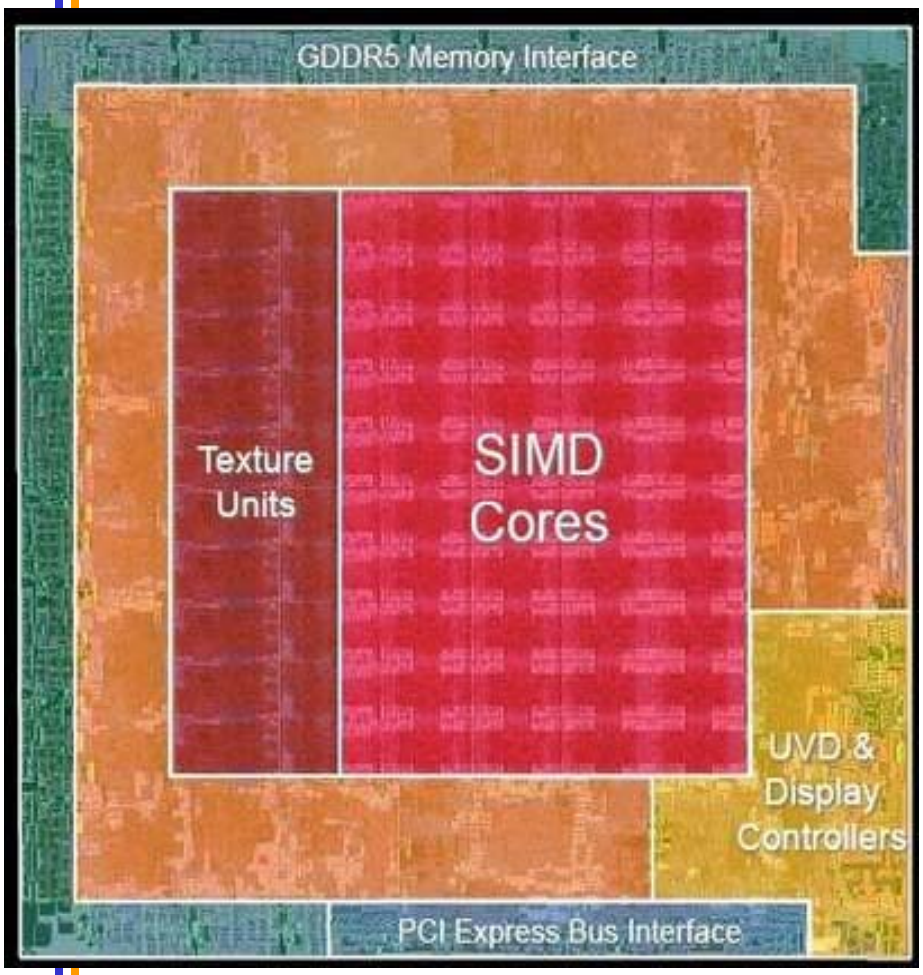

ATI Radeon™HD 4850
AMD
Smarter Choice
Designed to Perform in Single Slot
260 mm²
Preview
SP Compute Power
1.0 T-FLOPS
DP Compute Power
200 G-FLOPS
Core Clock Speed
625 Mhz
Stream Processors
800
Memory Type
GDDR3
Memory Capacity
512 MB
Max Board Power
110W
Memory Bandwidth
64 GB/Sec

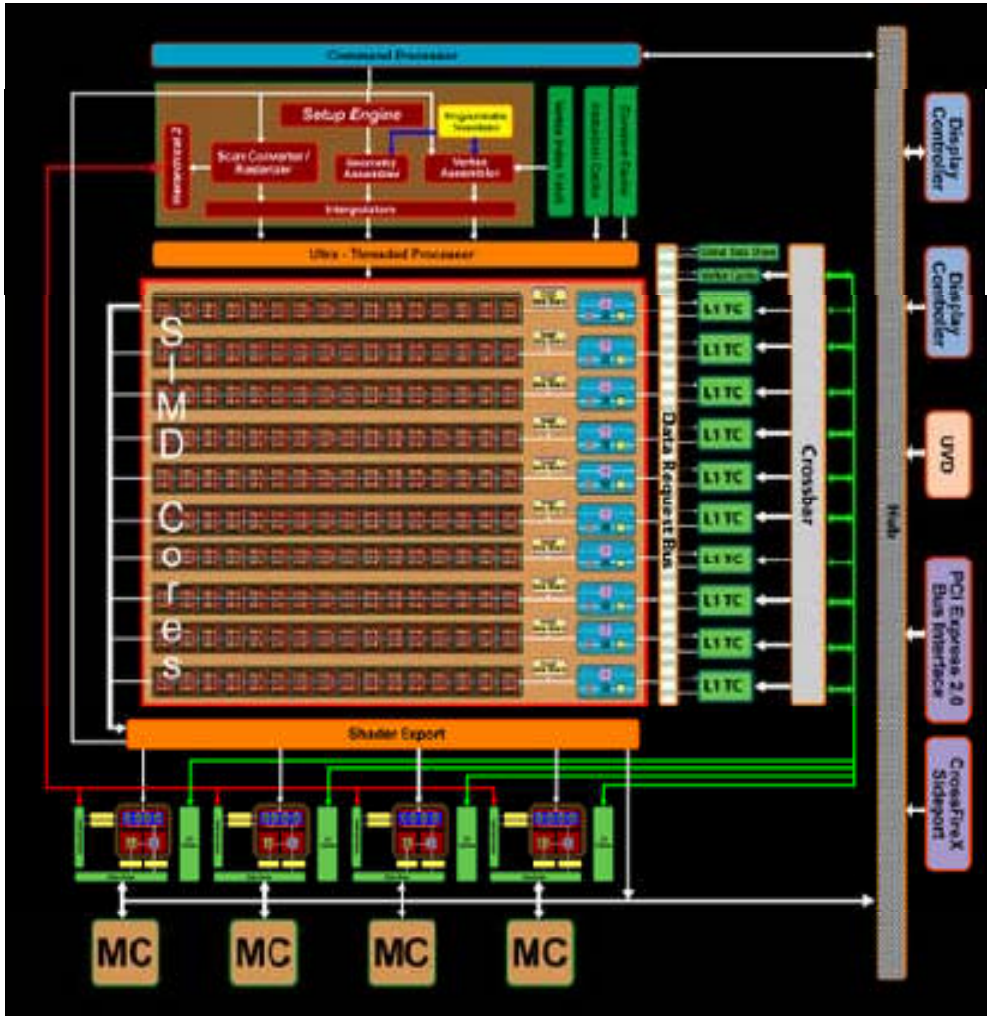
ATI 4870


ATI Radeon™HD 4870
AMD
Smarter Choice
First Graphics with GDDR5
RADEON
260 mm²
SP Compute Power
1.2 T-FLOPS
DP Compute Power
240 G-FLOPS
Core Clock Speed
750 Mhz
Stream Processors
800
Memory Type
GDDR5 3.6Gbps
Memory Capacity
512 MB
Max Board Power
160 W
Memory Bandwidth
115.2 GB/Sec

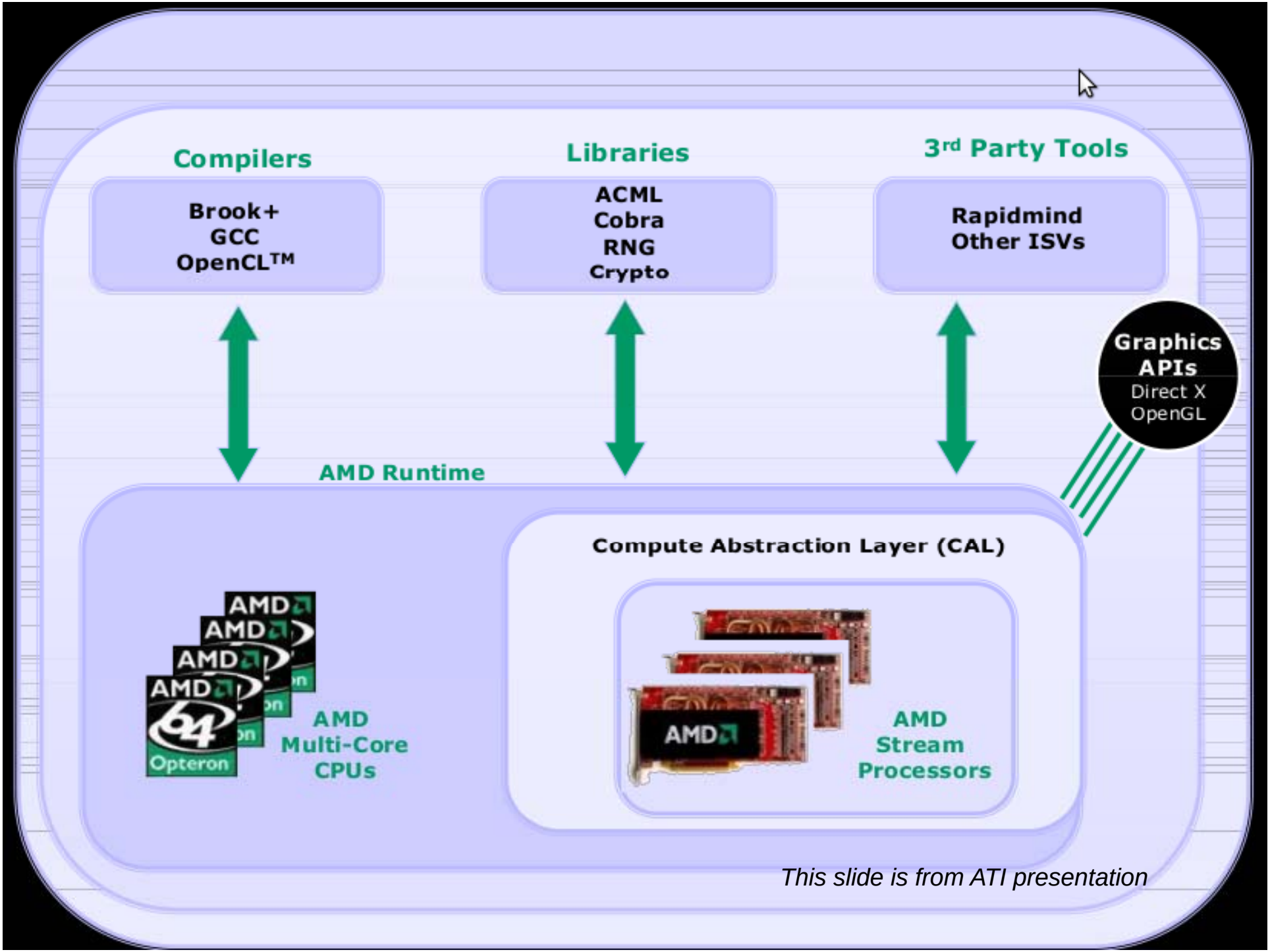
ATI 4870 X2


ATI Radeon™HD 4870 X2
AMD
Smarter Choice
Incredible Balance of Performance, Power, Price
RADEON
520 mm²
Compute Power
2.4 TFLOPS
DP Compute Power
240 G-FLOPS
Core Clock Speed
750 Mhz
Stream Processors
1600
Memory Type
GDDR5 3.6Gbps
Memory Capacity
2 GB
Max Board Power
285W
Memory Bandwidth
230 GB/Sec





Architecture of ATI Radeon 4000 series



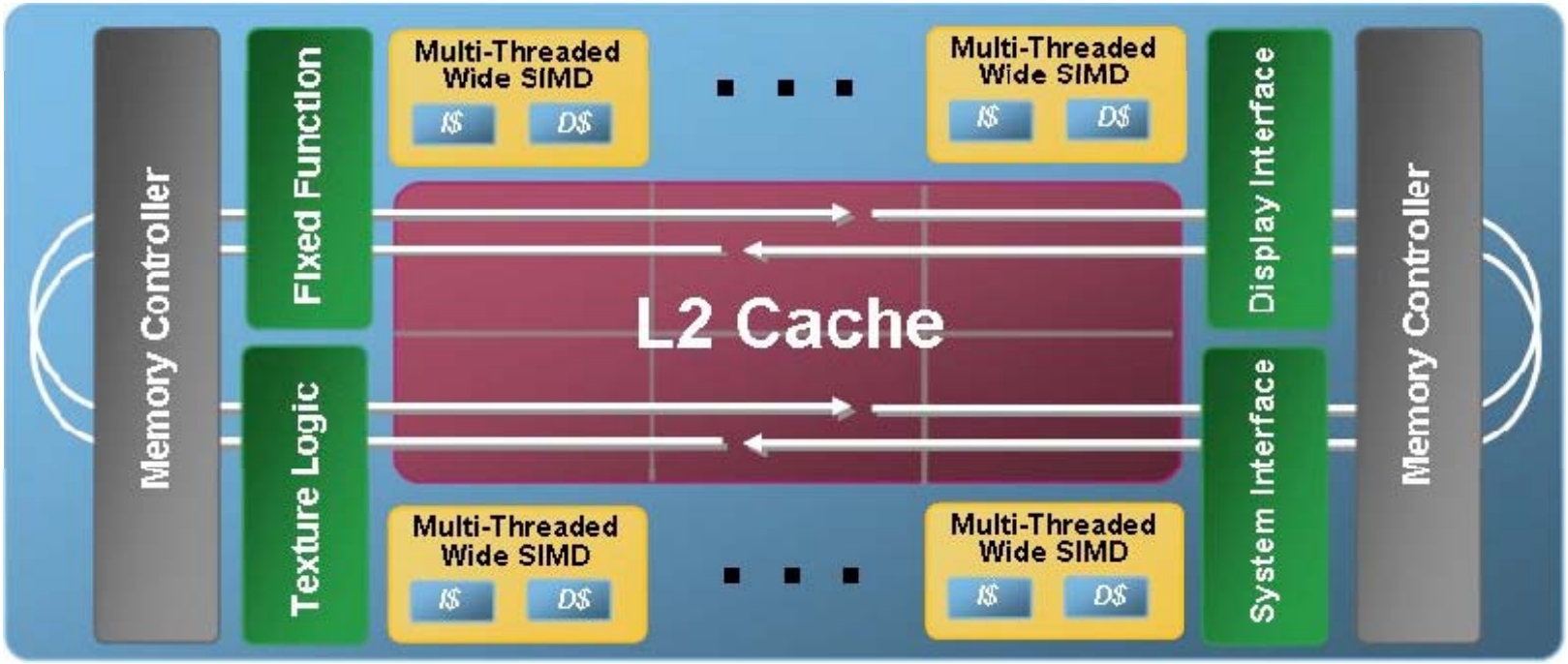
This slide is from ATI presentation

Specifications of the main cards					
GPU	HD 3870 X2	HD 4850	9800 GTX	9800 GTX +	280 GTX
GPU frequency	825 MHz	625 MHz	675 MHz	738 MHz	602 MHz
ALU frequency	825 MHz	625 MHz	1688 MHz	1836 MHz	1296 MHz
Memory frequency	900 MHz	1000 MHz	1100 MHz	1100 MHz	1107 MHz
Memory bus width	2x256 bits	256 bits	256 bits	256 bits	512 bits
Memory type	GDDR3	GDDR3	GDDR3	GDDR3	GDDR3
Memory quantity	2 x 512 MB	512 MB	512 MB	512 MB	1024 MB
Number of ALUs	640	800	128	128	240
Number of texture units	32	40	64	64	80
Number of ROPs	32	16	16	16	32
Shading power	1.06 TFlops	1 TFlops	(648) GFlops	(705) GFlops	933 GFlops
Memory bandwidth	115.2 GB/s	64 GB/s	70.4 GB/s	70.4 GB/s	141.7 GB/s
Number of transistors	1334 million	956 million	754 million	754 million	1400 million
Process	0.055µ	0.055µ	0.065µ	0.055µ	0.065µ
Die area	2 x 196 mm²	260 mm²	324 mm²	248 mm²	576 mm²
Generation	2008	2008	2008	2008	2008
Shader Model supported	4.1	4.1	4.0	4.0	4.0

This slide is from ATI presentation

Intel Larrabee

- a hybrid between a multi-core CPU and a GPU,
- coherent cache hierarchy and x86 architecture compatibility are CPU-like
- its wide SIMD vector units and texture sampling hardware are GPU-like.



Introduction to Open CL

Toward new approach in Computing

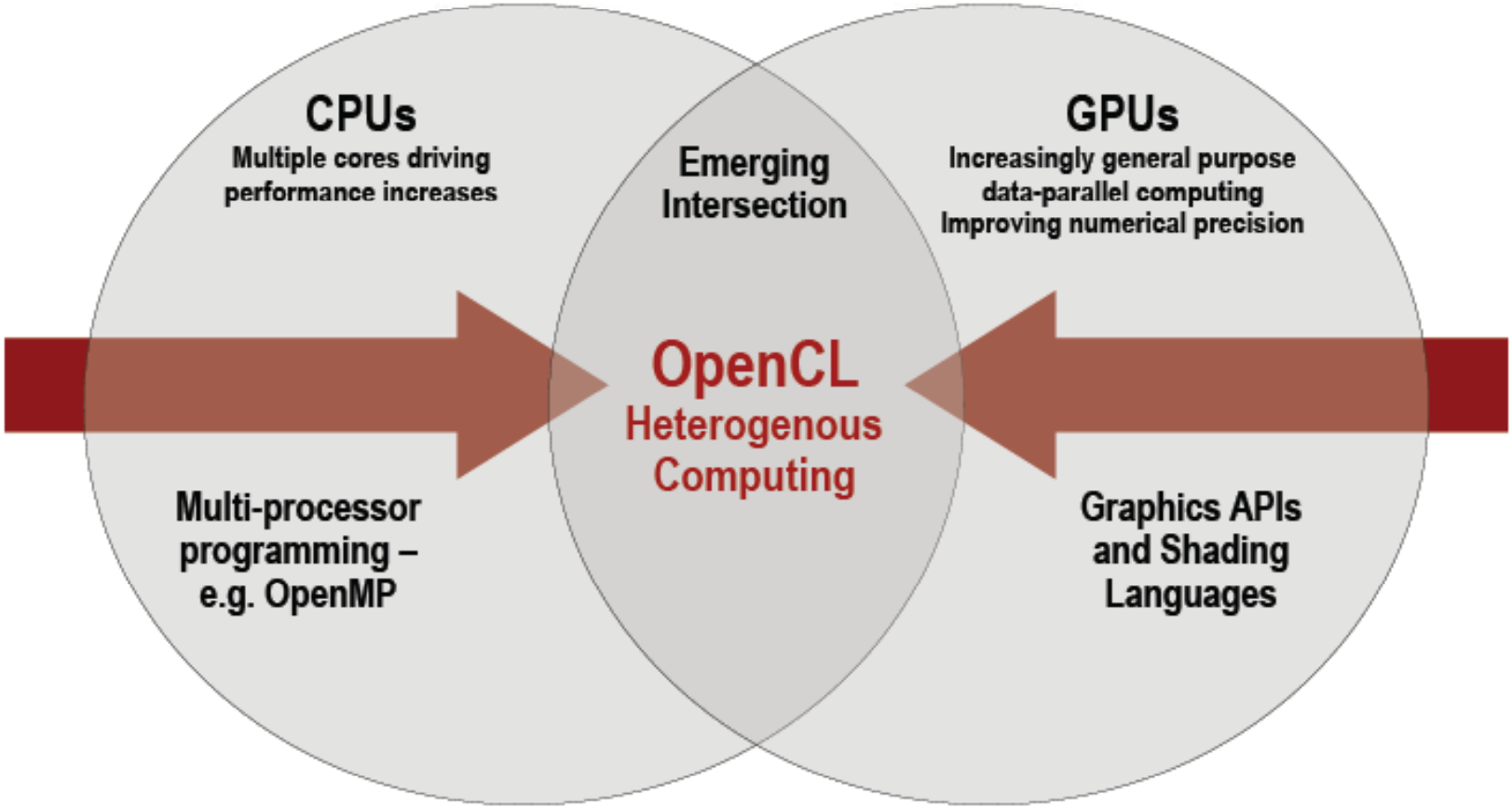
Introduction to openCL

- OpenCL stands for Open Computing Language.
- It is from consortium efforts such as Apple, NVDIA, AMD etc.
- The Khronos group who was responsible for OpenGL.
- Take 6 months to come up with the specifications.

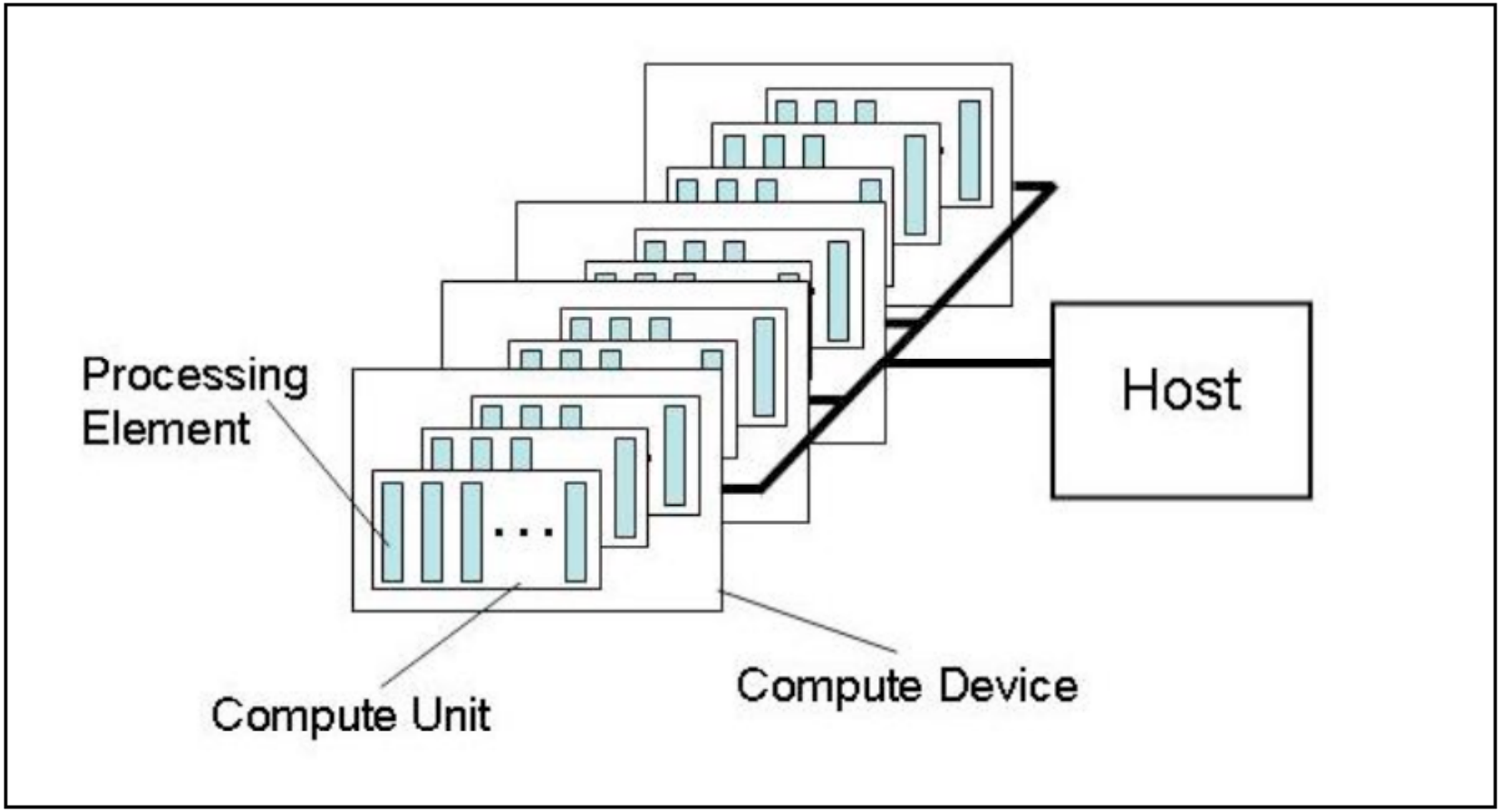
OpenCL

- 1. Royalty-free.
- 2. Support both task and data parallel programing modes.
- 3. Works for vendor-agnostic GPGPUs
- 4. including multi cores CPUs
- 5. Works on Cell processors.
- 6. Support handhelds and mobile devices.
- 7. Based on C language under C99.

Processor Parallelism

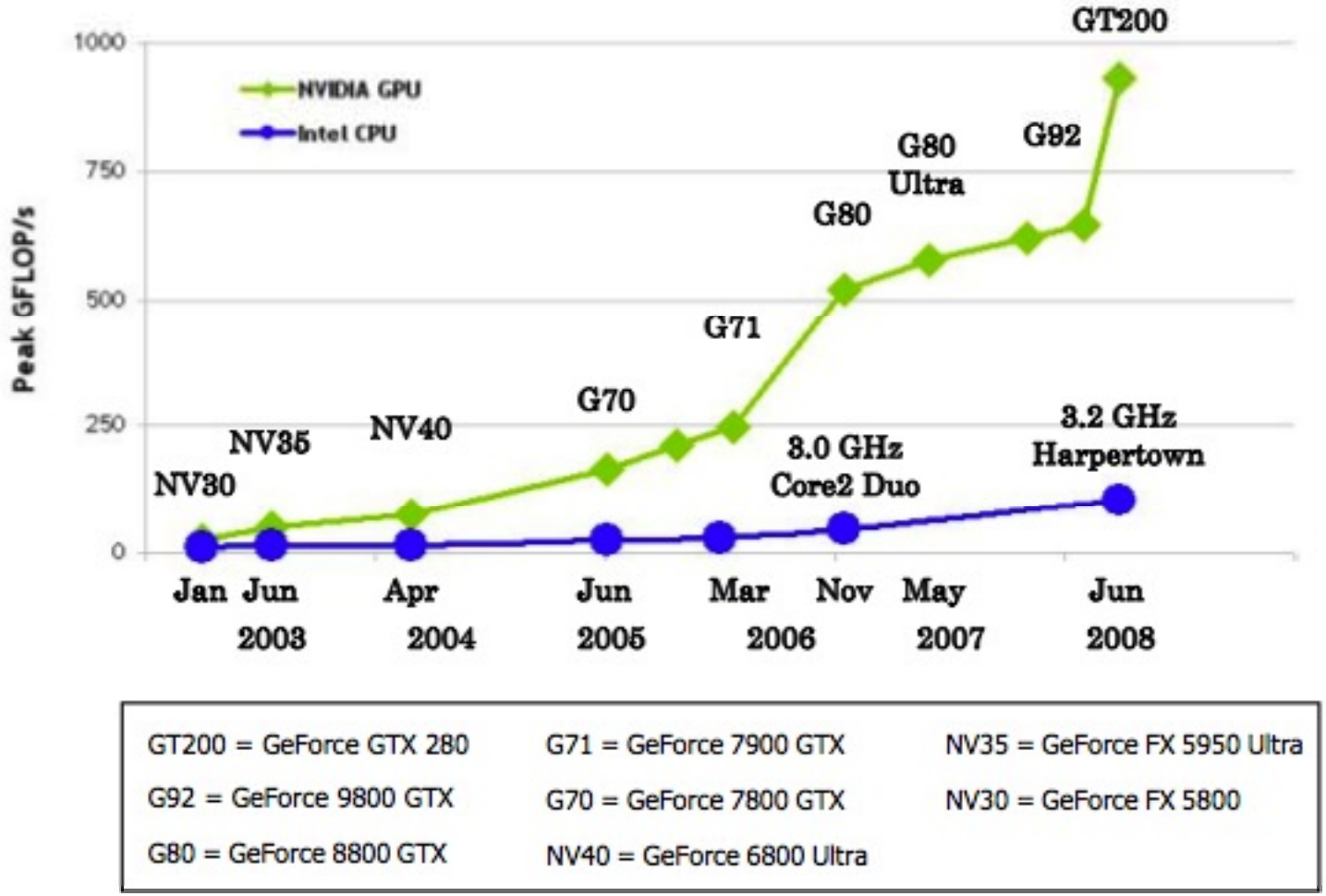


OpenCL Platform Model



CPUs+GPU platforms

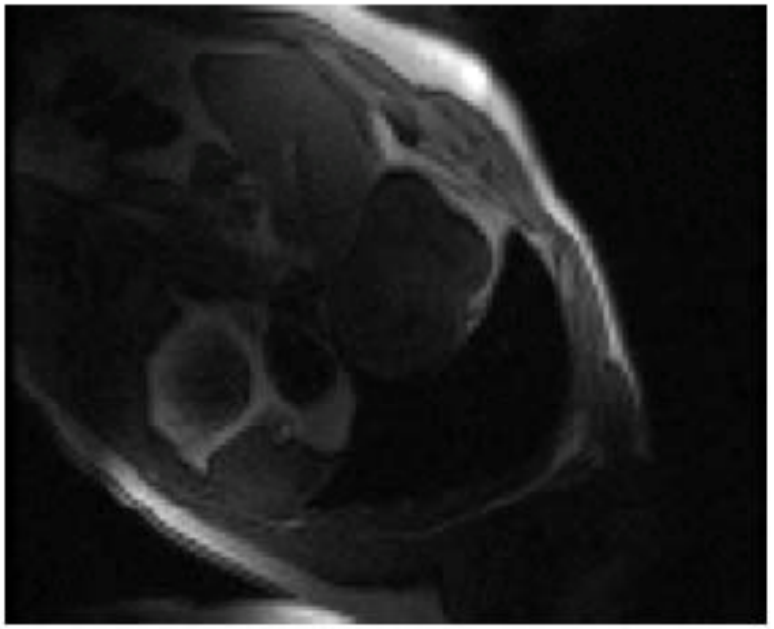
Performance of GPGPU



Note: A cluster of dual Xeon 2.8GZ 30 nodes, Peak performance ~336 GFLOPS

Dynamic Real-Time MRI

Bioengineering Institute, University of Auckland,
IUPS Physiome Project
http://www.bioeng.auckland.ac.nz/movies/database/
cardiovascular_system/textured-heart-beat.mpg



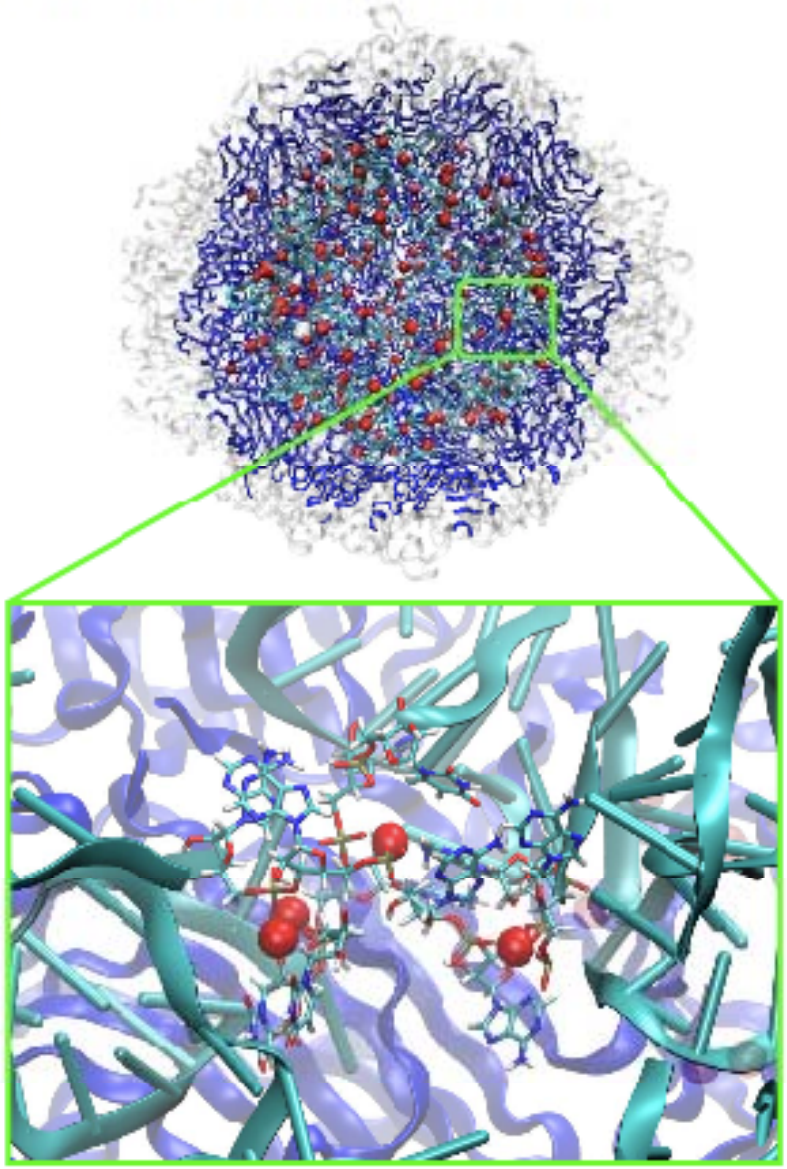
Zhi-Pei Liang's Research Group, Beckman Institute, UIUC
Used with permission of Justin Haldar

G80 GPU is 245x CPU

© Haoron Yi and Sam Stone, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
© David Kirk/NVIDIA

Preparing Virus for Molecular Simulation

- Key task: placement of ions inside and around the virus
- 110 CPU-hours on SGI Altix Itanium2
- Larger viruses could require thousands of CPU-hours
- 27 GPU-minutes on G80 GPU
- Over 240 times faster - ion placement can now be done on a desktop machine!



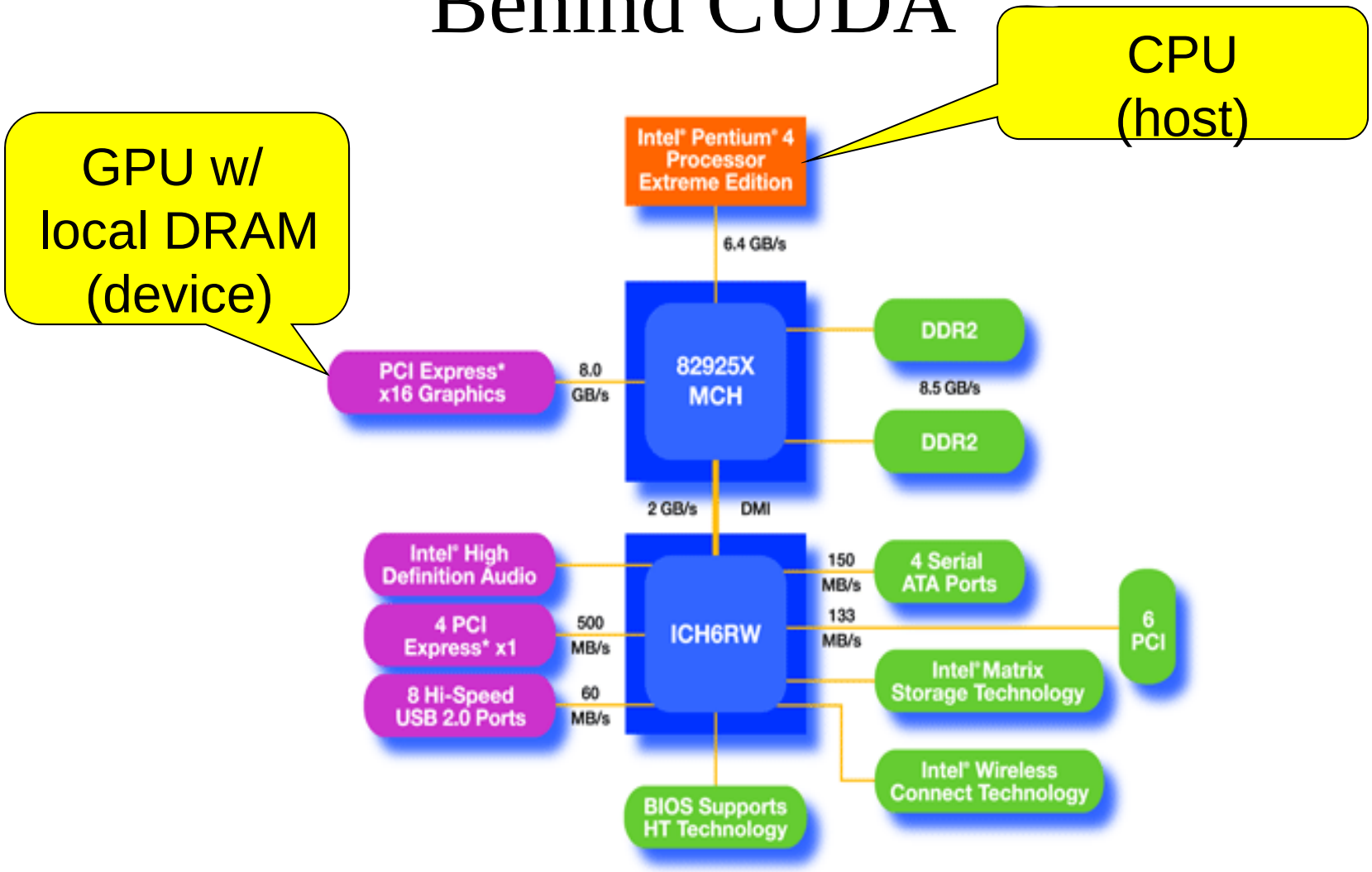
John Stone
Beckman Institute, University of Illinois

CUDA

- “Compute Unified Device Architecture”
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

An Example of Physical Reality Behind CUDA



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

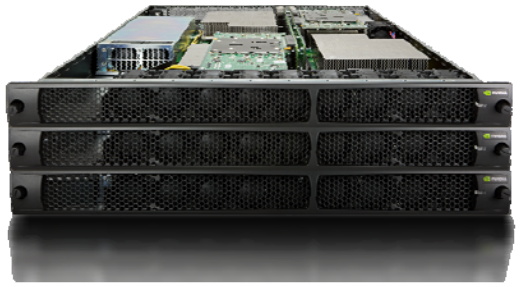
Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture
  - Via a separate HW interface
  - In laptops, desktops, workstations, servers

GeForce 8800

- Programmable in C with CUDA tools
- Multithreaded SIMD model uses application data parallelism and thread parallelism

Tesla D870

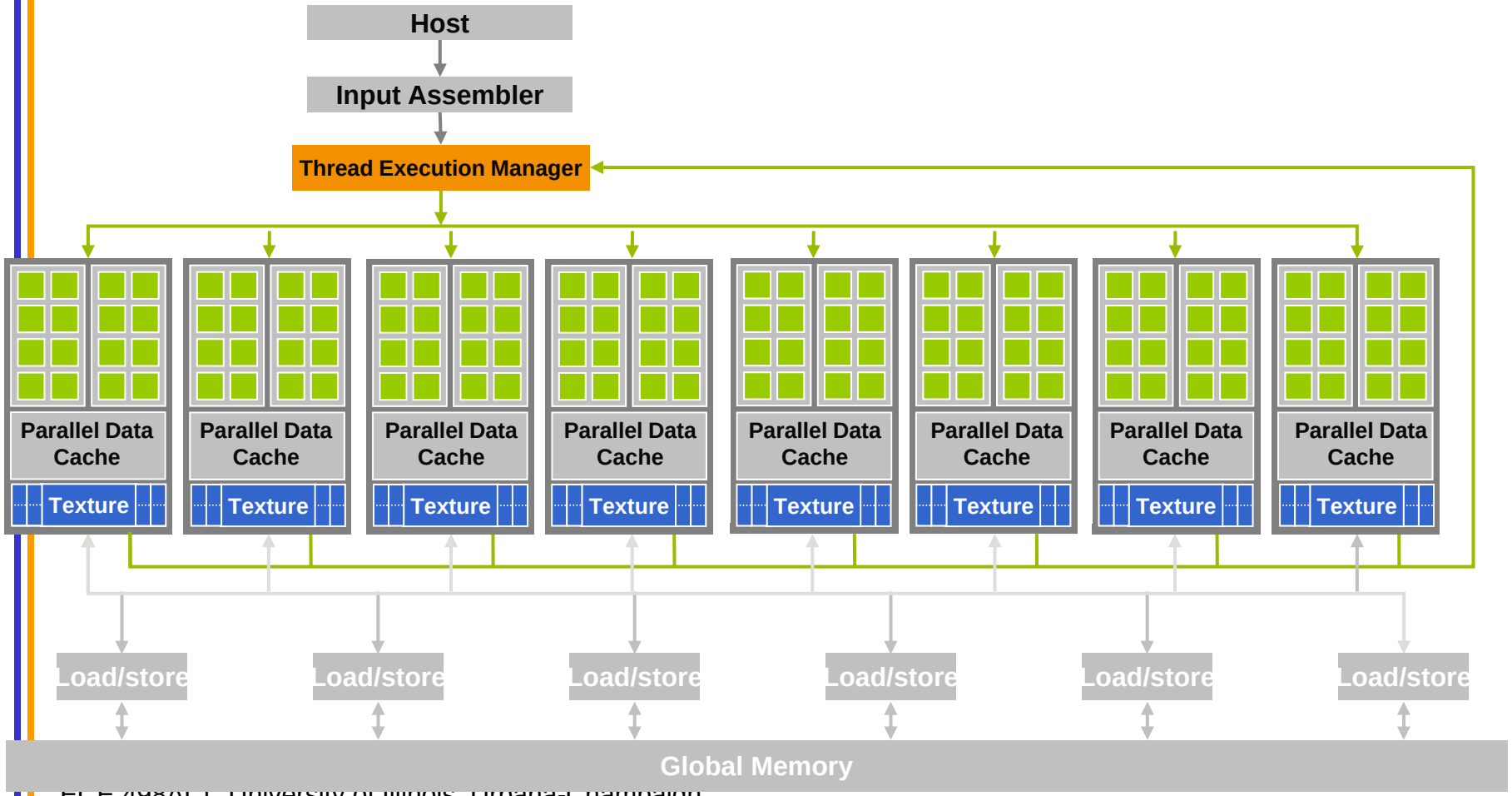


Tesla S870

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

GeForce 8800

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU



Introduction to CUDA programming

These materials are excerpted from David Kirk/NVIDIA and Wen-mei W. Hwu And Christian Trefftz / Greg Wolffe's SC08 GPU tutorials

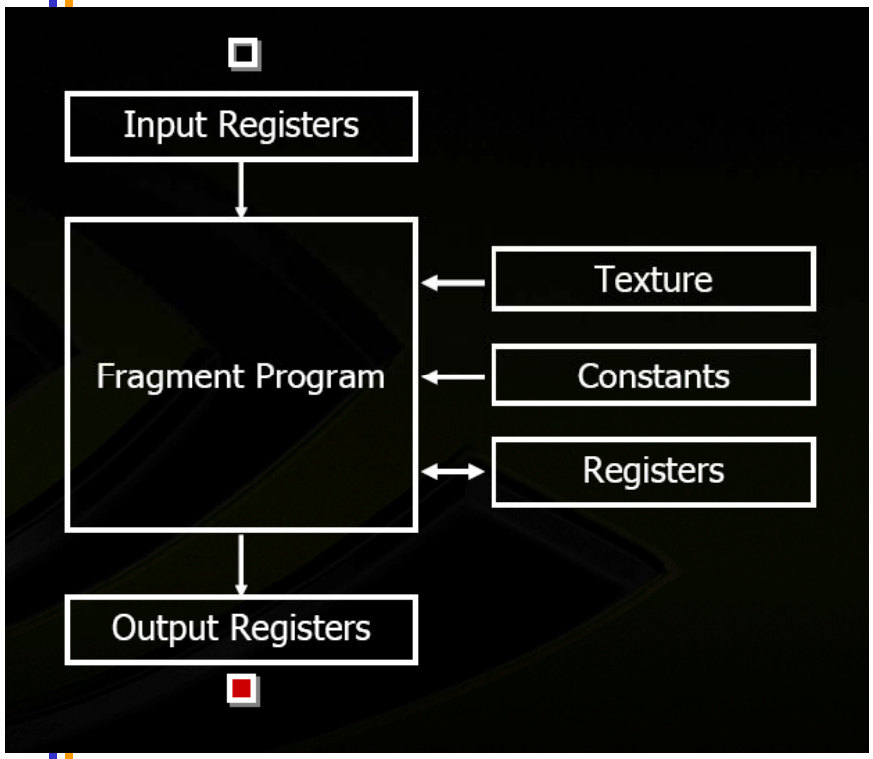
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

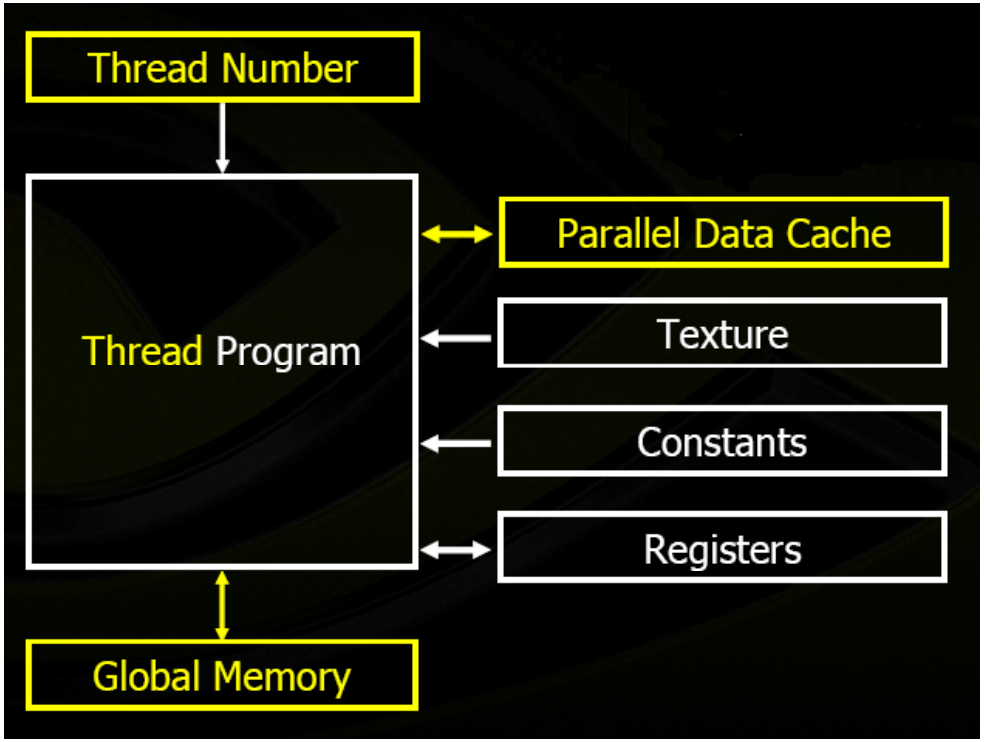
Data-parallel Programming

- Think of the CPU as a massively-threaded co-processor
- Write “kernel” functions that execute on the device -- processing multiple data elements in parallel

- Keep it busy! ⇨ massive threading
- Keep your data close! ⇨ local memory

Pixel / Thread Processing





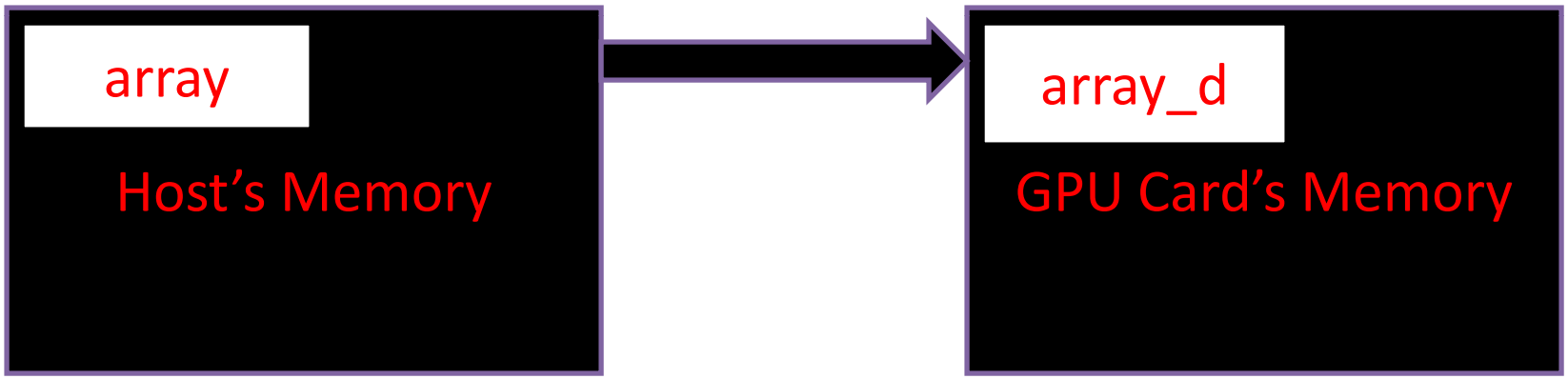
Steps for CUDA Programming

1. Device Initialization
2. Device memory allocation
3. Copies data to device memory
4. Executes kernel (Calling __global__ function)
5. Copies data from device memory (retrieve results)

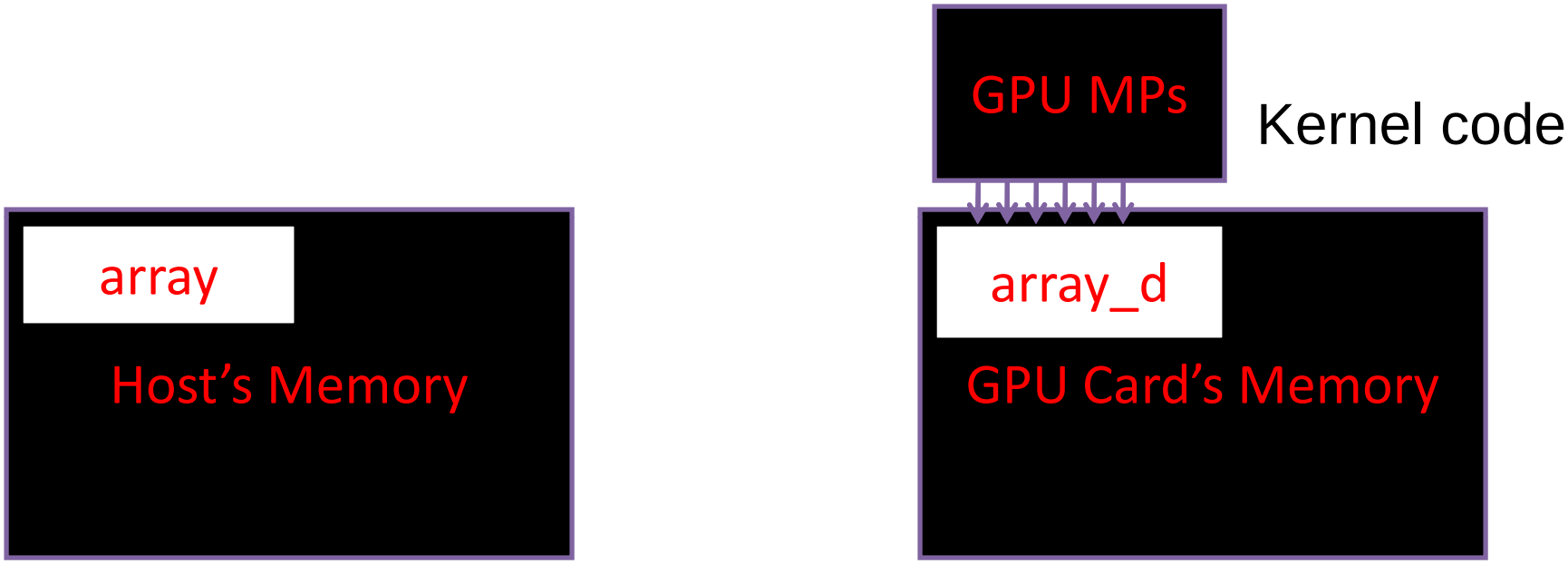
Initially:

Allocate Memory in the GPU card

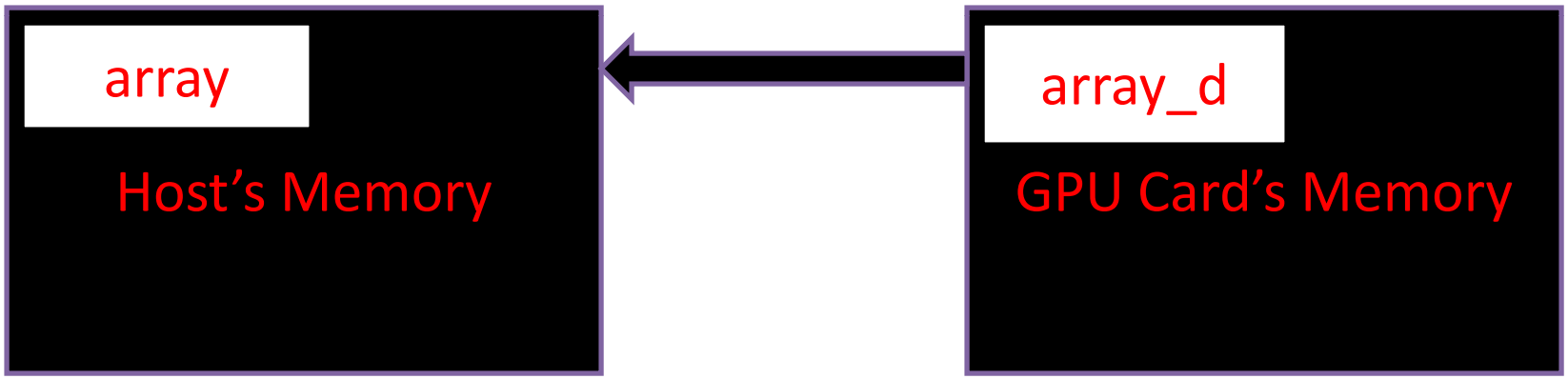
Copy content from the host's memory to the GPU card memory



Execute code on the GPU



Copy results back to the host memory



Steps for CUDA Programming

1. Device Initialization
2. Device memory allocation
3. Copies data to device memory
4. Executes kernel (Calling __global__ function)
5. Copies data from device memory (retrieve results)

Hello World

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
  {

}

int main() {      // Kernel invocation
  vecAdd<<<1, N>>>(A, B, C);
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

Hello World

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
   {
        int i = threadIdx.x;
        C[i] = A[i] + B[i];
}

int main() {      // Kernel invocation
   vecAdd<<<1, N>>>(A, B, C);
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

Extended C

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

Initialize Device calls

- *cudaSetDevice(device)* is for selecting the device associated to the host thread.
- *cudaGetDeviceCount(&devicecount)* is for getting number of devices.
- *cudaGetDeviceProperties(&deviceProp,device)* is for retrieving device's properties

- Note: *cudaSetDevice()* must be called before any __global__ function, otherwise device 0 is automatically selected.

CUDA Language concept

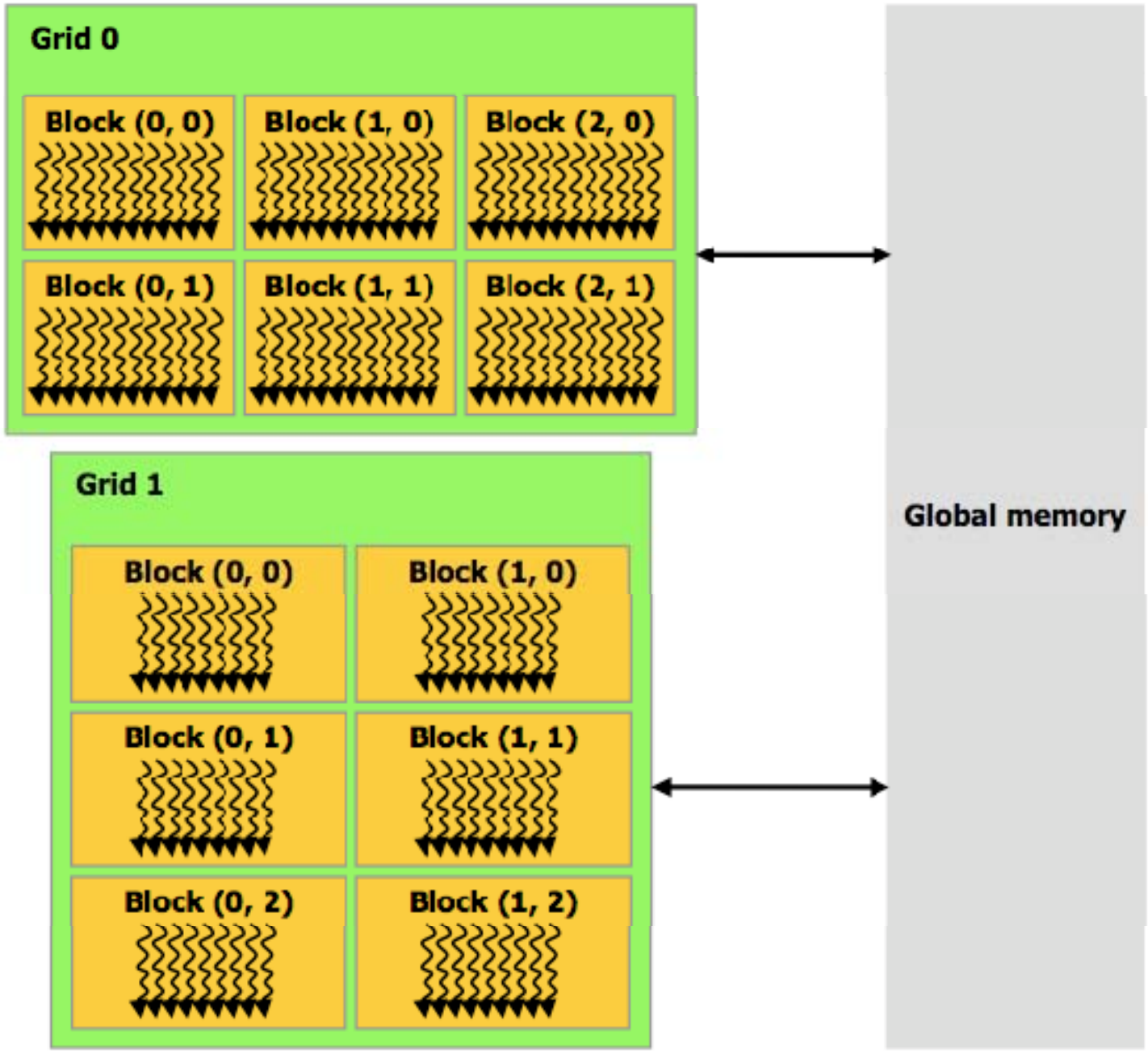
- CUDA Programming Model
- CUDA Memory Model

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

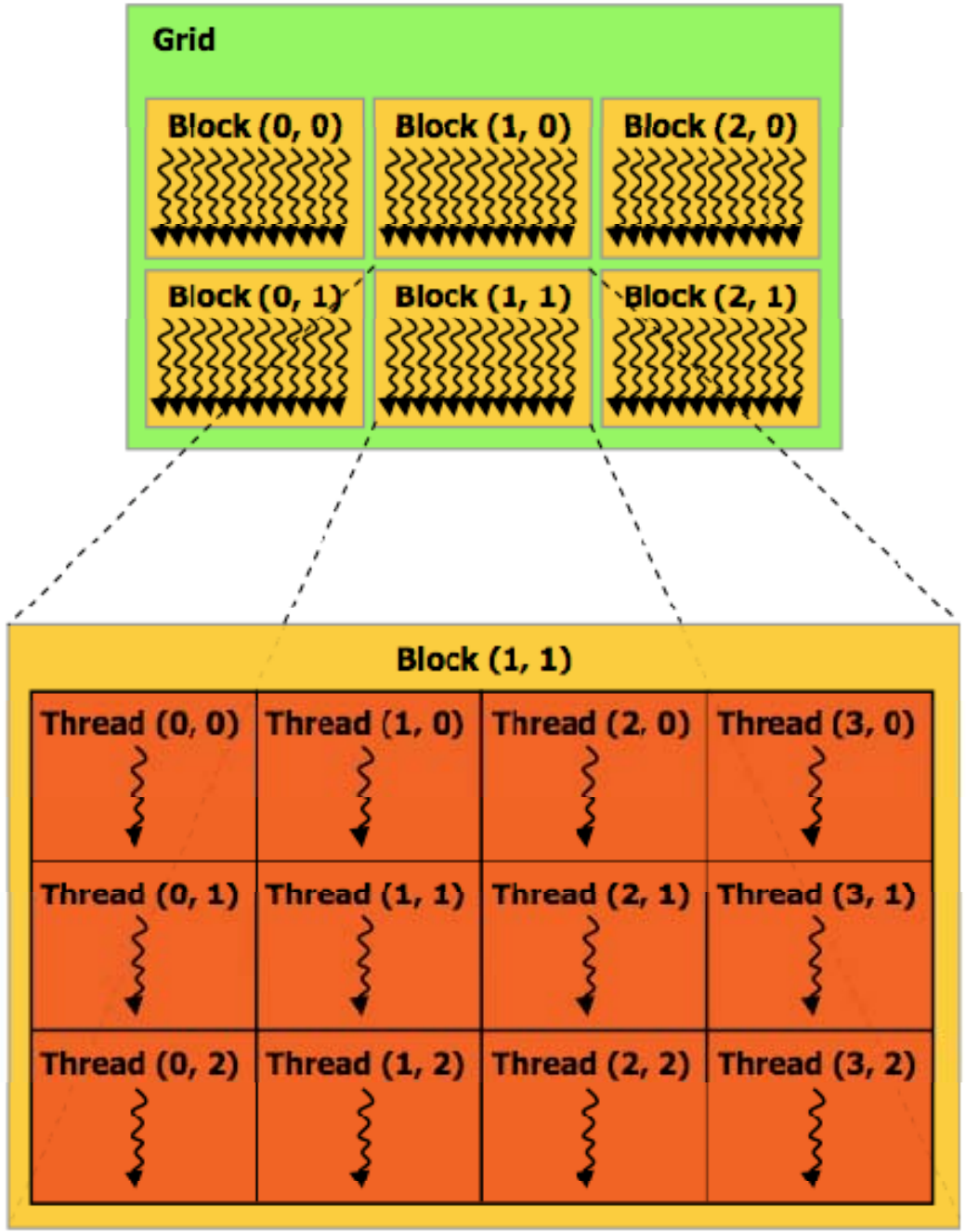
Some Terminology

- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = group of SIMD threads that execute a kernel and can communicate via shared memory

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007


Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 1
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
Block (0, 2)
Block (1, 2)
Global memory

© Da
ECE



© David K
ECE 498A

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

- CUDA threads vs CPU threads
  - CUDA thread switching is free
  - CUDA uses many threads per core

© NVIDIA Corporation 2007

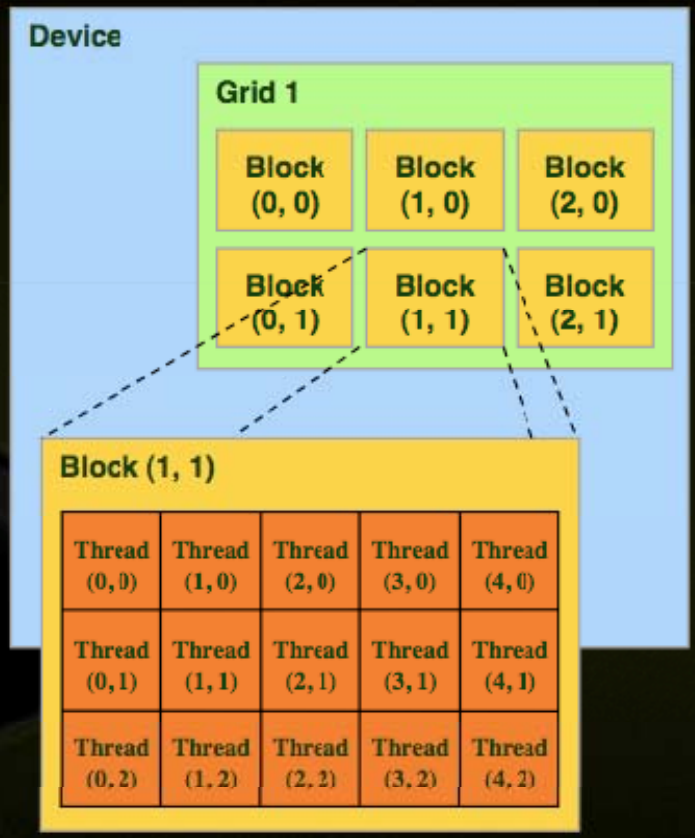
Thread Hierarchy

- 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

© NVIDIA Corporation 2007

IDs and Dimensions

- **Threads:**
  - 3D IDs, unique within a block
- **Blocks:**
  - 2D IDs, unique within a grid
- **Dimensions set at launch time**
  - Can be unique for each section
- **Built-in variables:**
  - threadIdx, blockIdx
  - blockDim, gridDim



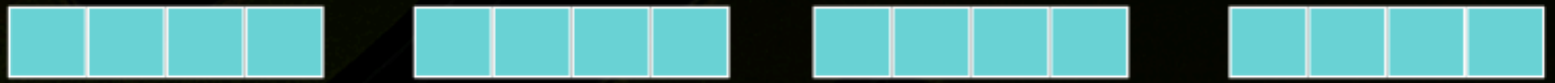
© NVIDIA Corporation 2007

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

```
int idx = blockDim.x * blockId.x + threadIdx.x;
```



blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

© NVIDIA Corporation 2007

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a, b, 16);
}
```

CUDA program

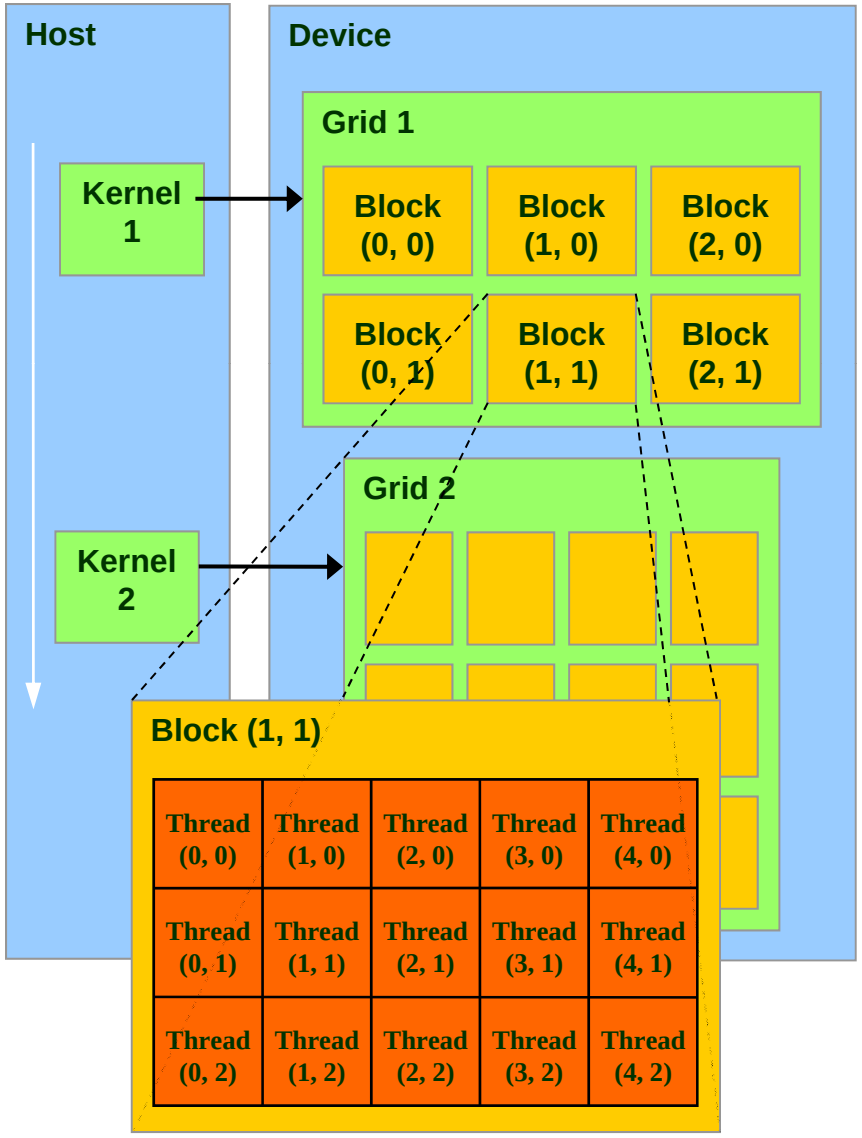
```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if( idx < N)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_gpu<<< 4, 4 >>>(a, b, 16);
}
```

© NVIDIA Corporation 2007

Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- 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



Courtesy: NDVIA

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

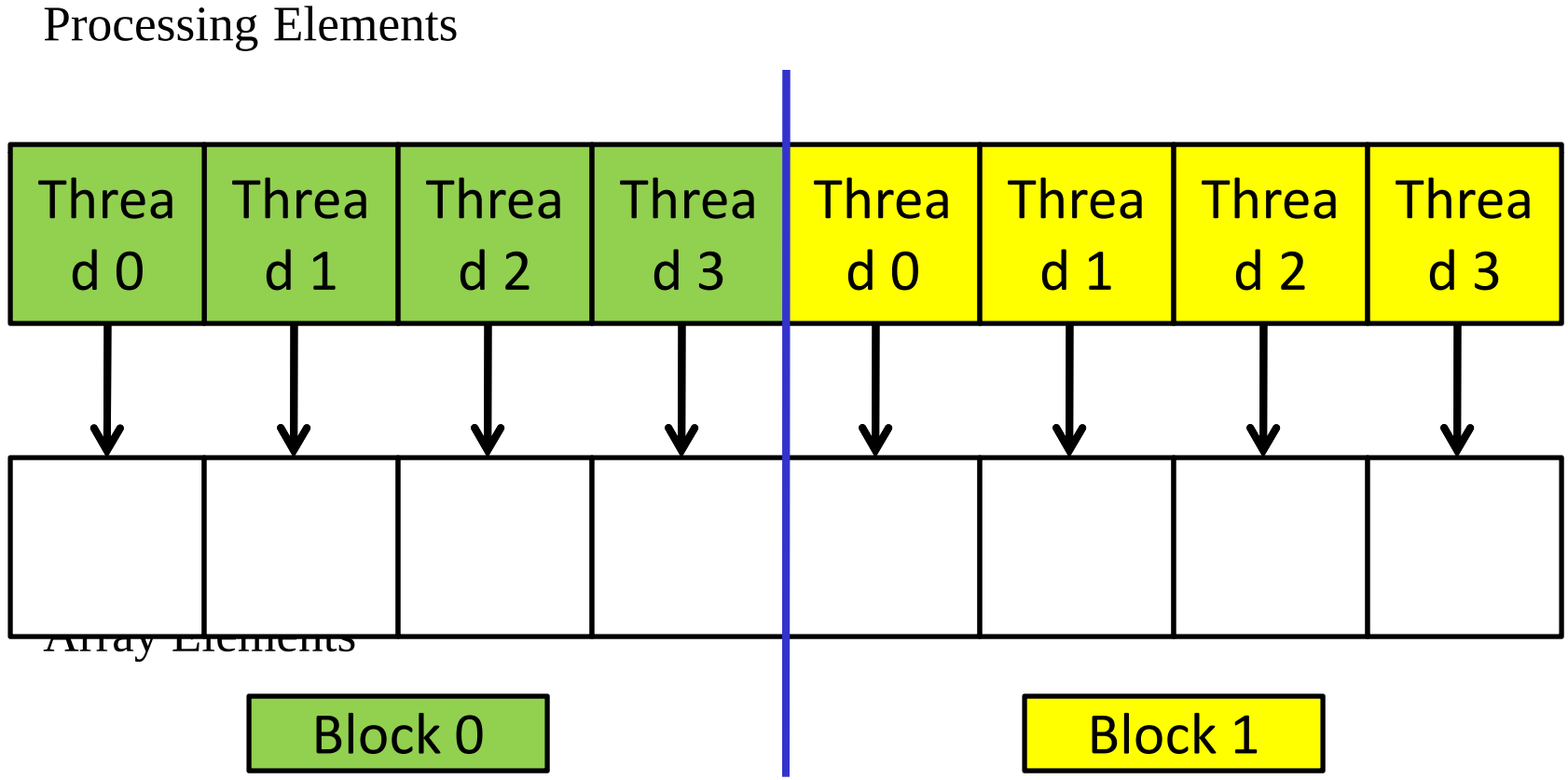
What are those blockIds and threadIds?

```
blockIdx.x is a built-in variable in
  CUDA that returns the blockId in the
  x axis of the block that is
  executing this block of code
threadIdx.x is another built-in
  variable returns the threadId in the
  x axis of the thread that is being
  executed by this stream processor
```

- Example code in the kernel:

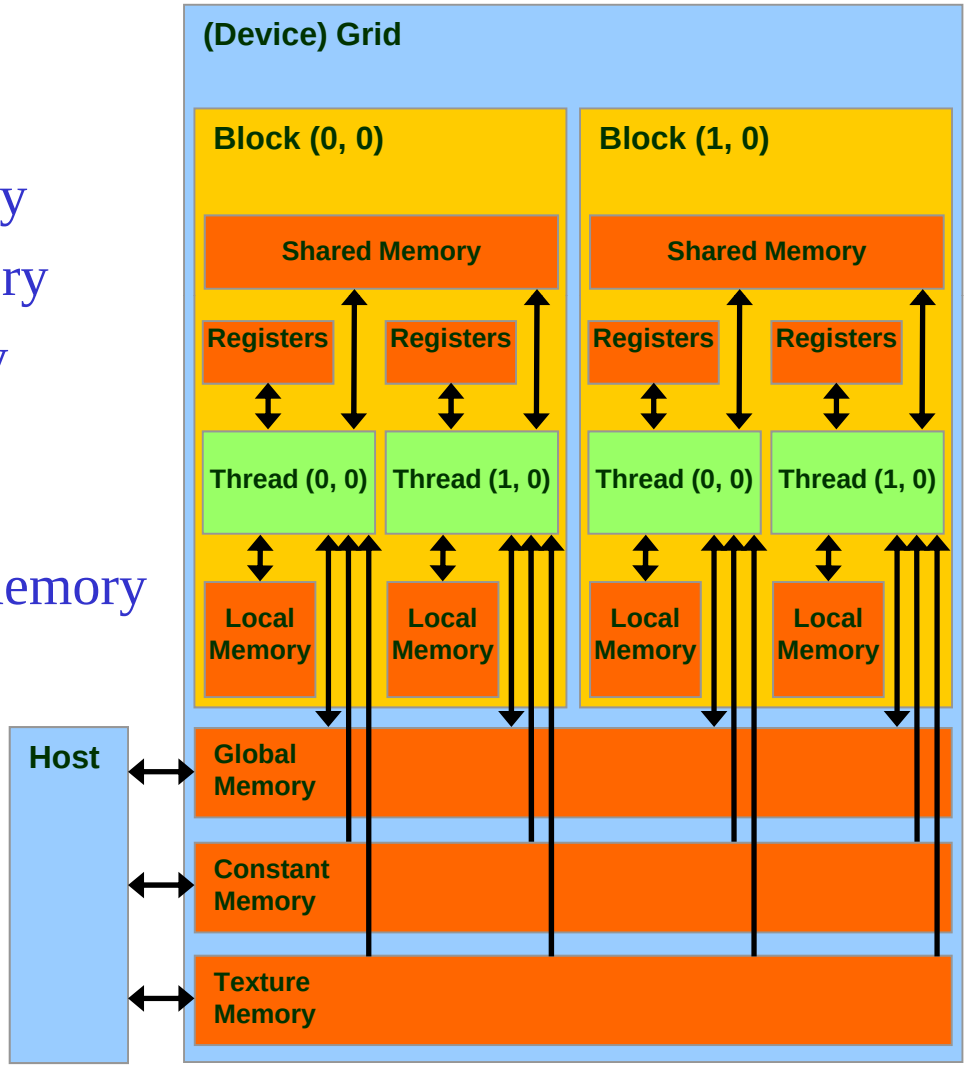
```
x=blockIdx.x*BLOCK_SIZE+threadIdx.x;
block_d[x] = blockIdx.x;
thread_d[x] = threadIdx.x;
```

In the GPU:


Processing Elements
Thread 0
Thread 1
Thread 2
Thread 3
Thread 0
Thread 1
Thread 2
Thread 3
Array Elements
Block 0
Block 1

CUDA Device Memory Model Overview

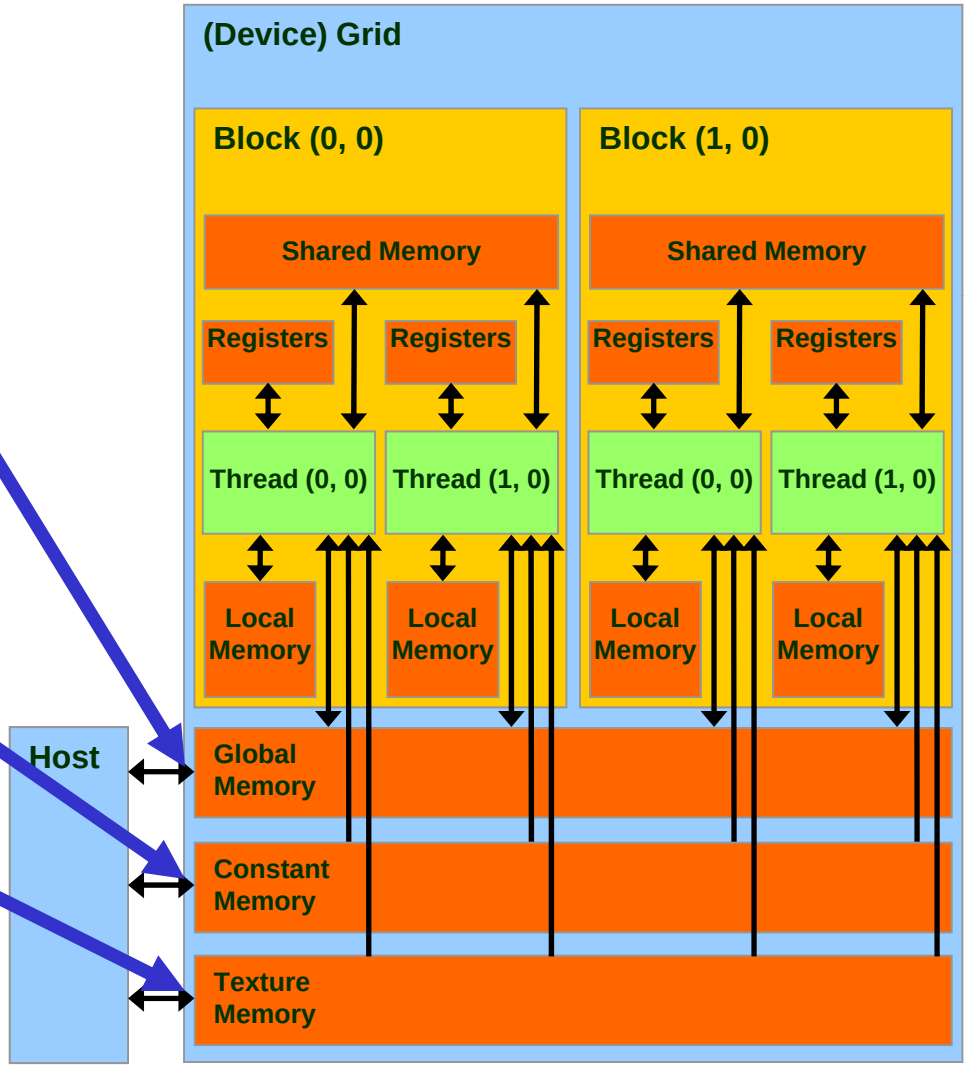
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



Courtesy: NDVIA

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

C-Code Example to Add Arrays

CPU C program

```
void add_matrix_cpu
            (float *a, float *b, float *c, int N)
{
    int i, j, index;
    for (i=0;i<N;i++) {
     for (j=0;j<N;j++) {
        index =i+j*N;
        c[index]=a[index]+b[index];
        }
    }
}
void main()
{
   ...
      add_matrix(a,b,c,N);
}
```

CUDA C program

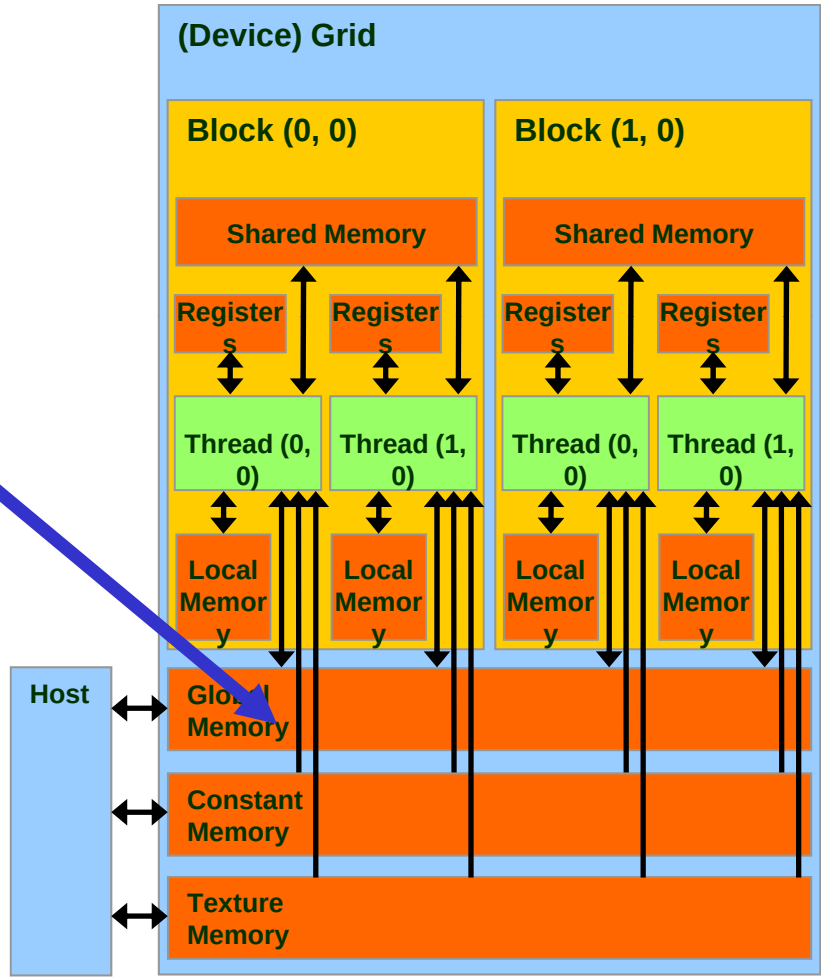
```
__global__ void add_matrix_gpu
            (float *a, float *b, float *c, int N)
{
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    int j=blockIdx.y*blockDim.y+threadIdx.y;
    int index =i+j*N;
    if( i <N && j <N) c[index]=a[index]+b[index];
}


void main()
{
    dim3 dimBlock (blocksize,blocksize);
    dim3 dimGrid (N/dimBlock.x,N/dimBlock.y);
    add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

CUDA Device Memory Allocation

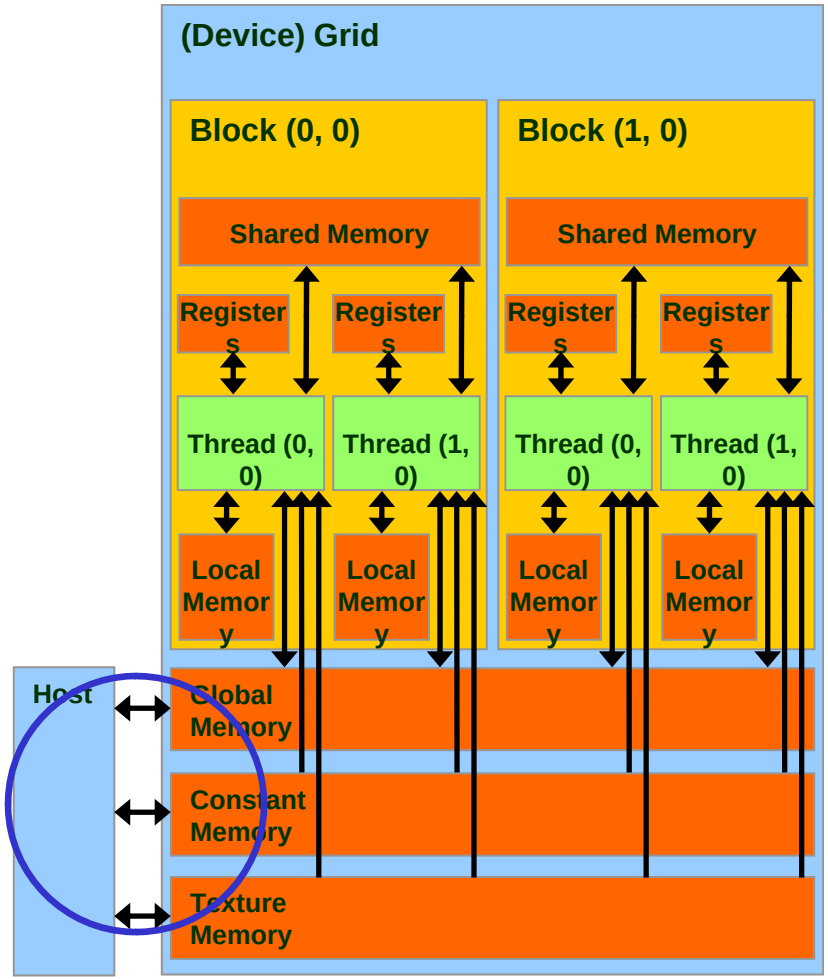
- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous in CUDA



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

Language Extensions: Variable Type Qualifiers

	Memory	Scope	Lifetime
`__device__ __local__ int LocalVar;`	local	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

Access Times

- 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, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

CUDA function calls restrictions

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

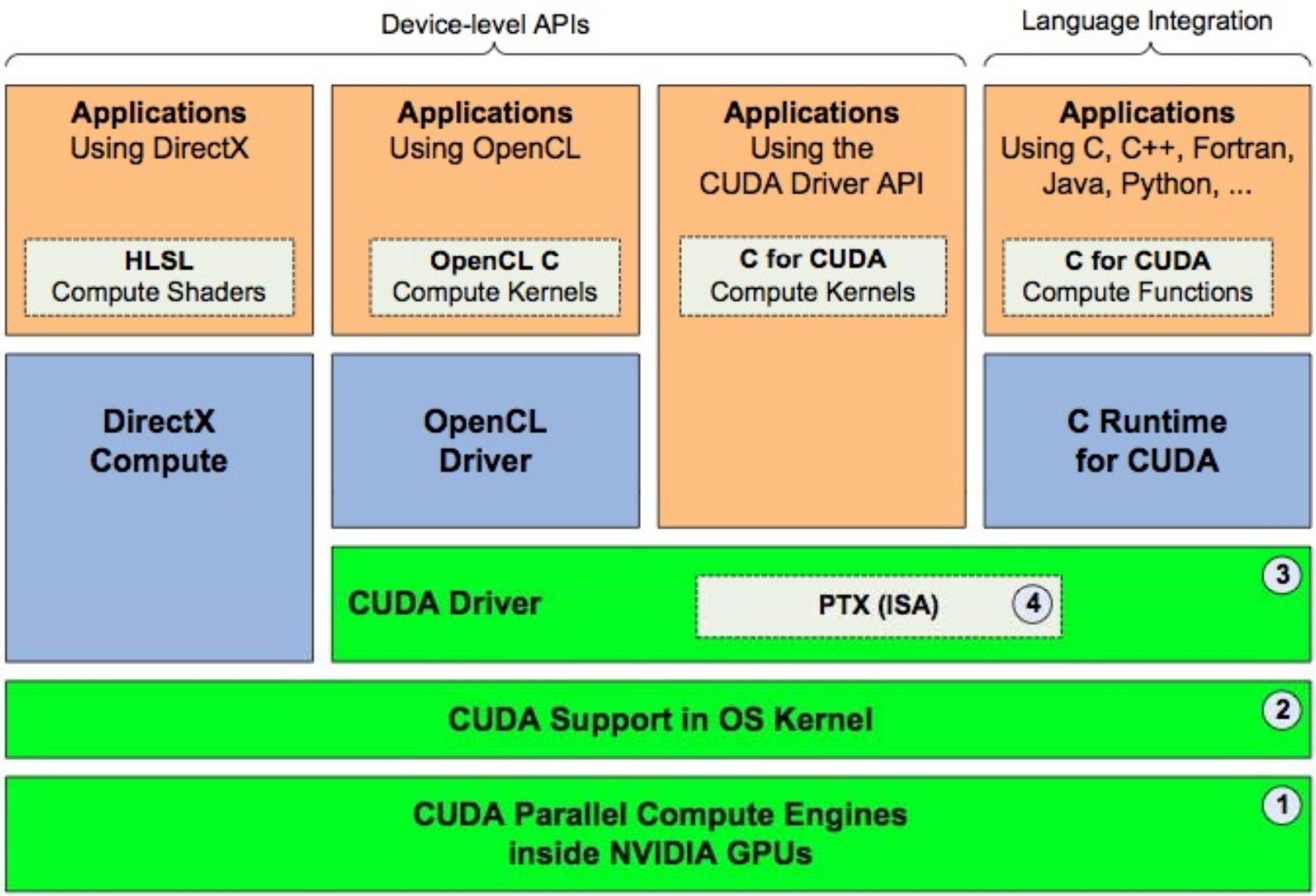
```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per
   block
size_t SharedMemBytes = 64; // 64 bytes of shared
   memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
   >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

Resources on line

- http://www.ddj.com/hpc-high-performance-computing/207200659
- http://www.nvidia.com/object/cuda_home.html#
- http://www.nvidia.com/object/cuda_learn.html


Device-level APIs
Language Integration
Applications
Using DirectX
HLSL
Compute Shaders
Applications
Using OpenCL
OpenCL C
Compute Kernels
Applications
Using the
CUDA Driver API
C for CUDA
Compute Kernels
Applications
Using C, C++, Fortran,
Java, Python, ...
C for CUDA
Compute Functions
DirectX
Compute
OpenCL
Driver
C Runtime
for CUDA
CUDA Driver
PTX (ISA)
4
3
CUDA Support in OS Kernel
2
CUDA Parallel Compute Engines
inside NVIDIA GPUs
1

© E

Case Studies

- GPU-enabled EM algorithm – applications in Bioinfomatics with Genome Institute, National BIOTEC center, Thailand
- GPU application for Monte Carlos Integration - with Amir Fabin UTA & Dick Greenwood
- GPU checkpoint/restart

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007

Thanks

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007