ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

OPERATING SYSTEMS & PARALLEL COMPUTING

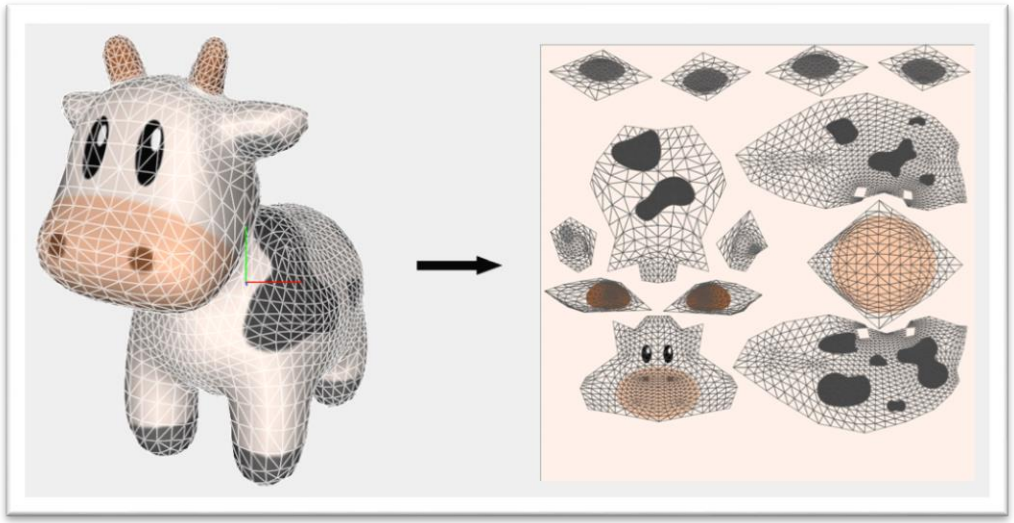
GPU & CUDA

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

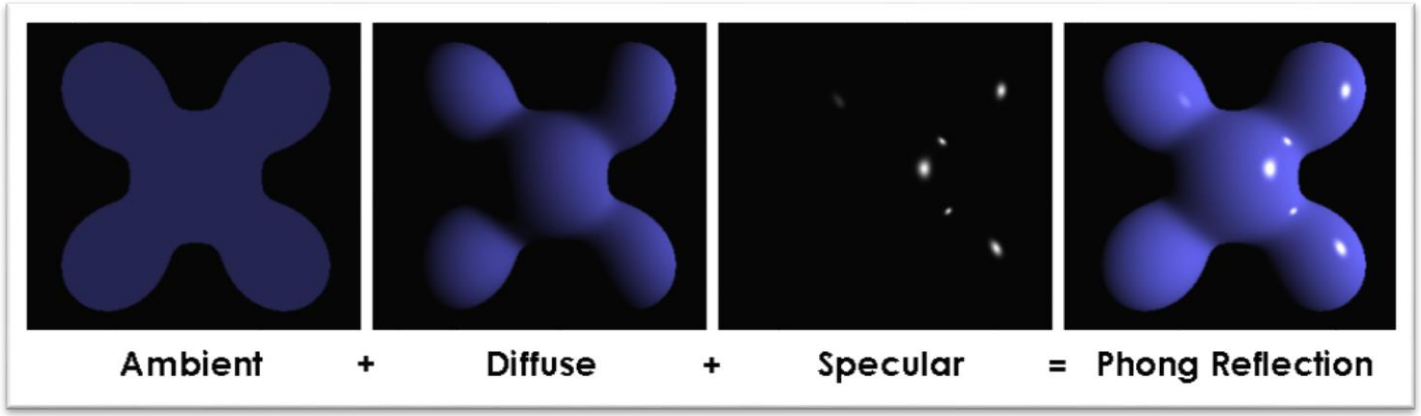
GPU Overview

Graphic Processing – Some History

- 1990s: Real-time 3D rendering for video games were becoming common
  - Doom, Quake, Descent, … (Nostalgia!)
- 3D graphics processing is immensely computation-intensive



Texture mapping



Shading

Warren Moore, “Textures and Samplers in Metal,” Metal by Example, 2014

Gray Olsen, “CSE 470 Assignment 3 Part 2 - Gourad/Phong Shading,” grayolsen.com, 2018

Graphic Processing – Some History

- Before 3D accelerators (GPUs) were common
- CPUs had to do all graphics computation, while maintaining framerate!
  - Many tricks were played

Doom (1993) : “Affine texture mapping”

- Linearly maps textures to screen location, disregarding depth
- Doom levels did not have slanted walls or ramps, to hide this

Graphic Processing – Some History

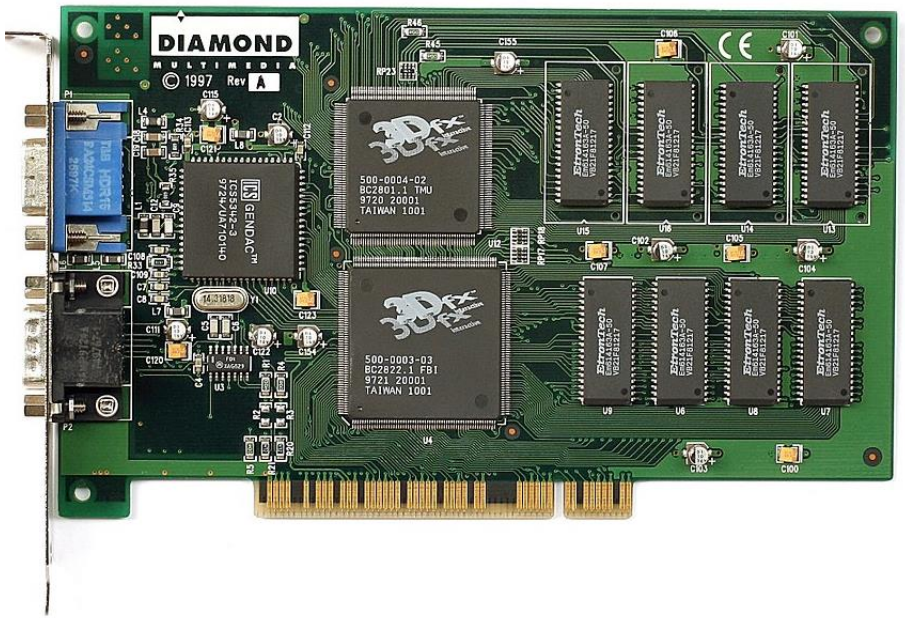
- Before 3D accelerators (GPUs) were common
- CPUs had to do all graphics computation, while maintaining framerate!
  - Many tricks were played

Quake III arena (1999) : “Fast inverse square root” magic!

```
float Q_rsqrt( float number )
{
    const float x2 = number * 0.5F;
    const float threehalfs = 1.5F;

    union {
        float f;
        uint32_t i;
    } conv = {number}; // member 'f' set to value of 'number'.
    conv.i  = 0x5f3759df - ( conv.i >> 1 );
    conv.f  *= ( threehalfs - ( x2 * conv.f * conv.f ) );
    return conv.f;
}
```

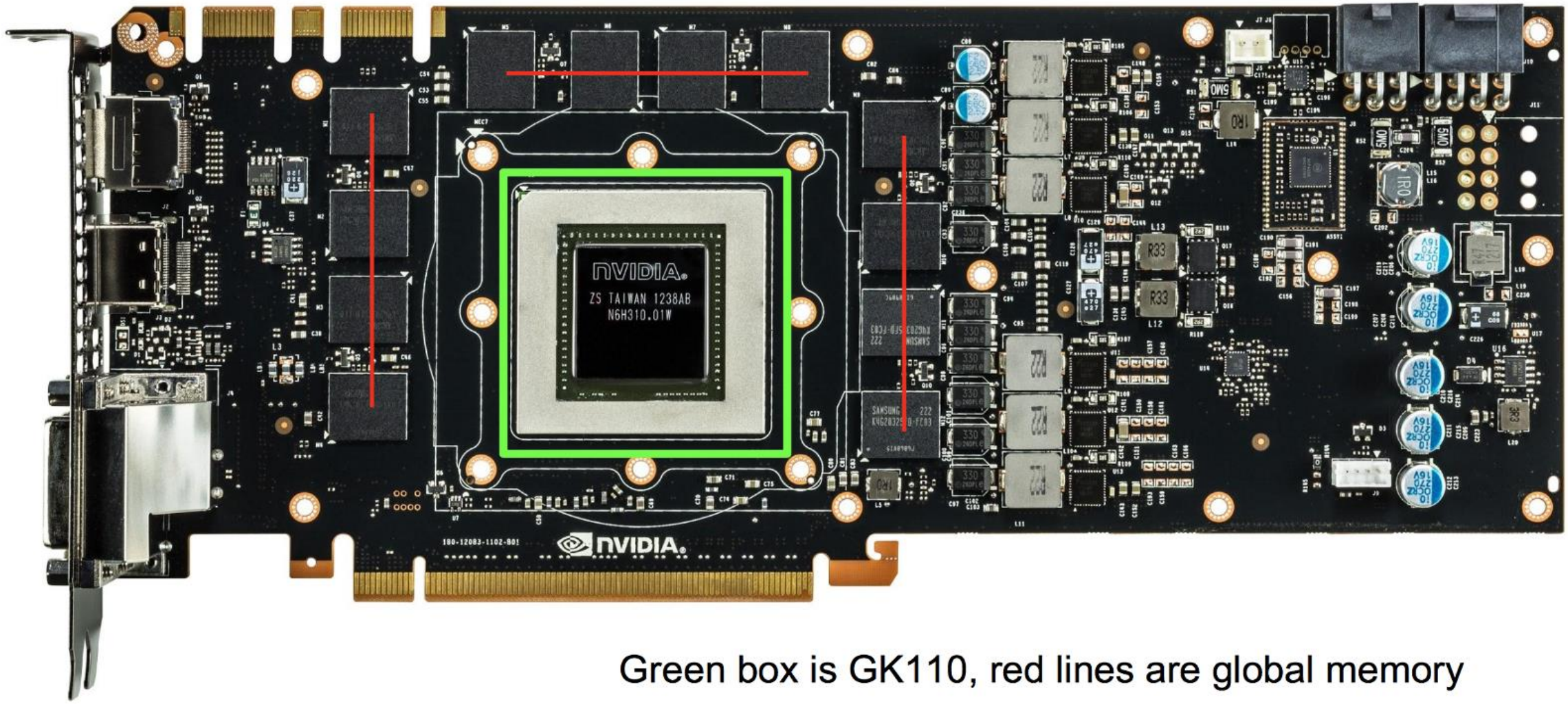
Introduction of 3D Accelerator Cards

- Much of 3D processing is short algorithms repeated on a lot of data
  - pixels, polygons, textures, …
- Dedicated accelerators with simple, massively parallel computation



A Diamond Monster 3D, using the Voodoo chipset (1997)
(Konstantin Lanzet, Wikipedia)

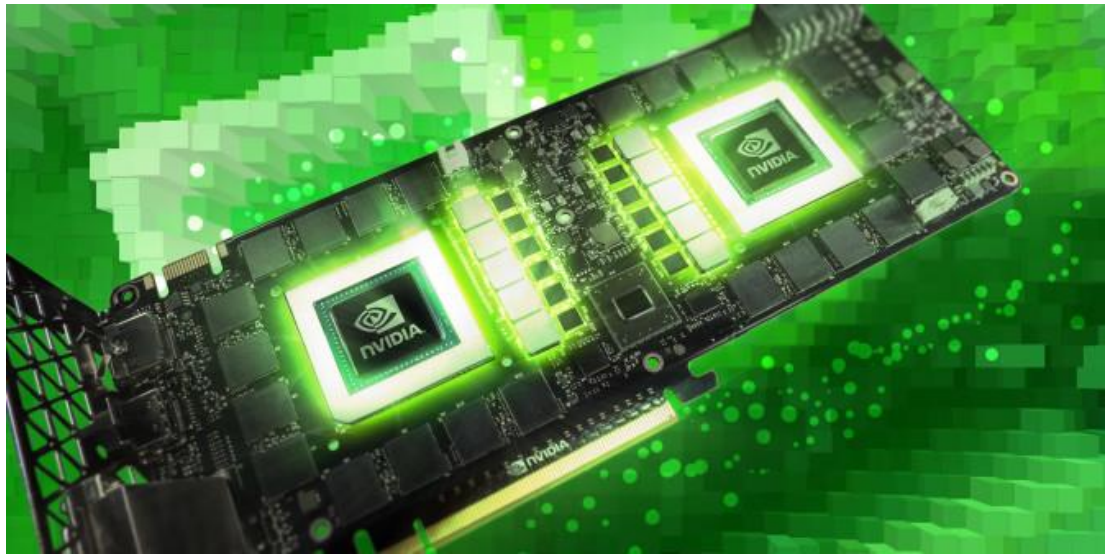
Graphical Processing Unit(GPU)



Green box is GK110, red lines are global memory

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

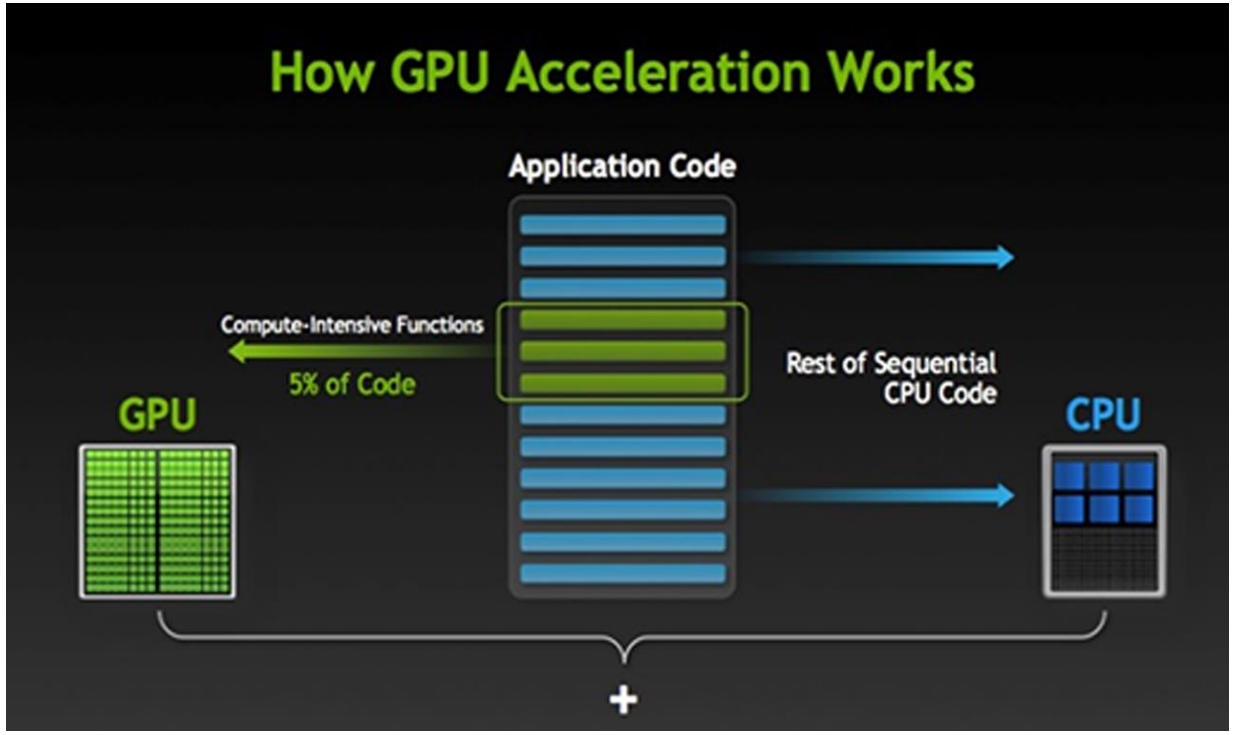
Contents:



- Overview
- Types Of Memory
- System Memory v/s Graphics Memory
- Most Used GPU’s
- Important Parts of GPU’s
- Evolution of GPU’s
- Types OF API’s(Application Programming Interfaces)
- DirectX/Direct3D

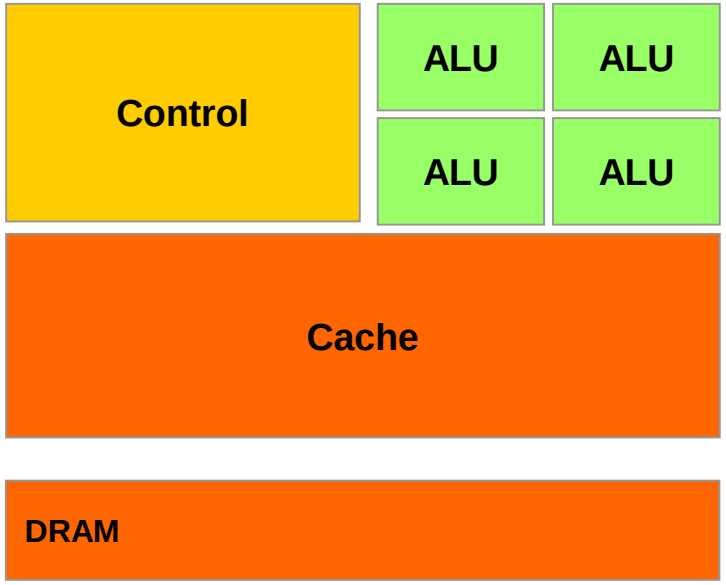
Overview

- A graphics processing unit (GPU), also occasionally called visual processing unit (VPU), is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the building of images in a frame buffer intended for output to a display.
- GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles. Modern GPUs are very efficient at manipulating computer graphics and image processing.
- Their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms that process large blocks of data in parallel. In a personal computer, a GPU can be present on a video card or embedded on the motherboard.

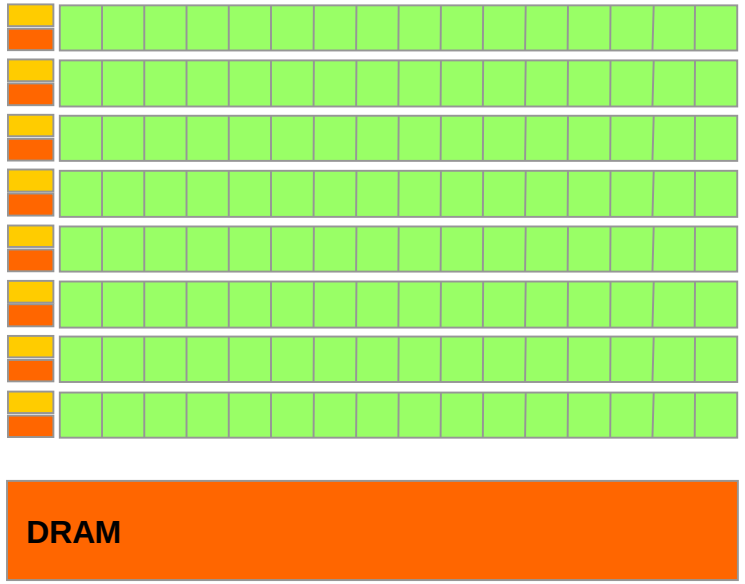


❑ CPU v/s GPU:

- CPUs and GPUs have a lot in common.
- They are both silicon-based microprocessors.
- At the same time, they are substantially different, and they are deployed for different roles.



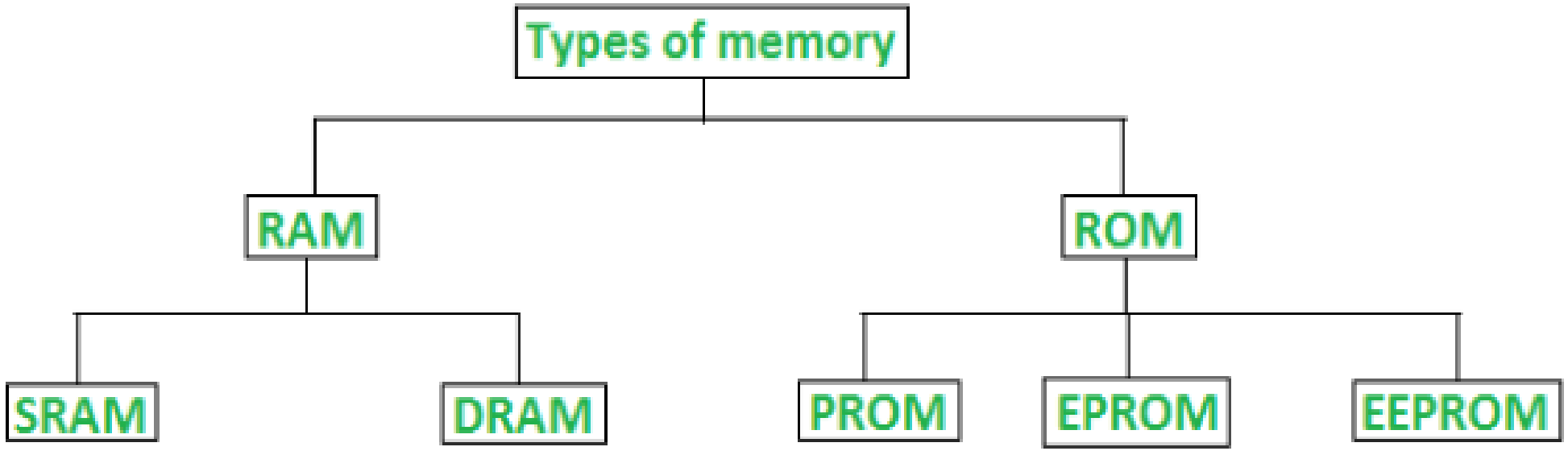
CPU



GPU

- A GPU uses thousands of smaller and more efficient cores for a massively parallel architecture aimed at handling multiple functions at the same time.
- They are 50–100 times faster in tasks that require multiple parallel processes, such as machine learning and big data analysis.

Types Of Memory



Classification of computer memory

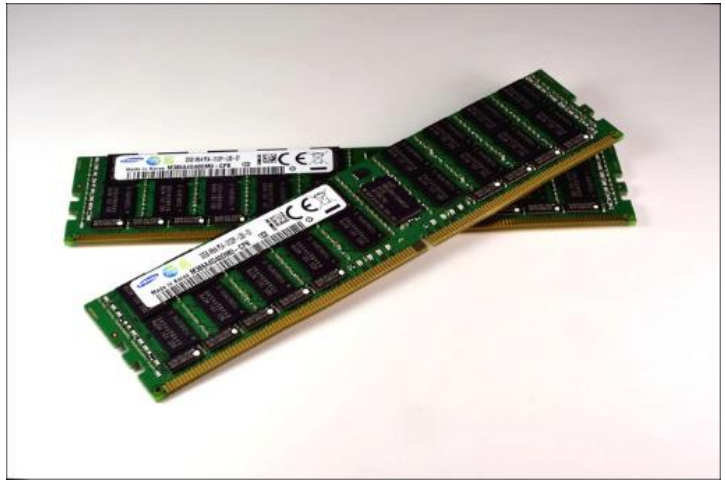
Types Of Memory

- DDR:Transfer Data at Twice the clock cycle
- DDR2:Faster and more efficient than DDR
- DDR3:Consumes 30% less Power and Faster
- DDR4:50% increased performance and bandwidth capabilities while decreasing voltage
- GDDR4: Multi-Preamble to reduce data transmission delay
- GDDR5:Type of Synchronous Graphics RAM and with a high bandwidth interface designed for use in graphics cards, game consoles, and high-performance computing.

Type	Memory clock rate (MHz)	Bandwidth (GB/s)
DDR	200-400	1.6-3.2
DDR2	400-1066.67	3.2-8.533
DDR3	800-2133.33	6.4-17.066
DDR4	1600-4866	12.8-25.6
GDDR4	3000–4000	160–256
GDDR5	1000–2000	288–336.5
GDDR5X	1000–1750	160–673
HBM	250–1000	512–1024

System Memory v/s Graphics Memory

System Memory	Graphics Memory
Considered as System RAM	Considered as Dedicated RAM
Lower Speed than Graphics Memory	Higher Speed Than System Memory
Quicker for performance	It is Wider and faster for bandwidth



Important Parts of Graphics Unit:

❑ Heat Sink:

- A heat sink is mounted on most modern graphics cards. A heat sink spreads out the heat produced by the graphics processing unit evenly throughout the heat sink and unit itself. The heat sink commonly has a fan mounted as well to cool the heat sink and the graphics processing unit.
- Not all cards have heat sinks, for example, some cards are liquid cooled, and instead have a waterblock; additionally, cards from the 1980s and early 1990s did not produce much heat, and did not require heatsinks. Most modern graphics cards need a proper thermal solution.

❑ Video BIOS:

- The video BIOS or firmware contains a minimal program for initial set up and control of the video card.
- It may contain information on the memory timing, operating speeds and voltages of the graphics processor, RAM, and other details which can sometimes be changed.

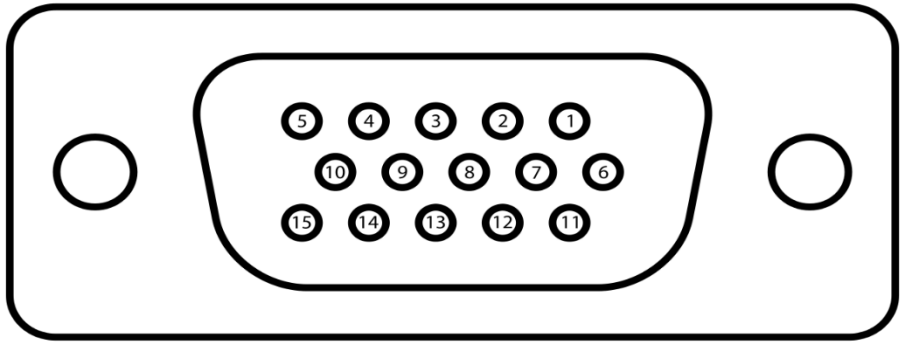
❑ Video Memory:

- The memory capacity of most modern video cards ranges from 1 GB to 12 GB. Since video memory needs to be accessed by the GPU and the display circuitry, it often uses special high-speed or multi-port memory

❑ Output Interfaces:

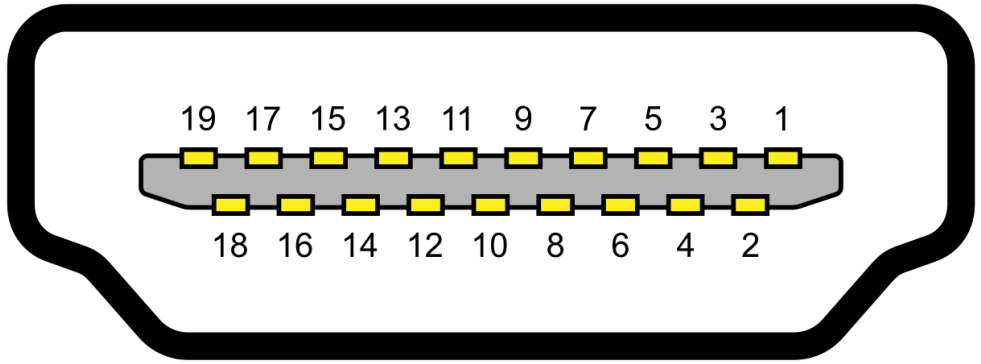
- **Video Graphics Array(VGA)**

The VGA analog interface is used for high definition video including 1080p and higher.
While the VGA transmission bandwidth is high enough to support even higher resolution playback



- **High Definition Multimedia Interface(HDMI)**

HDMI is a compact audio/video interface for transferring uncompressed video data and compressed/uncompressed digital audio data from an HDMI-compliant device

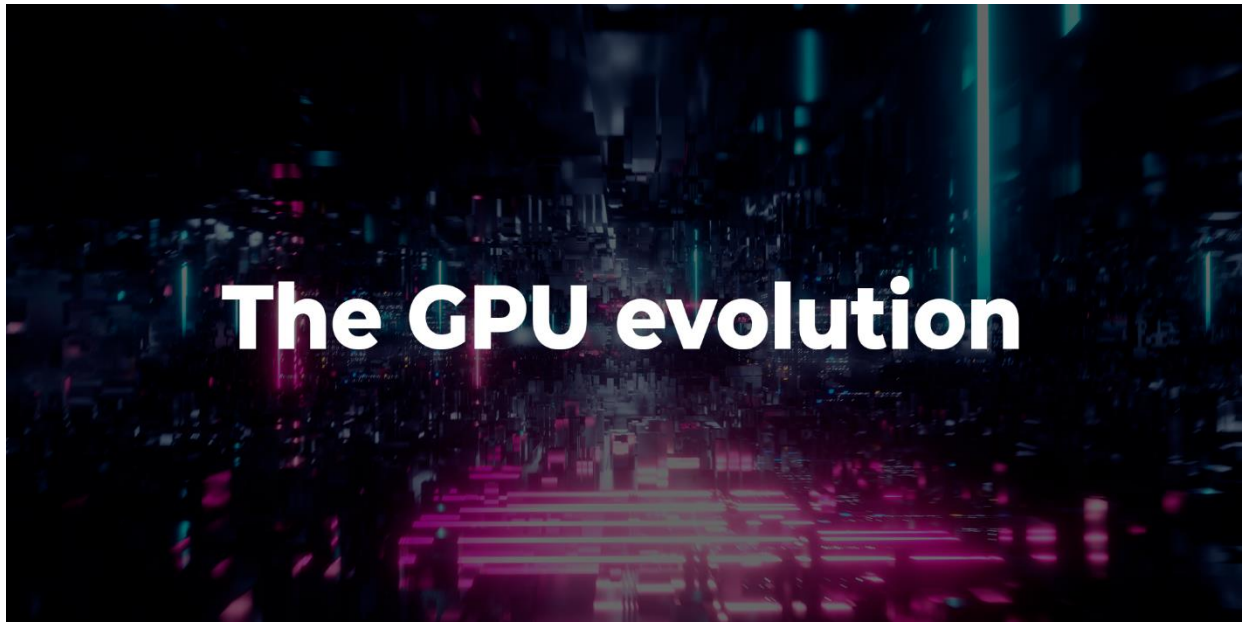


❑ External GPU (EGPU):

- An external GPU is a graphics processor located outside of the housing of the computer.
- External graphics processors are sometimes used with laptop computers.
- Laptops might have a substantial amount of RAM and a sufficiently powerful central processing unit (CPU), but often lack a powerful graphics processor, and instead have a less powerful but more energy-efficient on-board graphics chip..

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

❑ Evolution OF GPU's:



- 1980's – No GPU. PC used VGA controller
- 1990's – Add more function into VGA controller
- 1997 – 3D acceleration functions:
  Hardware for triangle setup and rasterization
  Texture mapping
  Shading
- 2000 – A single chip graphics processor (beginning of GPU term)
- 2005 – Massively parallel programmable processors
- 2007 – CUDA (Compute Unified Device Architecture)

Most Used GPU's:

Most GPUs are designed for a specific usage, real-time 3D graphics or other mass calculations:

1.Gaming

1. GeForce GTX, RTX
2. nVidia Titan X
3. Radeon HD
4. Radeon r5, r7, r9, RX, and Vega series

2.Cloud Gaming

1. nVidia Grid
2. Radeon Sky

3.Workstation

1. nVidia Quadro
2. nVidia Titan X
3. AMD FirePro
4. Radeon Pro

4.Cloud Workstation

1. nVidia Tesla
2. AMD FireStream

5.Artificial Intelligence Cloud

1. nVidia Tesla
2. Radeon Instinct

6.Automated/Driverless car

1. nVidia Drive PX

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

❑ Types of API's(Application Programming Interfaces):

- **Low Level 3D API's:**
  - Direct3D (a subset of DirectX)
  - Glide.
  - Mantle developed by AMD.
  - Metal developed by Apple.
  - MonoGame.
  - OpenGL and the OpenGL Shading Language.
- **High Level 3D API's:**
  - Crystal Space 3D
  - Java 3D
  - OpenGL performer
  - OpenSG
- **Flash Based API's:**
  - Stage 3D is library in Flash11 or later version

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

❑ Direct3D(DirectX):

- Microsoft DirectX is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming and video, on Microsoft platforms.
- Originally, the names of these APIs all began with Direct, such as Direct3D, DirectDraw, DirectMusic, DirectPlay, DirectSound, and so forth. The name DirectX was coined as a shorthand term for all of these APIs (the X standing in for the particular API names) and soon became the name of the collection.
- When Microsoft later set out to develop a gaming console, the X was used as the basis of the name Xbox to indicate that the console was based on DirectX technology.
- Direct3D is widely used in the development of video games for Microsoft Windows and the Xbox line of consoles.
- As Direct3D is the most widely publicized component of DirectX, it is common to see the names "DirectX" and "Direct3D" used interchangeably.

▪ Types OF DirectX:

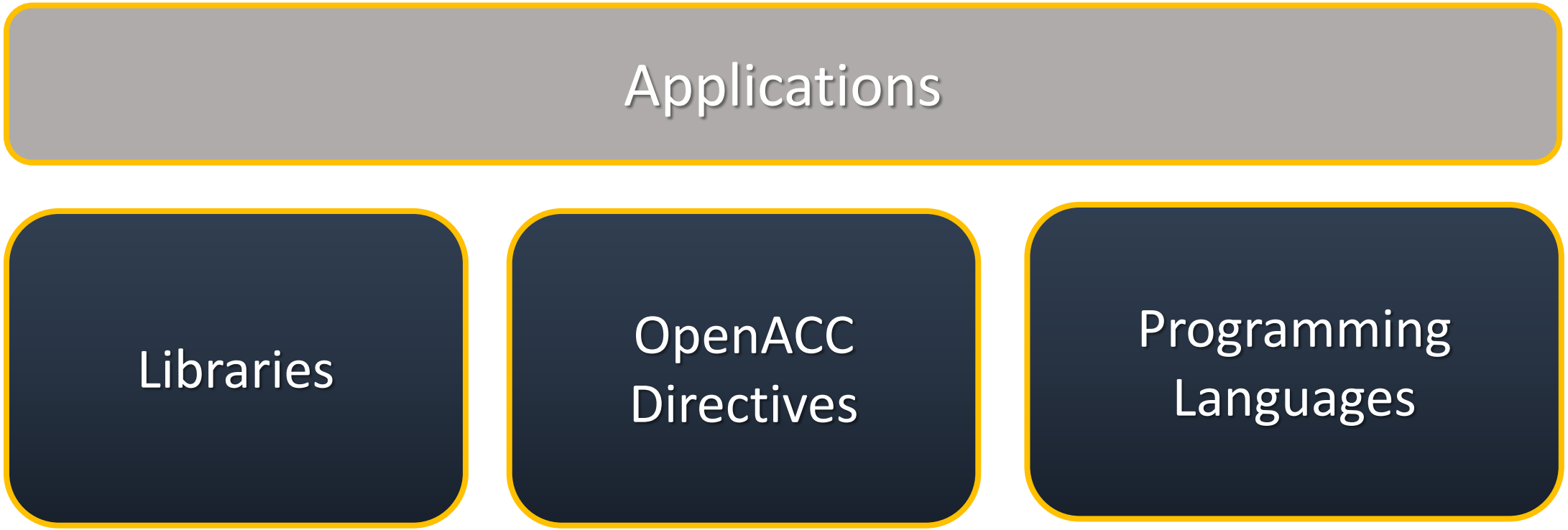
- DirectX 9(2002)- For Windows 98 and XP
- DirectX 10 (2003)-For Windows Vista
- DirectX 11(2009)-For Windows 7
- DirectX 12(2015)-For Windows 10 and XBOX ONE

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Introduction to GPU

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

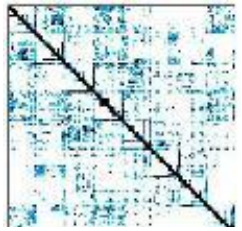
Access The Power of GPU



ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

GPU Accelerated Libraries
"Drop-in" Acceleration for your Applications

NVIDIA cuBLAS



NVIDIA cuSPARSE

NVIDIA NPP

NVIDIA cuFFT

MAGMA

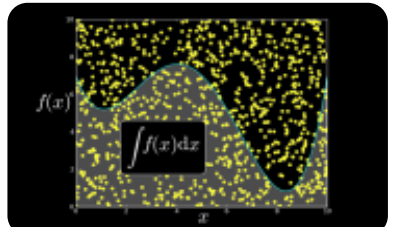
Matrix Algebra on GPU and Multicore

CULA | tools

GPU Accelerated Linear Algebra

GPU VSIPL

Vector Signal Image Processing



NVIDIA cuRAND

ROGUE WAVE SOFTWARE

IMSL Library

CenterSpace NMath

ArrayFire

Building-block Algorithms

Thrust

C++ Templated Parallel Algorithms

GPU Programming Languages

Numerical analytics ▶	MATLAB, Mathematica, LabVIEW
Fortran ▶	OpenACC, CUDA Fortran
C ▶	OpenACC, CUDA C
C++ ▶	CUDA C++, Thrust, Hemi, ArrayFire
Python ▶	Anaconda Accelerate, PyCUDA, Copperhead
.NET ▶	CUDAfy.NET, Alea.cuBase

developer.nvidia.com/language-solutions

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

GPU Architecture

GPU: Massively Parallel Coprocessor

- A GPU is
  - Coprocessor to the CPU or Host
  - Has its own DRAM
  - Runs 1000s of threads in parallel
  - Single Precision: 4.58TFlop/s
  - Double Precision: 1.31TFlop/s

Heterogeneous Parallel Computing

Logic()

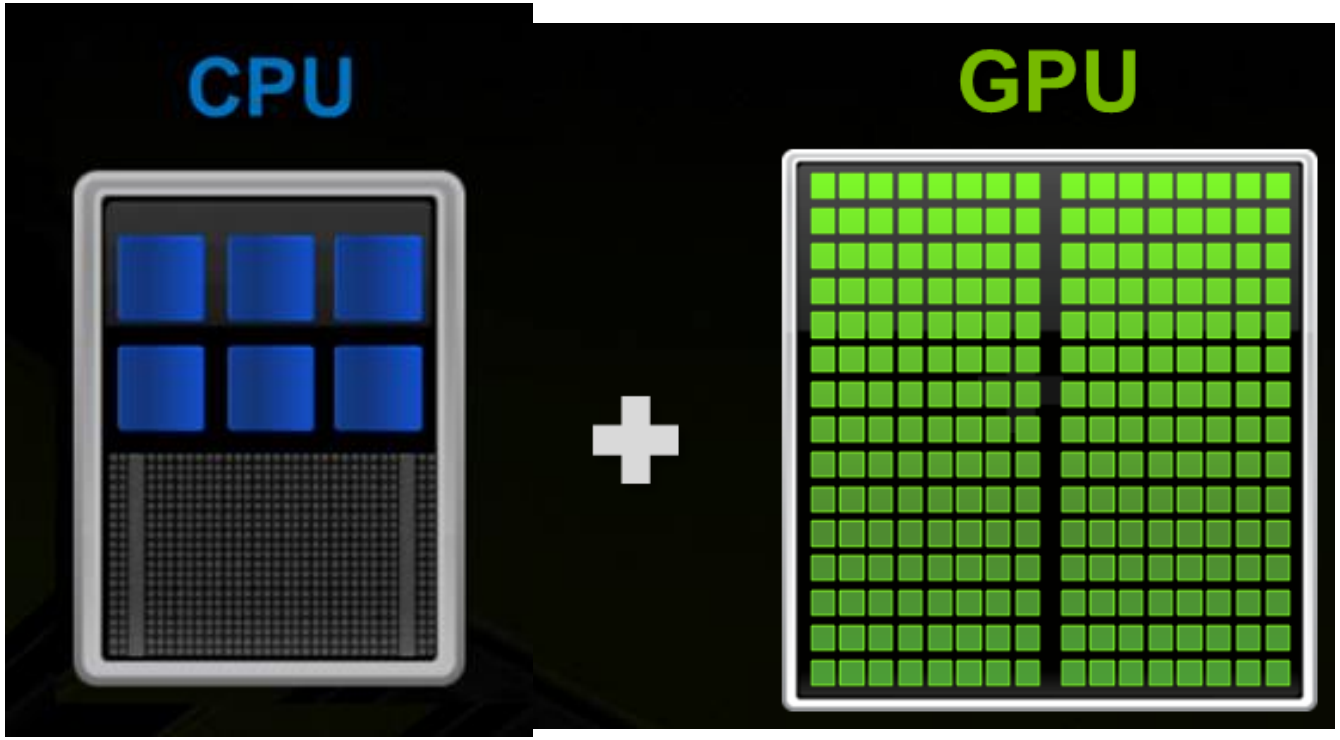
Compute()

Latency-Optimized
Fast Serial Processing

Heterogeneous Parallel Computing

Logic()

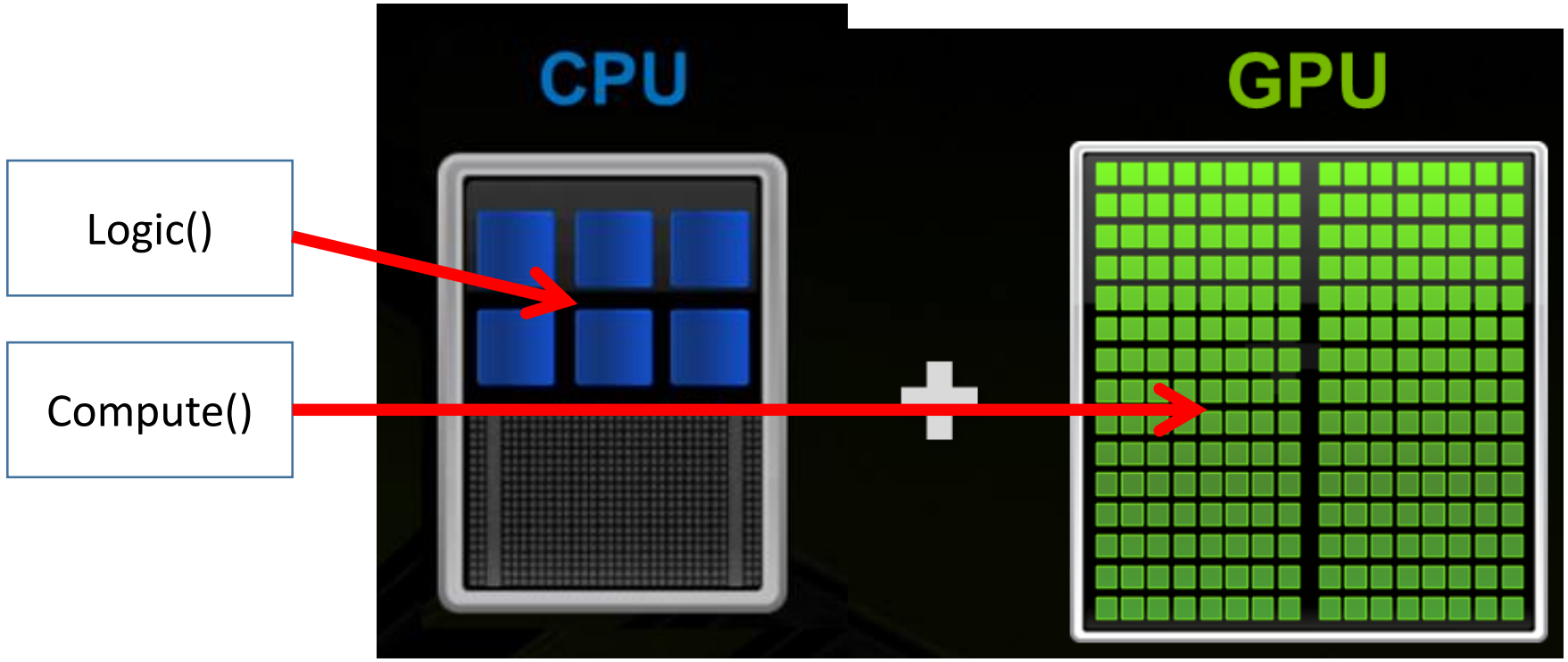
Compute()



Latency-Optimized
Fast Serial Processing

Throughput-Optimized
Fast Parallel Processing

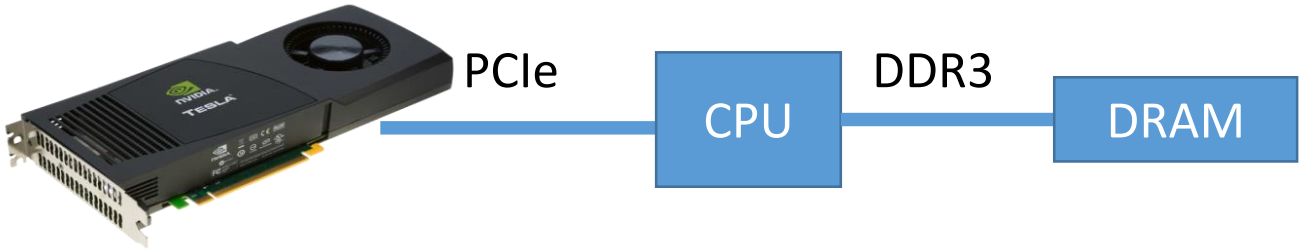
Heterogeneous Parallel Computing



Latency-Optimized
Fast Serial Processing

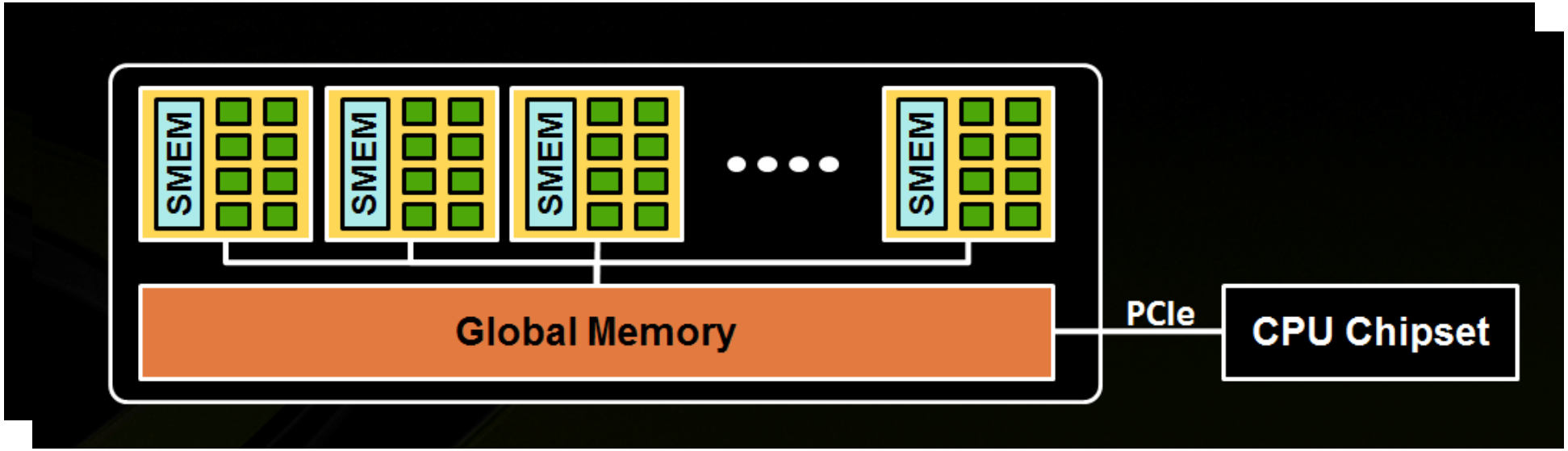
Throughput-Optimized
Fast Parallel Processing

GPU in Computer System



- **Connected to CPU chipset by PCIe**
- **16GB/s One Way, 32GB/s in both way**

GPU High Level View



- **Streaming Multiprocessor (SM)**
  - **A set of CUDA cores**
- **Global memory**

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

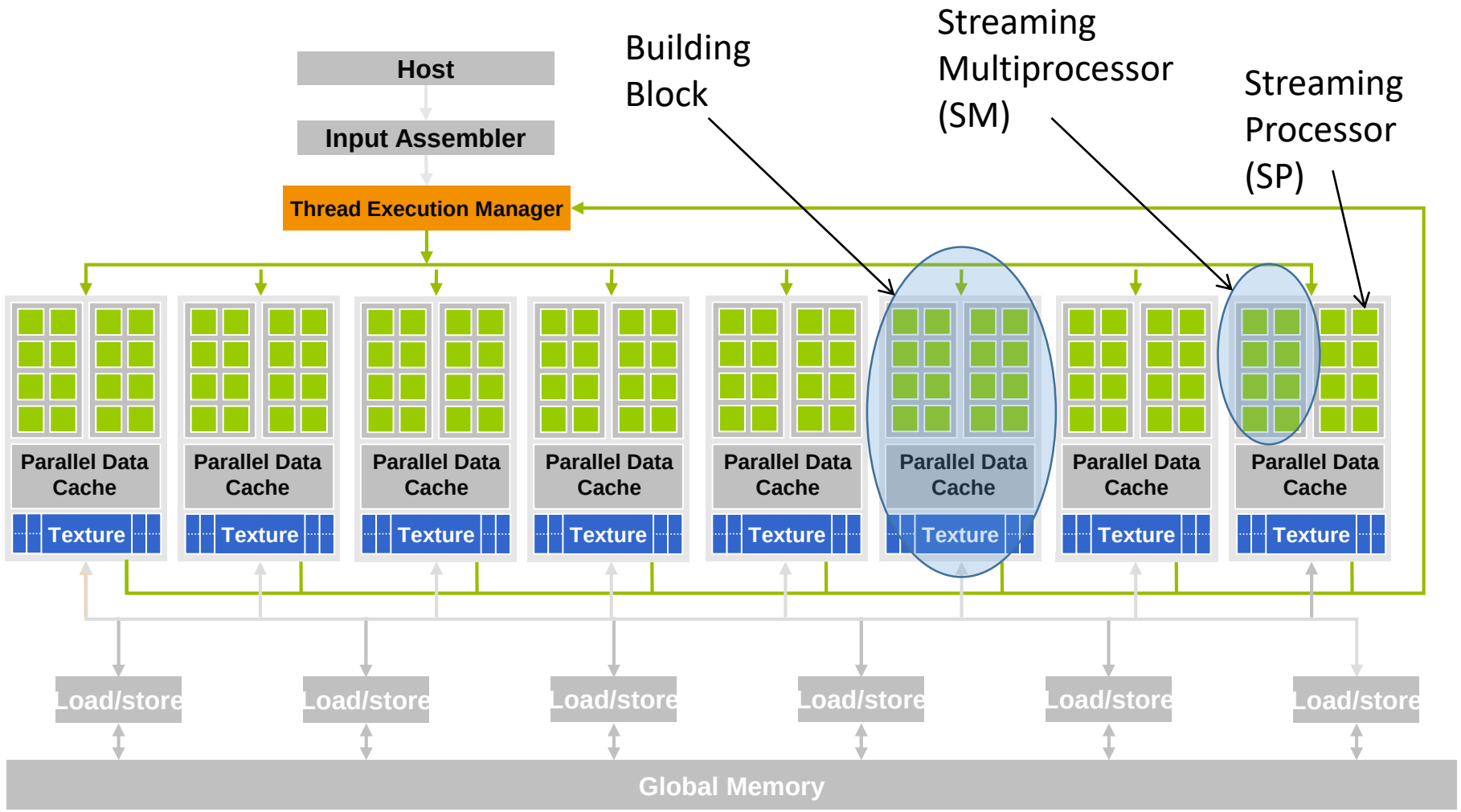
GPUs: The Big Development

- Up to **448 cores** on a single chip
- Simplified logic (no out-of-order execution, no branch prediction) means much more of the chip is devoted to computation
- Arranged as multiple units with each unit being effectively a vector unit, all cores doing the same thing at the same time
- Very high bandwidth (up to 140GB/s) to graphics memory (up to 4GB)
- Not general purpose – for **parallel applications like graphics and Monte Carlo simulations**
- Can also build big clusters out of GPUs

GPUs: The Big Development

- Four major vendors:
  - **NVIDIA**
  - **AMD**: bought ATI several years ago
  - **IBM**: co-developed Cell processor with Sony and Toshiba for Sony Playstation, but now dropped it for high-performance computing
  - **Intel**: was developing “Larrabee” chip as GPU, but now aimed for high-performance computing

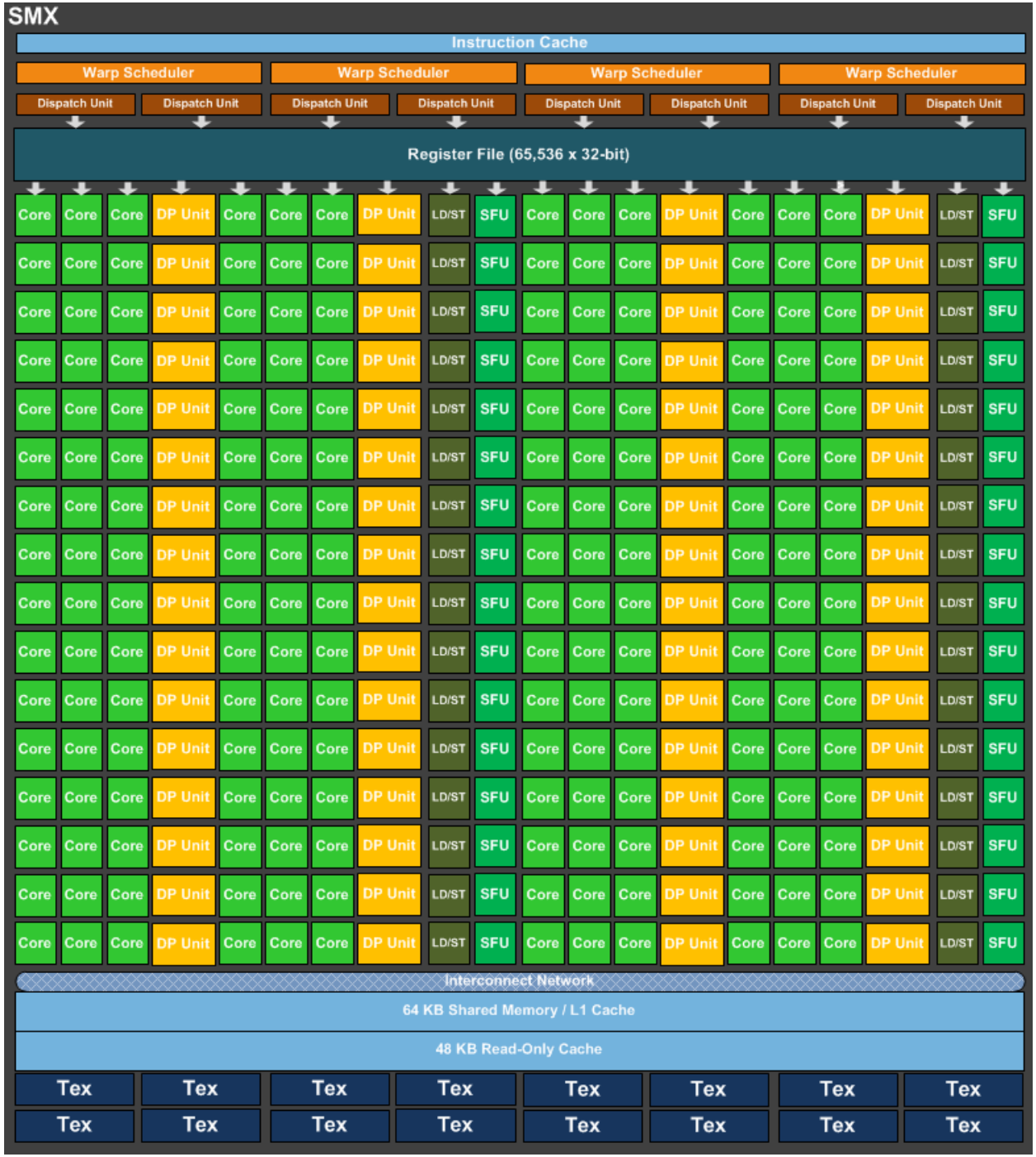
Architecture of a CUDA-capable GPU



30 SM's each with 8 SP's on the C1060

GK110 SM

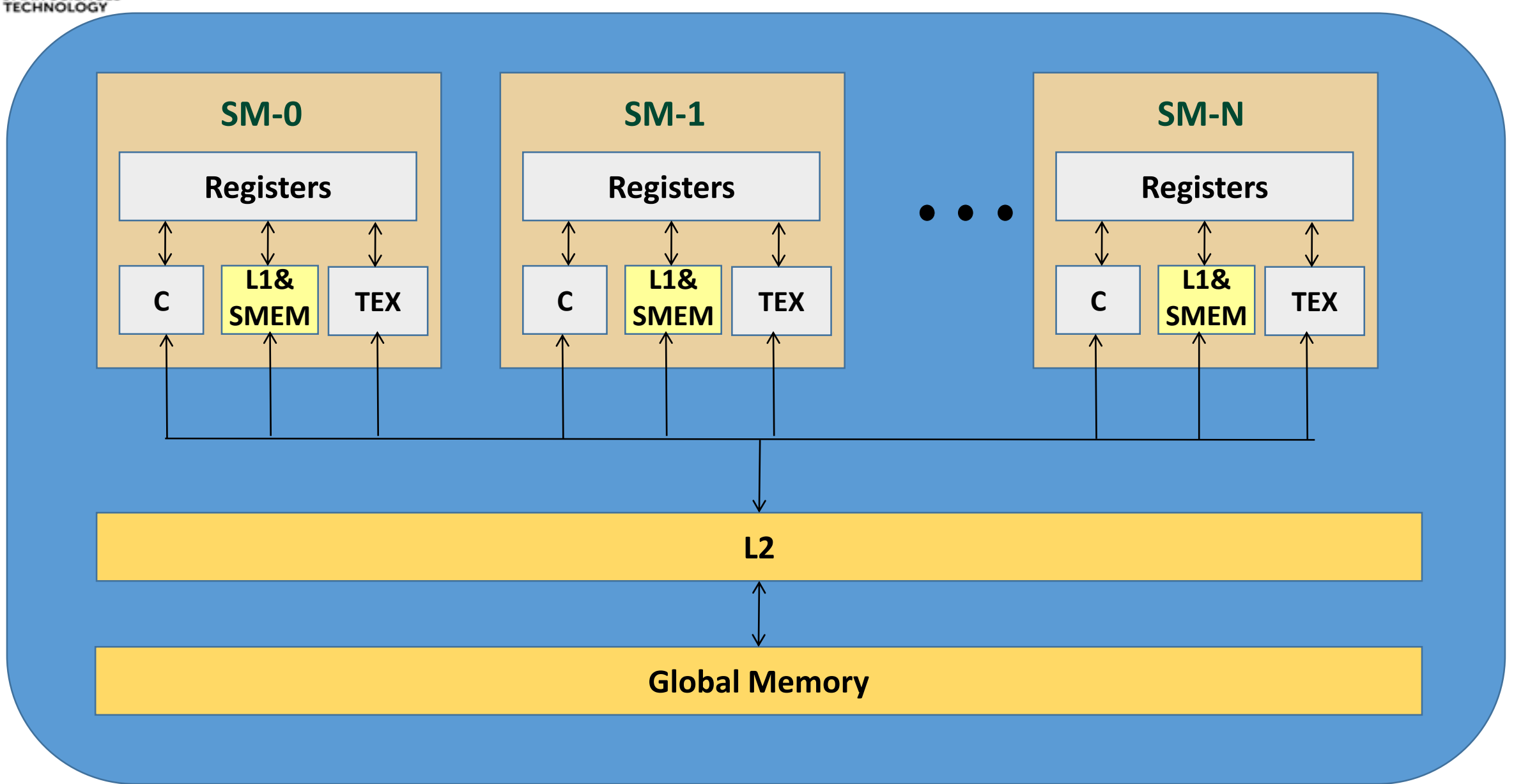
- Control unit
  - 4 Warp Scheduler
  - 8 instruction dispatcher
- Execution unit
  - 192 single-precision CUDA Cores
  - 64 double-precision CUDA Cores
  - 32 SFU, 32 LD/ST
- Memory
  - Registers: 64K 32-bit
  - Cache
    - L1+shared memory (64 KB)
    - Texture
    - Constant



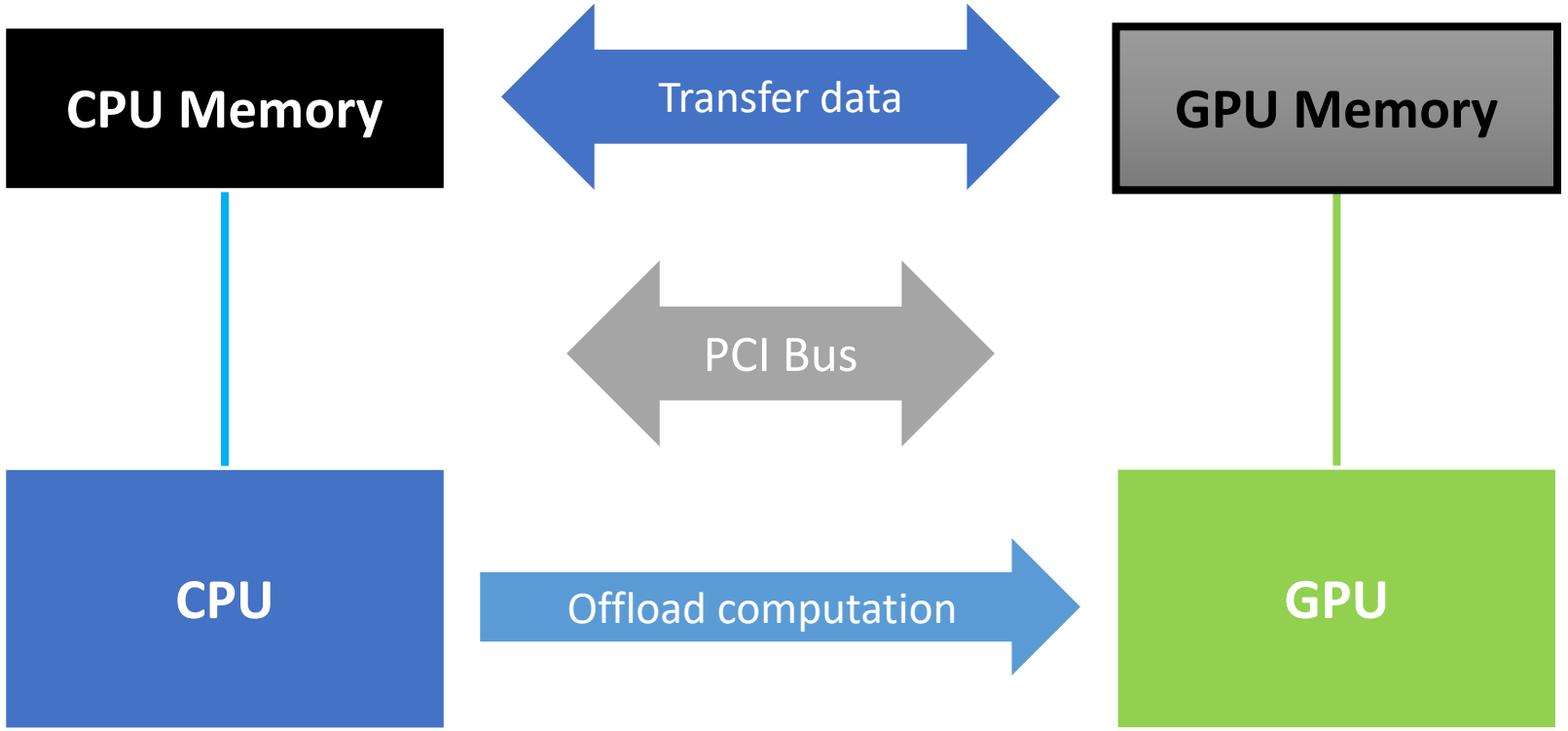
Kepler/Fermi Memory Hierarchy

- 3 levels, very similar to CPU
- Register
  - Spills to local memory
- Caches
  - Shared memory
  - L1 cache
  - L2 cache
  - Constant cache
  - Texture cache
- Global memory

Kepler/Fermi Memory Hierarchy



Basic Concepts



GPU computing is all about 2 things:

- Transfer data between CPU-GPU
- Do parallel computing on GPU

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

GPU Programming Basics

How To Get Start

- CUDA C/C++: download CUDA drivers & compilers & samples (All In One Package) free from:

  http://developer.nvidia.com/cuda/cuda-downloads

- CUDA Fortran: PGI

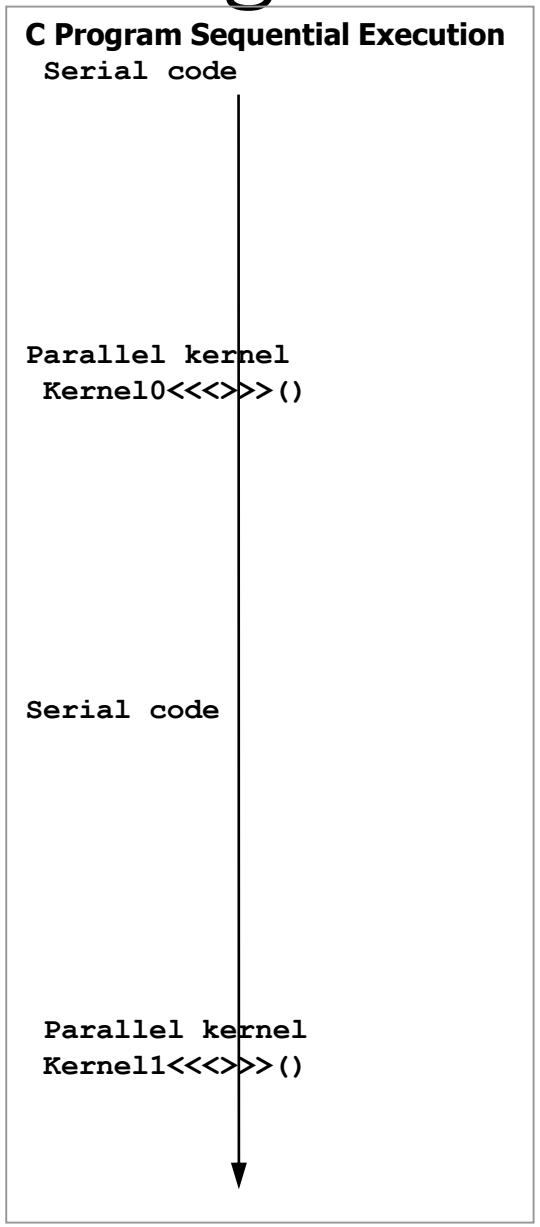
- OpenACC: PGI, CAPS, Cray

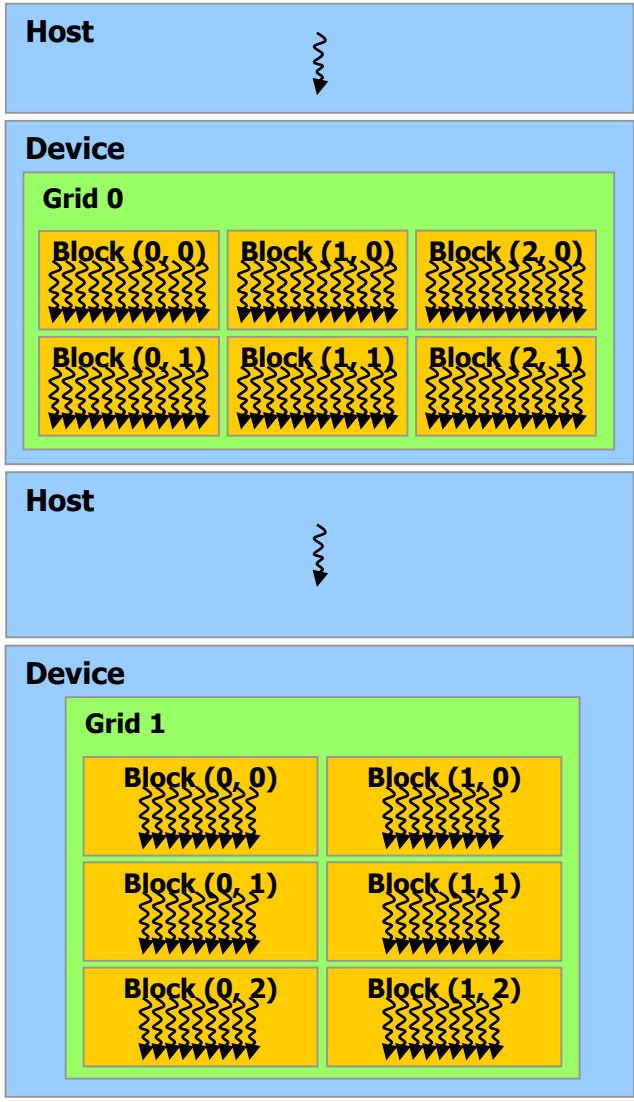
CUDA Programming Basics

- Hello World
  - Basic syntax, compile & run
- GPU memory management
  - Malloc/free
  - memcpy
- Writing parallel kernels
  - Threads & block
  - Memory hierarchy

Heterogeneous Computing

- Executes on both CPU & GPU
  - Similar to OpenMP's fork-join pattern
- Accelerated kernels
  - CUDA: simple extensions to C/C++





Hello World on CPU

hello_world.c:

```
#include <stdio.h>

void hello_world_kernel()
{
   printf(“Hello World\n”);
}

int main()
{
   hello_world_kernel();
}
```

Compile & Run:

```
gcc hello_world.c
./a.out
```

Hello World on GPU

hello_world.cu:

```
#include <stdio.h>

__global__ void hello_world_kernel()
{
   printf(“Hello World\n”);
}

int main()
{
   hello_world_kernel<<<1,1>>>();
}
```

Compile & Run:
nvcc hello_world.cu
./a.out

Hello World on GPU

hello_world.cu:

```
#include <stdio.h>

__global__ void hello_world_kernel()
{
   printf("Hello World\n");
}

int main()
{
   hello_world_kernel<<<1,1>>>();
}
```

Compile & Run:

```
nvcc hello_world.cu
./a.out
```

- CUDA kernel within .cu files
- .cu files compiled by nvcc
- CUDA kernels preceded by "__global__"
- CUDA kernels launched with "<<<…,…>>>"

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
- Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions

CUDA C/C++ Memory Allocation / Release

- Host (CPU) manages device (GPU) memory:
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)

```
int nbytes = 1024*sizeof(int);
int * d_a = 0;
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);
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- Non-blocking memcopies are provided

Code Walkthrough 1

- Allocate CPU memory for *n* integers
- Allocate GPU memory for *n* integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Compile & Run

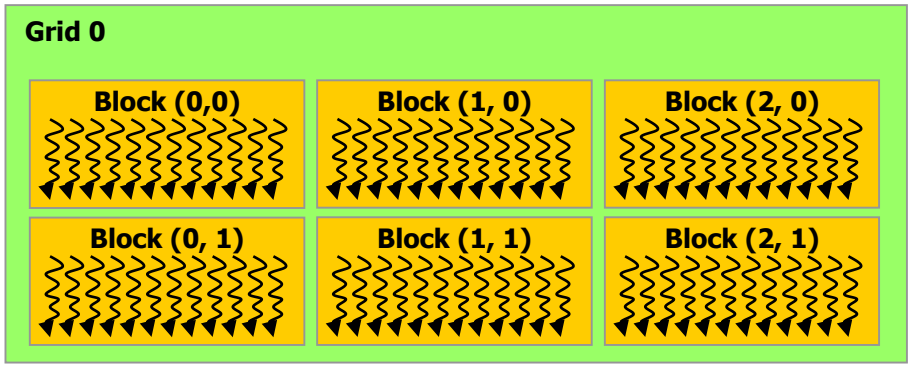
- nvcc main.cu

- ./a.out

  0000000000000000

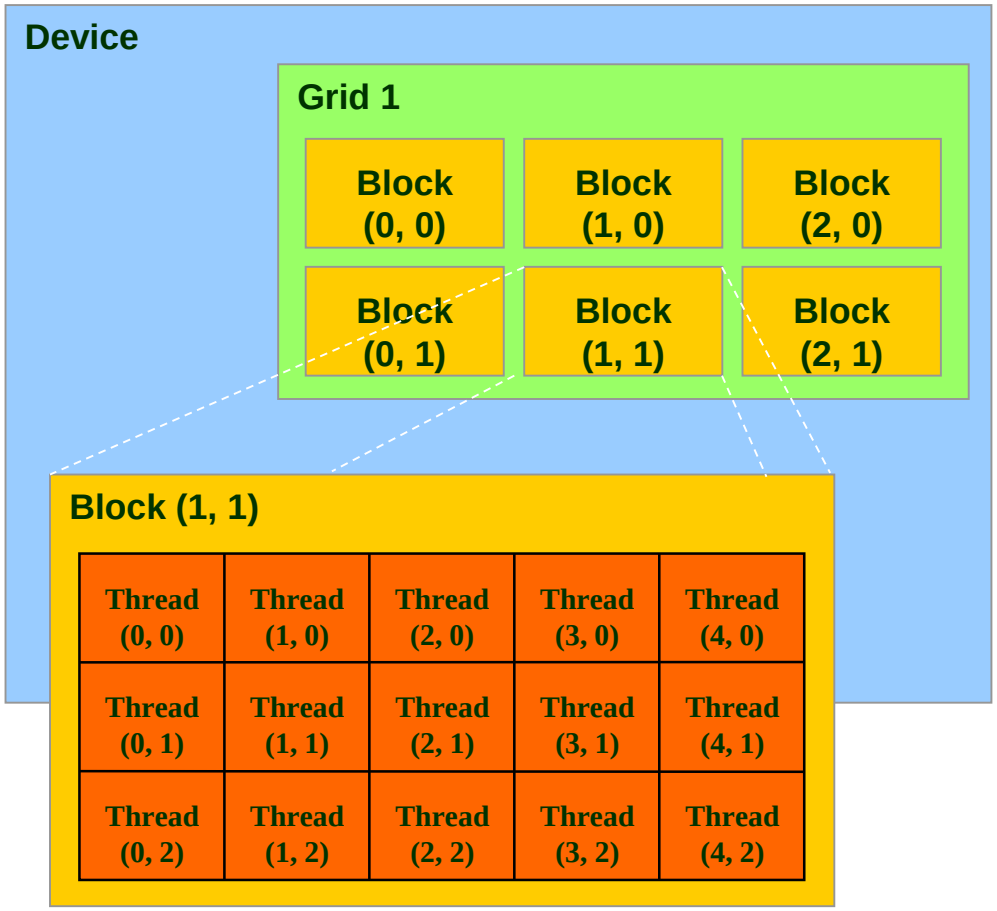
Thread Hierarchy

- 2-level hierarchy: blocks and grid
  - Block = a group of up to 1024 threads
  - Grid = all blocks for a given kernel launch
  - E.g. total 72 threads
    - blockDim=12, gridDim=6
- A block can:
  - Synchronize their execution
  - Communicate via shared memory
- Size of grid and blocks are specified during kernel launch
  - dim3 grid(6,1,1), block(12,1,1);
    kernel<<<grid, block>>>(…);



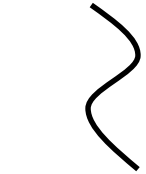
IDs and Dimensions

- Threads:
  - 3D IDs, unique within a block
- Blocks:
  - 3D IDs, unique within a grid
- Built-in variables:
  - `threadIdx: idx within a block`
  - `blockIdx: idx within the grid`
  - `blockDim: block dimension`
  - `gridDim: grid dimension`

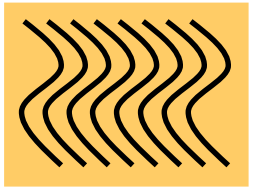


GPU and Programming Model

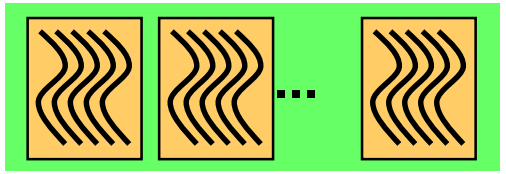
Software



Thread



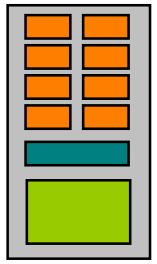
Thread
Block



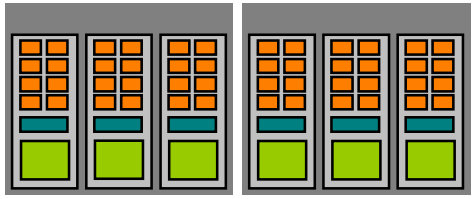
Grid

GPU

CUDA Core



Multiprocessor



Device

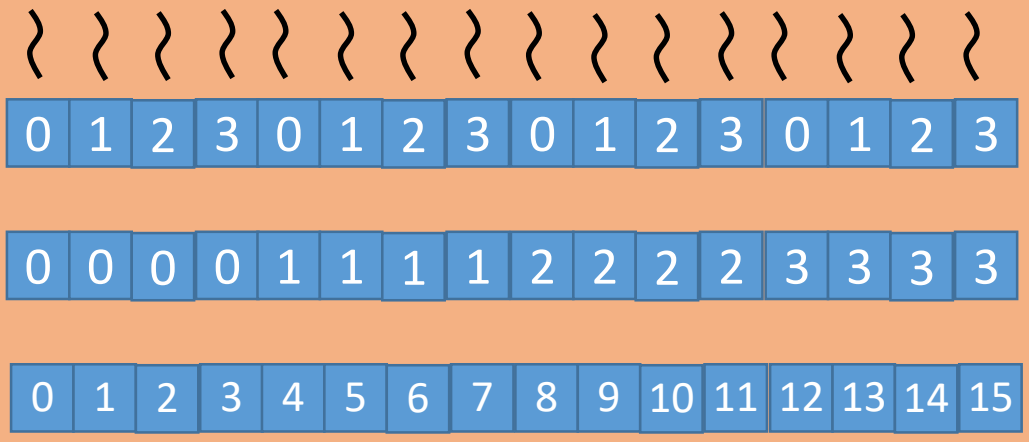
Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

A kernel is launched as a grid of thread blocks

Which thread do I belong to?

blockDim.x = 4, gridDim.x = 4



Code Walkthrough 2: Simple Kernel

- Allocate memory on GPU
- Copy the data from CPU to GPU
- Write a kernel to perform a vector addition
- Copy the result to CPU
- Free the memory

Vector Addition using C

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

float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
vec_add(x,y,n);
free(x);
free(y);
```

Vector Addition using CUDA C

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

float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostToDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
cudaFree(d_x);
cudaFree(d_y);
```

Vector Addition using CUDA C

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

float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostToDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
cudaFree(d_x);
cudaFree(d_y);
```

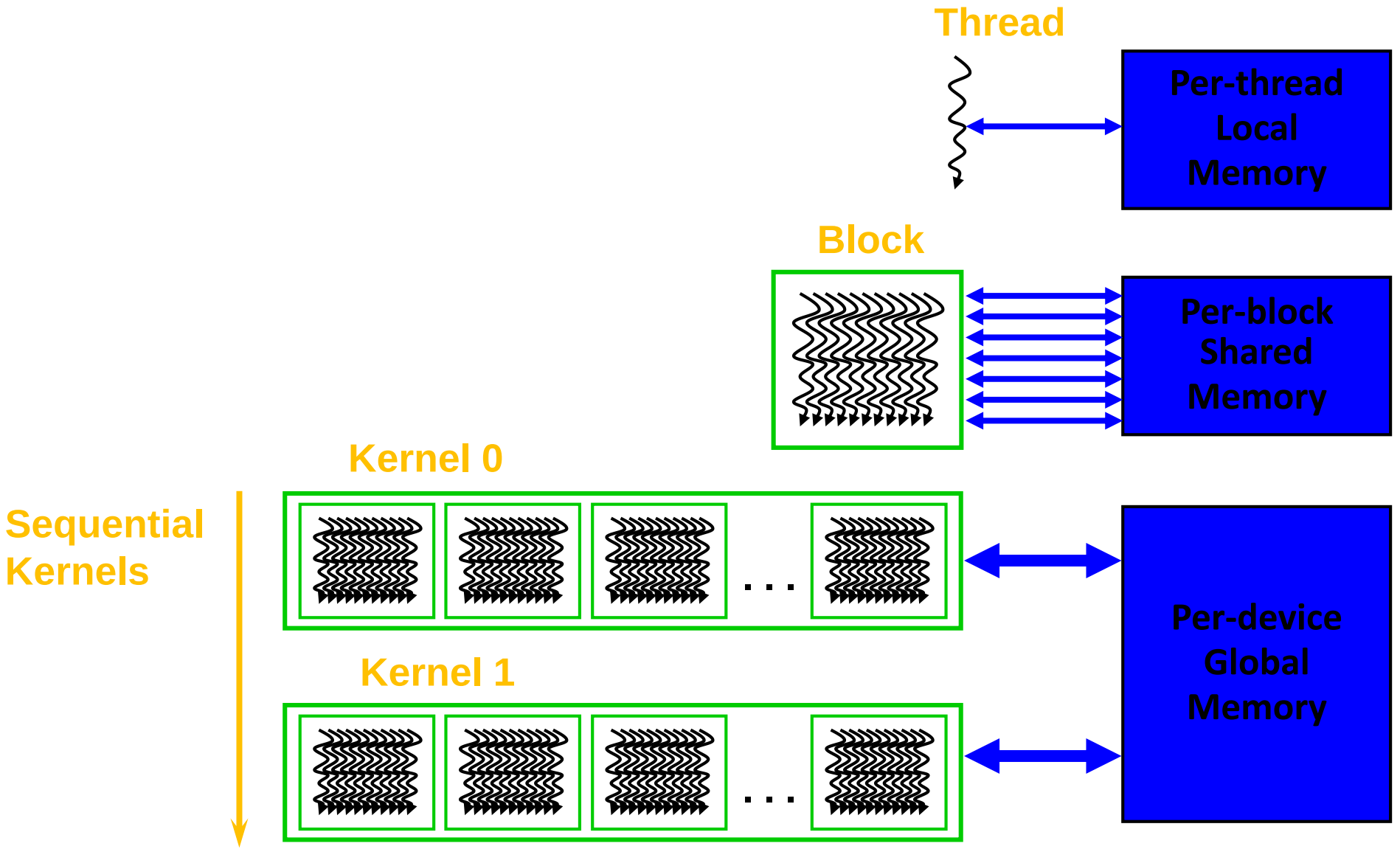
Keyword for CUDA kernel

Vector Addition using CUDA C

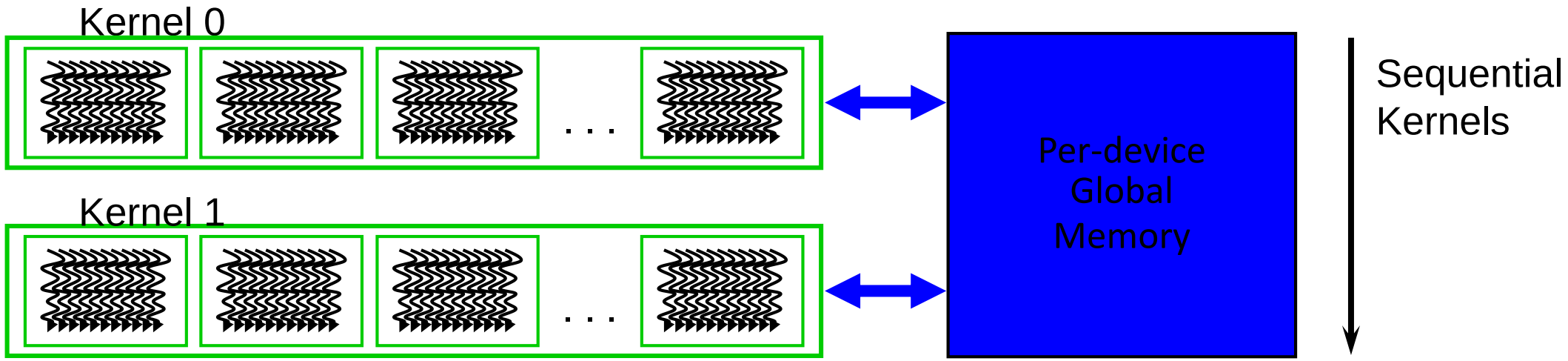
```
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}

float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostToDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
cudaFree(d_x);
cudaFree(d_y);
```

Thread index computation to replace loop

GPU Memory Model Review



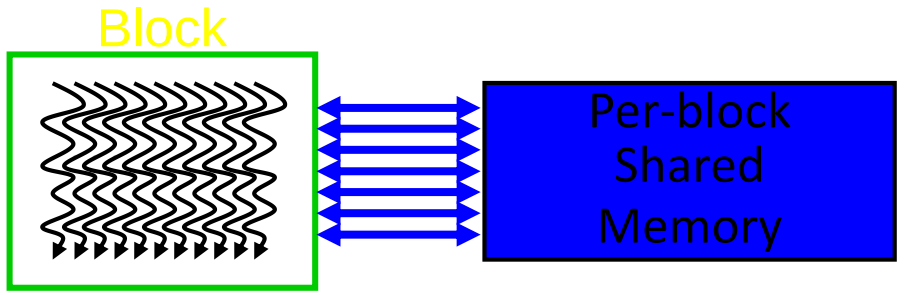
Global Memory



- **Data lifetime = from allocation to deallocation**
- **Accessible by all threads as well as host (CPU)**

Shared Memory

- C/C++: __shared__ int a[SIZE];
- Allocated per threadblock
- Data lifetime = block lifetime
- Accessible by any thread in the threadblock
  - Not accessible to other threadblocks

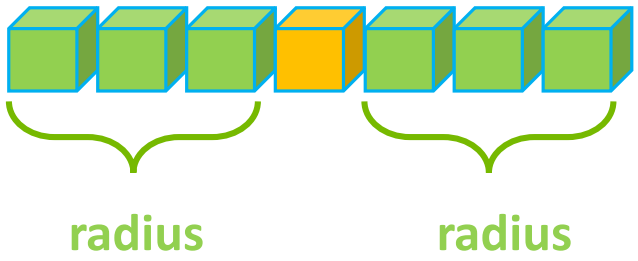


Registers

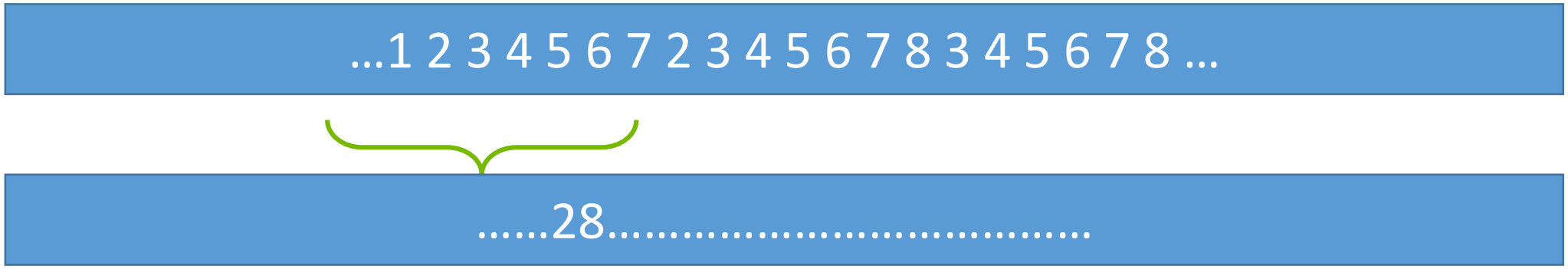
- **Automatic variables (scalar/array) inside kernels**
- Data lifetime = thread lifetime
- Accessible only by the thread declares it

Example of Using Shared Memory

- Applying a 1D stencil to a 1D array of elements:
  - Each output element is the sum of all elements within a radius
- For example, for radius = 3, each output element is the sum of 7 input elements:



Example of Using Shared Memory



Kernel Code Using Global Memory

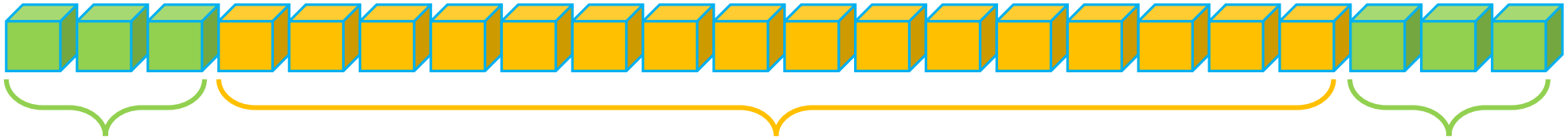
One element per thread

```
__global__ void stencil(int* in, int* out) {
  int globIdx = blockIdx.x * blockDim.x + threadIdx.x;
  int value = 0;
  for (offset = - RADIUS; offset <= RADIUS; offset++)
    value += in[globIdx + offset];
  out[globIdx] = value;
}
```

A lot of redundant read in neighboring threads: not an optimized way

Implementation with Shared Memory

- One element per thread
  - Read (BLOCK_SIZE + 2 * RADIUS) elements from global memory to shared memory
  - Compute BLOCK_SIZE output elements in shared memory
  - Write BLOCK_SIZE output elements to global memory



"halo"
= RADIUS elements on the left

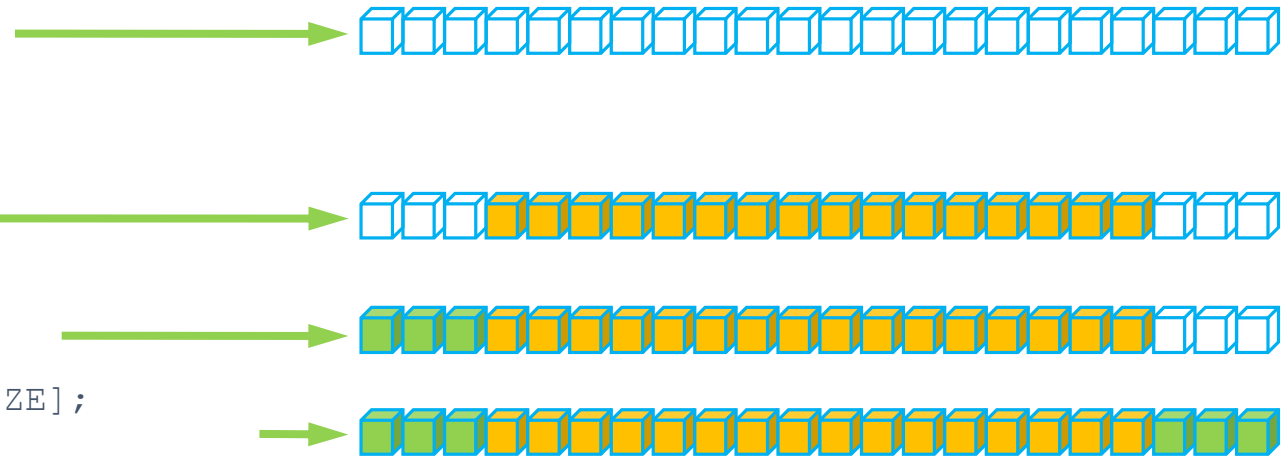
The BLOCK_SIZE input elements corresponding to the output elements

"halo"
= RADIUS elements on the right

Kernel Code

```
__global__ void stencil(int* in, int* out) {
  __shared__ int shared[BLOCK_SIZE + 2 * RADIUS];
  int globIdx = blockIdx.x * blockDim.x + threadIdx.x;
  int locIdx  = threadIdx.x + RADIUS;
  shared[locIdx] = in[globIdx];
  if (threadIdx.x < RADIUS) {
    shared[locIdx - RADIUS] = in[globIdx - RADIUS];
    shared[locIdx + BLOCK_DIMX] = in[globIdx + BLOCK_SIZE];
  }
  __syncthreads();
  int value = 0;
  for (offset = - RADIUS; offset <= RADIUS; offset++)
    value += shared[locIdx + offset];
  out[globIdx] = value;
}
```

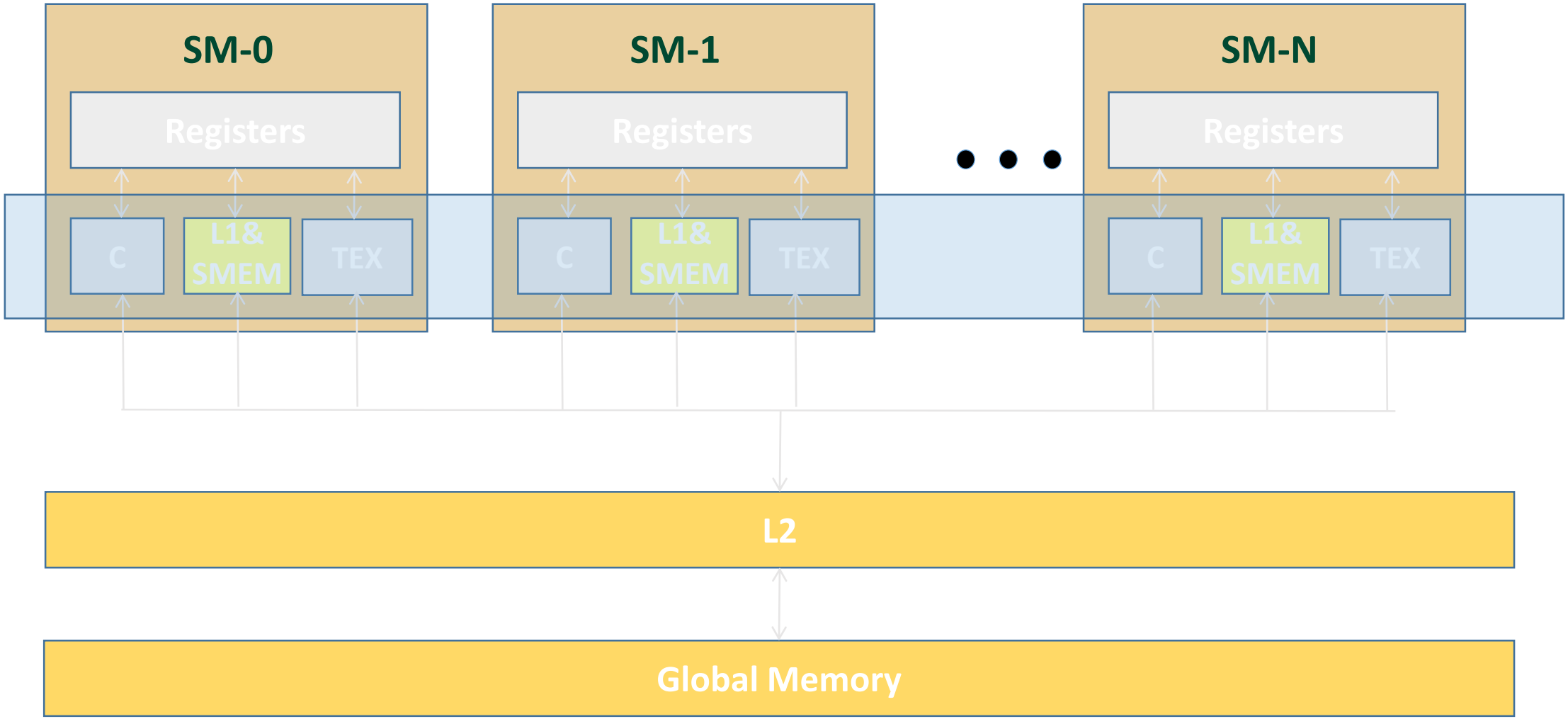
RADIUS = 3
BLOCK_SIZE = 16



Thread Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a thread block
  - Since threads are scheduled at run-time
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Should be used in conditional code only if the conditional is uniform across the entire thread block
  - Otherwise may lead to deadlock

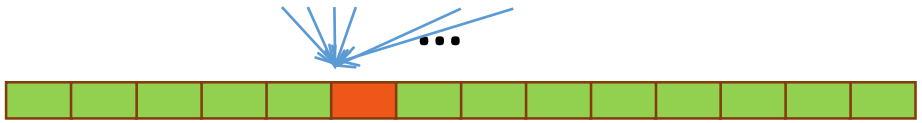
Kepler/Fermi Memory Hierarchy



Constant Cache

- Global variables marked by __constant__ are constant and can’t be changed in device.
- Will be cached by Constant Cache
- Located in global memory
- Good for threads access the same address

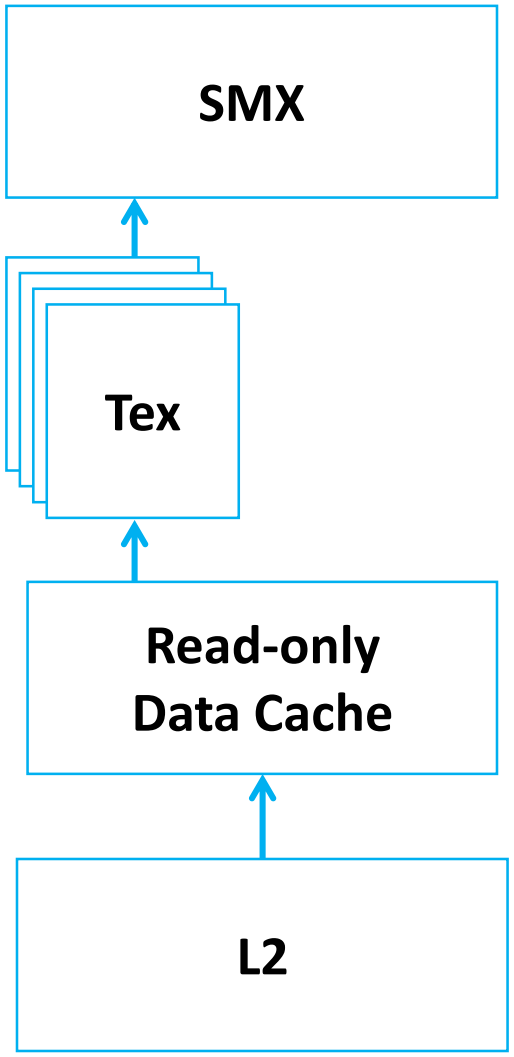
```
__constant__ int a=10;
__global__ void kernel()
{
        a++; //error
}
```



Memory addresses

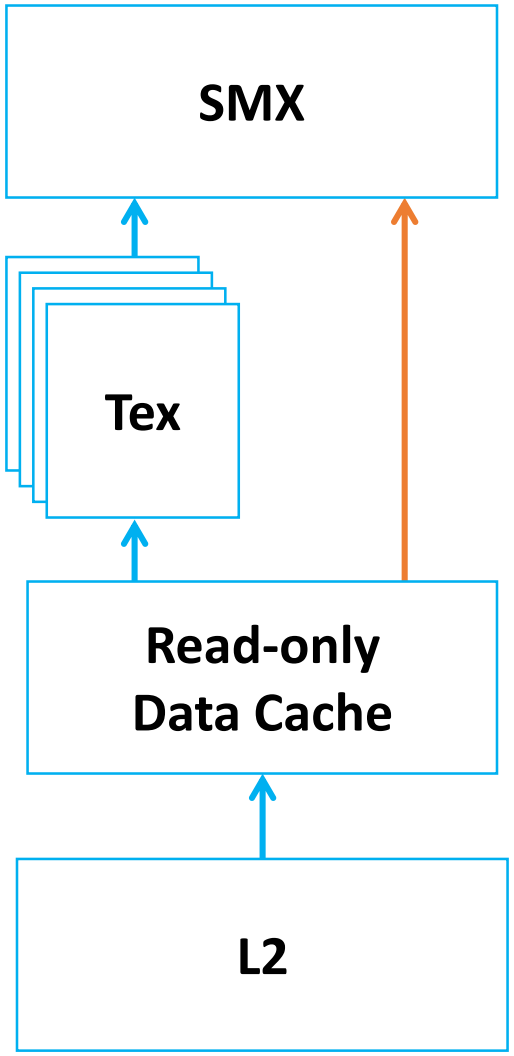
Texture Cache

- Save Data as Texture :
  - Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
  - Read-only data cache holds fetched samples
  - Backed up by the L2 cache
- Why use it?
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses



Texture Cache Unlocked In GK110

- Added a new path for compute
  - Avoids the texture unit
  - Allows a global address to be fetched and cached
  - Eliminates texture setup
- Managed automatically by compiler
  - “const __restrict” indicates eligibility



const __restrict

- Annotate eligible kernel parameters with `const __restrict`
- Compiler will automatically map loads to use read-only data cache path

```
__global__ void saxpy(float x, float y,
                      const float * __restrict input,
                      float * output)
{
    size_t offset = threadIdx.x +
                    (blockIdx.x * blockDim.x);

    // Compiler will automatically use texture
    // for "input"
    output[offset] = (input[offset] * x) + y;
}
```

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

pyCUDA Implementation

https://mathema.tician.de/software/pycuda/

Performance

- Speed(C) >> Speed(Python)
- For most code, it doesn't matter
- Does matter for inner loops
- Solution: PyCUDA
- Hybrid Python + CUDA code

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Why Python

- Mature language
- Multi-paradigm language
- Has comprehensive built-in functionalities
- Dynamically typed
- Emphasizes readability

Why Python

Value programmer time over computer time

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

PyCUDA saves you time

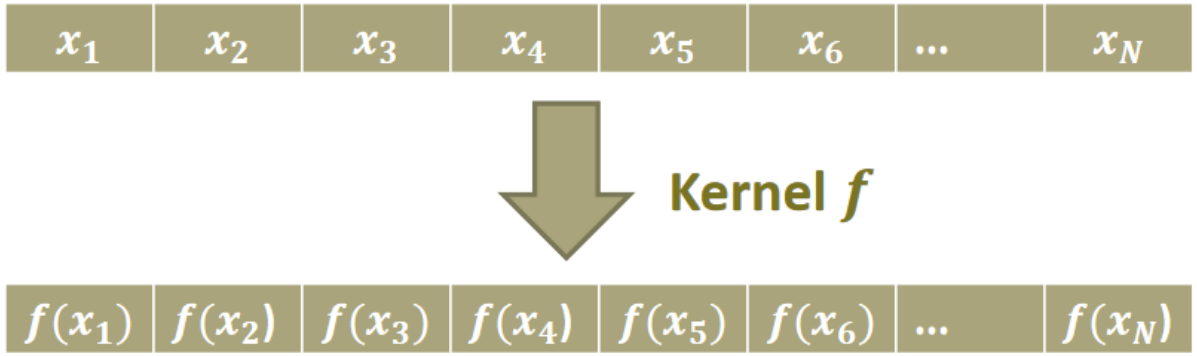
Think about the tools you use
Use the right tool for the job

Just as powerful

- Arrays and textures
- Pagelocked host memory
- Memory transfers (asynchronous, structured)
- Streams and events
- Device queries
- Supports every OS that CUDA supports

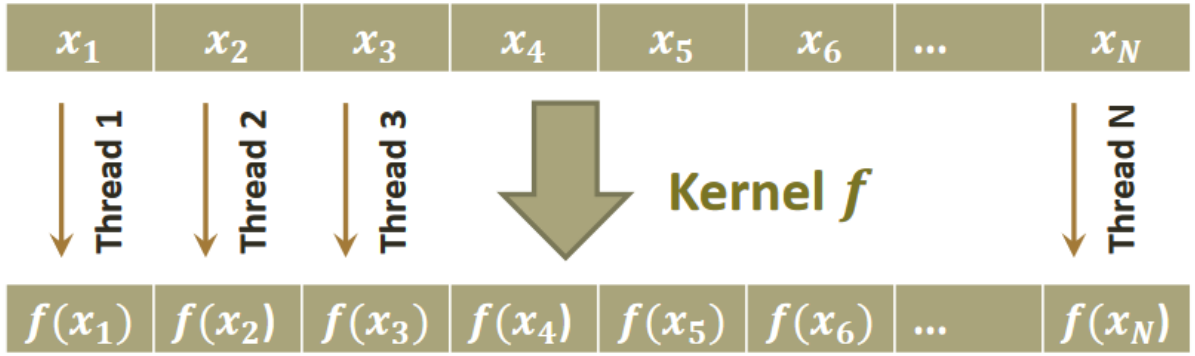
Data parallelism

- When can we make use of the GPU's power?
- Simplest case:



Data parallelism

- When can we make use of the GPU's power?
- Simplest case:



- Each element gets handles by one thread
- Compare to Map()

Terminologies

- A kernel is a function run by a thread
- A thread is the abstraction of a function call on some data
- Threads are launched in groups called Blocks
- Blocks are in turn organizes in a grid

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

PyCUDA

- Automatically manage resources
- Similar concept to strong and weak reference
- Scare resources (memory) can be explicitly freed
- Provide abstractions
- Integrate tightly with numpy

Elementwise operations

We’re used to numpy operating on elements like this

```
import numpy

size = 1e7
X = numpy.linspace(1,size,size).astype(numpy.float32)
Y = numpy.sin(X)
```

Elementwise operations

```
import pycuda.gpuarray as gpuarray
import pycuda.cumath as cumath
import pycuda.autoinit
import numpy

size = 1e7
X = numpy.linspace(1,size,size).astype(numpy.float32)
X_gpu = gpuarray.to_gpu(X) # 1. transfer -> gpu
Y_gpu = cumath.sin(X_gpu)  # 2. execute kernel
Y = Y_gpu.get()                    # 3. retrieve result
```

Elementwise operations

- Numpy: 438 ms
- PyCUDA: 162 ms

https://wiki.tiker.net/PyCuda

https://wiki.tiker.net/PyCuda/Examples

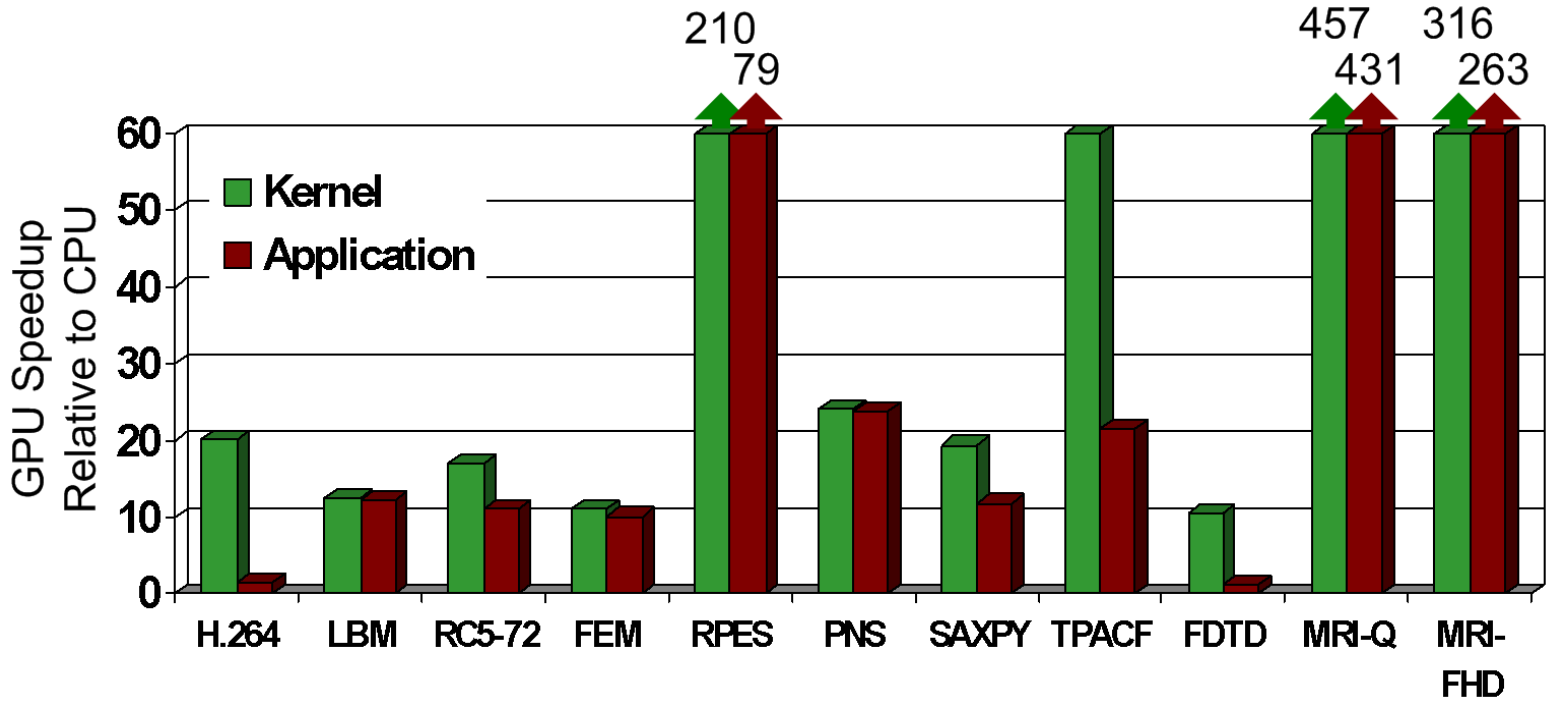
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Case Studies

Previous Projects, UIUC ECE 498AL

Application	Description	Source	Kernel	% time
H.264	SPEC ‘06 version, change in guess vector	34,811	194	35%
LBM	SPEC ‘06 version, change to single precision and print fewer reports	1,481	285	>99%
RC5-72	Distributed.net RC5-72 challenge client code	1,979	218	>99%
FEM	Finite element modeling, simulation of 3D graded materials	1,874	146	99%
RPES	Rye Polynomial Equation Solver, quantum chem, 2-electron repulsion	1,104	281	99%
PNS	Petri Net simulation of a distributed system	322	160	>99%
SAXPY	Single-precision implementation of saxpy, used in Linpack’s Gaussian elim. routine	952	31	>99%
TPACF	Two Point Angular Correlation Function	536	98	96%
FDTD	Finite-Difference Time Domain analysis of 2D electromagnetic wave propagation	1,365	93	16%
MRI-Q	Computing a matrix Q, a scanner’s configuration in MRI reconstruction	490	33	>99%

Speedup of Applications



- GeForce 8800 GTX vs. 2.2 GHz Opteron 248

Speedup of Applications

- 10x speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads

- 25x to 400x speedup if the function's data requirements and control flow suit the GPU and the application is optimized

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Final Thoughts

- Parallel hardware is here to stay
- GPUs are massively parallel manycore processors
  - Easily available and fully programmable
- Parallelism and scalability are crucial for success
- This presents many important research challenges
  - Not to mention the educational challenges

References

- Manuals
  - Programming Guide
  - Best Practice Guide
- Books
  - CUDA By Examples, Tsinghua University Press
- Training videos
  - GTC talks online: optimization, advanced optimization + hundreds of other GPU computing talks
    http://www.gputechconf.com/gtcnew/on-demand-gtc.php
  - NVIDIA GPU Computing webinars
    http://developer.nvidia.com/gpu-computing-webinars
- Forum
  - http://cudazone.nvidia.cn/forum/forum.php

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

GPU Appliance or Box

NVIDIA Appliance

FEATURE	DETAILS
GPUs	8x NVIDIA High-End Quadro / Tesla GPUs
GPU Memory	24 GB per GPU
NVIDIA CUDA® Cores	30,720 (combined across all GPUs)
CPU	4 GB 64 bit LPDDR4 25.6 GB/s
CPU Cores	40 with Hyper Threading
System Memory	256 GB
Storage	2TB SSD
Network	1 x 1 GigE Network Interface 1x Mellanox ConnectX-3 Infiniband
Installed Software	Linux CentOS 6.6 VCA SW Manager OptiX 3.8 or newer

https://www.nvidia.com/en-us/design-visualization/visual-computing-appliance/

SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY