**CUDA**

**GPU vs Multicore computers**

- **Multicore machines**
  - Emphasize multiple full-blown processor cores, implementing the complete instruction set of the CPU
  - The cores are out-of-order implying that they could be doing different tasks
  - They may additionally support hyperthreading with two hardware threads
  - Designed to maximize the execution speed of sequential programs

- **GPUs (Nvidia)**
  - Typically have hundreds of cores
  - Cores are heavily multithreaded, in-order, and single-instruction issue processors
  - Each core shares control and instruction cache with seven other cores
  - Achieve about 10X performance compared to single core machines

- **Design philosophy of general-purpose multicore CPUs**
  - Design of CPU optimized for sequential code performance
  - Emphasis of parallelizing different parts of instruction (pipelining) to improve performance of sequential processes
  - Large cache memories to reduce instruction and data latency
  - Core i7 (Announced for Q1 2012)
    - Two load/store operations per cycle for each memory channel
    - Allows for 32KB data and 32KB instruction L1 cache at 3 clocks and 256KB L2 cache at 8 clocks per core
    - Up to 8 physical cores and 16 logical cores through hyperthreading

- **Design philosophy of many-core GPUs**
  - Shaped by video game industry; large number of floating point calculations per video frame
  - Maximize chip area and power budget for floating point computation
  - Optimize for the execution throughput of large number of threads
  - Large bandwidth to move data (10X compared to CPU)
  - Nvidia GeForce 8800 GTX moves data at 85GB/s in and out of its main DRAM; GT200 chips support 150GB/s; CPU-based systems support about 50GB/s
  - Simpler memory models
  - No legacy applications, OS, or I/O devices to restrict increase in memory bandwidth
  - Small cache memories to control bandwidth requirements for applications
  - Designed as numeric computing engines
  - Applications designed to execute sequential part on CPU and numerically intensive part on GPU

- **CUDA**
  - Compute Unified Device Architecture
  - Designed to support joint CPU/GPU execution of applications
  - Set of developing tools to create applications to execute on GPU
  - CUDA compiler uses a variation of C with some C++ extensions
  - Avoids the performance overhead of graphics layer APIs by compiling software directly to the hardware
Includes a unified shader pipeline allowing each ALU on the GPU to be used for general-purpose computations
* ALUs built to comply with IEEE single precision floating point standard
* Execution units on GPU allowed arbitrary read/write access to memory as well as software-managed cache known as *shared memory*

**Modern GPU architecture**

- Array of highly threaded streaming multiprocessors SMs
  - SMs share control logic and instruction cache
- A set of SMs combined into a building block
  - Above figure (GeForce 8800) contains 16 highly threaded SMs per block
  - 128 FPU	* Each has a multiply-add unit and additional multiply unit
  - Special function units perform floating point functions such as square root
  - Each SP can run multiple threads (768 threads/SM)
  - 367 GFLOPS
  - 768 MB DRAM
  - 86.4 GB/s memory bandwidth
  - 8 GB/s bandwidth to CPU
    * 4GB/s from device to system
    * 4GB/s from system to device
  - 4GB of graphics double data rate (GDDR) DRAM – global memory – on GPU
    * GDDR DRAM differs from system DRAM on CPU motherboard
CUDA Introduction

- GDDR DRAM is frame buffer memory used for graphics
- Functions as very high bandwidth off-chip memory for computing

CUDA-enabled GPUs

- Operate as a co-processor within the host computer
- Each GPU has its own memory and PEs
- Data needs to be transferred from host to device memory and device to host memory
  - Memory transfers affect performance times
- Use the nvcc compiler to convert C code to run on a GPU
- Preferred to use .cu as extensions to CUDA C source code
- Program to move data between host and device memory

Fixed-function graphics pipeline

- Configurable but non-programmable graphics hardware
- Graphics APIs to use software or hardware functionality
- Send commands to a GPU to display an object being drawn
- Graphics pipeline
  - Host interface receives graphics commands and data from CPU
  - Commands given by host through API
  - DMA hardware in host interface to efficiently transfer bulk data between host and device memory
  - Vertex
    * Corner of a polygon
    * Graphics pipeline optimized to render triangles
    * Vertex in GeForce refers to vertex of a triangle
    * Surface of an object drawn as a set of triangles
    * Smaller triangles imply better image quality
  - Addressing modes for limited texture size/dimension
  - Vertex control stage
    * Receives parameterized triangle data from CPU
    * Converts it to a form understood by hardware
    * Places prepared data into vertex cache
  - Vertex shading, transform, and lighting stage
    * Transforms vertices and assigns per-vertex values
      - Color, normal, texture coordinates, tangents
    * Shading done by pixel shading hardware
    * Vertex shader assigns color to each vertex but color is not immediately applied to triangle pixels
    * Edge equations to interpolate colors and other per-vertex data (texture coordinates) across the pixels touched by triangle
  - Raster stage
CUDA Introduction

- Determines pixels contained in each triangle
- Interpolates per vertex values to shade the pixels
- Performs color raster operations to blend the colors of overlapping/adjacent colors for transparency and antialiasing
- Determines the visible objects for a given viewpoint and discards occluded pixels
- Antialiasing gives each pixel a color that is blended from the colors of objects that partially overlap the pixel

- Shader stage
  - Determines final color of each pixel
  - Combined effect of many techniques: interpolation of vertex colors, texture mapping, per-pixel lighting mathematics, reflections, and so on

- Frame buffer interface stage
  - Memory read/write to the display frame buffer memory
  - High bandwidth requirement for high resolution displays
  - Two techniques
    1. Special memory design for higher bandwidth
    2. Simultaneously manage multiple memory channels connected to multiple memory banks

- Evolution of programmable real-time graphics
  - CPU die area dominated by cache memories
  - GPU dominated by floating point datapath and fixed-function logic
  - GPU memory interfaces emphasize bandwidth over latency
    - Latency can be hidden by massively parallel execution

CUDA Programming Model

- 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 and readback speeds
  - Explicit GPU memory management

- C with no shader limitations
  - Integrated host+device app C program
    - Serial or modestly parallel parts in host C code
    - Highly parallel parts in device SPMD kernel C code

- Kernel call
- "Hello world" program
- An empty function named `kernel()` qualified with `__global__`
- A call to the empty function, embellished with `<<<1,1>>>
- Nvidia tools feed the code into the host compiler (gcc)
- `__global__` qualifier alerts the compiler that a function should be compiled to run on the device instead of host
- Kernel is a function callable from the host and executed on the CUDA device – simultaneously by many threads in parallel
- Host calls the kernel by specifying the name of the kernel and an execution configuration
  * Execution configuration defines the number of parallel threads in a group and the number of groups to use when running the kernel for CUDA device
  * Execution configuration defined in angular brackets
    - Angular brackets are not arguments to the device code
    - Parameters to influence how the runtime will launch device code
- Code for addnum
  * Can pass parameters to a kernel just like any C function
  * Need to allocate memory on device to do anything useful
- Code for incr_arr
  * No loop in the device code
  * Function is simultaneously executed by an array of threads on device
  * Each thread is provided with a unique ID that is used to compute different array indices or to make control decisions
  * Unique ID calculated in the register variable `idx` used to refer each element in the array
  * Number of threads can be larger than the size of the array
    - `idx` is checked against `n` to see if the index element needs to be operated on
  * Call to kernel `incr_arr_dev` queues the launch of kernel function on device by an asynchronous call
    - Execution configuration contains the number of blocks and block size
    - Arguments to kernel passed by standard C parameter list
    - Since the device is idle, kernel immediately starts to execute as per execution configuration and function arguments
    - Both host and device execute concurrently their separate code
    - Host calls `cudaMemcpy` which waits until all threads have finished on device (returned from `incr_arr_dev`)
    - Specification of kernel configuration by number of blocks and block size allows the code to be portable without recompilation
  * Built-in variables in the kernel
    - `blockIdx` Block index within the grid
    - `threadIdx` Thread index within the block
    - `blockDim` Number of threads in a block
      - Structures containing integer components of the variables
      - Blocks have `x`, `y`, and `z` components because they are 3D
      - Grids are 2D and contain only `x` and `y` components
      - We used only `x` component because the input array is 1D
      - We added an extra block if `n` was not evenly divisible by `blk_sz`; this may lead to some threads not having any work in the last block
      - Important: *Each thread should be able to access the entire array `a`, `c` on device*
      - No data partitioning when the kernel is launched
• **cudaMalloc**
  - The code should not try to dereference the pointer returned on host
  - Host code can pass this pointer around, perform arithmetic on it, or cast it to a different type
  - Just cannot read or write in memory using this pointer
  - Memory allocated must be freed by using a call to `cudaFree`
  - Pointers within the device code are used exactly as you would on host
    
    ```
    *c = a + b;
    ```
  - In a similar way, do not try to access host memory from within device

**CUDA error handling**

• Every CUDA call, with the exception of kernel launches, returns an error of type `cudaError_t`
  - The successful calls return `cudaSuccess`
  - A failure returns an error code
• The error code can be converted to human readable form by the function
  
  ```
  char * cudaGetErrorString ( cudaError_t code );
  ```
• `cudaGetLastError` reports last error for any previous run time call in host thread
  - Kernel launches are asynchronous; so cannot explicitly check with `cudaGetLastError`
    * Use `cudaThreadSynchronize` to block until the device has completed all previous calls, including kernel calls
    * Returns an error if one of the preceding calls fails
    * Queuing multiple kernel launches implies that error checking can only be done after all the kernels have completed
  - Errors are reported to the correct host thread
    * If host runs multiple threads on different CUDA devices, error is reported to the correct host thread
– If there are multiple errors, only the last error is reported

Querying devices

- Find out memory capacity and other capabilities of CUDA device
- Some machines may have more than one CUDA capable device
  - The class machine gpu.umsl.edu has two such devices
- Iterate through each device to get the relevant information on each
  - Properties returned in a structure of type cudaDeviceProp

\* Defined in /usr/local/cuda/include/driver_types.h

```c
struct cudaDeviceProp {
    char name[256]; // ASCII string to identify the device
    size_t totalGlobalMem; // Device memory in bytes
    size_t sharedMemPerBlock; // Maximum amount of shared memory in bytes
    // usable in a single block
    int regsPerBlock; // Number of 32-bit registers per block
    int warpSize; // Number of threads in a warp
    size_t memPitch; // Maximum pitch allowed for memory copies
    // in bytes
    int maxThreadsPerBlock; // Maximum number of threads in a block
    int maxThreadsDim[3]; // Max number of threads along each dimension
    int maxGridSize[3]; // Number of blocks along each side of grid
    size_t totalConstMem; // Available constant memory
    int major; // Major revision of device compute capability
    int minor; // Minor revision of device compute capability
    int clockRate; // Clock frequency in kHz
    size_t textureAlignment; // Device requirement for texture alignment
    int deviceOverlap; // Can device simultaneously perform a
    // cudaMemcpy and kernel execution
    int multiProcessorCount; // Number of PEs on device
    int kernelExecTimeoutEnabled; // Is there a runtime limit for
    // kernels executed on this device
    int integrated; // Is the device an integrated GPU?
    // Integrated GPU is part of chipset and
    // not a discrete GPU
    int canMapHostMemory; // Device can map host memory into CUDA
    // address space
    int computeMode; // Device computing mode: default,
    // exclusive, or prohibited
    int maxTexture1D; // Max size for 1D textures
    int maxTexture2D[2]; // Max dimensions for 2D textures
    int maxTexture3D[3]; // Max dimensions for 3D textures
    int maxTexture2DArray[3]; // Max dimensions for 2D texture arrays
    int concurrentKernels; // Does device support executing multiple
    // kernels within same context simultaneously
};
```

- Using device properties
  - Device properties useful in optimizing the code to take advantage of the underlying hardware and software
- Double precision floating point math is supported in cards with compute capability 1.3 or higher
  * To support double precision floating point, we need to work with the device that has compute capability of 1.3 or higher
- Find the device that can achieve the goals and use it

```c
int dev;
cudaDeviceProp prop;
memset(&prop, 0, sizeof(cudaDeviceProp));
prop.major = 1;
prop.minor = 3;
cudaChooseDevice(&dev, &prop);
printf( "CUDA device closest to revision 1.3 is: %d\n", dev);
cudaSetDevice(dev);
```

**CUDA parallel programming**

- `__global__` qualifier
  - Adding this qualifier and calling it with special angle bracket syntax, the function is executed on GPU
- Kernel to increment an array defined as follows:

```c
__global__
void incr_arr_dev(
    float * arr, // Array to be incremented
    int n       // Number of elements in array
)
{
    int idx;     // To define thread index

    idx = blockIdx.x * blockDim.x + threadIdx.x;

    if ( idx < n )
        arr[idx] += 1.0f;
}
```

- Kernel launch
  - Requires specification of an execution configuration
    * Number of threads that compose a block
    * Number of blocks that form a grid
    * A block can only be processed on a single multi-processor
  - The call to launch kernel is in terms of:

```c
int n = 256;        // Size of array
int a_d[256];      // Array on device
int blk_sz = 4;
int num_blks = (int)ceil((float)n / blk_sz);
incr_arr_dev <<<num_blks, blk_sz >>>(a_d, n);
```
  * This creates `num_blks` copies of the kernel and executes those concurrently
  * Within the kernel, the block is identified as a CUDA built-in variable `blockIdx`; variable defined by CUDA runtime
    * Blocks can be defined in two dimensions
- Since we are adding a 1D array, we use `blockIdx.x` to get the first dimension; the second one being 1
- Collection of parallel blocks is called a *grid*
- The above call interprets kernel as a grid of 1D blocks
- Scalar values are interpreted as 1D

**Data Parallelism**

- Many arithmetic operations can be concurrently and safely performed on data structures
- Example: Matrix-matrix multiplication $C = A \cdot B$
  - Each element of product matrix $C$ is a result of dot product of a row from $A$ and column from $B$
  - The dot product for each element of $C$ can be performed concurrently

**Shared memory**

- Global memory can deliver over 60GB/s or 15GF/s for single touch use of data
- Reuse of local data can achieve higher performance
  - May hide global memory latency and global memory bandwidth restrictions
- Kernel launch requires specification of an execution configuration as
  - Number of threads that compose a block
  - Number of blocks that compose a grid
- A block can only be processed on a single multiprocessor
  - Threads within a block can communicate with each other through local multiprocessor resources such as local shared memory
- Balancing hardware and software resources against cost
  - Developers want large amounts of local multiprocessor resources such as registers and shared memory
  - Hardware needs to be inexpensive but fast local multiprocessor memory is expensive
  - Different hardware with different capabilities and cost
  - CUDA occupancy calculator:
    - Autoconfiguring the application by querying the device helps in optimization
- **CUDA execution model**
  - Each hardware multiprocessor can concurrently process multiple blocks
    - Capability depends on number of registers per thread and the amount of shared memory needed by the kernel
    - Blocks processed by one multiprocessor at one time are called *active*
    - Kernels with minimal resource requirements are better because multiprocessor resources are split among all threads of active blocks
    - If there are not enough resources to process at least one block, kernel will fail to launch
      - Kernel failure can be caught by checking for errors
  - Warp
    - Each active block split into SIMD groups of threads called warps
    - Each warp has the same number of threads (warp size), executed in SIMD fashion
    - Efficient and cost-effective model from hardware point
    - Serializes conditionals in the sense that both branches of the condition must be evaluated