Graphic Processing Units – GPU

History of GPUs

• VGA in early 90’s -- A memory controller and display generator connected to some (video) RAM
• By 1997, VGA controllers were incorporating some acceleration functions
• In 2000, a single chip graphics processor incorporated almost every detail of the traditional high-end workstation graphics pipeline
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization
• More recently, processor instructions and memory hardware were added to support general-purpose programming languages
• OpenGL: A standard specification defining an API for writing applications that produce 2D and 3D computer graphics
• CUDA (compute unified device architecture): A scalable parallel programming model and language for GPUs based on C/C++
Historical PC architecture

- CPU
  - Front Side Bus
  - North Bridge
  - Memory
  - PCI Bus
  - South Bridge
  - VGA Controller
  - Video Graphics Array
  - Framebuffer Memory
  - VGA Display
  - LAN
  - UART
Contemporary PC architecture
Basic unified GPU architecture

ROP = Raster Operations Pipeline
TPC = Texture Processing Cluster
Tutorial CUDA

Cyril Zeller
NVIDIA Developer Technology
Note: These slides are truncated from a longer version which is publicly available on the web.
Enter the GPU

GPU = *Graphics Processing Unit*

- Chip in computer video cards, PlayStation 3, Xbox, etc.
- Two major vendors: NVIDIA and ATI (now AMD)
Enter the GPU

- GPUs are massively multithreaded manycore chips
  - NVIDIA Tesla products have up to 128 scalar processors
  - Over 12,000 concurrent threads in flight
  - Over 470 GFLOPS sustained performance

- Users across science & engineering disciplines are achieving 100x or better speedups on GPUs

- CS researchers can use GPUs as a research platform for manycore computing: arch, PL, numeric, …
Enter CUDA

CUDA is a scalable parallel programming model and a software environment for parallel computing
- Minimal extensions to familiar C/C++ environment
- Heterogeneous serial-parallel programming model

NVIDIA’s TESLA GPU architecture accelerates CUDA
- Expose the computational horsepower of NVIDIA GPUs
- Enable general-purpose GPU computing

CUDA also maps well to multicore CPUs!
CUDA Programming Model
total_hits = 0;
sample_points_per_thread = sample_points / num_threads;

for (i = 0; i < num_threads; i++) {
    my_arg[i].t_seed = i;           /* can chose any seed – here i is chosen*/
    pthread_create (&p_threads[i], &attr, compute_pi, (void*) &my_arg[i]);
}

for (i = 0; i < num_threads; i++) {
    pthread_join (p_threads[i], NULL);
    total_hits += my_arg[i].hits;
}

computed_pi = 4.0*(double) total_hits / ((double) (sample_points));
Heterogeneous Programming

CUDA = serial program with parallel kernels, all in C
- Serial C code executes in a host thread (i.e. CPU thread)
- Parallel kernel C code executes in many device threads across multiple processing elements (i.e. GPU threads)

<table>
<thead>
<tr>
<th>Serial Code</th>
<th>Host</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Kernel KernelA (args);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serial Code</td>
<td>Host</td>
<td>Device</td>
</tr>
<tr>
<td>Parallel Kernel KernelB (args);</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© NVIDIA Corporation 2008
Kernel = Many Concurrent Threads

- One kernel is executed at a time on the device
- Many threads execute each kernel
  - Each thread executes the same code…
  - … on different data based on its threadID

CUDA threads might be
- **Physical** threads
  - As on NVIDIA GPUs
  - GPU thread creation and context switching are essentially free
- **Virtual** threads
  - E.g. 1 CPU core might execute multiple CUDA threads

```plaintext
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...```

© NVIDIA Corporation 2008
Hierarchy of Concurrent Threads

- Threads are grouped into **thread blocks**
- **Kernel = grid of thread blocks**

```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

By definition, threads in the same block may **synchronize with barriers**

```
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID - 1];
```

Threads wait at the barrier until all threads in the same block reach the barrier
Transparent Scalability

- Thread blocks cannot synchronize
  - So they can run in any order, concurrently or sequentially
- This independence gives scalability:
  - A kernel scales across any number of parallel cores

Implicit barrier between dependent kernels
vec_minus<<<nblocks, blksize>>>(a, b, c);
vec_dot<<<nblocks, blksize>>>(c, c);
Heterogeneous Memory Model

Host memory

CUDA Memcpy()

Device 0 memory

Device 1 memory
Kernel Memory Access

- **Per-thread**
  - Registers: On-chip
  - Local Memory: Off-chip, uncached

- **Per-block**
  - Block
  - Shared Memory: On-chip, small, Fast

- **Per-device**
  - Global Memory:
    - Off-chip, large
    - Uncached
    - Persistent across kernel launches
    - Kernel I/O
Physical Memory Layout

- “Local” memory resides in device DRAM
- Use registers and shared memory to minimize local memory use
- Host can read and write global memory but not shared memory
10-Series Architecture

- 240 *thread processors* execute kernel threads
- 30 *multiprocessors*, each contains
  - 8 thread processors
  - One double-precision unit
- *Shared memory* enables thread cooperation
Execution Model

**Software**

- Thread
- Thread Block

**Hardware**

- Thread Processor
- Multiprocessor
- Grid
- Device

**Threads are executed by thread processors**

**Thread blocks are executed on multiprocessors**

**Thread blocks do not migrate**

**Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)**

**A kernel is launched as a grid of thread blocks**

**Only one kernel can execute on a device at one time**
CUDA Programming Basics

Part I - Software Stack and Memory Management
Compiler

Any source file containing language extensions, like “<<< >>>”, must be compiled with `nvcc`

`nvcc` is a compiler driver
- Invokes all the necessary tools and compilers like cudacc, g++, cl, ...

`nvcc` can output either:
- C code (CPU code)
  - That must then be compiled with the rest of the application using another tool
- PTX or object code directly

An executable requires linking to:
- Runtime library (`cudart`)
- Core library (`cuda`)
Compiling

CPU/GPU Source

NVCC

PTX Code

Virtual

PTX to Target Compiler

G80
...
GPU

Physical

Target code
**GPU 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)`

```c
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc((void**)&a_d, nbytes);
cudaMemset(a_d, 0, nbytes);
cudaFree(a_d);
```
Data Copies

`cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`

- `direction` specifies locations (host or device) of `src` and `dst`
- Blocks CPU thread: returns after the copy is complete
- Doesn’t start copying until previous CUDA calls complete
- `enum cudaMemcpyKind`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToDevice`
  - `cudaMemcpyDeviceToHost`
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}

© NVIDIA Corporation 2009
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
a_h = (float *)malloc(nBytes);
b_h = (float *)malloc(nBytes);
cudaMalloc((void **) &a_d, nBytes);
cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaMemcpyHostToDevice;
    return 0;
}
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}

© NVIDIA Corporation 2009
```c
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```
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
Executing Code on the GPU

Kernels are C functions with some restrictions

- Cannot access host memory
- Must have `void` return type
- No variable number of arguments ("varargs")
- Not recursive
- No static variables

Function arguments automatically copied from host to device
Function Qualifiers

- Kernels designated by function qualifier: __global__
  - Function called from host and executed on device
  - Must return void

- Other CUDA function qualifiers __device__
  - Function called from device and run on device
  - Cannot be called from host code

- __host__
  - Function called from host and executed on host (default)
  - __host__ and __device__ qualifiers can be combined to generate both CPU and GPU code
Launching Kernels

- Modified C function call syntax:
  
  ```c
  kernel<<<dim3 dG, dim3 dB>>>(...)
  ```

- Execution Configuration ("<<< >>>")
  - `dG` - dimension and size of grid in blocks
    - Two-dimensional: `x` and `y`
    - Blocks launched in the grid: `dG.x*dG.y`
  
  - `dB` - dimension and size of blocks in threads:
    - Three-dimensional: `x`, `y`, and `z`
    - Threads per block: `dB.x*dB.y*dB.z`
  
- Unspecified `dim3` fields initialize to 1
More on Thread and Block IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on

- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
Execution Configuration Examples

```c
dim3 grid, block;
grid.x = 2; grid.y = 4;
block.x = 8; block.y = 16;
kernel<<<grid, block>>>(...);
```

```c
dim3 grid(2, 4), block(8,16);
kernell<<<grid, block>>>(...);
```

```c
kernel<<<32,512>>>(...);
```

Equivalent assignment using constructor functions
CUDA Built-in Device Variables

All `__global__` and `__device__` functions have access to these automatically defined variables:

- `dim3 gridDim;`  
  Dimensions of the grid in blocks (at most 2D)
- `dim3 blockDim;`  
  Dimensions of the block in threads
- `dim3 blockIdx;`  
  Block index within the grid
- `dim3 threadIdx;`  
  Thread index within the block
Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
  - Map from local thread ID (threadIdx) to a global ID which can be used as array indices

**Grid**

<table>
<thead>
<tr>
<th>blockIdx.x</th>
<th>blockIdx.x*blockDim.x + threadIdx.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>1</td>
<td>5 6 7 8 9</td>
</tr>
<tr>
<td>2</td>
<td>10 11 12 13 14</td>
</tr>
</tbody>
</table>

blockIdx.x = 5

threadIdx.x

© NVIDIA Corporation 2009
Minimal Kernels

```cpp
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```cpp
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 2 2 2 2

```cpp
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
Increment Array Example

CPU program

```c
void inc_cpu(int *a, int N) {
    int idx;
    for (idx = 0; idx < N; idx++)
        a[idx] = a[idx] + 1;
}

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

CUDA program

```c
__global__ void inc_gpu(int *a_d, int N) {
    int idx = blockIdx.x * blockDim.x
               + threadIdx.x;
    if (idx < N)
        a_d[idx] = a_d[idx] + 1;
}

void main()
{
    ... dim3 dimBlock (blocksize);
    dim3 dimGrid(ceil(N/(float)blocksize));
    inc_gpu<<<dimGrid, dimBlock>>>(a_d, N);
    ...}
```
Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed

- `cudaMemcpy()` is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed

- `cudaThreadSynchronize()`
  - blocks until all previous CUDA calls complete
Host Synchronization Example

...  
// copy data from host to device
cudaMemcpy(a_d, a_h, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
inc_gpu<<<ceil(N/(float)blocksize), blocksize>>>(a_d, N);

// run independent CPU code
run_cpu_stuff();

// copy data from device back to host
cudaMemcpy(a_h, a_d, numBytes, cudaMemcpyDeviceToHost);

...
Variable Qualifiers (GPU code)

**device**
- Stored in global memory (large, high latency, no cache)
- Allocated with `cudaMalloc(__device__ qualifier implied)`
- Accessible by all threads
- Lifetime: application

**shared**
- Stored in on-chip shared memory (very low latency)
- Specified by execution configuration or at compile time
- Accessible by all threads in the same thread block
- Lifetime: thread block

**Unqualified variables:**
- Scalars and built-in vector types are stored in registers
- Arrays may be in registers or local memory
GPU Thread Synchronization

- void __syncthreads();
- Synchronizes all threads in a block
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block
GPU Atomic Integer Operations

- Requires hardware with compute capability $\geq 1.1$
  - G80 = Compute capability 1.0
  - G84/G86/G92 = Compute capability 1.1
  - GT200 = Compute capability 1.3

- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
    - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap

- Atomic operations on integers in shared memory
  - Requires compute capability $\geq 1.2$