Graphic Processing Units – GPU (Section 7.7)

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
Contemporary PC architecture
Basic unified GPU architecture

ROP = raster operations pipeline
Note: The following slides are extracted from different presentations by NVIDIA (publicly available on the web)

For more details on CUDA see:
http://docs.nvidia.com/cuda/cuda-c-programming-guide

(or search for "CUDA programming guide" on Google)
The World Leader in Parallel Processing
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, …
GTX Titan: For High Performance Gaming Enthusiasts

- **CUDA Cores**: 2688
- **Single Precision**: ~4.5 Tflops
- **Double Precision**: ~1.27 Tflops
- **Memory Size**: 6GB
- **Memory B/W**: 288GB/s
Heterogeneous Computing

- **Terminology:**
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)
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));
}
CUDA Accelerates Computing

Choose the right processor for the right task

CPU
Several sequential cores

CUDA GPU
Thousands of parallel cores
Heterogeneous Computing

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }

  // Synchronize (ensure all the data is available)
  __syncthreads();

  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS; offset <= RADIUS; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}

void fill_ints(int *x, int n) {
  fill_n(x, n, 1);
}

int main(void) {
  int *in, *out;    // host copies of a, b, c
  int *d_in, *d_out; // device copies of a, b, c
  int size = (N + 2*RADIUS) * sizeof(int);

  // Alloc space for host copies and setup values
  in  = ( int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
  out = ( int *)malloc(size); fill_ints(out, N + 2*RADIUS);

  // Alloc space for device copies
  cudaMalloc((void **)& d_in,  size);
  cudaMalloc((void **)&d_out, size);

  // Copy to device
  cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

  // Launch stencil_1d() kernel on GPU
  stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

  // Copy result back to host
  cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

  // Cleanup
  free(in); free(out);
  cudaFree(d_in); cudaFree(d_out);
  return 0;
}
```
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
Simple Processing Flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
CUDA Kernels: Parallel Threads

A **kernel** is a function executed on the GPU as an array of threads in parallel.

All threads execute the same code, can take different paths.

Each thread has an ID:
- Select input/output data
- Control decisions

```c
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
CUDA Kernels

- Threads are grouped into **blocks**
- **Blocks** are grouped into a **grid**
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
Kernel Execution

Each kernel is executed on one device. Multiple kernels can execute on a device at one time.

Each thread is executed by a core.

Each block is executed by one SM and does not migrate. Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources.

Each kernel is executed on one device. Multiple kernels can execute on a device at one time.
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store memory that they all use
  - Share results with each other
  - Cooperate to produce a single result
  - Synchronize with each other
Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs
Blocks are divided into 32 thread wide units called warps
  - Size of warps is implementation specific and can change in the future

The SM creates, manages, schedules and executes threads at warp granularity
  - Each warp consists of 32 threads of contiguous threadIds

All threads in a warp execute the same instruction
  - If threads of a warp diverge the warp serially executes each branch path taken

When a warp executes an instruction that accesses global memory it coalesces the memory accesses of the threads within the warp into as few transactions as possible
Hierarchy of Concurrent Threads

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

### Thread Block 0
```plaintext
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

### Thread Block 1
```plaintext
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

### Thread Block N - 1
```plaintext
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

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

```plaintext
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.
Heterogeneous Memory Model

Host memory

Device 0 memory
cudaMemcpy()

Device 1 memory
Kernel Memory Access

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

- **Per-block**
  - Shared Memory: On-chip, small
    - Fast
  - Global Memory: Off-chip, large
    - Uncached
    - Persistent across kernel launches
    - Kernel I/O

- **Per-device**

© NVIDIA Corporation 2009
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

Hardware

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
Host (CPU) manages device (GPU) memory

- cudaMalloc(void **pointer, size_t nbytes)
- cudaMemcpy(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 );
cudaMemcpy( a_d, 0, nbytes );
cudaFree(a_d);
```
Data Copies

```c
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

```c
enum cudaMemcpyKind
```

- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`
```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++) 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);
    cudaMemcpy((void **) &a_d, nBytes);
    cudaMemcpy((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;
}
```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++) 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_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);
    return 0;
}
```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);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, 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++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```
```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);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, 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++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

```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);
    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);
    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;
}