Most of these slides were taken from a NVIDIA corporation tutorial for CUDA programming in C.
Announcements

(1) Project 1 has been posted. Please see project description on our class web site.

Measure performance, power, energy for CPU, OpenMP, and CUDA versions of quickshift (an image smoothing application) under different parameter settings. Project is a tradeoff study.

(2) Please pick your papers to present.
#pragma omp parallel for private(i, hash)
    for (j = 0; j < num_hf; j++) {
        for (i = 0; i < wl_size; i++) {
            hash = hf[j] (get_word(wl, i));
            hash %= bv_size;
            bv[hash] = 1;
        }
    }

• outermost (j-loop) is parallel
• each thread will get its own copy of variables i and hash, eliminating loop carried anti and output dependences on i and hash.
• output dependence on bv remains
#define CHUNK_SIZE 2
int chunk = CHUNK_SIZE
#pragma omp parallel for \
  schedule (dynamic, chunk) \
  private(i, hash)
for (j = 0; j < num_hf; j++) {
  for (i = 0; i < wl_size; i++) {
    hash = hf[j] (get_word(wl, i));
    hash %= bv_size;
    bv[hash] = 1; }
}

• Scheduling optimization: Outermost parallel j-loop with CHUNK SIZE iterations scheduled as a group; default chunk size=1
• three basic scheduling strategies: static, dynamic, or guided
My View of the World

GPGPU

CPU

GPU
CUDA C/C++ BASICS

NVIDIA Corporation 2013

CUDA: Compute Unified Device Architecture - NVIDIA 2006

Most of these slides were taken from a NVIDIA corporation tutorial for CUDA programming in C
What is CUDA?

- **CUDA Architecture**
  - Expose GPU parallelism for general-purpose computing
  - Retain performance

- **CUDA C/C++**
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - "Straightforward" APIs to manage devices, memory etc.

- This lecture introduces CUDA C/C++
Terminology:

- **Host**  
  The CPU and its memory (host memory)
- **Device**  
  The GPU and its memory (device memory)
```c
#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];
    // (threadIdx.x < RADIUS) ?
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];

    __syncthreads();

    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];
    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
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

**Parallel and Serial Code**

---

**Parallel Function**

```c
__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];
    // (threadIdx.x < RADIUS) ?
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];

    __syncthreads();

    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];
    out[gindex] = result;
}
```

**Serial Code**

```c
void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}
```
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
int main(void) {
    printf("Hello World!\n");
    return 0;
}

- Standard C that runs on the host

- NVIDIA compiler (nvcc) can be used to compile programs with no device code

Output:

$ nvcc hello_world.cu
$ a.out
Hello World!
$
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Two new syntactic elements...
__global__ void mykernel(void) {

}

CUDA C/C++ keyword __global__ indicates a function that:

→ Runs on the device
→ Is called from host code

nvcc separates source code into host and device components

→ Device functions (e.g. mykernel()) processed by NVIDIA compiler
→ Host functions (e.g. main()) processed by standard host compiler
  • gcc, cl.exe
mykernel<<<1,1>>>();

Triple angle brackets mark a call from host code to device code
→ Also called a “kernel launch”
→ We’ll return to the parameters (1,1) in a moment

That’s all that is required to execute a function on the GPU!
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

• `mykernel()` does nothing, somewhat anticlimactic!

Output:

```
$ nvcc hello.cu
hello.cu
$ a.out
Hello World!
$ 
```
• But wait... GPU computing is about massive parallelism!

• We need a more interesting example...

• We’ll start by adding two integers and build up to vector addition
A simple kernel to add two integers

```c
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

As before `__global__` is a CUDA C/C++ keyword meaning:
- `add()` will execute on the device
- `add()` will be called from the host
Addition on the Device

• Note that we use pointers for the variables

```c
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

• `add()` runs on the device, so `a`, `b` and `c` must point to device memory

• We need to allocate memory on the GPU
Memory Management

- Host and device memory are separate entities
  - **Device** pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - **Host** pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`
• Returning to our `add()` kernel

```c
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

• Let’s take a look at `main()`...
int main(void) {
    int a, b, c;  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Setup input values
    a = 2;
    b = 7;
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices
GPU computing is about massive parallelism

So how do we run code in parallel on the device?

\[ \text{add}<<<1, 1>>>(); \]

\[ \text{add}<<<N, 1>>>(); \]

Instead of executing \text{add()} once, execute \text{N} times in parallel
• With `add()` running in parallel we can do vector addition

• Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

```c
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

• By using `blockIdx.x` to index into the array, each block handles a different index
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

On the device, each block can execute in parallel:

Block 0
   c[0] = a[0] + b[0];

Block 1
   c[1] = a[1] + b[1];

Block 2

Block 3
• Returning to our parallelized `add()` kernel

```c
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

• Let’s take a look at main()...
#define N 512
int main(void) {
    int *a *b *c    // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
• Difference between host and device
  → Host CPU
  → Device GPU

• Using __global__ to declare a function as device code
  → Executes on the device
  → Called from the host

• Passing parameters from host code to a device function
Review (2 of 2)

- **Basic device memory management**
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- **Launching parallel kernels**
  - Launch \( N \) copies of `add()` with `add<<<N,1>>>(...);`
  - Use `blockIdx.x` to access block index