A Few Words About Final Project

- **Purpose:**
  - A chance to dig in deeper into a parallel programming model and explore concepts.
  - Research experience:
    - Freedom to pick your own problem and solution, get feedback
    - Thrill of victory, agony of defeat
  - Communication skills
    - Present results to communicate technical ideas
- Write a non-trivial parallel program that combines two parallel programming languages/models. In some cases, just do two separate implementations:
  - OpenMP + SSE
  - OpenMP + CUDA (but need to do this in separate parts of the code)
  - MPI + OpenMP
  - MPI + SSE
  - MPI + CUDA
- Present results in a poster session on the last day of class.

Example Projects

- Look in the textbook or look on-line
  - Chapter 6: N-body, Tree search
  - Chapters 3 and 5: Sorting
  - Image and signal processing algorithms
  - Graphics algorithms
  - Stencil computations
  - FFT
  - Graph algorithms
  - Other domains...
- Must change it up in some way from text
  - Different language/strategy

Details and Schedule

- 2-3 person projects
  - Let me know if you need help finding a team
- Ok to combine with project for other class, but expectations will be higher and professors will discuss
- Each group must talk to me about their project between now and November 10
  - Before/after class, during office hours or by appointment
  - Bring written description of project, slides are fine
  - Must include your plan and how the work will be shared across the team
- I must sign off by November 22 (in writing)
- Dry run on December 6
- Poster presentation on December 8
Strategy

- A lesson in research
  - Big vision is great, but make sure you have an evolutionary plan where success comes in stages
  - Sometimes even the best students get too ambitious and struggle
- Parallel programming is hard
  - Some of you will pick problems that don’t speed up well and we’ll need to figure out what to do
  - There are many opportunities to recover if you have problems
    - I’ll check in with you a few times and redirect if needed
    - Feel free to ask for help
  - Optional final report can boost your grade, particularly if things are not working on the last day of classes

Remainder of Semester

- Next programming assignment
  - Posted later today, due November 4
  - Combines task and data parallelism with locality and possibly SIMD in OpenMP
- Project 4
  - CUDA assignment, due November 22
- Homework 4
  - MPI, algorithms, preparation for final exam, due December 1

Outline

- Overview of the CUDA Programming Model for NVIDIA systems
  - Presentation of basic syntax
- Simple working examples
  - See http://www.cs.utah.edu/~mhall/cs6963s09
- Architecture
- Execution Model
- Heterogeneous Memory Hierarchy
  - This lecture includes slides provided by Wen-mei Hwu (UIUC) and David Kirk (NVIDIA)
  - See http://courses.ece.uiuc.edu/eece498/af1
  - and Austin Robison (NVIDIA)

Reading

- David Kirk and Wen-mei Hwu manuscript or book
- CUDA Manual, particularly Chapters 2 and 4
  (download from nvidia.com/cudazone)
- Nice series from Dr. Dobbs Journal by Rob Farber
  - http://www.ddj.com/cpp/207200659
Today’s Lecture

• Goal is to enable writing CUDA programs right away
  - Not efficient ones – need to explain architecture and mapping for that
  - Not correct ones (mostly shared memory, so similar to OpenMP)
  - Limited discussion of why these constructs are used or comparison with other programming
  - Limited discussion of how to use CUDA environment
  - No discussion of how to debug.

Why Massively Parallel Processor

• A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until last year, programmed through graphics API

 GFLOPS

NV40 = GeForce 6800 Ultra
NV35 = GeForce FX 5950 Ultra
NV30 = GeForce FX 5800

Why Massively Parallel Processor

• Computation partitioning (where does computation occur?)
  - Declarations on functions __host__, __global__, __device__
  - Mapping of threads to device: compute <<<gs, bs>>>(<args>)

• Data partitioning (where does data reside, who may access it and how?)
  - Declarations on data __shared__, __device__, __constant__, ...
• Data management and orchestration
  - Copying to/from host: e.g., cudaMemcpy(h_obj, d_obj, cudaMemcpyDeviceToHost)
  - Concurrency management
    - E.g., __syncthreads()

Minimal Extensions to C + API

• Declspecs
  - __device__ float filter[N];
  - __global__ void convolve (float *image)
  - __shared__ float region[N];

• Keywords
  - threadIdx, blockIdx
  - __syncthreads
  - __syncthreads()
  - image[i] = result;

• Runtime API
  - Memory, symbol, execution management
    // Allocate GPU memory
    void *myimage = cudaMalloc(bytes)

• Function launch
  // 100 blocks, 10 threads per block
  convolve<<<100, 10>>>(myimage)
NVCC Compiler’s Role: Partition Code and Compile for Device

```
int main_data;
__shared__ int sdata;

Main() {}
__host__ hfunc() {
int hdata;
<<<gfunc(g,b,m)>>>()
}
```

Compiled by native compiler: gcc, icc, cc
Compiled by nvcc compiler

CUDA Programming Model:
A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
  - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate

Block and Thread IDs
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.x, threadIdx.y)
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...
**Simple working code example: Count 6**

- **Goal for this example:**
  - Really simple but illustrative of key concepts
  - Fits in one file with simple compile command
  - Can absorb during lecture

- **What does it do?**
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does "6" appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

  Known as a cyclic data distribution

**CUDA Pseudo-Code**

**MAIN PROGRAM:**

- **Initialization**
  - Allocate memory on host for input and output
  - Assign random numbers to input array
  - Call function
  - Calculate final output from per-thread output
  - Print result

**HOST FUNCTION:**

- Allocate memory on device for copy of input and output
- Copy input to device
- Set up grid/block
- Call global function
- Synchronize after completion
- Copy device output to host

**GLOBAL FUNCTION:**

- Thread scans subset of array elements
- Call device function to compare with "6"
- Compute local result

**DEVICE FUNCTION:**

- Compare current element and "6"
- Return 1 if same, else 0

---

**Main Program: Preliminaries**

**MAIN PROGRAM:**

- `#include <stdio.h>`
- `#define SIZE 16`
- `#define BLOCKSIZE 4`
- `int main(int argc, char **argv)`
  - `int *in_array, *out_array;`
  - `...`

---

**Main Program: Invoke Global Function**

**MAIN PROGRAM:**

- `#include <stdio.h>`
- `#define SIZE 16`
- `#define BLOCKSIZE 4`
- `__host__ void outer_compute(int *in_arr, int *out_arr);`
- `int main(int argc, char **argv)`
  - `int *in_array, *out_array;`
  - `...`
- `outer_compute(in_array, out_array);`
Main Program: Calculate Output & Print Result

INITIALIZATION (OMIT)
• Allocate memory on host for input and output
• Assign random numbers to input array
Call host function
Calculate final output from per-thread output
Print result

#include <stdio.h>
define SIZE 16
define BLOCKSIZE 4
__host__ void outer_compute (int *in_arr, int *out_arr);
int main(int argc, char **argv) {
    int *in_array, *out_array;
    int sum = 0;
    /* initialization */ ...
    outer_compute(in_array, out_array);
    for (int i=0; i<BLOCKSIZE; i++) {
        sum+=out_array[i];
    }
    printf ("Result = %d\n",sum);
}

Host Function: Preliminaries & Allocation

HOST FUNCTION:
Allocate memory on device for copy of input and output
Copy input to device
Set up grid/block
Call global function
Synchronize after completion
Copy device output to host

__host__ void outer_compute (int *h_in_array, int *h_out_array) {
    int *d_in_array, *d_out_array;
    cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));
    cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));
    cudaMemcpy(d_in_array, h_in_array, SIZE*sizeof(int), cudaMemcpyHostToDevice);
    // do computation ... 
    cudaMemcpy(d_out_array, h_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
    // ... compute ... 
    cudaMemcpy(h_out_array, d_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}
Global Function

GLOBAL FUNCTION:
Thread scans subset of array elements
Call device function to compare with "6"
Compute local result

__global__ void compute(int *d_in, int *d_out) {
    d_out[threadIdx.x] = 0;
    for (int i=0; i<SIZE/BLOCKSIZE; i++) {
        int val = d_in[i*BLOCKSIZE + threadIdx.x];
        d_out[threadIdx.x] += compare(val, 6);
    }
}

Device Function

DEVICE FUNCTION:
Compare current element and "6"
Return 1 if same, else 0

__device__ int compare(int a, int b) {
    if (a == b) return 1;
    return 0;
}

Reductions

- This type of computation is called a parallel reduction
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) [dimensionality reduced]
- Why might parallel reductions be well-suited to GPUs?
- What if we tried to compute the final sum on the GPUs?

Standard Parallel Construct

- Sometimes called "embarrassingly parallel" or "pleasingly parallel"
- Each thread is completely independent of the others
- Final result copied to CPU
- Another example, adding two matrices:
  - A more careful examination of decomposing computation into grids and thread blocks
Summary of Lecture

- Introduction to CUDA

- Essentially, a few extensions to C++ API supporting heterogeneous data-parallel CPU+GPU execution
  - Computation partitioning
  - Data partitioning (parts of this implied by decomposition into threads)
  - Data organization and management
  - Concurrency management

- Compiler nvcc takes as input a .cu program and produces
  - C Code for host processor (CPU), compiled by native C compiler
  - Code for device processor (GPU), compiled by nvcc compiler

- Two examples
  - Parallel reduction
  - Embarassingly/Pleasingly parallel computation (your assignment)