Overview

- GP-GPU: What and why
- OpenCL, CUDA, and programming CPUs
- GPU Performance demo
A Quick Review: Classes of Parallelism

- **ILP:**
  - Run multiple instructions from one stream in parallel (e.g. pipelining)

- **TLP:**
  - Run multiple instruction streams simultaneously (e.g. openMP)

- **DLP:**
  - Run the same operation on multiple data at the same time (e.g. SSE intrinsics)

GPUs are here
GPUs

- Hardware specialized for graphics calculations
- Graphics calculations are extremely data parallel
  - e.g. double the brightness of every pixel in an image
- Programmers found that they could rephrase some of their problems as graphics manipulations and run them on the GPU
  - Incredibly burdensome for the programmer to use
  - More usable these days – openCL, CUDA
CPU vs. GPU

- Latency optimized
- A couple threads of execution
- Each thread executes quickly
- Serial code
- Lots of caching

- Throughput optimized
- Many, many threads of execution
- Each thread executes slowly
- Parallel code
- Lots of memory bandwidth
OpenCL and CUDA

- Extensions to C which allow for relatively easy GPU programming
- CUDA is NVIDIA proprietary
  - NVIDIA cards only
- OpenCL is opensource
  - Can be used with NVIDIA or ATI cards
- Similar tools, but different jargon
Kernels

- Kernels define the computation for one array index
- The GPU runs the kernel on each index of a specified range
- Similar functionality to map, but you get to know the array index and the array value.
- Call the work at a given index a *work-item*, a *cuda thread*, or a *μthread*.
- The entire range is called an *index-space* or *grid*. 
OpenCL vvadd

/* C version. */

void vvadd(float *dst, float *a, float *b, unsigned n) {
    for(int i = 0; i < n; i++)
        dst[i] = a[i] + b[i]
}

/* openCL Kernel. */

__kernel void vvadd(__global float *dst, __global float *a,
                    __global float *b, unsigned n) {
    unsigned tid = get_global_id(0);
    if (tid < n)
        dst[tid] = a[tid] + b[tid];
}
Programmer's View of Execution

Create enough work groups to cover input vector
(openCL calls this ensemble of work groups an index space, can be 3-dimensional in openCL, 2-dimensional in CUDA)

Local work size (programmer can choose)
Conditional \((i < n)\) turns off unused threads in last block
Hardware Execution Model

- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor.
- CPU sends whole index-space over to GPU, which distributes work-groups among cores (each work-group executes on one core)
  - Programmer unaware of number of cores
GPUs use a SIMT model, where individual scalar instruction streams for each work item are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a *warp*. OpenCL refers to them as *wavefronts*.)

```
Scalar instruction stream

ld x  
mul a
ld y  
add  
st y

μT0 μT1 μT2 μT3 μT4 μT5 μT6 μT7

SIMD execution across wavefront
```
Terminology Summary

- **Kernel**: The function that is mapped across the input.

- **Work-item**: The basic unit of execution. Takes care of one index. Also called a microthread or cuda thread.

- **Work-group**: A group of work-items. Each work-group is sent to one core in the GPU.

- **Index-space**: The range of indices over which the kernel is applied.

- **Wavefront**: A group of microthreads (work-items) scheduled to be SIMD executed with each other.
Administrivia

- Project 4 is due May 5th.
- Lab this week is free time to work on the project.
Conditionals in the SIMT Model

- Simple if-then-else are compiled into predicated execution, equivalent to vector masking
- More complex control flow compiled into branches
- How to execute a vector of branches?

Scalar instruction stream

\[
\begin{align*}
\text{tid} &= \text{threadid} \\
\text{If (tid} &\geq n) \text{ skip} \\
\text{Call func1} \\
\text{add} \\
\text{st y} \\
\text{skip:}
\end{align*}
\]

SIMD execution across warp
Branch Divergence

- Hardware tracks which µthreads take or don’t take branch
- If all go the same way, then keep going in SIMD fashion
- If not, create mask vector indicating taken/not-taken
- Keep executing not-taken path under mask, push taken branch PC+mask onto a hardware stack and execute later
- When can execution of µthreads in warp reconverge?
Warps (wavefronts) are multithreaded on a single core

- One warp of 32 µthreads is a single thread in the hardware
- Multiple warp threads are interleaved in execution on a single core to hide latencies (memory and functional unit)
- A single thread block can contain multiple warps (up to 512 µT max in CUDA), all mapped to single core
- Can have multiple blocks executing on one core

[Nvidia, 2010]
OpenCL Memory Model

- **Global** – read and write by all work-items and work-groups
- **Constant** – read-only by work-items; read and write by host
- **Local** – used for data sharing; read/write by work-items in the same work group
- **Private** – only accessible to one work-item
SIMT

- Illusion of many independent threads
- But for efficiency, programmer must try and keep μthreads aligned in a SIMD fashion
- Try to do unit-stride loads and store so memory coalescing kicks in
- Avoid branch divergence so most instruction slots execute useful work and are not masked off
VVADD

/* C version. */
void vvadd(float *dst, float *a, float *b, unsigned n) {
    #pragma omp parallel for
    for(int i = 0; i < n; i++)
        dst[i] = a[i] + b[i]
}

/* openCL Kernel. */
__kernel void vvadd(__global float *dst, __global float *a,
                    __global float *b, unsigned n) {
    unsigned tid = get_global_id(0);
    if (tid < n)
        dst[tid] = a[tid] + b[tid];
}

A: CPU faster
B: GPU faster
/* C version. */
void vvadd(float *dst, float *a, float *b, unsigned n) {
    #pragma omp parallel for
    for(int i = 0; i < n; i++)
        dst[i] = a[i] + b[i]
}

• Only 1 flop per three memory accesses => memory bound calculation.

• “A many core processor ≡ A device for turning a compute bound problem into a memory bound problem” – Kathy Yelick
VECTOR_COP

/* C version. */
void vector_cop(float *dst, float *a, float *b, unsigned n) {
    #pragma omp parallel for
    for(int i = 0; i < n; i++) {
        dst[i] = 0;
        for (int j = 0; j < A_LARGE_NUMBER; j++)
            dst[i] += a[i]*2*b[i] - a[i]*a[i] - b[i]*b[i];
    }
}

/* OpenCL kernel. */
__kernel void vector_cop(__global float *dst, __global float *a, __global float *b, unsigned n) {
    unsigned i = get_global_id(0);
    if (tid < n) {
        dst[i] = 0;
        for (int j = 0; j < A_LARGE_NUMBER; j++)
            dst[i] += a[i]*2*b[i] - a[i]*a[i] - b[i]*b[i];
    }
}

A: CPU faster
B: GPU faster
GP-GPU in the future

- High-end desktops have separate GPU chip, but trend towards integrating GPU on same die as CPU (already in laptops, tablets and smartphones)
  - Advantage is shared memory with CPU, no need to transfer data
  - Disadvantage is reduced memory bandwidth compared to dedicated smaller-capacity specialized memory system
    - Graphics DRAM (GDDR) versus regular DRAM (DDR3)
- Will GP-GPU survive? Or will improvements in CPU DLP make GP-GPU redundant?
  - On same die, CPU and GPU should have same memory bandwidth
  - GPU might have more FLOPS as needed for graphics anyway
Acknowledgements

- These slides contain materials developed and copyright by
  - Krste Asanovic (UCB)
  - AMD
  - codeproject.com
And in conclusion…

- GPUs thrive when
  - The calculation is data parallel
  - The calculation is CPU-bound
  - The calculation is large

- CPUs thrive when
  - The calculation is largely serial
  - The calculation is small
  - The programmer is lazy
Bonus

- OpenCL source code for vvadd and vector_cop demos available at
  http://www-inst.eecs.berkeley.edu/~cs61c/sp13/lec/39/demo.tar.gz