ST: CDA 6938 Multi-Core/Many-Core Architectures and Programming

http://csl.cs.ucf.edu/courses/CDA6938/

Prof. Huiyang Zhou

Outline

- Administration
- Motivation
  - Why multi-core many core processors? Why GPGPU?
- CPU vs. GPU
- An overview of Nvidia G80 and CUDA
Description (Syllabus)

- High performance computing on multi-core / many-core architectures

- Focus:
  - Data-level parallelism, thread-level parallelism
  - How to express them in various programming models
  - Architectural features with high impact on the performance

- Prerequisite
  - CDA5106: Advanced Computer Architecture I
  - C programming

Description (cont.)

- Textbook
  - No required textbooks, four optional ones
  - Papers & Notes

- Tentative grading policy
  - +/- policy will be used
  - Homework: 25%
  - In-class presentation: 10%
  - Participation in discussion: 5% (Not applicable to FEEDS students)
  - Project: 60%
    - Including another in-class presentation
  - A:90~100  B+: 85~90  B: 80~85  B-: 75~80.
Who am I

• Assistant Professor at School of EECS, UCF.

• My research area: computer architecture, back-end compiler, embedded systems
  – High Performance, Power/Energy Efficient, Fault Tolerant Microarchitectures, Multi-core/many-core architectures (e.g., GPGPU), Architectural support for software debugging, Architectural support for information security

Topics

• Introduction to multi-core/many-core architecture
• Introduction to multi-core/many-core programming
• NVidia GPU architectures and the programming model for GPGPU (CUDA)
• AMD/ATI GPU architectures and the programming model for GPGPU (CTM or Brook+) (4 guest lectures from AMD)
• IBM Cell BE architecture and the programming model for GPGPU
• CPU/GPU trade-offs
• Stream processors
• Vector processors
• Data-level parallelism and the associated programming patterns
• Thread-level parallelism and the associated programming patterns
• Future multi-core/many-core architectures
• Future programming support for multi-core/many-core processors
Assignments

- **Homework**
  - #0 “Hello world!” using an emulator of Nvidia G80 processors
  - #1 A small program: Parallel reduction (both on an emulator and the actual graphics processors)
  - #2 Matrix Multiplication
  - #3 Prefix Sum Computation

- **Presentation**
  - an in-depth presentation based on some research papers (either on a particular processor or on GPGPU in general)

- **Projects**
  - Select one processor model from Nvidia G80, ATI streaming processors, and IBM Cell processors.
  - Select (or find your own) an application
  - Try to improve the performance using the GPU that you selected

- **Cross platform comparison**

Experiments

- **Lab: HEC 238**

- Get the keys to HEC 238 from
  - Martin Dimitrov (in charge of experimental environment for AMD/ATI GPUs) dimitrov@cs.ucf.edu
  - Hongliang Gao (in charge of experimental environment for IBM Cell processors) hgao@cs.ucf.edu
  - Jingfei Kong (in charge of experimental environment for Nvidia G80 GPUs) jfkong@cs.ucf.edu

- Schedule the time within the group
Acknowledgement

• Some material including lecture notes are based on the lecture notes of the following courses:
  
  • Programming Massively Parallel Processors (UIUC)
  • Multicore Programming Premier: Learn and Compete Programming for the PS3 Cell Processors (MIT)
  • Multicore and GPU Programming for Video Games (GaTech)

Computer Science at a Crossroads (D. Patterson)

• Old CW: Uniprocessor performance 2X / 1.5 yrs
• New CW: Power Wall + ILP Wall + Memory Wall = Brick Wall
  - Uniprocessor performance now 2X / 5(?) yrs
  ⇒ Sea change in chip design: multiple “cores”
    (2X processors per chip / ~ 2 years)
  • More simpler processors are more power efficient
• The Free (performance) Lunch is over: A Fundamental Turn Toward Concurrency in Software
  - The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency (by Herb Sutter)
Problems with Sea Change

- Algorithms, Programming Languages, Compilers, Operating Systems, Architectures, Libraries, … not ready to supply Thread Level Parallelism or Data Level Parallelism for 1000 CPUs / chip,
- Architectures not ready for 1000 CPUs / chip
  - Unlike Instruction Level Parallelism, cannot be solved by just by computer architects and compiler writers alone, but also cannot be solved without participation of computer architects
  - Modern GPUs run hundreds or thousands threads / chip
- Shifts from Instruction Level Parallelism to Thread Level Parallelism / Data Level Parallelism
  - GPGPU is one such example

GPU at a Glance

- 1st: Designed for graphics applications
- Trend: converging the different functions into a programmable model
- To suit graphics applications
  - High memory bandwidth
    - 86.4 GB/s (GPU) vs. 8.4 GB/s (CPU)
  - High FP processing power
    - 400~500 GFLOPS (GPU) vs. 30~40 GFLOPS (CPU)
- Can we utilize the processing power to perform computing besides graphics?
  - GPGPU
GPU vs. CPU

G80 Die (90 nm tech.) Photo  IBM Power 6 Die (65 nm tech.) Photo

IBM Power 6

- Outstanding Feature: 4.7 GHz; 2 cores with symmetric multiprocessing (SMP) support; 8MB L2 cache
Inside the CPU core (CDA5106)

- Power 5 die

NVidia G80

- Some Outstanding features:
  - 16 highly threaded SM’s, >128 FPU’s
  - Shared memory per SM: 16KB
  - Constant memory: 64 KB
GPU vs. CPU

- The GPU is specialized for compute-intensive, highly data parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than data caching and flow control

CPU

- CPU: all these on-chip estate are used to achieve performance improvement transparent to software developers
  - Sequential programming model
  - Moving towards multi-core and many-core

GPU

- GPU: more on-chip resources used for floating-point computation
  - Requires data parallel programming model
  - Expose architecture features to software developers and software needs to explicitly taking advantage of those features to achieve high performance
Things to know for a GPU processor

- Thread execution model
  - How the threads are executed, how to synchronize threads
  - How the instructions in each/multiple thread(s) are executed
- Memory model
  - How the memory is organized
  - Speed and Size considerations for different types of memories
  - Shared or private memory. If shared, how to ensure the memory ordering
- Control flow handling
- Instruction Set Architecture

Support:
- Programming environment
  - Compiler, debugger, emulator, etc.

HW and SW support for GPGPU

- Nvidia Geforce 8800 GTX vs Geforce 7800
  - Slides from the Nvidia talk given at Stanford Univ.

- Programming models (candidates for course presentation)
  - CUDA
  - Brook+
  - Peak Stream
  - Rapid Mind
GeForce-8 Series HW Overview

CUDA Processor Terminology

- **SPA**
  - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)
- **TPC**
  - Texture Processor Cluster (2 SM + TEX)
- **SM**
  - Streaming Multiprocessor (8 SP)
  - Multi-threaded processor core
  - Fundamental processing unit for CUDA thread block
- **SP**
  - Streaming Processor
  - Scalar ALU for a single CUDA thread

From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu
Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 768 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- DRAM texture and memory access

G80 Thread Computing Pipeline

- The future of GPUs is programmable processing
- So build the architecture around the processor

From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu
CUDA

- “Compute Unified Device Architecture”
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel coprocessor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu

Extended C

- Declspec
  - global, device, shared, local, constant
  - __device__ float filter[N];
  - __global__ void convolve (float *image) {
    __shared__ float region[M];
    ...
  }
- Keywords
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads()
  - __syncthreads()
- Runtime API
  - Memory, symbol, execution management
  - image[j] = result;
  - // Allocate GPU memory
  - void *myimage = cudaMalloc(bytes)
- Function launch
  - // 100 blocks, 10 threads per block
  - convolve<<<100, 10>>>(myimage);

From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu
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:
  - 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
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...
Programming Model: Memory Spaces

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can read/write global, constant, and texture memory

Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached
Compilation

- Any source file containing CUDA language extensions must be compiled with `nvcc`
- `nvcc` is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- `nvcc` can output:
  - Either C code
    - That must then be compiled with the rest of the application using another tool
  - Or object code directly

From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu

Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)

From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu
Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (nvcc -deviceemu) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. printf) and vice-versa
  - Detect deadlock situations caused by improper usage of __syncthreads

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode.
- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host

From the lecture notes of ECE 498 AL by D. Kirk and W. Hwu