University of Central Florida

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

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

Prof. Huiyang Zhou

School of Electrical Engineering and Computer Science
University of Central Florida

Outline

- Administration
- Motivation
  - Why multi-core many core processors? Why GPGPU?
- CPU vs. GPU
- The brief history of GPGPU
- An overview of AMD/ATI streaming processors and the software development toolset (Brook+ and CAL)
- 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%
  - Participation in discussion: 10%
  - Project: 65%
    - Including two in-class presentations
  - 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
- AMD/ATI GPU architectures and the programming model for GPGPU (Brook+ and CAL) (several guest lectures from AMD)
- NVidia GPU architectures and the programming model for GPGPU (CUDA)
- IBM Cell BE architecture and the programming model for GPGPU
- CPU/GPU trade-offs
- 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 emulators (running on CPU) of GPUs
  - Programming assignments (3 sets)
- 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 (PS3) and HEC 242 (Computers with ATI / Nvidia Graphics cards)
- Get the access to the lab and Q & A
  - Yi Yang (yangyi@gmail.com)
- Schedule the time

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)

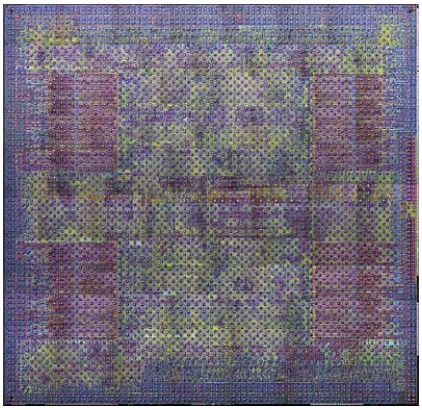
Problems with Sea Change

- 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

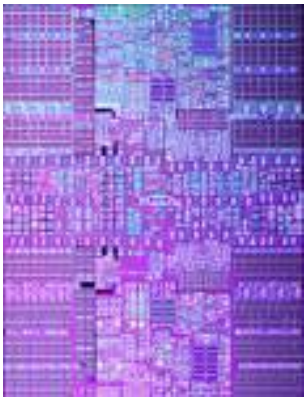
GPU at a Glance

- 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) (last spring)
    - 115.2 (ATI HD 4870); 141.7 GB/s (GTX 280)
  - High FP processing power
    - 400~500 GFLOPS (GPU) vs. 30~40 GFLOPS (CPU) (last spring)
    - 1.2 TFLOPS (ATI HD 4870); 933 GFLOPS (GPU) (GTX 280)
- Can we utilize the processing power to perform computing besides graphics?
  - GPGPU

GPU vs. CPU



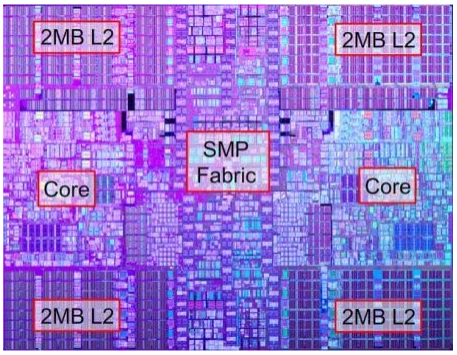
G80 Die (90 nm tech.) Photo



IBM Power 6 Die (65 nm tech.) Photo

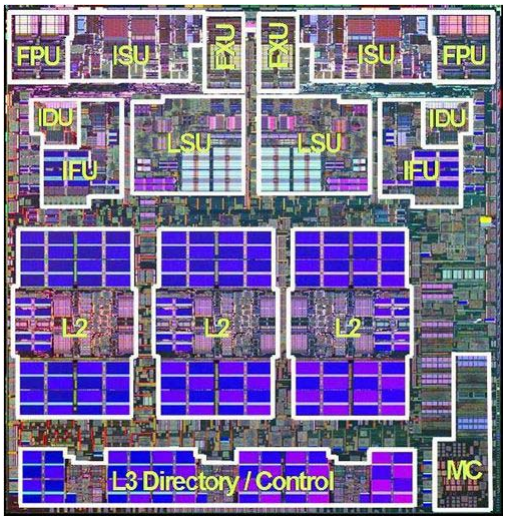
IBM Power 6

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

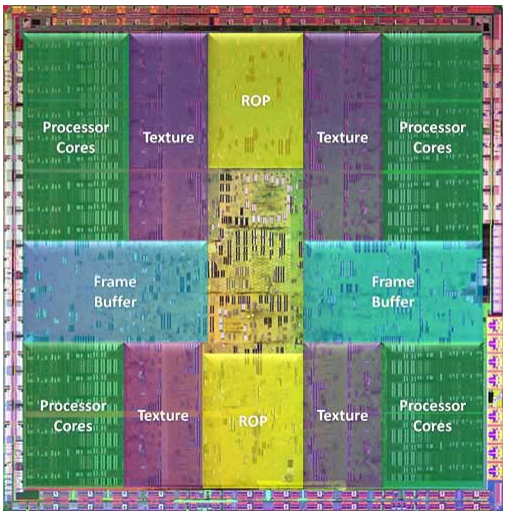


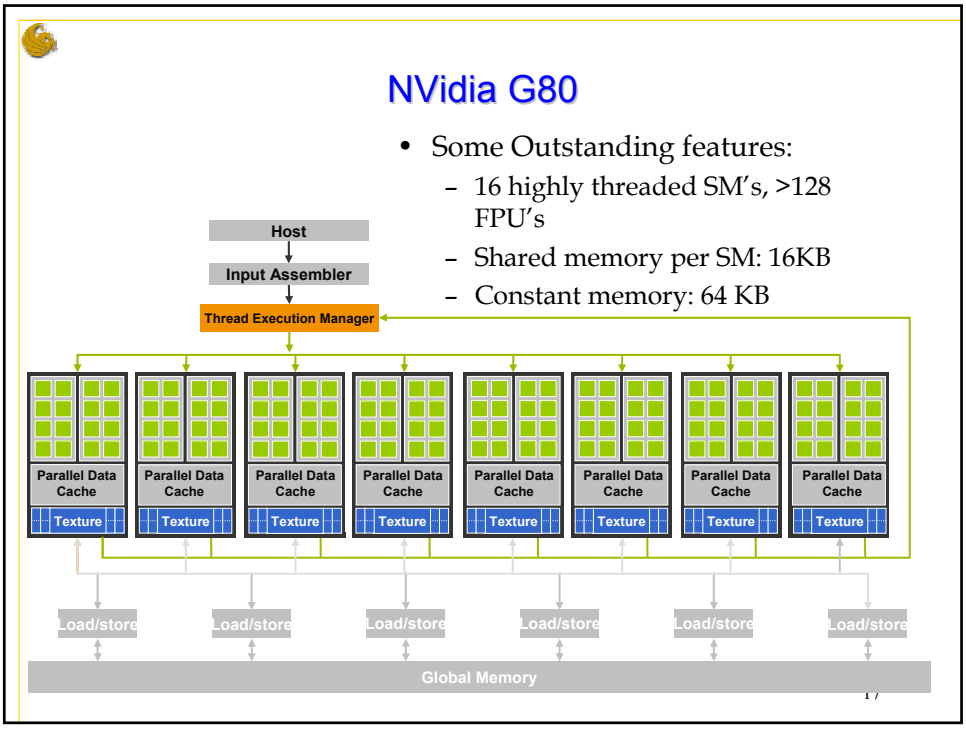
Inside the CPU core (CDA5106)

- Power 5 die



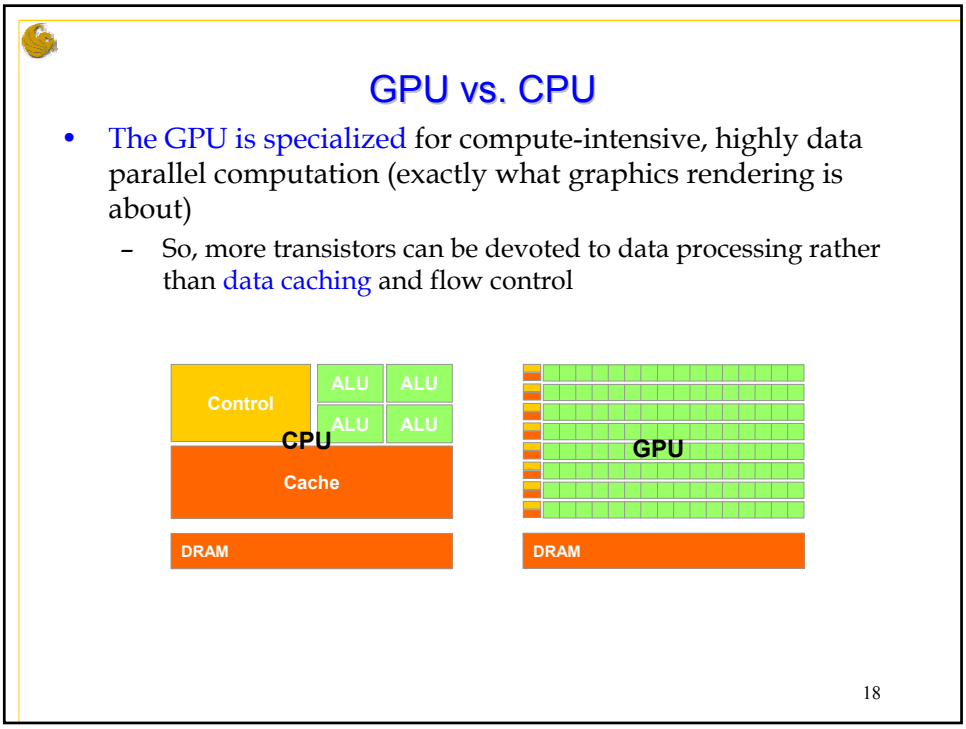
GPU Die (GTX280 65 nm)





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

GPU vs. 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: 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

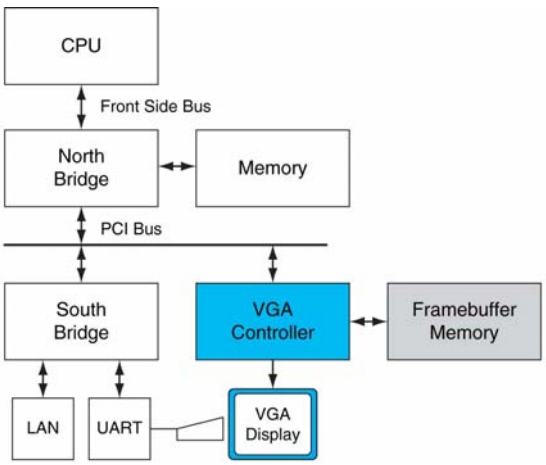


FIGURE A.2.1 Historical PC. VGA controller drives graphics display from framebuffer memory.

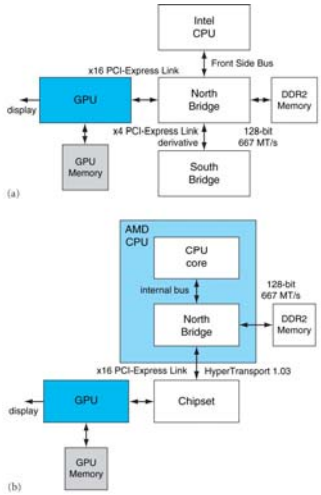


FIGURE A.2.2 Contemporary PCs with Intel and AMD CPUs. See Chapter 6 for an explanation of the components and interconnects in this figure. Copyright © 2009 Elsevier, Inc. All rights reserved.

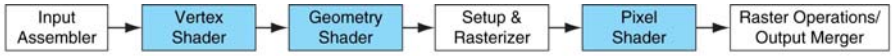


FIGURE A.2.3 Graphics logical pipeline. Programmable graphics shader stages are blue, and fixed-function blocks are white. Copyright © 2009 Elsevier, Inc. All rights reserved.

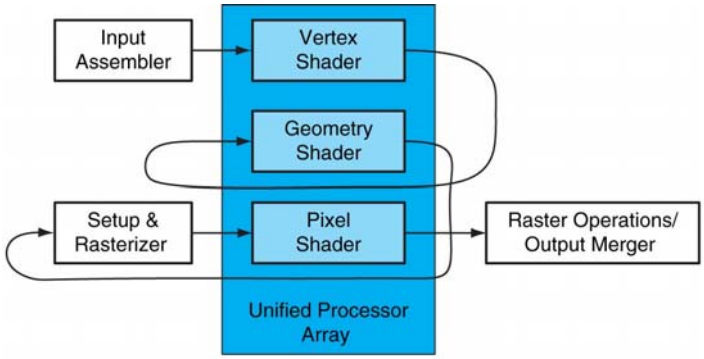


FIGURE A.2.4 Logical pipeline mapped to physical processors. The programmable shader stages execute on the array of unified processors, and the logical graphics pipeline dataflow recirculates through the processors. Copyright © 2009 Elsevier, Inc. All rights reserved.

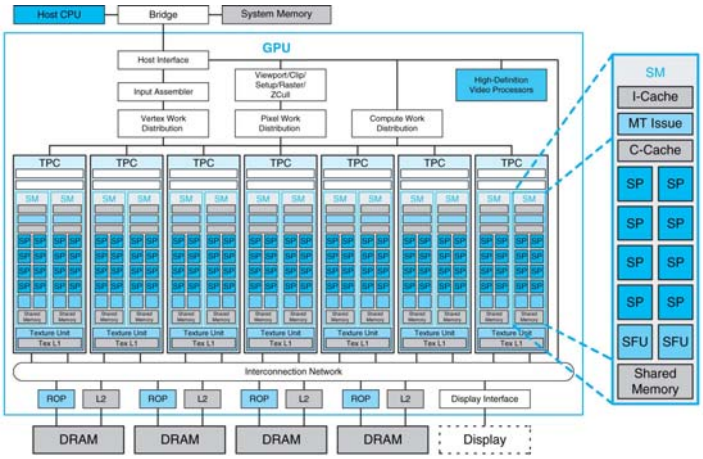


FIGURE A.2.5 Basic unified GPU architecture. Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory. Copyright © 2009 Elsevier, Inc. All rights reserved.

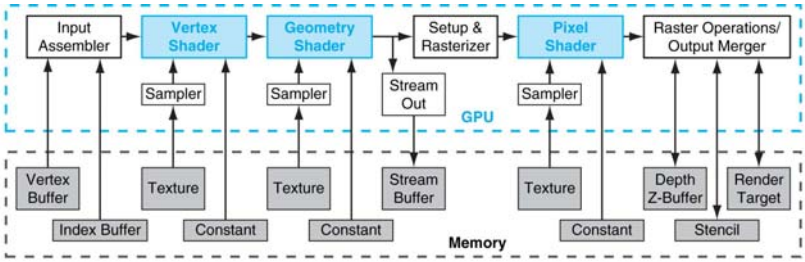


FIGURE A.3.1 Direct3D 10 graphics pipeline. Each logical pipeline stage maps to GPU hardware or to a GPU processor. Programmable shader stages are blue, fixed-function blocks are white, and memory objects are grey. Each stage processes a vertex, geometric primitive, or pixel in a streaming dataflow fashion. Copyright © 2009 Elsevier, Inc. All rights reserved.

FIGURE A.3.2 GPU-rendered image. To give the skin visual depth and translucency, the pixel shader program models three separate skin layers, each with unique subsurface scattering behavior. It executes 1400 instructions to render the red, green, blue, and alpha color components of each skin pixel fragment. Copyright © 2009 Elsevier, Inc. All rights reserved.

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
  - CUDA
  - Brook+
  - OpenCL
  - CAL
  - Peak Stream
  - Rapid Mind