Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Processor Architectures

Schedule

- **Friday, April 13th** – **Pacific Day – No class**
- **Exam 3** – Friday, April 20th
  - Caches
  - Virtual Memory
  - Input / Output
  - Operating Systems
  - Compilers & Assemblers
  - Processor Architecture
  - **Review the lecture notes before the exam (not just the homework!)**
  - **No calculators for this exam**

Flynn's Taxonomy

Flynn's Taxonomy

- Many attempts have been made to come up with a way to categorize computer architectures
- **Flynn's Taxonomy** has been the most enduring of these
  - But it is not perfect!
- Considerations
  - Number of processors?
  - Number of data paths? (or data streams)

Flynn's Taxonomy

- **SISD**: Single instruction stream, single data stream
  - Classic uniprocessor system (e.g. MARIE)
- **SIMD**: Single instruction stream, multiple data streams
  - Execute the same instruction on multiple data values
  - Example: Vector processor
- **MIMD:** Multiple instruction streams, multiple data streams
  - **Today's parallel architectures**
- **MISD:** Multiple instruction streams, single data stream
  - Uncommon – used for fault tolerance

Instruction-Level Parallelism

- Example program: *(imagine it was in assembly)*

```
① e = a + b;
② f = c + d;
③ g = e * h;
```

- Assume we have a processor with "lots" of ALUs
  - **What instructions <u>can</u> be executed in parallel?**
  - **What instructions <u>cannot</u> be executed in parallel?**

Instruction-Level Parallelism

- Example program 2: *(imagine it was in assembly)*

```
① e = a + b;
② f = c + d;
③ if(e > f)
④   a = 15;
⑤ else
⑥   a = 18;
⑦ g = h + 30;
```

- Assume we have a processor with "lots" of ALUs
  - **What instructions can be executed in parallel?**
  - **What instructions cannot be executed in parallel?**
    - ***If we tried really hard, could we run them in parallel?***

Instruction-Level Parallelism

- This is **instruction-level parallelism**
  - Finding instructions in the *same* program that be executed in parallel
  - **Different** from multi-core parallelism, which executes instructions from *different* programs in parallel
- You can do this in a single "core" of a CPU
  - Adding more ALUs to the chip is easy
  - Finding the parallelism to exploit is harder…
  - Getting the data to the ALUs is harder…

Instruction-Level Parallelism

- Instruction-level parallelism is good
  - Let's find as much of it as possible and use it to decrease execution time!
- Two competing methods:
  - **Superscalar**: the *hardware* finds the parallelism
  - **VLIW**: the *compiler* finds the parallelism
- Both designs have **multiple execution units** (e.g. ALUs) in a **single** processor core

MIMD – Superscalar

- **Superscalar** designs – the *hardware* finds the **instruction-level parallelism** while the program is running
- Challenges
  - CPU *instruction fetch unit* must simultaneously retrieve several instructions from memory
  - CPU *instruction decoding unit* determines which of these instructions can be executed in parallel and combines them accordingly
    - **Complicated!**

MIMD – VLIW

- **Very long instruction word (VLIW)** designs – the *compiler* finds the **instruction-level parallelism** before the program executes
  - The *compiler* packs multiple instructions into one **long** instructions that the hardware executes in parallel
- Arguments:
  - **For**: Simplifies hardware, plus the compiler can better identify instruction dependencies (it has more time to work)
  - **Against**: Compilers cannot have a view of the run time code, and must plan for all possible branches and code paths
- Examples: Intel Itanium, ATI R600-R900 GPUs

Instruction-Level Parallelism

- Back to the example program:

```
①   e = a + b;
②   f = c + d;
③   if(e > f)
④     a = 15;
⑤   else
⑥     a = 18;
⑦   g = h + 30;
```

- More techniques for ILP
- **Speculative execution** (or **branch prediction**)
  - Guess that e>f, and execute line 4 immediately…
- **Out-of-order execution**
  - Execute line 7 before 4-6, since it doesn't depend on them

Shared Memory Multiprocessors

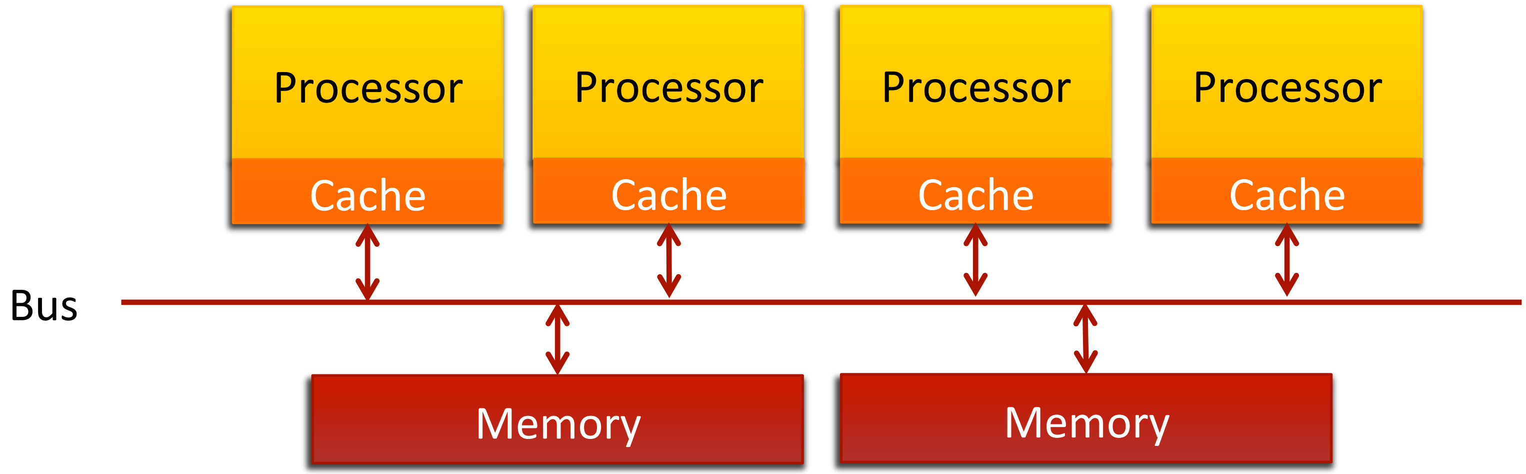
- Imagine a multi-core CPU. How do different cores (running different programs) communicate with each other?
  - One common approach – use main memory!
  - Referred to as **symmetric multiprocessing (SMP)**
- The processors do not necessarily have to share the same block of physical memory
  - Each processor can have its own memory, but it must share it with the other processors

Shared Memory Multiprocessors

- Shared memory MIMD machines can be divided into two categories based upon how they access memory
  - **Uniform memory access (UMA)**
  - **Non-uniform memory access (NUMA)**

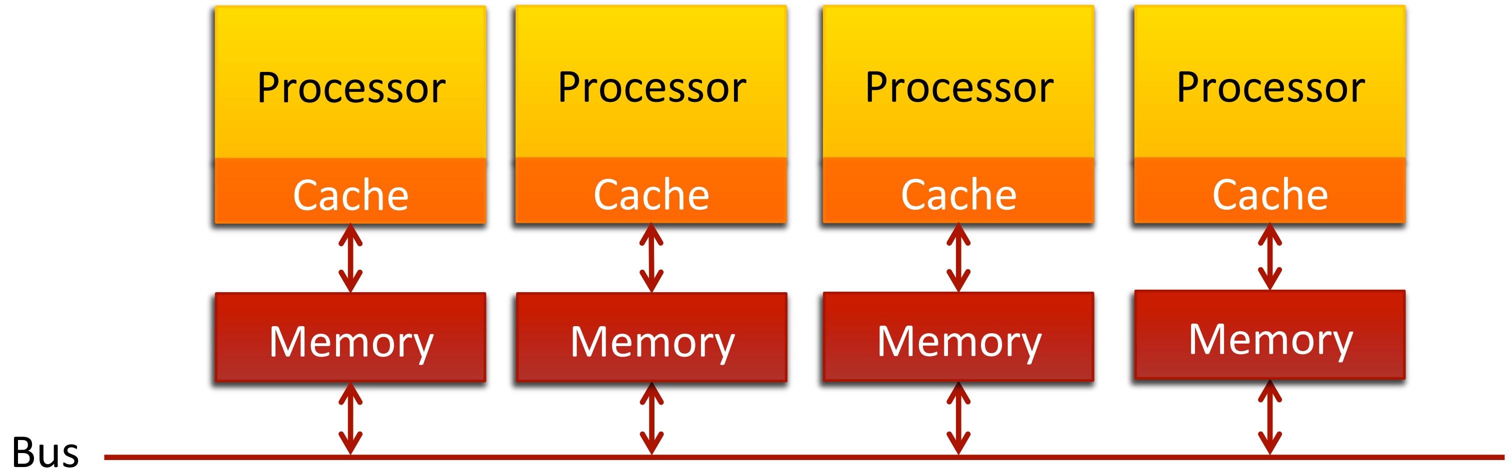
Shared Memory Multiprocessors

- MIMD **uniform memory access (UMA)**
  - All memory accesses take the <u>same amount of time</u>
- Hard to scale to large numbers of processors!
  - Bus becomes a bottleneck



Shared Memory Multiprocessors

- MIMD **nonuniform memory access (NUMA)**
  - A processor can access its own memory much more quickly than it can access memory that is elsewhere
  - Each processor has its own memory and cache
- **More scalable** / **cache coherence challenges**!



Cache Coherence

- What if main memory is changed by processor A, but the cached copy of the data in processor B is *not* changed?
  - Cache coherence problems!
    (We say that the cached value is **stale**)
- Solution? Add even more hardware!
  - Cache coherent NUMA systems
    (e.g. AMD Opteron, Intel Core)
  - Each core monitors the cache writes by the other cores, and updates their own caches

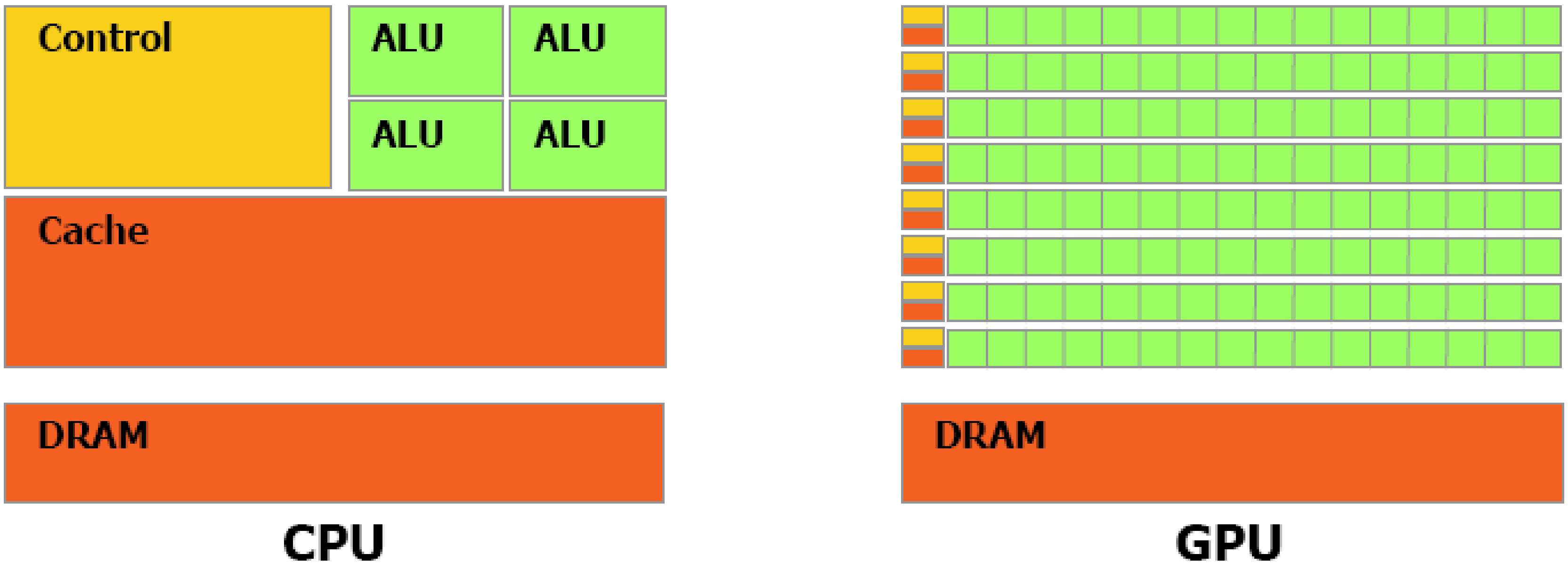
GPUs

What about GPUs?

- **GPU – Graphics Processing Unit**
- GPUs are a specialized processor
  - Target application: 2D and 3D graphics rendering
- GPUs are optimized for highly parallel operation over a finite data set
  - CPU sends data to GPU over PCIe bus
  - CPU tells GPU: render scene and display!
  - GPU operates autonomously

GPU versus CPU Design

- Both Intel and Nvidia have a similar "transistor budget"
  - How do they "spend" those transistors?



GPU versus CPU Design

- **Flexibility?**
  - CPU is the winner
  - Designed for broad range of applications and has a large ISA (instruction set architecture)
- **Single-thread performance?**
  - CPU is the winner
  - CPU cores have transistor-expensive features like out-of-order execution, large caches, branch prediction, etc… that improve single-thread performance
- **Massively-parallel application performance?**
  - GPU is the winner
  - Hundreds of cores, but each is very simple (no/small cache, in-order execution, limited instruction set, limited floating-point support)

GPGPU

- Can we use GPUs for more than just gaming?
- Yes!
  - **General Purpose Computing on GPUs (GPGPU)**
  - Send the data to the GPU along with a program
  - Process it
  - Retrieve the finished data from GPU (instead of displaying it on screen)
- *Only true if your application shares some high-level attributes with game rendering*

GPGPU Strengths / Weaknesses

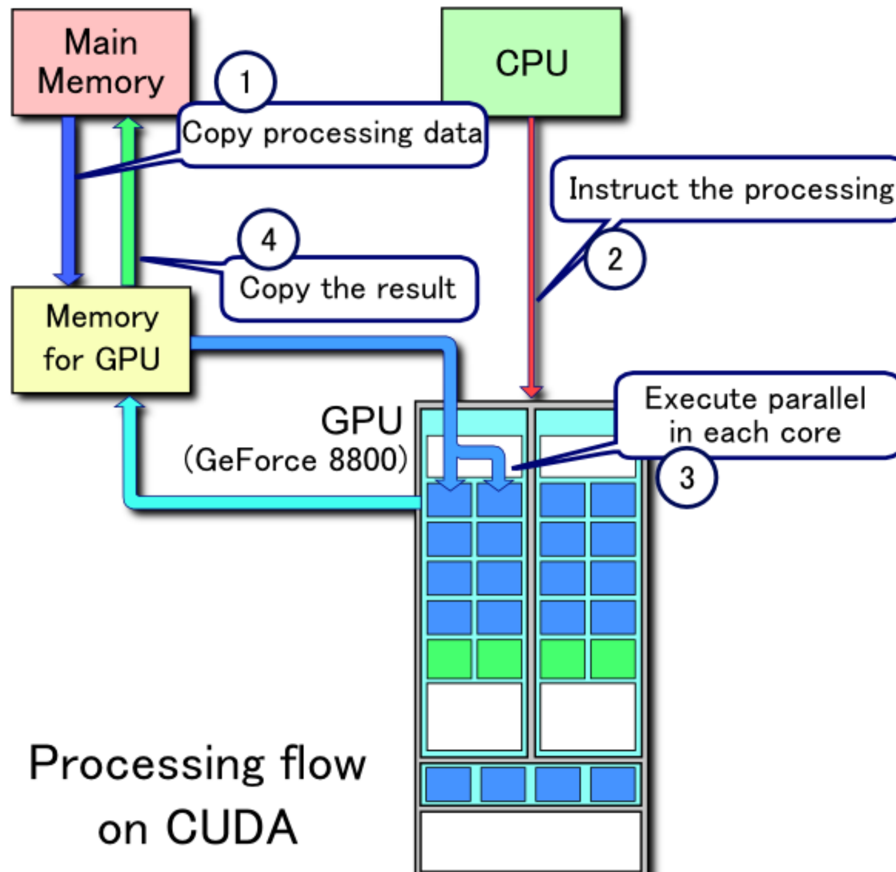
- Fast if your program involves:
  - Large data sets
  - Many parallel integer or floating-point operations
  - Minimal dependency between data elements (i.e. SIMD)
- Slower if your program involves:
  - Double precision floating-point
  - Logical operations on integer data
    - Lots of branches!
  - Random access / memory-intensive operations beyond the size of GPU memory

GPGPU Programming

- Challenge:
  - GPU architecture changes all the time!
    - # of independent threads, ALUs, memory size, etc…
  - How can we write one program that runs on many different GPU models?
- One solution from NVIDIA: CUDA
  - Compute Unified Device Architecture
  - Extension to the C programming language

CUDA Programming

- CUDA provides a mechanism to
  - Transfer data to from main memory to GPU
  - Initiate hundreds/ thousands of threads on the GPU for data-parallel parts of the algorithm
    - GPU needs many threads (thousands) in order to run efficiently!
  - Transfer results from GPU back to main memory



Processing flow on CUDA