UE18CS342
Heterogeneous Parallelism
Unit 2

Aronya Baksy

March 2021

1 Superscalar Architectures

- Superscalar CPU architectures implement ILP inside a single processor that allows for greater throughput at the same clock rate as a scalar pipelined processor.
-

1.1 Parallel Pipeline

- Multiple instruction pipelines that run instructions simultaneously
- Instructions are issued from a sequential stream. The CPU dynamically checks for dependencies between instructions at runtime.
- Multiple instructions can be executed in a single clock cycle, as against a single pipeline that has an upper throughput limit of 1 Instruction per Cycle.

1.2 Diversified Pipeline

- Within a single pipeline, maintain different functional units that can operate simultaneously
- Each FU is not a separate CPU core, rather an execution resouce within the CPU (such as an ALU, FPU, multiplier, barrel shifter etc.)
- Requires scheduling to maximize the utilization of the Functional Units, as well as to minimize any structural hazards

1.3 Dynamic Pipelines

- Making use of dispatch units and reorder buffers to allow for in-order issue, out of order execution and in-order completion
- This allows the processor to schedule instructions *around* stalls (i.e. If a stall occurs, the processor can schedule other instructions to be executed until the stall is resolved)

1.4 Approaches to Multithreading

1.4.1 Fine-Grained Multithreading

- Switches threads on every clock cycle. One instruction from each thread is executed before context switches to the next thread
- Threads are scheduled in a round-robin manner, skipping over any stalled threads
- **Advantage**: It can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- **Disadvantage**: Slows down the execution of threads that are waiting and can execute without any stalls

1.4.2 Coarse-Grained Multithreading

- Threads switched only on costly stalls, such as L2 cache misses
- **Advantages**: no need for fast thread-switching, high performance of current thread (as long as it doesn't stall for too long)
- **Disadvantages**: Short stalls cause high throughput loss due to the time needed to empty the current pipeline and refill it with instructions from the new thread

1.4.3 Simultaneous Multithreading

- In simultaneous multithreading, instructions from more than one thread can be executed in any given pipeline stage at a time.
- The main architecture changes needed are the ability to fetch instructions from multiple threads in a cycle, and a larger register file to hold data from multiple threads
- The independent state of each thread is duplicated. Thread state involves:
  1. Program Counter (PC)
  2. Register file, and mapping between physical and logical registers
  3. I and D caches
  4. Branch predictor
  5. Re-order Buffer
- Thread switching hardware allows for faster context switching than processes (100s or at most 1000s of clock cycles for threads)

1.4.4 Resource Utilization

- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Fine-grained multithreading can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot

2 Laws of Parallelism

2.1 Amdahl's Law

- "The speedup of a program using multiple processors in parallel is limited by the time needed for the sequential fraction of the program" - Gene Amdahl, 1967
- Mathematically,

$$S = \frac{1}{s + \frac{1-s}{P}} \tag{1}$$

  where S is the speedup obtained, s is the sequential (serial) fraction of the program (hence $1 - s$ is the parallelizable fraction) and P is the number of processing units available to work in parallel.
- As $P \to \infty$, we have $S \to \frac{1}{s}$ (the **maximum theoretical speedup**, which clearly depends only on the serial fraction s)

2.2 Generalized Amdahl's Law

- Let s and f be two parts of a program. We have

$$s + f = 1$$

- Now, the part f is sped up by F times. As a result of this, the speedup S is calculated as:

$$S = \frac{s + f}{s + \frac{f}{F}} \tag{2}$$

$$\implies S = \frac{1}{s + \frac{f}{F}} \tag{3}$$

$$\implies S = \frac{1}{(1 - f) + \frac{f}{F}} \tag{4}$$

2.2.1 Takeaway from Amdahl's Law

- Make the common case fast; reduce the serialized portion, rather than focus on adding more processing units
- Only for programs where parallel fraction is large, does adding more processors help
- Amdahl's law works only for a fixed problem size (does not scale with larger problems)

2.3 Gustafson's Law

- Scaling the problem size along with the increase of machine size within the same execution time
- The speedup gained from N processors is given as:

$$S = N + (1 - N)s \tag{5}$$

where S is the speedup, s is the serial fraction (i.e. the one that is not affected by increase in resources)

2.3.1 Takeaway from Gustafson Law

- Derived by fixing the parallel execution time (rather than the problem size that is fixed by Amdahl's Law)
- Amdahl's law turns out to be too conservative for high performance computing.
- For many real world applications, Gustafson's Law makes more sense (If you have more resources you can do more work, LOL)

3 Drawbacks of both laws

- Neither take into account the overhead of synchronization, communication, OS, etc., or the fact that loads may not be equally balanced between processors

4 Parallel Computing

4.1 Approaches to Parallel Computing

4.1.1 Extend language

- Add functionality for creating, terminating and synchronizing processes, as well as IPC mechanisms
- Pros: Easy, quick, least expensive, leverages existing compiler technology, fast delivery of software after new parallel hardware is released
- Cons: Lack of support for bug detection, easy to incorporate bugs that can be hard to catch

4.1.2 Extend compilers

- Compilers detect parallelism in sequential programs, and produce a parallelized executable
- Pros: leverage existing sequential code, no reskilling needed for programmers, easier to program sequentially than in parallel
- Cons: Performance of parallelizing compilers on broad range of applications still not very encouraging

4.2 Multithreaded Programming Model

- Programmer explicitly creates multiple threads, uses shared memory for load/store
- Pre-emptive multithreading (part of thread scheduling) is managed by the Operating System
- Use cases: GUI programs, handling I/O latencies, expressing parallelism using TLP

4.3 GPGPU Computing

- GPUs are specialized processors designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.
- GPUs are mainly optimized for 2D graphics (that involve bit-block transfers), and raster graphics (that are expressed as 2D rectangular arrays of pixels)
- GPUs are examples of throughput-oriented architecture:
  - Small caches for high memory throughput
  - Simple control logic with no forwarding and branch prediction
  - Large number of energy efficient ALUs that are heavily pipelined for high throughput
  - Require massive number of threads to have latencies in tolerable range
- Devote more transistors to data processing rather than flow control and caching
- GPUs have fine-grained and lightweight threads, with poor single-thread performance

4.3.1 CUDA

- Compute Unified Device Architecture, an Nvidia proprietary scalable parallel programming model and software environment for parallel programming
- CUDA is the most common platform for General Purpose computing using GPUs. It is exposed to programmers using extensions for C, C++ and Fortran languages.
- Easy integration, high efficiency with mathematical computations (especially matrix-vector computations). Can execute code with high levels of data parallelism, reuse and regularity
- Disadvantages: GPU harder to program (poor compiler support, architecture differences), architecture (especially poor single-thread performance) is a bottleneck for serious GP computing

4.4 Parallel Programming

- Most common approach is thread-level. Programmers launch threads (lightweight sub-units of a process) and each thread performs a sub-task
- The OS takes care of allocating threads to the relevant hardware, and thread scheduling
- Most thread-based programming uses the *fork/join model*, wherein the program starts with a main thread, then multiple threads are launched (for the parallel execution) and these threads are joined to the main thread once they complete their task
- Threads within a process share code and shared variables, as well as resources (like open files, signals). But each thread has its own PC, register set and stack.

- When many threads are simultaneously manipulating a single shared variable, there is a chance of inconsistency depending on the sequence in which the variable is manipulated. This is called a **race condition**
- Synchronization primitives (like mutexes and semaphores) are used to tell the compiler that a shared variable is to be protected from random parallel access.

4.5 False Sharing

- False Sharing occurs when threads on different processors modify variables that reside in the same cache block
- It is called false sharing because each thread is actually not sharing access to the same variable
- False sharing does not occur when processor word size and cache block size are equal. It does occur when cache block is larger than processor word size.
- If one core writes, the cache line holding the memory line is invalidated on other cores. Even though another core may not be using that data (reading or writing), it may be using another element of data on the same cache line. The second core will need to reload the line before it can access its own data again.
- The cache hardware ensures data coherency, but at a potentially high performance cost if false sharing is frequent.
- Solution is to force the variables into 2 different cache lines, using **padding**.