# Computer architecture

Parallelism - single and multi-core processors

# Types of parallelism

- Parallelism in hardware
  - single-core - pipelining, superscalar, VLIW
  - SIMD instructions, Vector processors, GPU (example: MMX, SSE)
  - multi-core - SMP, distributed-memory MP
  - multicomputer (clusters)
- Parallelism in software
  - Instruction-level parallelism
  - Task-level parallelism
  - Data parallelism
  - Transaction level parallelism

# Instruction-level parallelism

- Multiple instructions from the same instruction stream can be executed concurrently
- Generated and managed by hardware (**superscalar**) or by compiler (**VLIW**)
- Limited in practice by data and control dependences

# Thread-level or task-level parallelism

- Multiple threads or instruction sequences from the same application can be executed concurrently
- Generated by compiler/user and managed by compiler and hardware
- Limited in practice by communication/synchronization overheads and by algorithm characteristics

# Data-level parallelism

- Instructions from a single stream operate concurrently on several data
- Limited by non-regular data manipulation patterns and by memory bandwidth

# Transaction-level parallelism

- Multiple threads/processes from different transactions can be executed concurrently
- Limited by concurrency overheads

# Term recap

Other types of processors use the following techniques to improve performance:

- pipelining (break the instruction into subparts);
- superscalar processor (independently execute the instructions in different parts of the processor);
- out-of-order-execution (execute order is different to the program code);

Problem: each of these methods add to the complexity of the hardware

# Superpipelined and Superscalar Processors

In practice, it has proved better to produce superscalar processors, often with deep pipelines, rather than purely superpipelined processors:

- Practical limits to clock frequency
- Some operations or modules are difficult to pipeline.
- The need to balance logic in pipeline stages

But:

- It's difficult to schedule instructions for these (very complex!)

There are other approaches - use intrinsic parallelism in instruction stream, complexity, branching by resolving them in a higher instruction set architecture called **VLIW** (Very Long Instruction Word)

# VLIW

- uses ILP - it has programs to control parallel execution of the instructions
- specifically, it uses a compiler to resolve all of these issues
- subprograms decide the parallel flow of the instructions, and resolve conflicts
- this obviously increases **compiler** complexity, but it also reduces **hardware** complexity

# VLIW features

- The processors in this architecture have multiple functional units, fetch from the Instruction cache that have the Very Long Instruction Word.
- Multiple independent operations are grouped together in a single VLIW Instruction. They are initialized in the same clock cycle.
- Each operation is assigned an independent functional unit.
- All the functional units share a common register file.
- Instruction words are typically of the length 64-1024 bits depending on the number of execution unit and the code length required to control each unit.
- Instruction scheduling and parallel dispatch of the word is done statically by the compiler.
- The compiler checks for dependencies before scheduling parallel execution of the instructions.

# VLIW advantages

- Reduces hardware complexity.
- Reduces power consumption because of reduction of hardware complexity.
- Since compiler takes care of data dependency check, decoding, instruction issues, it becomes a lot simpler.
- Increases potential clock rate.
- Functional units are positioned corresponding to the instruction pocket by compiler.

# VLIW problems

- Complex compilers are required which are hard to design.
- Increased program code size.
- Larger memory bandwidth and register-file bandwidth.
- Unscheduled events, for example a cache miss could lead to a stall which will stall the entire processor.
- In case of un-filled opcodes in a VLIW, there is waste of memory space and instruction bandwidth.

# SIMD (Single-Instruction, Multiple-Data)

- A SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams
- Machines based on a SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations
- Information can be passed to all the processing elements (PEs) organized data elements of vectors can be divided into multiple sets(N-sets for N PE systems) and each PE can process one data set

# ARM SIMD instructions (NEON)

"Arm Neon technology is an advanced Single Instruction Multiple Data (SIMD) architecture extension for the A-profile and R-profile processors. Neon technology is a packed SIMD architecture. Neon registers are considered as vectors of elements of the same data type, with Neon instructions operating on multiple elements simultaneously. Multiple data types are supported by the technology, including floating-point and integer operations. Neon technology is intended to improve the multimedia user experience by accelerating audio and video encoding and decoding, user interface, 2D and 3D graphics, and gaming. Neon can also accelerate signal processing algorithms and functions to speed up applications such as audio and video processing, voice and facial recognition, computer vision, and deep learning."

Source: ARM Developer website, https://developer.arm.com/Architectures/Neon

# Intel SIMD instructions (Instruction Set Extensions)

- MultiMedia eXtension (1996) - MMX (Pentium MMX, Pentium II)
- Streaming SIMD Extensions - SSE (Pentium III), SSE2 (Pentium 4), SSE3 (Pentium 4 with HT), SSE4 (Intel Silvermont, 2013)
- Advanced Vector eXtensions - AVX (2008.), AVX2 (2013), AVX-512 (2013)

# Vector processors

"Although what constitutes a vector processor has changed over the years, their key characteristic is that they can operate on arrays or vectors of data, while conventional CPUs operate on individual data elements or scalars."

- Peter S. Pacheco, Matthew Malensek: An Introduction to Parallel Programming (Second Edition), 2022.

# Typical parts of vector processors

- Vector registers. These are registers capable of storing a vector of operands and operating simultaneously on their contents. The vector length is fixed by the system, and can range from 4 to 128 64-bit elements.
- Vectorized and pipelined functional units. Note that the same operation is applied to each element in the vector, or, in the case of operations like addition, the same operation is applied to each pair of corresponding elements in the two vectors. Thus, vector operations are SIMD.
- Interleaved memory. The memory system consists of multiple “banks” of memory, which can be accessed more or less independently. After accessing one bank, there will be a delay before it can be reaccessed, but a different bank can be accessed much sooner. So if the elements of a vector are distributed across multiple banks, there can be little to no delay in loading/storing successive elements.
- Strided memory access and hardware scatter/gather. In strided memory access, the program accesses elements of a vector located at fixed intervals. For example, accessing the first element, the fifth element, the ninth element, and so on, would be strided access with a stride of four. Scatter/gather (in this context) is writing (scatter) or reading (gather) elements of a vector located at irregular intervals— for example, accessing the first element, the second element, the fourth element, the eighth element, and so on. Typical vector systems provide special hardware to accelerate strided access and scatter/gather.

# Vector processors advantages

- for many applications, they are very fast and very easy to use
- vectorizing compilers are quite good at identifying code that can be vectorized
- compilers identify loops that cannot be vectorized, and they often provide information about why a loop couldn’t be vectorized - user can thereby make informed decisions about whether it’s possible to rewrite the loop so that it will vectorize.
- they have very high memory bandwidth, and every data item that’s loaded is actually used, unlike cache-based systems that may not make use of every item in a cache line

# Vector processors disadvantages

- they don’t handle irregular data structures as well as other parallel architectures
- there seems to be a very finite limit to their scalability, that is, their ability to handle ever larger problems
- it's difficult to create systems that would operate on ever longer vectors
- scalability is achieved by increasing the number of vector processors, not the vector length
- current commodity systems provide limited support for operations on very short vectors, while processors that operate on long vectors are custom manufactured, and, consequently, very expensive.

# GPUs

- Real-time graphics application programming interfaces, or APIs, use points, lines,and triangles to internally represent the surface of an object
- They use a graphics processing pipeline to convert the internal representation into an array of pixels that can be sent to a computer screen
- Several of the stages of this pipeline are programmable
- The behavior of the programmable stages is specified by functions called shader functions
- The shader functions are typically quite short—often just a few lines of C code
- They're also implicitly parallel, since they can be applied to multiple elements (e.g., vertices, points where two or more line segments meet) in the graphics stream

# GPUs

- Since the application of a shader function to nearby elements often results in the same flow of control, GPUs can optimize performance by using SIMD parallelism - all GPUs use it
- This is achieved by including a large number of ALUs (e.g., 80) on each GPU processing core

# GPU processing

- Processing a single image can require very large amounts of data—hundreds of megabytes of data for a single image is not unusual
- GPUs therefore need to maintain very high rates of data movement, and in order to avoid stalls on memory accesses, they rely heavily on hardware multithreading; some systems are capable of storing the state of more than a hundred suspended threads for each executing thread
- The actual number of threads depends on the amount of resources (e.g., registers) needed by the shader function
- Drawback: many threads processing a lot of data are needed to keep the ALUs busy, and GPUs may have relatively poor performance on small problems

# Other GPU characteristics

- GPUs are not pure SIMD systems
- Although the ALUs on a given core do use SIMD parallelism, current generation GPUs can have dozens of cores, which are capable of executing independent instruction streams
- GPUs are becoming increasingly popular for general, high-performance computing, and several languages have been developed that allow users to exploit their power either as a part of the system that includes a CPU (OpenCL), or solo-devices (CUDA, various Fortan, C/C++ compilers, ...)

# Speedup - Scalability

- Scalability: adding x times more resources to the machine yields close to x times better “performance”
  - Usually resources are processors (but can also be memory size or interconnect bandwidth)
  - Usually means that with x times more processors we can get ~x times speedup for the same problem
  - In other words: How does efficiency (see Lecture 1) hold as the number of processors increases?
- In reality we have different scalability models:
  - Problem constrained
  - Time constrained
- Most appropriate scalability model depends on the user interests

# Problem-constrained scaling

- Problem size is kept fixed
- Wall-clock execution time reduction is the goal
- Number of processors and memory size are increased
- “Speedup” is then defined as:

$$S_{PC} = \frac{\text{Time(1 processor)}}{\text{Time(p processors)}}$$

- Example: Weather simulation that does not complete in reasonable time

# Time-constrained (TC) scaling

- Maximum allowable execution time is kept fixed
- Problem size increase is the goal
- Number of processors and memory size are increased
- "Speedup" is then defined as:

$$S_{TC} = \frac{\text{Work(p processors)}}{\text{Work(1 processor)}}$$

- Example: weather simulation with refined grid

# Motivation

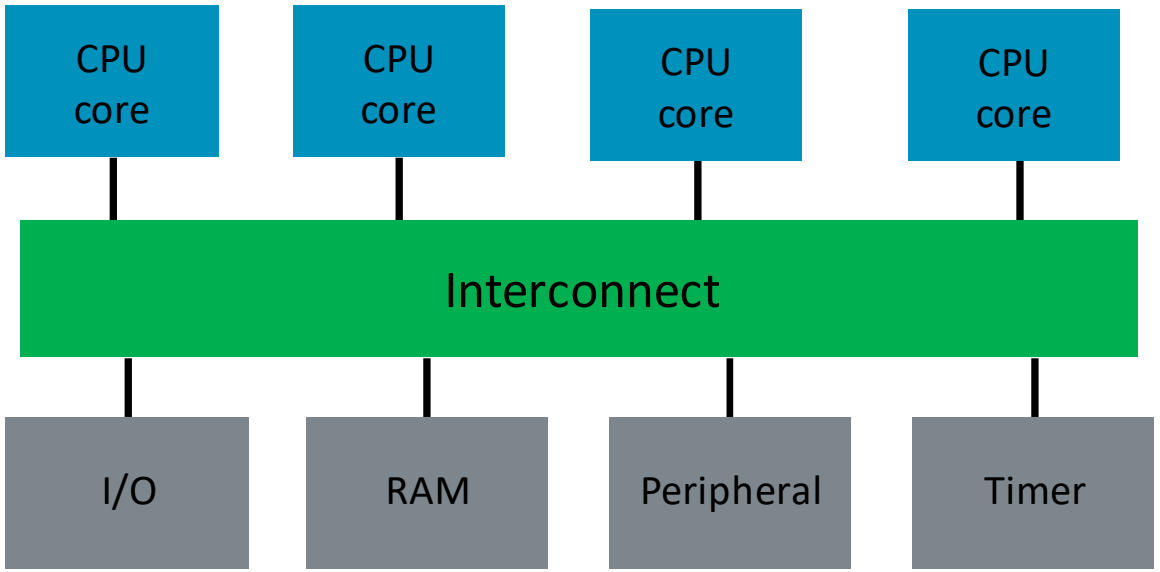
- Moore's law meant that the number of transistors available to an architect kept increasing.
  - Historically, these were used to improve the performance of a single core processor.
  - Usually through increased speculation support, but this gives sublinear performance improvement
- However, the breakdown of Dennard scaling meant these schemes were no longer viable.
  - If they consumed large amounts of power without giving commensurate performance improvements.
- Multicore architectures are an efficient way of using these transistors instead.
  - Performance comes from parallelism, specifically thread-level or process-level parallelism.
  - Note that we had multi-processor systems long before Dennard scaling failed; this just pushed them to mainstream.
- What are the challenges in providing multiple cores and how do they communicate?

# Multicore

Overview

- In a multicore processor, multiple CPU cores are provided within the chip.
- Cores are connected together through some form of interconnect.
- Cores share some components on chip.
  - For example, memory interface, or a cache level

An example of multicore system

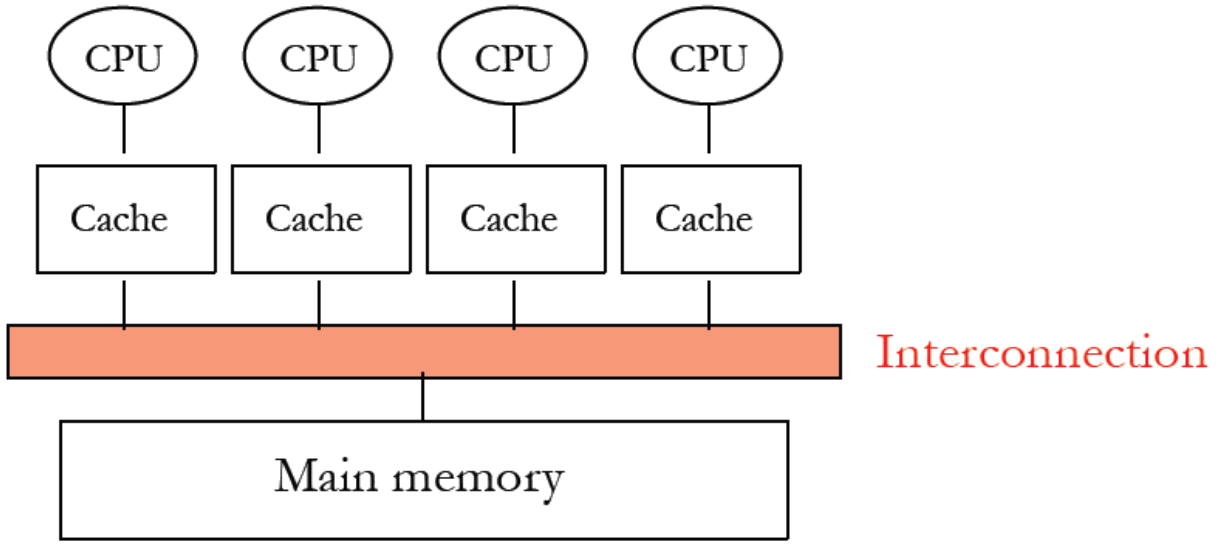


# Multicore

- Cores in a multicore processor are connected together and can collaborate.
- There are a number of challenges to consider when creating a system like this.
  - How do cores communicate with each other?
  - How is data synchronized?
  - How do we ensure that cores don't get stale data when it's been modified by other cores?
  - How do cores see the ordering of events coming from different cores?
- We'll explore each of these by considering the concepts of
  - Shared memory and message passing
  - Cache coherence
  - Memory consistency

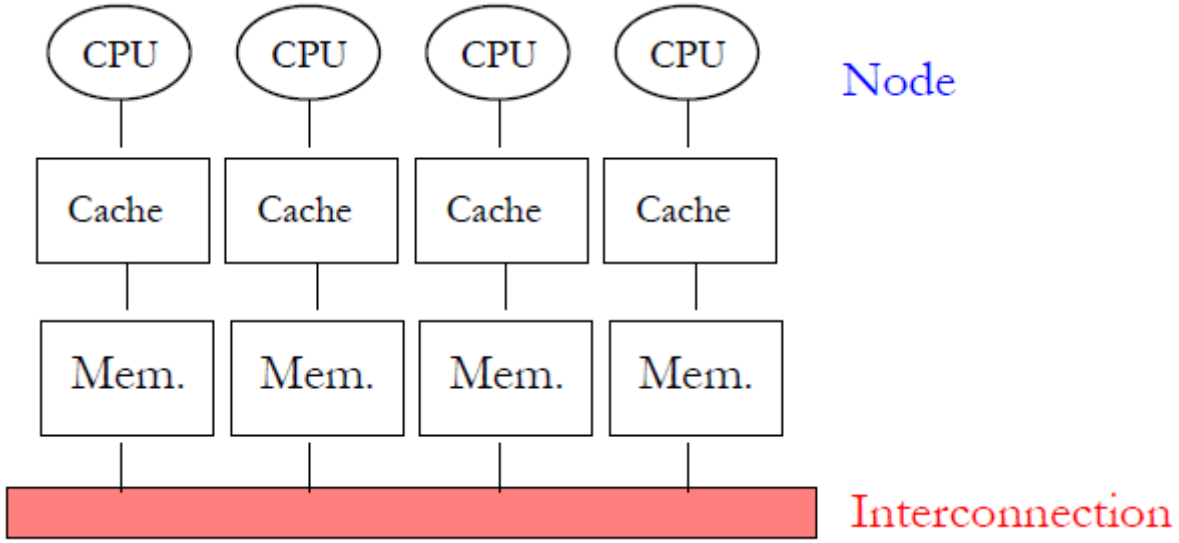
# UMA/SMP

- According to physical organization of processors and memory:
  - Physically centralized memory, uniform memory access (UMA)
  - All memory is allocated at same distance from all processors
  - Also called symmetric multiprocessors (SMP)
  - Memory bandwidth is fixed and must accommodate all processors → does not scale to large number of processors
  - Used today (single-socket systems)



# NUMA

- According to physical organization of processors and memory:
- Physically distributed memory, non-uniform memory access (NUMA)
  - A portion of memory is allocated with each processor (node)
  - Accessing local memory is much faster than remote memory
  - If most accesses are to local memory than overall memory bandwidth increases linearly with the number of processors
  - Used in multi-socket CMP (Chip Multi-Processing) like Intel Nehalem and later



# Thank you for your attention!