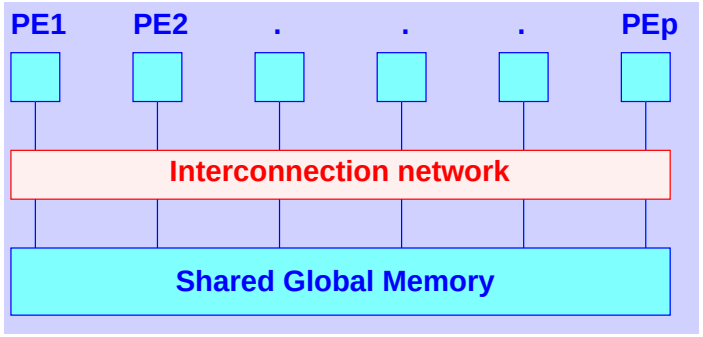
MEMORY ORGANIZATION

- **Memory and cache performance issues**
- **Hierarchical memories**
- **Latency, bandwidth, ..**
- **Caches**
- **Examples**

Terminology: Symmetric Multiprocessing (SMP)

Main feature: a global memory that is accessed by p processors through some interconnection network or high-speed bus

➤ Same view, same addressing, of memory by all PEs



Terminology: UMA / NUMA

UMA : Uniform Memory Access – any memory word can be accessed in the same amount of time - regardless of location.

NUMA : Non-Uniform Memory Access – time to access a certain memory word depends on location.

Often NUMA architectures correspond to distributed memory computers with an OS that allows a global address space.

➤ This idea adopted in microprocessor chip: "cluster in a box". [see example later.]

Memory and Cache performance issues

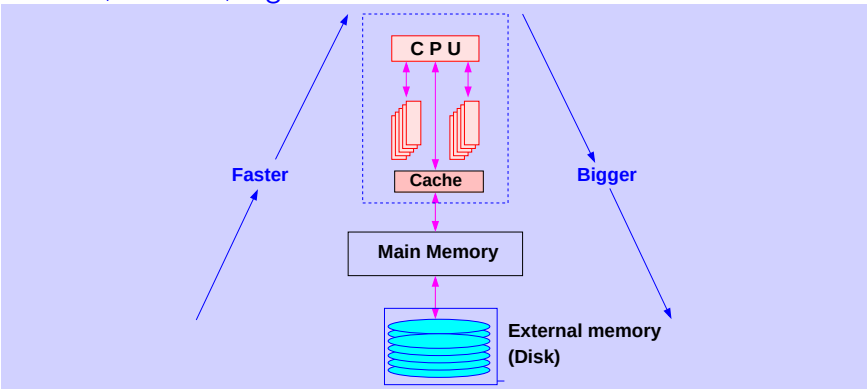
➤ Main problem: memory speed much smaller than CPU speed.

➤ Fast memory is expensive

➤ Idea: organize memory hierarchically to exploit "locality of data"

Cache: Small fast memory with very fast access from/to CPU.

Memory is hierarchical: Remote / External memory (Disk), RAM, Chache 1, Cache 2, registers



➤ Data is first searched in cache. If not available (cache miss) it is moved from main memory to cache.

➤ However, a whole "line" is moved from memory in anticipation for the use of nearby data items..

➤ Memory is often organized in banks (or "modules")

➤ Low order interleaving:

Banks	1	2	3	4	5	6	7	8
	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39
	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.

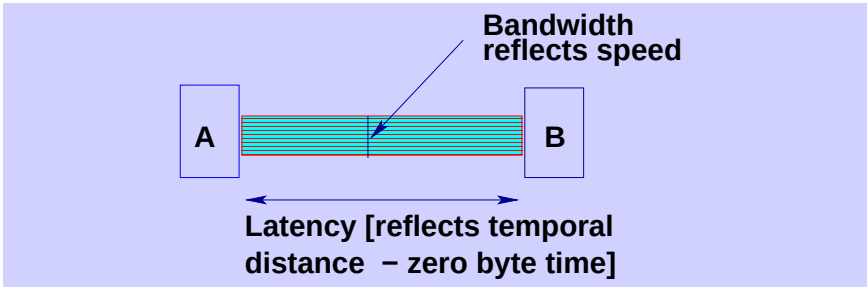
➤ Access to items from different banks is simultaneous but access to same bank causes a "memory-bank conflict"

➤ "High-order inverleaving": use the high-order bits as the module address ➤ Contiguous memory locations assigned to the same bank.

➤ Vector computers often use "low-order interleaving"./

Latency and Bandwidth

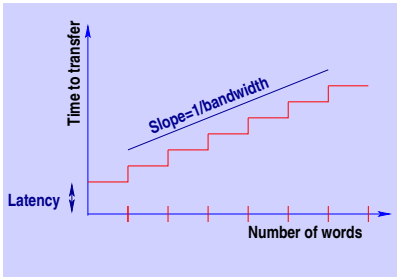
➤ Access to and from memory works much like other transportation systems:



Bandwidth: Max. number of bytes that can be moved per second.

Latency: Time to access a word from any given bank

➤ Typical curve to tansfer n words from memory to CPU (no cache)



➤ Stair-like shape because words are moved as single units.

➤ Roughly speaking : Time to transfer n words from memory to CPU or to cache [when there is one] takes the form

$$t(n) = \eta + \frac{n-1}{\beta}$$

where η is the latency, and β = Bandwidth.

Examples

✍1 Assume a memory has 16 banks (organized in words of 4 bytes) and that each bank cycle time is 8, i.e., it takes 8 clock cycles to access one 4-byte word from the bank. What is the Maximum memory bandwidth if the clock cycle is 6 nanoseconds?

✍2 In the same example as above, assume you need to access a vector with a stride of 8 (low-order interleaving). What is the actual bandwidth achieved?

Example: A CPU operates at 1GHz. Latency to DRAM = 100ns (no caches - Assume: can access a block of one word each 100ns) Proc. has 2 *, + units. Executes 4 OPS per cycle. ➤ CPU Peak Speed 4 GFlop/s. But doing an inner product of 2 vectors would give a speed of only 10 MFlop/s.

Cache

➤ Fast, low latency memories (usually small) between processor and main memory.

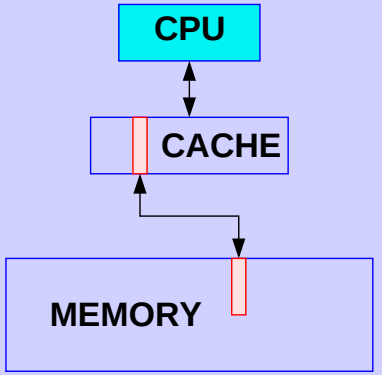
➤ Main principles exploited:

(1) Data reuse: References are often made to the same memory location many times, and

(2)Temporal Locality: when accessing a given entry in memory it is likely that entries next to it will be references soon after

➤ Minimum amount of information copied into cache is a "line". A line is copied from mem. to cache – with various addressing schemes.

Two-level memory hierarchy. Shaded area = a line copied to cache from main memory



Memory and Cache performance

- The performance can be degraded dramatically due to

1. Poor cache utilization (frequent cache misses)
2. Memory bank conflicts

Example: Consider the following code segment:

```
1. for (j = 0; j < n; j++) {
2.     c_sum[j] = 0.0;
3.     for (i = 0; i < m; i++)
4.         c_sum[j] += b[i][j];
5. }
```

- Computes the column sums of a certain matrix b
- Recall: In C, matrices are laid row-wise in memory.

- Stored as a one-dimensional array :

$$b[0][0], b[0][1], \cdots, b[0][n-1],$$
$$b[1][0], b[1][1], \cdots, b[1][n-1], \ldots$$
$$\cdots, b[m-1][0], b[m-1][1], \cdots, b[m-1][n-1]$$

Problem 1: i loop accesses entries in columns. Corresponds to accessing every n-th entry in the above one-dimensional array.

- If for example $n = 1024$ and the cache line is 16 words then: cache miss at each instance of the loop.

Problem 2: Strided access to memory can lead to bank conflicts. For example if, again, $n = 1024$ and if the memory is organized in 32 banks (low-order interleaving), then entries 0, 1024, 2048,.. cannot be fetched simultanously.

- Rewrite by swapping the i and j to

Rewrite as

```
1. for (j = 0; j < n; j++)
2.     c_sum[j] = 0.0;
3. for (i = 0; i < m; i++) {
4.     for (j = 0; j < n; j++)
5.         c_sum[j] += b[i][j];
6.     }
```

- An important ingredient in optimizing code relies on loop reordering and loop "unrolling" [especially on vector computers.]

- Use of loop unrolling:

```
0.    t = 0.0;
1.    for (j = 0; j < n; j++)
2.        t+ = x[j];
```

becomes (for example)

```
.    t = 0.0;
.    for (j = 0; j < n; j+ = 4)
.        t+ = x[j] + x[j + 1] + x[j + 2] + x[j + 3];
```

- Seldom needed these days [compilers do the job]

How do caches work?

➤ Three designs: (1) Fully associative caches (expensive) (2) Direct mapping caches (simplest, cheapest) and (3) set associative caches (combine direct mapping with associative memories)

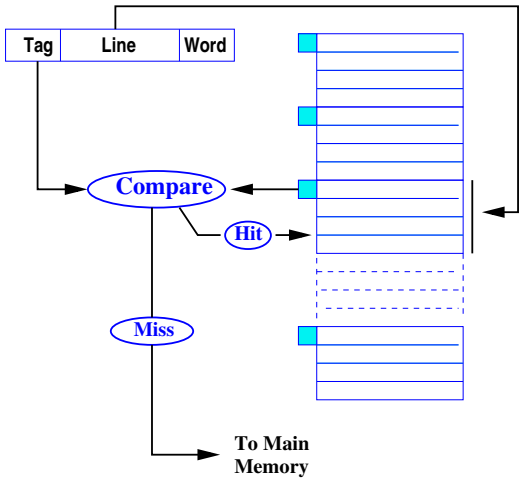
Idea of direct mapping:

➤ Address split in 3 pieces: (1) tag, (2) block (or line), and (3) word.

➤ Block is located in address 'BLOCK' of cache (2^w words)

➤ OFFSET = address of the word within the line [Also known as WORD]

➤ It suffices to compare the tags of the block and address being searched to determine if it is in cache. If it is not (cache miss) then copy the line from memory (slow).

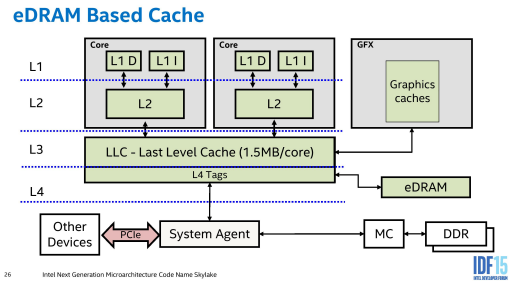


Caches of different levels

➤ Processors may have several types of cache: level 1, level 2, ...

➤ Up to level 4: Present in Intel microarchitectures of Haswell and Broadwell. [roughly from 2013 – 2016] – Disappeared with Skylake which appeared in 2015.

source: Ian Cutress in Anandtech, 09/01/2015

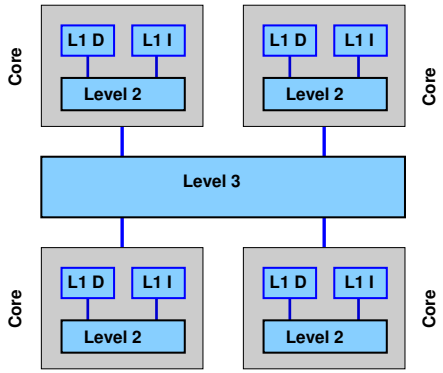


➤ More common: 3 'Levels', L1, L2, and L3.

➤ L1 closer to CPU ⟶ faster

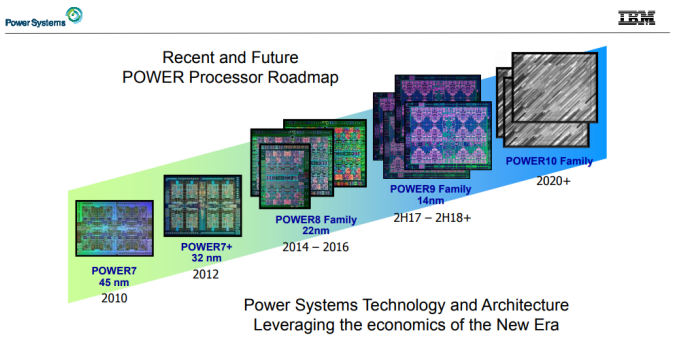
➤ The higher the level the larger the cache: e.g., 32 KB to 1MB for L1. L2: up to ≈ 8MB, and L3: up to ≈ 128MB.

➤ L1 is often divided in Instruction cache and Data cache



Example: The IBM Power9

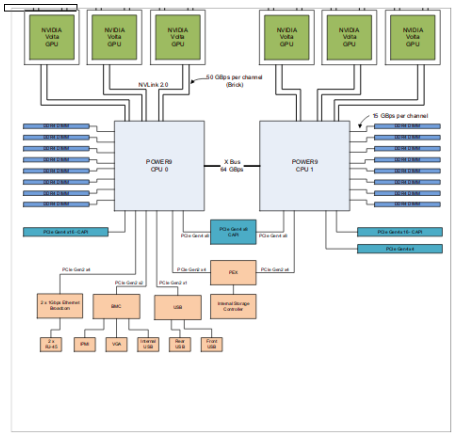
➤ IBM roadmap for High-Performance Chips [source: IBM]



- Starting with Power-9 IBM aims at Data-related applications.

- IBM Power system AC 922 that powers the Summit machine in Oak-Ridge. Each node: $2\times$ POWER9 SMT4 Monza, with $6\times$ Nvidia Volta GPUs

source: www.redbooks.ibm.com



- The two POWER9 processors together have 44 cores and 6 attached NVIDIA Tesla V100.
- L1 Data cache = 32KB [128B/line], 8 way - 8 Banks L2 cache can supply 64B/ clock cycle. L1 Instr. Cache: 32KB [2x64B sectors]
- 512 KB of L2 cache per core,
- Up to 120 MB of L3 cache per chip
- Optional: Level-4 cache via Centaur chip
- Up to four threads per core
- 120 GBps memory bandwidth per chip
- 64 GBps SMP interconnect between POWER9 chips
- Clock rate: 4GHz

➤ IBM is now rolling-out the new Power 10. Doubled up functional units, 2MB L2 cache, 18 B. Transistors (7nm technology!), ...

See read more on this site

DDR4 memory:

- Sixteen dual inline memory module (DIMM) memory slots
- Maximum of 1024 GB DDR4 system memory (8335-GTG)
- Improved clock from 1333 MHz to 2666 MHz for reduced latency

- Lots of Details in

https://www.redbooks.ibm.com/redpapers/pdfs/redp5472.pdf

and [for a slightly different Power9 chip]

https://www.7-cpu.com/cpu/Power9.html

Example: The 'phixx' cluster in cselabs

✍D3 To do in class: Look at the 'phixx' cluster – Analyze one node: "phi01.cselabs.umn.edu" – Number of sockets, cores, CPUs, addressable CPU ('NUMA nodes'), cache sizes, etc.

➤ Before class: explore the unix command *lscpu* and try to decipher the file /proc/cpuinfo

✍D4 Remember *Mangi* and its *Rome* nodes? visit this site