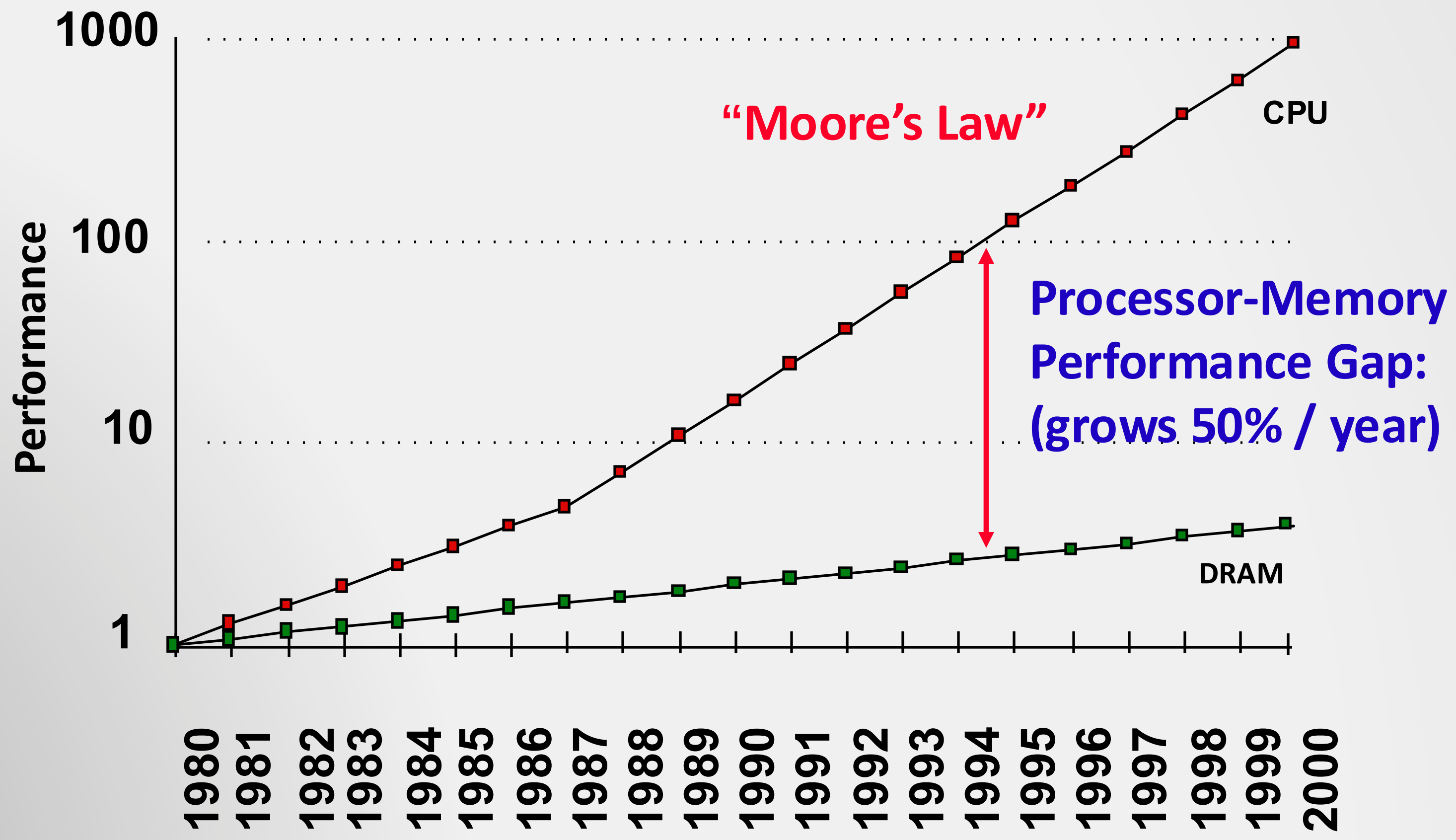
Performance Engineering on CPUs and GPUs:

- CPU and Memory: Things to be Careful for Performance -

Kamer Kaya, Sabancı University

ncc@ulakbim.gov.tr

Computer: Memory



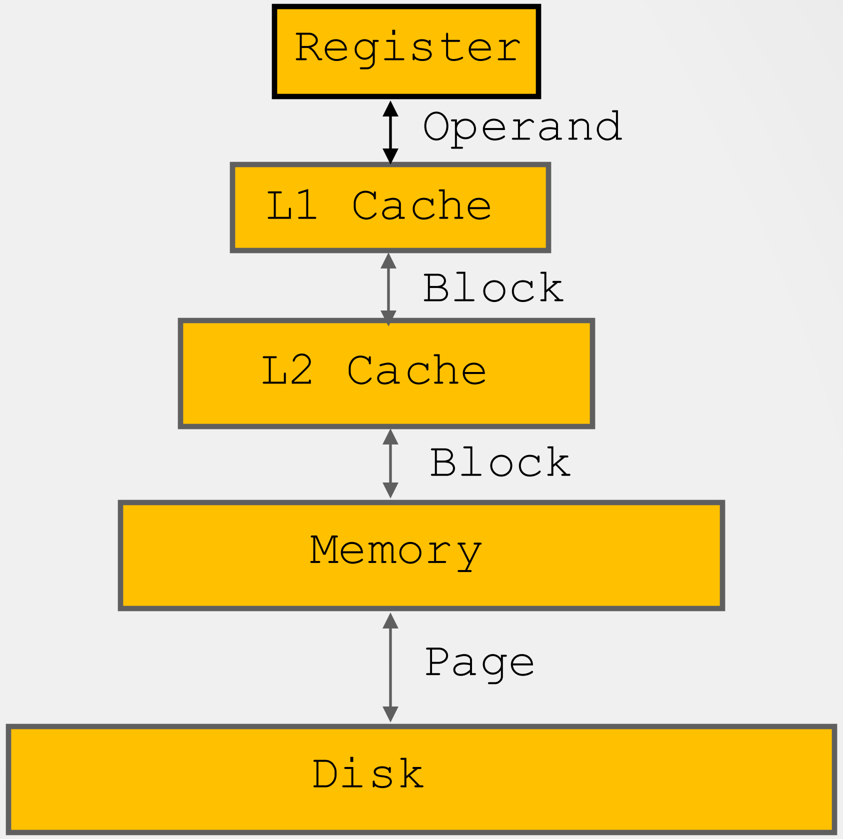
Computer: Memory

CPU registers
0.3-0.5ns

L1 and L2 cache
10ns-20ns
$1000s/GByte

G Bytes
80ns-200ns
~$100/GByte

T Byte, 10 ms
(10,000,000ns)
~$1/GByte



compiler
1-8 byte

Cache manager
32-64 byte

Cache manager
64-128 byte

OS
4K-8K byte

Computer: Memory

- **Memory system, and not processor speed, is often the bottleneck for many applications.**
- Memory system performance is largely captured by two parameters, **latency** and **bandwidth**.
  - **Latency** is the time from the issue of a memory request to the time the data is available at the processor.
  - **Bandwidth** is the rate at which data can be pumped to the processor by the memory system.

Computer: Memory

- **It is very important to understand the difference between latency and bandwidth.**
- Consider the example of a fire-hose (or an assembly line). If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.
- Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.
- If you want **immediate response** from the hydrant, it is important to reduce **latency**.
- If you want to **fight big fires**, you want high **bandwidth**.

Computer: Memory

- Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and hence is capable of executing four instructions (in total) in each cycle of 1 ns.
- The following observations follow:
  - The peak processor rating is
    - **4 GFLOPS.**
  - Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made (i.e., for each word), the processor must wait each data for
    - **100 cycles**

Computer: Memory

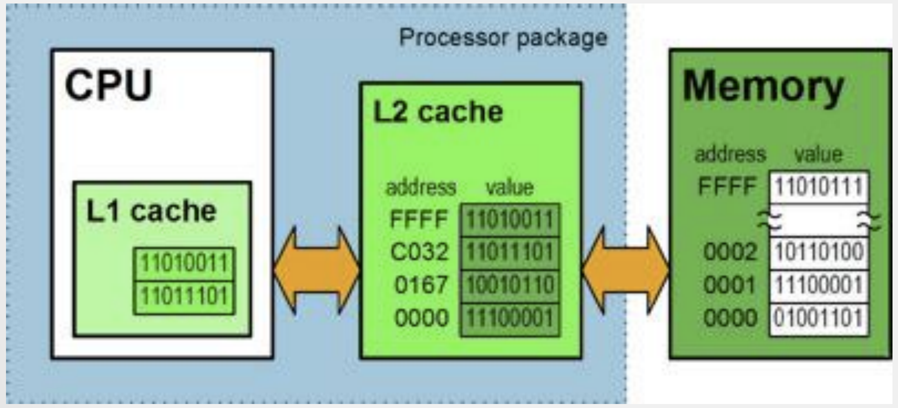
- On the above architecture, consider the problem of computing a dot-product of two vectors.
  - A dot-product computation performs one multiply-add (2 flops) on a single pair of vector elements, i.e., each floating-point operation requires
    - **one**

    data fetch.
  - It follows that the peak speed of this computation is limited to one floating point operation every
    - **100 ns, or a speed of**
    - **10 MFLOPS**

    which is a very small fraction of the peak processor rating!

Computer: Memory

- Caches are small and fast memory elements between the processor and DRAM.
- This memory acts as a low-latency high-bandwidth storage.
- If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.



- Can we improve the dot-product example with a cache?

Memory: Caches

- The fraction of data references satisfied by the cache is called the **cache-hit ratio** of the computation on the system.
- Cache-hit ratio achieved by a code on a memory system often determines its performance.

Memory: Caches

Consider the architecture from the previous example. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle. We use this setup to multiply two matrices A and B of dimensions 32 × 32.

So, the cache is large enough to store matrices A and B, as well as the result matrix C.

What is the peak performance **without the cache**?

10 MFlops

Memory: Caches

These observations are made (100ns/word, 1 ns/cycle, 4 instructions/cycle):

- Fetching the two matrices into the cache takes approximately
  - **200 µs.**
- Multiplying two $n \times n$ matrices takes $2n^3$ operations which can be performed in
  - **16K cycles (or 16 µs) at four instructions per cycle.**
- The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e.,
  - **200 + 16 µs.**
- This corresponds to a peak computation rate of
  - **64K FLOP/216 µs or 303 MFLOPS**.

Caches: Temporal locality

- **Data reuse is critical for cache performance.**
  - In our example, we had *$O(n^2)$* data accesses and *$O(n^3)$* computation. This asymptotic difference makes the above example particularly desirable for caches.
- Repeated references to the same data item is related to **temporal locality**.
  - Temporal locality in the example does not have an impact since the cache can store everything.

Computer: Memory

- **Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.**
- Memory bandwidth can be improved by increasing the size of memory blocks.
- The underlying system takes
  - *l* time units (where *l* is the latency of the system) to deliver
  - *b* units of data (where *b* is the block size).

Computer: Memory

- Consider the same setup as before (no cache), except in this case, the block size is 4 words instead of 1 word. We repeat the dot-product computation in this scenario:
  - Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in
    - **200 cycles**.
  - This is because a single memory access fetches four consecutive words in the vector.
  - This corresponds to a FLOP every
    - **25 ns, for a peak speed of**
    - **40 MFLOPS.**

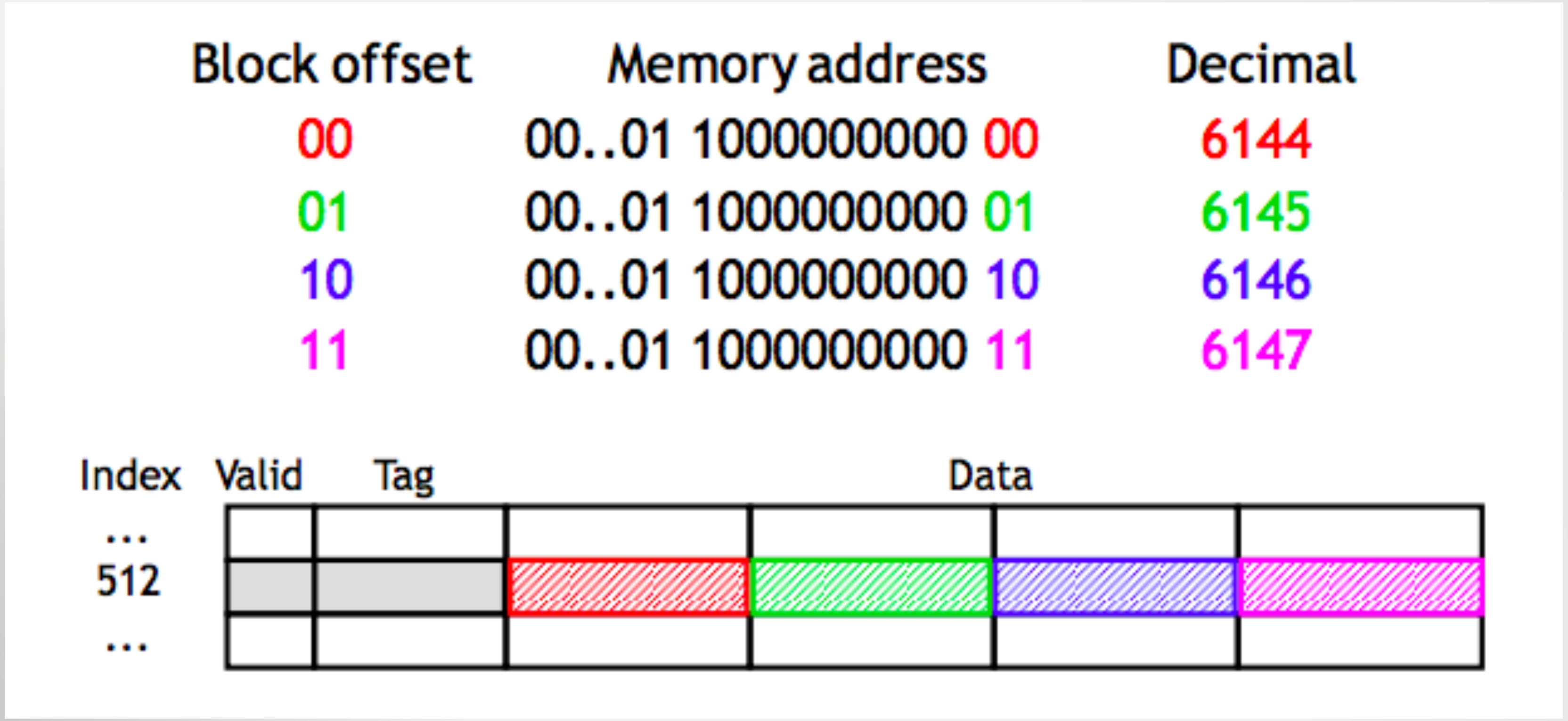
Computer: Memory

- **It is important to note that increasing block size does not change latency of the system**.
- Physically, the scenario illustrated here can be viewed as a wide data bus (4 words or 128/256 bits) connected to multiple memory banks.
- In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.

Computer: Memory

- The example clearly illustrated how increased bandwidth results in higher peak computation rates.
- The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions (**spatial locality of reference**).
- If we take a **data-layout centric view**, computations must be reordered to enhance spatial locality of reference.

Memory: Caches



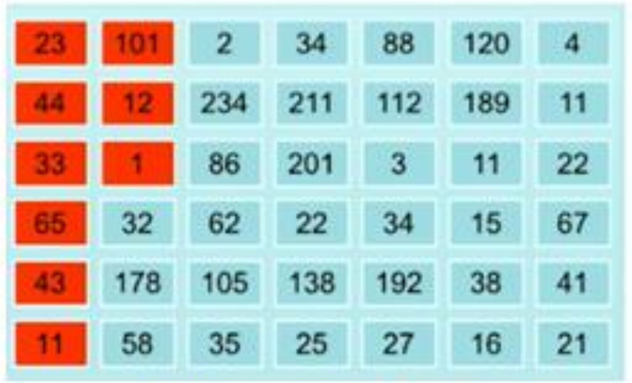
Block offset	Memory address	Decimal
00	00..01 1000000000 00	6144
01	00..01 1000000000 01	6145
10	00..01 1000000000 10	6146
11	00..01 1000000000 11	6147

Caches: Spatial Locality

Consider the following code fragment:

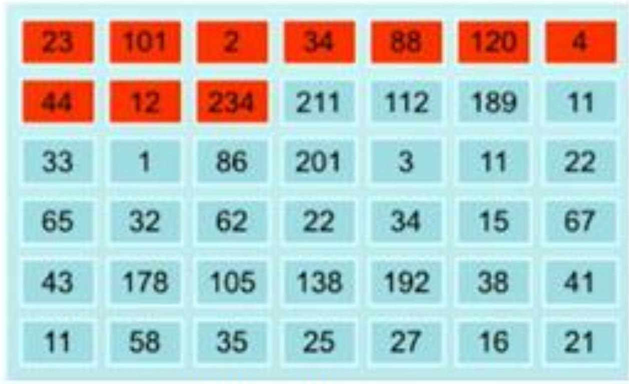
```
for (i = 0; i < 1000; i++)
    for (j = 0; j < 1000; j++)
        column_sum[i] += b[j][i];
```

The code fragment sums columns of the matrix b into a vector `column_sum`.



23	**101**	2	34	88	120	4
44	**12**	234	211	112	189	11
33	**1**	86	201	3	11	22
65	32	62	22	34	15	67
43	178	105	138	192	38	41
11	58	35	25	27	16	21

Column major access



23	**101**	**2**	**34**	**88**	**120**	**4**
44	**12**	**234**	211	112	189	11
33	1	86	201	3	11	22
65	32	62	22	34	15	67
43	178	105	138	192	38	41
11	58	35	25	27	16	21

Row major access

Caches: Spatial Locality

Let’s see an example

Thanks

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101101903. The JU receives support from the Digital Europe Programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Türkiye, Republic of North Macedonia, Iceland, Montenegro, Serbia