Principle of Locality:

- Programs tend to reuse data and instructions near those used recently, or that were recently referenced.

- *Temporal locality:* Recently referenced items are likely to be referenced in the near future.

- *Spatial locality:* Items with nearby addresses tend to be referenced close together in time.

```c
sum = 0;
for ( i = 0; i < n; i++ )
    sum += a[i];
return sum;
```
Data:
- Reference array elements in succession (stride-1): Spatial
- Reference sum each iteration: Temporal

Instructions:
- Reference instructions in sequence: Spatial
- Cycle through loop repeatedly: Temporal
Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

```c
int sumarrayrows1( int a[M][N] )
{
    int i, j, sum = 0;

    for ( i = 0; i < M; i++ )
        for ( j = 0; j < N; j++ )
            sum += a[i][j];

    return sum;
}
```

Does this function have good locality?
Does this compute the same function as sumarrayrows1? Does this function have good locality? How does it compare to the previous version?
A *stride-1 reference pattern* means that successive references are 1 “unit” apart. Here unit means the size of the data type, not necessarily one byte.

Can you permute the loops so that this function scans the 3-d array `a` with a stride-1 reference pattern (and thus has good spatial locality)?

```c
int sumarray3d( int a[N][N][N] )
{
    int i, j, k, sum = 0;

    for ( i = 0; i < N; i++ )
        for ( j = 0; j < N; j++ )
            for ( k = 0; k < N; k++ )
                sum += a[k][i][j];

    return sum;
}
```
Remember the CPU-Memory Gap

CPU speed increases *faster* than memory speed, meaning that:
- memory is more and more a limiting factor on performance;
- increased importance for caching and similar techniques.
Some fundamental and enduring properties of hardware and software:

- Fast storage technologies typically cost more per byte and have less capacity than slower ones.
- The gap between CPU and main memory speed is widening.
- Well-written programs tend to exhibit good locality.
- Memory systems access “blocks” of data, not individual bytes.

These fundamental properties complement each other beautifully.

They suggest an approach for organizing memory and storage systems known as a memory hierarchy.
Example Memory Hierarchy

- **L0**: Registers
  - CPU registers hold words retrieved from L1 cache

- **L1**: L1 cache (SRAM)
  - L1 cache holds cache lines retrieved from L2 cache

- **L2**: L2 cache (SRAM)
  - L2 cache holds cache lines retrieved from main memory

- **L3**: Main memory (DRAM)
  - Main memory holds disk blocks retrieved from local disks

- **L4**: Local secondary storage (local disks)
  - Local disks hold files retrieved from disks on remote network servers

- **L5**: Remote secondary storage (tapes, distributed file systems, Web servers)

---

Smaller, faster, costlier per byte

Larger, slower, cheaper per byte
**Cache**: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

The fundamental idea of a memory hierarchy: For each $k$, the faster, smaller device at level $k$ serves as a cache for the larger, slower device at level $k+1$. 

![Diagram of CPU, L1 Cache, and L2 Cache](image)
### Examples of Caching in the Hierarchy

<table>
<thead>
<tr>
<th>Cache type</th>
<th>What</th>
<th>Where</th>
<th>Latency (cycles)</th>
<th>Managed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>8-byte word</td>
<td>CPU registers</td>
<td>0</td>
<td>compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>address translations</td>
<td>On-chip TLB</td>
<td>0</td>
<td>hardware</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32-byte block</td>
<td>On-chip L1</td>
<td>1</td>
<td>hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>32-byte block</td>
<td>On-chip L2</td>
<td>10</td>
<td>hardware</td>
</tr>
<tr>
<td>Virtual Memory</td>
<td>4KB page</td>
<td>main memory</td>
<td>100</td>
<td>hw + OS</td>
</tr>
<tr>
<td>Buffer cache</td>
<td>parts of files</td>
<td>main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>Network buffer cache</td>
<td>parts of files</td>
<td>local disk</td>
<td>10M</td>
<td>AFS/NFS client</td>
</tr>
<tr>
<td>Browser cache</td>
<td>web pages</td>
<td>local disk</td>
<td>10M</td>
<td>web browser</td>
</tr>
<tr>
<td>Web cache</td>
<td>web pages</td>
<td>remote server disks</td>
<td>1000M</td>
<td>web proxy server</td>
</tr>
</tbody>
</table>
Why do memory hierarchies work?

- Programs tend to access the data at level $k$ more often than they access the data at level $k+1$.
- Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.
- *Net effect:* A large pool of memory that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.
- We use a combination of small fast memory and big slow memory to give the illusion of big fast memory.
Smaller, faster, more expensive device at level $k$ caches a subset of the blocks from level $k+1$.

Data is copied between levels in block-sized transfer units.

Larger, slower, cheaper storage device at level $k+1$ is partitioned into blocks.
Program needs object d, stored in some block b.

*Cache hit*: program finds b in the level k cache, e.g., block 14.

*Cache miss*: b is not at level k, so must fetch it from level k+1, e.g., block 12.

- If level k cache is full, then some current block (the victim) must be replaced (evicted).
- *Placement policy*: where can the new block go? E.g., b mod 4.
- *Replacement policy*: Which block should be evicted? E.g., LRU.
Types of cache misses:

*Cold (compulsary) miss*: the cache is empty.

*Conflict miss*: all available positions at level $k$ are occupied.
- Most caches limit blocks at level $k+1$ to a small subset (sometimes only one) of the block positions at level $k$.
- E.g., Block $i$ at level $k+1$ must be placed in block $(i \mod 4)$ at level $k$.
- Conflict misses occur when multiple data objects all map to the same level $k$ block. Note: there still may be empty slots in the cache.
- E.g., Referencing blocks $0, 8, 0, 8, 0, 8, \ldots$ would miss every time.

*Capacity miss*: the set of active cache blocks (working set) is larger than the cache.
L1 and L2 cache memories are small, fast SRAM-based memories managed automatically in hardware. They hold frequently accessed blocks of main memory.

CPU looks first for data in L1, then in L2, then in main memory.

The typical bus structure is shown below.
The tiny, very fast CPU register file has room for a small number of 8-byte words.

The transfer unit between register file and cache is an 8-byte block.

The small, fast L1 cache has room for k lines (each containing several words).

The transfer unit between cache and main memory is a block of bytes.

The big, slow main memory has room for many blocks.
Options: separate data and instruction caches, or a unified cache. Having separate data and instruction memories characterizes the Harvard architecture.
### Getting Info on Your CPU

```
oscar:~/cs429/c> lscpu
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 8
On-line CPU(s) list: 0–7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1
NUMA node(s): 1
Model name: Intel(R) Xeon(R) CPU E3-1270 v3 @ 3.50 GHz
Stepping: 3
CPU MHz: 3574.375
CPU max MHz: 3900.0000
CPU min MHz: 800.0000
BogoMIPS: 6984.06
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 8192K
NUMA node0 CPU(s): 0–7
```
This slideset:
- Locality: Spatial and Temporal
- Cache principles
- Multi-level cache hierarchies

Next time:
- Cache organization
- Replacement and writes
- Programming considerations