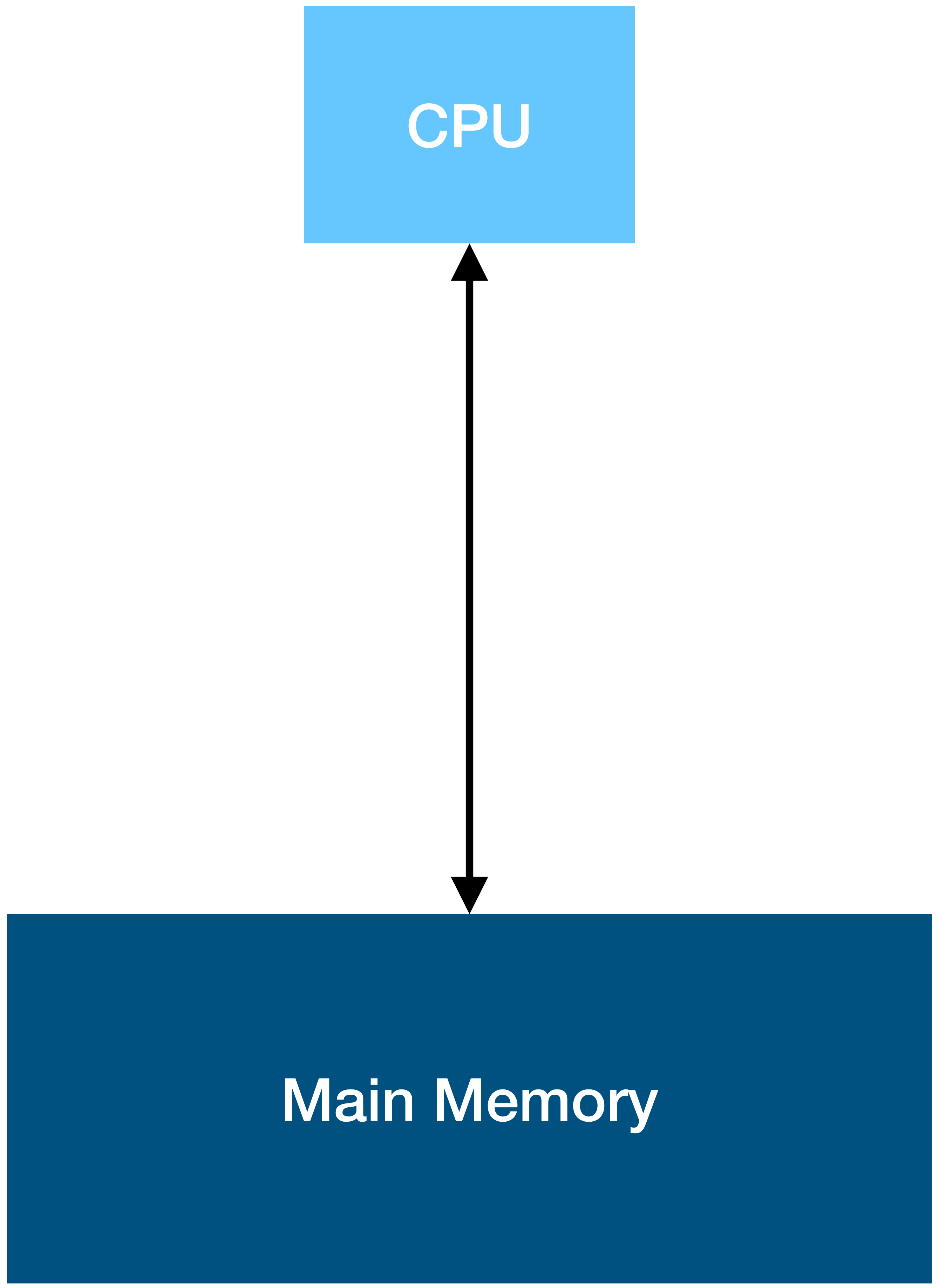
# High-Level Loop Optimizations

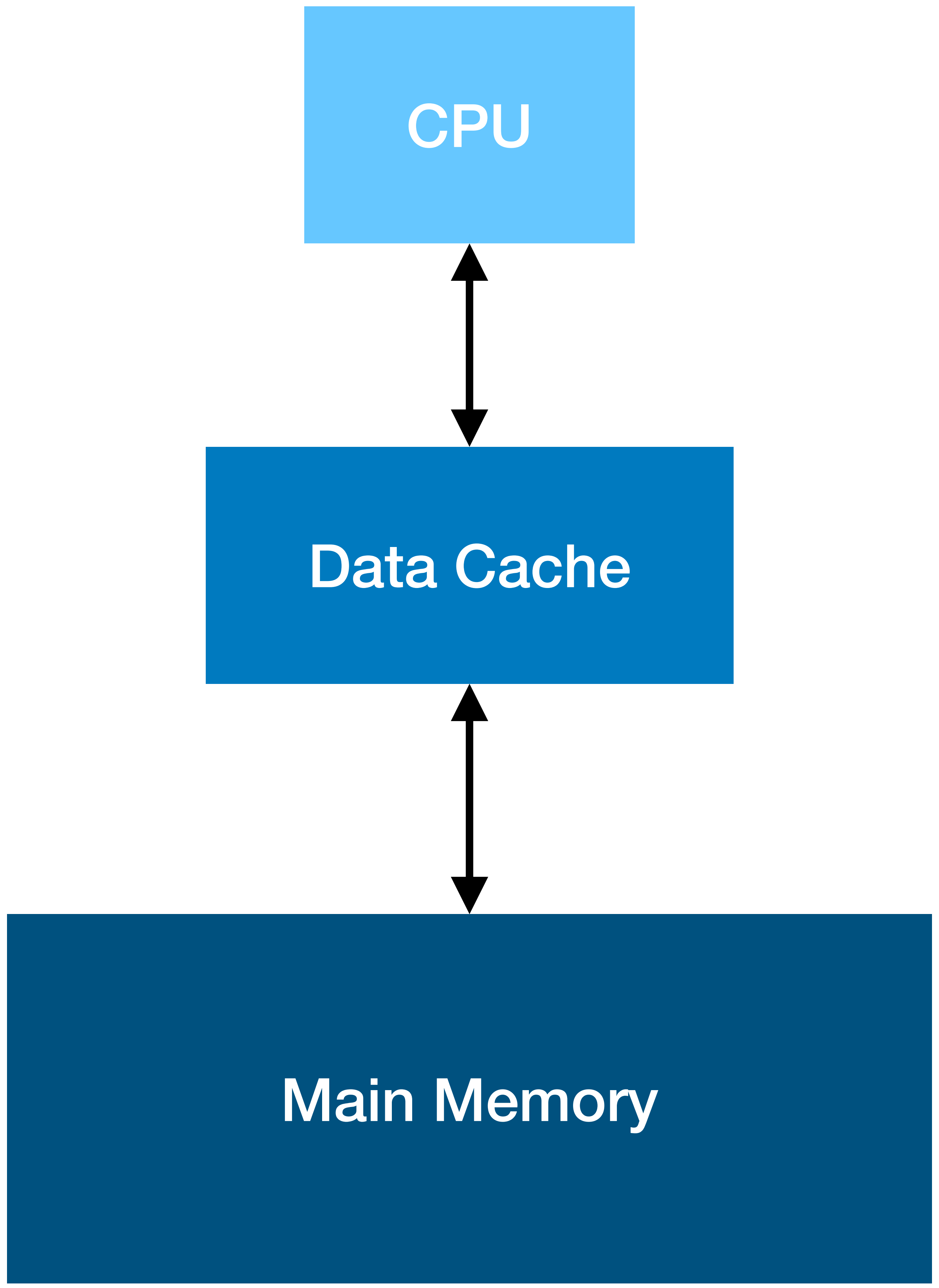
# Caches

- Modern machines have very large main memories
    - Making large, inexpensive memory means access is quite slow (hundreds of cycles to perform a load)
    - Fast memory is both small and expensive
- But programs perform *lots* of loads and stores
- Idea: add small, fast memory to hold some of your data → a cache



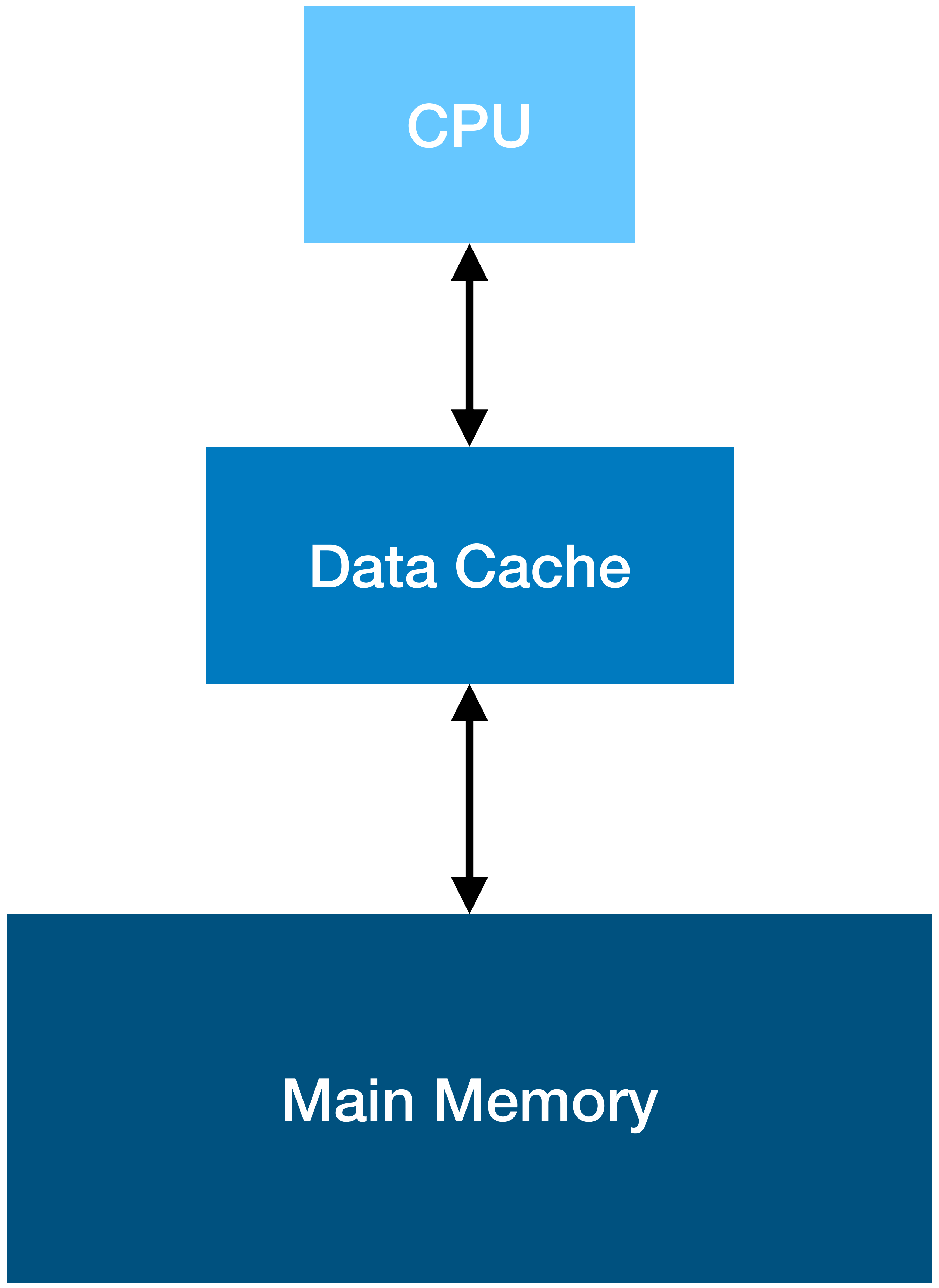
# Caches

- Modern machines have very large main memories
  - Making large, inexpensive memory means access is quite slow (hundreds of cycles to perform a load)
  - Fast memory is both small and expensive
- But programs perform *lots* of loads and stores
- Idea: add small, fast memory to hold some of your data → a cache



# Cache behavior

- Caches keep *recently used data* in fast memory
    - Caches use least recently used policy for keeping data: data that hasn't been used in a while is kicked out of cache
    - Intuition: program accessed a piece of data, so it is likely to access it again soon
- A program that reuses data quickly has good temporal locality → data likely to still be in cache
- A program that doesn't reuse data quickly has bad temporal locality → data likely to not be in cache
- The *same set of accesses* in a different order can have different behavior depending on how good the locality is



# Reuse distance

- How can we measure how good the locality in a program is? **reuse distance**
- Consider a stream of accesses:
  - For each access, count *how many other memory locations* have been accessed since the last time this location has been accessed
  - Important: not *number of accesses* — number of *unique other locations*

```
- - 1 1 - - 2 3
A B A B C D B A
```

# Locality using reuse distance

- On a memory access you can get a
  - Cache miss: first time a location is touched (cold miss) or because a location has not been touched in a while (capacity miss)
  - Cache hit: location has been touched recently, so is still in cache
  - Can also consider **spatial locality** — caches move memory around at the granularity of *cache lines*, so if A and B are next to each other in memory, accessing B right after A will result in a cache hit
- Reuse distance predicts cache hits: if reuse distance of an access is *less* than the number of elements the cache can hold, likely to be a cache hit

# Optimization for locality

- A program can have good or bad locality
- Can rearrange the order of accesses to reduce reuse distance, and hence get better locality

```
- - 1 1 - - 2 3
A B A B C D B A
```

vs

```
- 0 0 - 0 - -
A A A B B C D
```

# High level loop optimizations

- Many useful compiler optimizations require *restructuring* loops or sets of loops
  - E.g., change the order of a nested loop (*interchange*), running a loop in parallel (*parallelization*)
- Do not necessarily reduce the number of instructions; just changes when instructions are executed
- Goal: leverage hardware features like caches to execute instructions faster
  - Reschedule computations to improve reuse distance