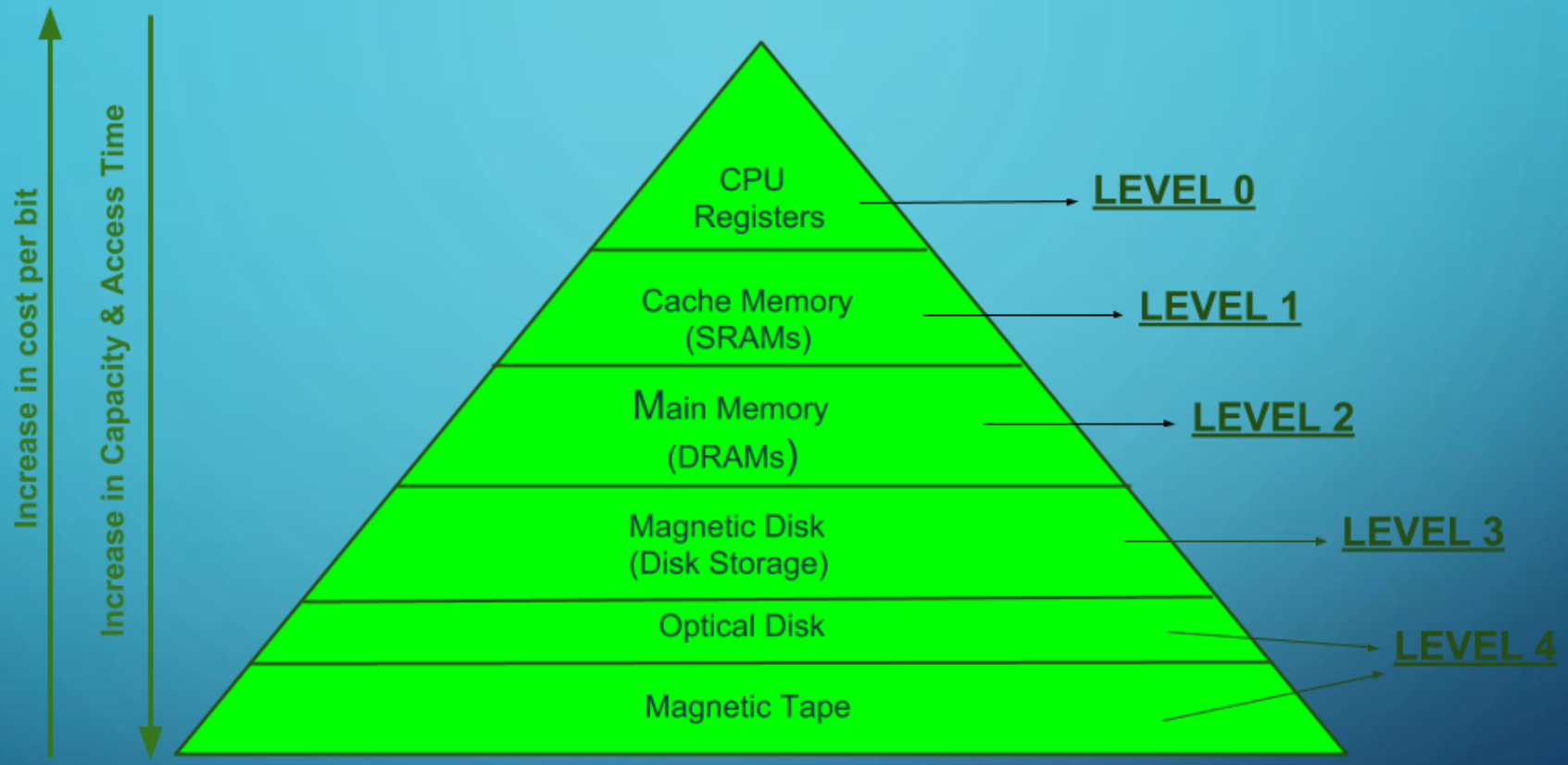
FALSE SHARING

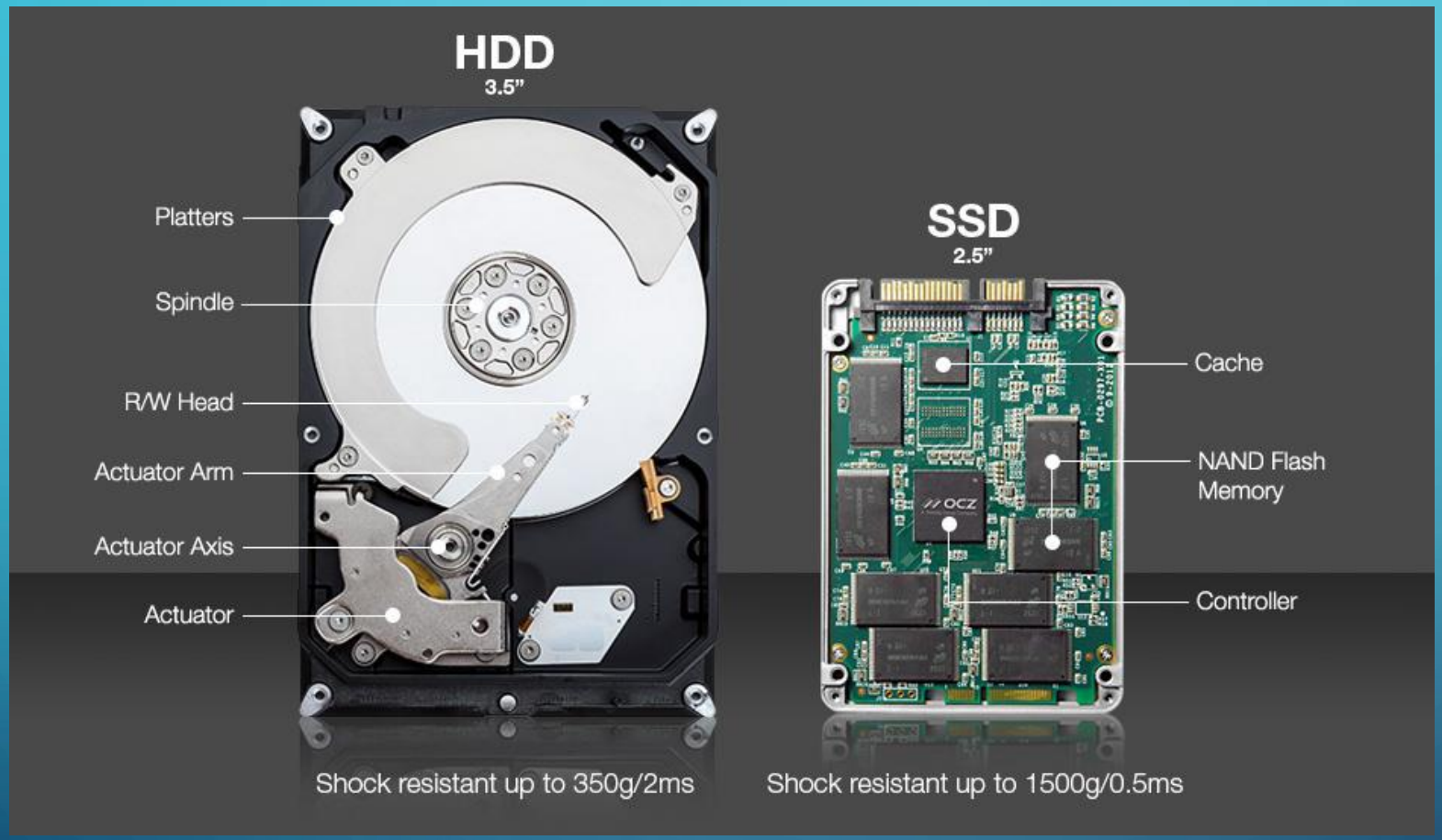
GPU621NSA

KEVIN CHOU

MEMORY HIERARCHY



SECONDARY MEMORY



- large storage capacity
- non-volatile
- affordable
- durable & reliable

- relatively slow
- not ideal for the extremely fast CPU

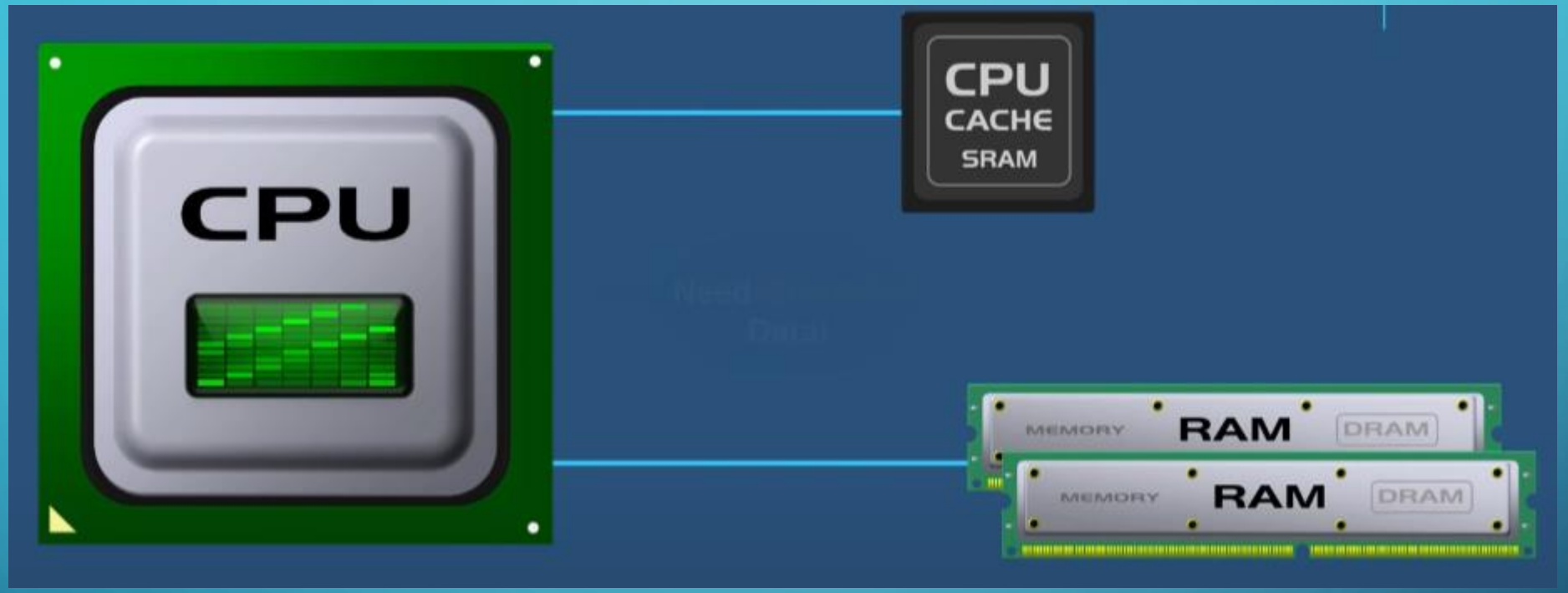
DYNAMIC RANDOM ACCESS MEMORY (DRAM)

- use capacitors, transistors and electricity to store data
- must be constantly refreshed to store data (volatile)
- smaller capacity
- fast, but still not fast enough for CPU

STATIC RANDOM ACCESS MEMORY (SRAM)

- does not need to be refreshed
- used in CPU cache
- very small storage capacity
- very expensive
- much faster than DRAM

CPU CACHE



- CPU always searches cache first
- “cache hit” if data is there
- “cache miss” if data is not there

CACHE OPERATION

- want to minimize the number of cache misses to avoid performance loss
- data is brought into cache ahead of time

Locality of Reference

- the tendency of a program to access the same set of memory locations over a short period of time

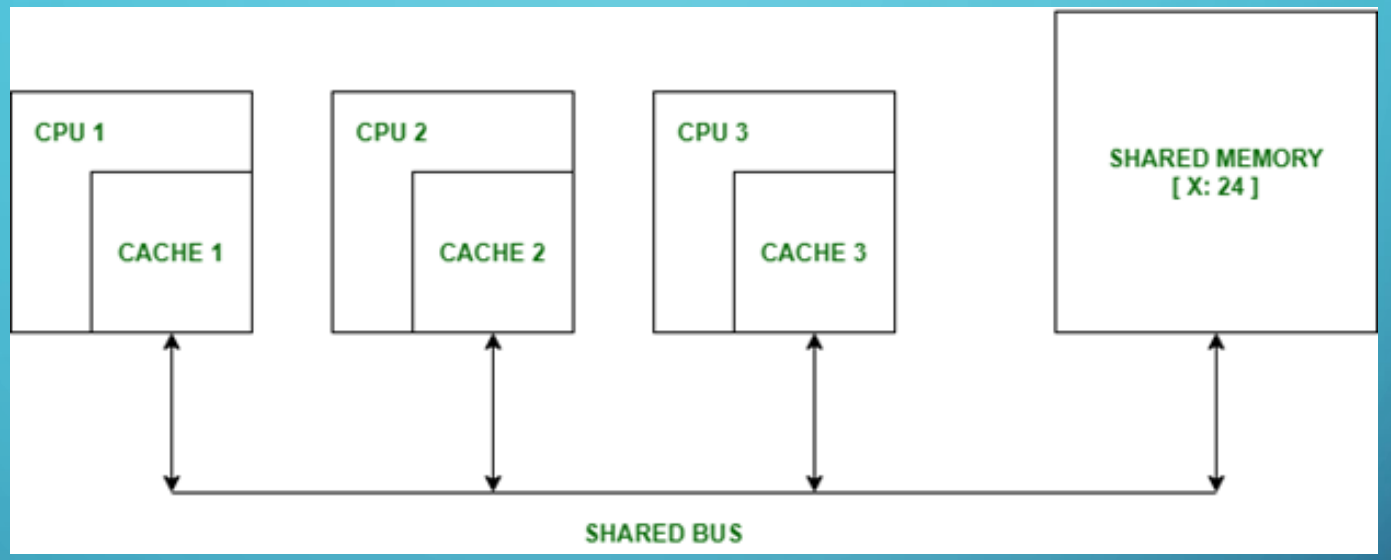
Temporal Locality

- an accessed memory location will likely be accessed again in the near future

Spatial Locality

- nearby memory locations of an accessed memory location will likely be needed in the near future

CACHE COHERENCE & CACHE LINE



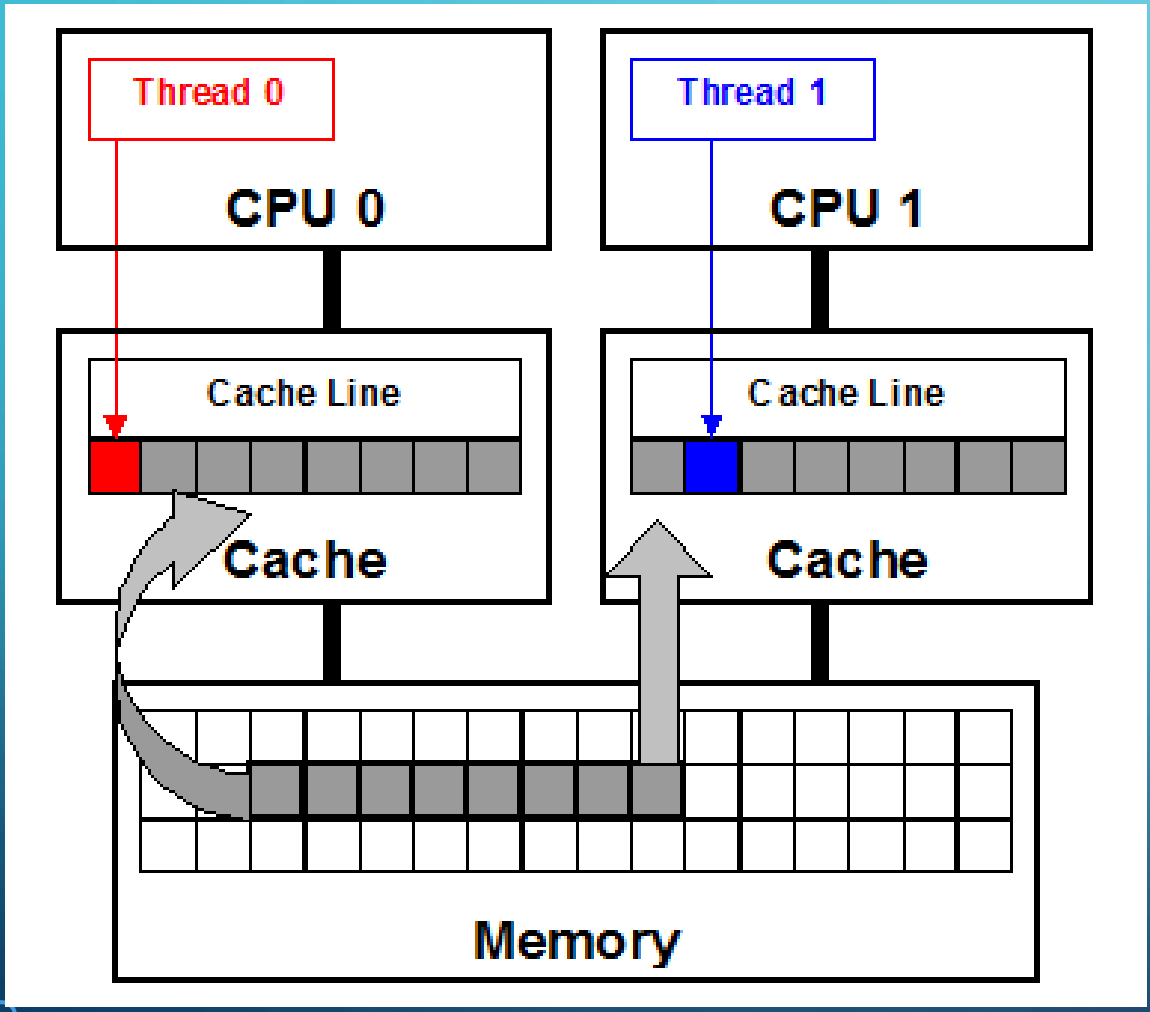
- cache lines are blocks of memory transferred to the cache
- requires data synchronization

CACHE COHERENCE & CACHE LINE

Cache Coherence: the uniformity of shared resource data that ends up stored in multiple local caches

- solving this is known as the cache coherence problem
- modern systems use cache coherence protocols to manage and maintain the cache
- cache coherence protocols prevent data discrepancies

FALSE SHARING



- occurs when multiple processors modify data that resides on the same cache line
- does not have to be same piece of data
- cache coherency maintained on cache-line basis
- modified shared cache lines become invalidated or “dirty”
- must fetch updated data from memory affecting performance

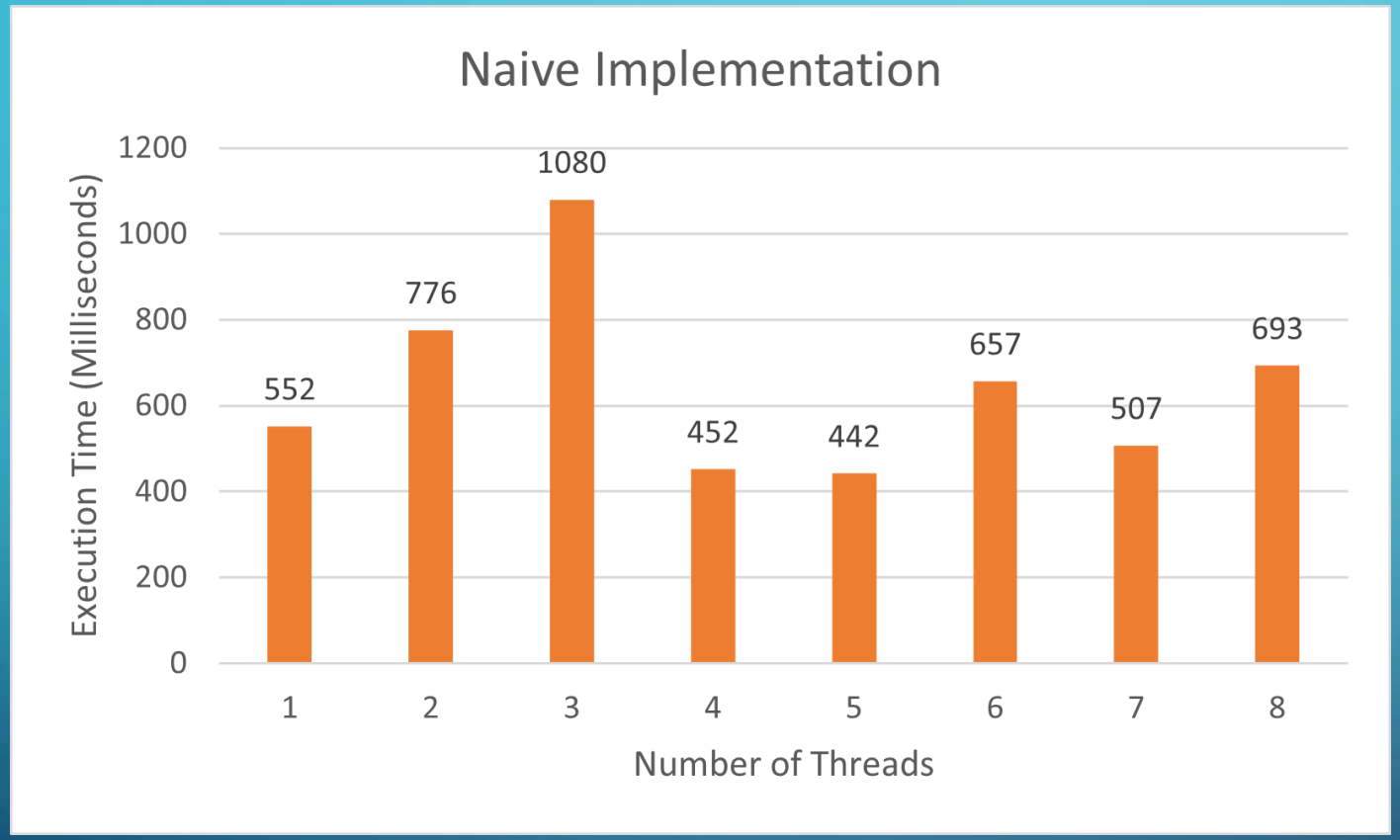
FALSE SHARING IN WORKSHOP 2

```
int actual_thread_count;
double pi = 0.0f;
double sum[NUM_THREADS] = { 0.0f };
double step = 1.0 / (double)n;

omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int id, num_threads;
    double x;

    id = omp_get_thread_num();
    num_threads = omp_get_num_threads();

    // get master thread to return how many threads were actually created
    if (id == 0)
    {
        actual_thread_count = num_threads;
    }

    // each thread is responsible for calculating the area of a specific set of sections underneath the curve
    for (int i = id; i < n; i = i + num_threads)
    {
        x = ((double)i + 0.5f) * step;
        sum[id] += 1.0f / (1.0f + x * x);
    }
}

// sum up each calculation to get approximation of pi
for (int i = 0; i < actual_thread_count; i++)
{
    pi += 4 * sum[i] * step;
}
```

← Promote sum to an array

← Threads store their calculations in their own element of the array

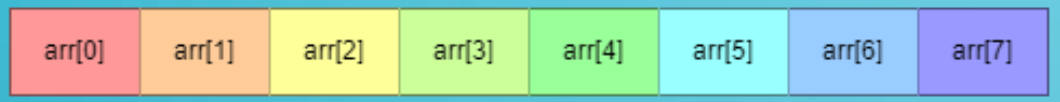
FALSE SHARING IN WORKSHOP 2



- arrays are a contiguous block of memory in C++
- nearby elements of the array are brought into cache due to spatial locality
- likely share the same cache line causing false sharing

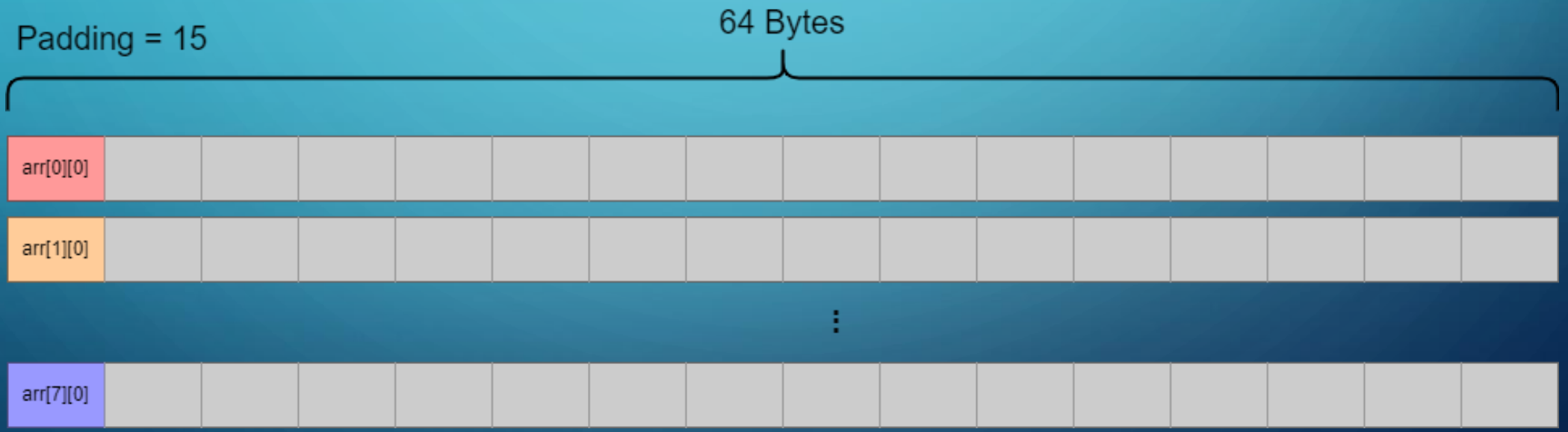
PADDING

Naive Implementation's Array

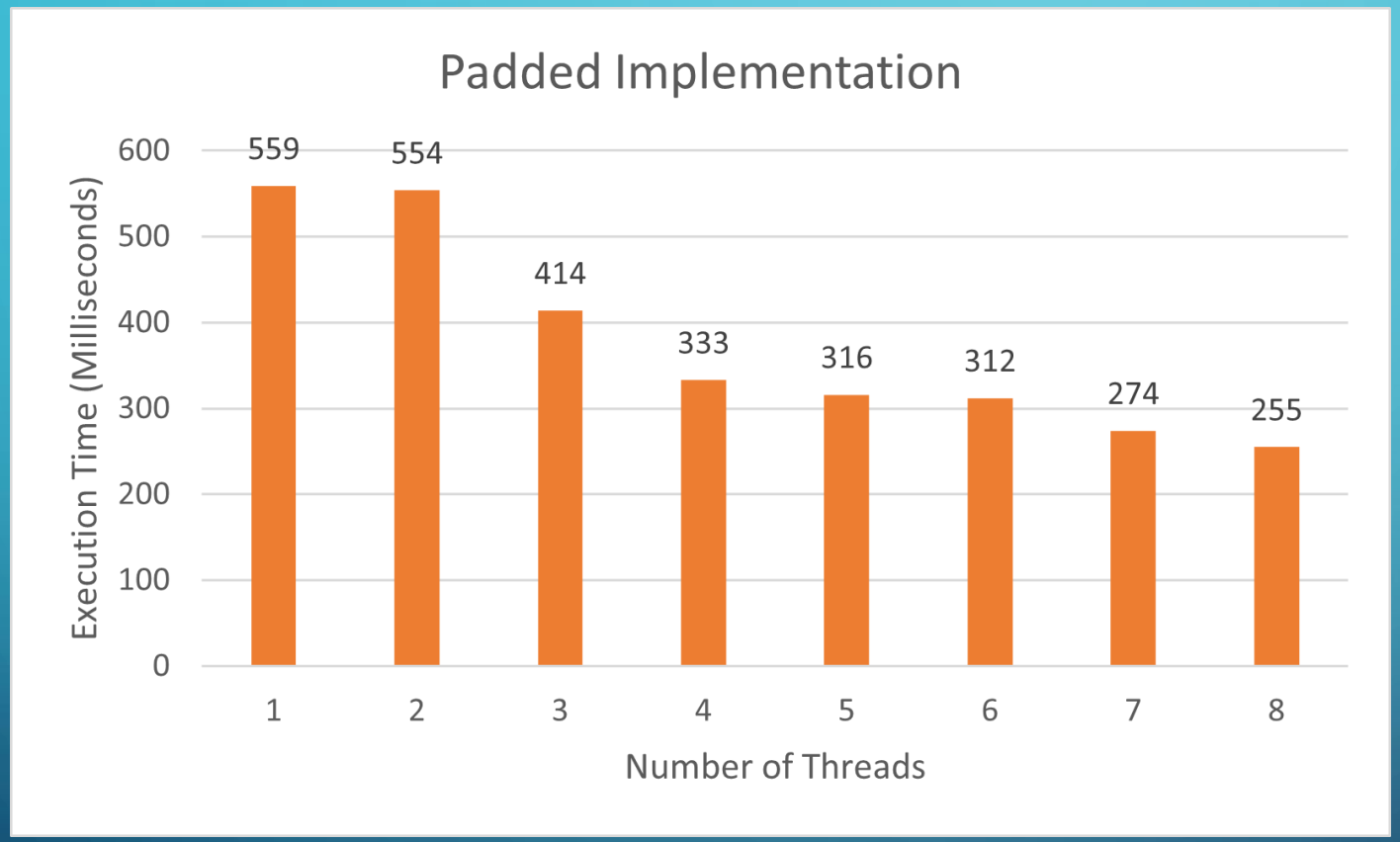


- pad out a 2D array, so that only the individual element is brought into cache

Padded Implementation's Array



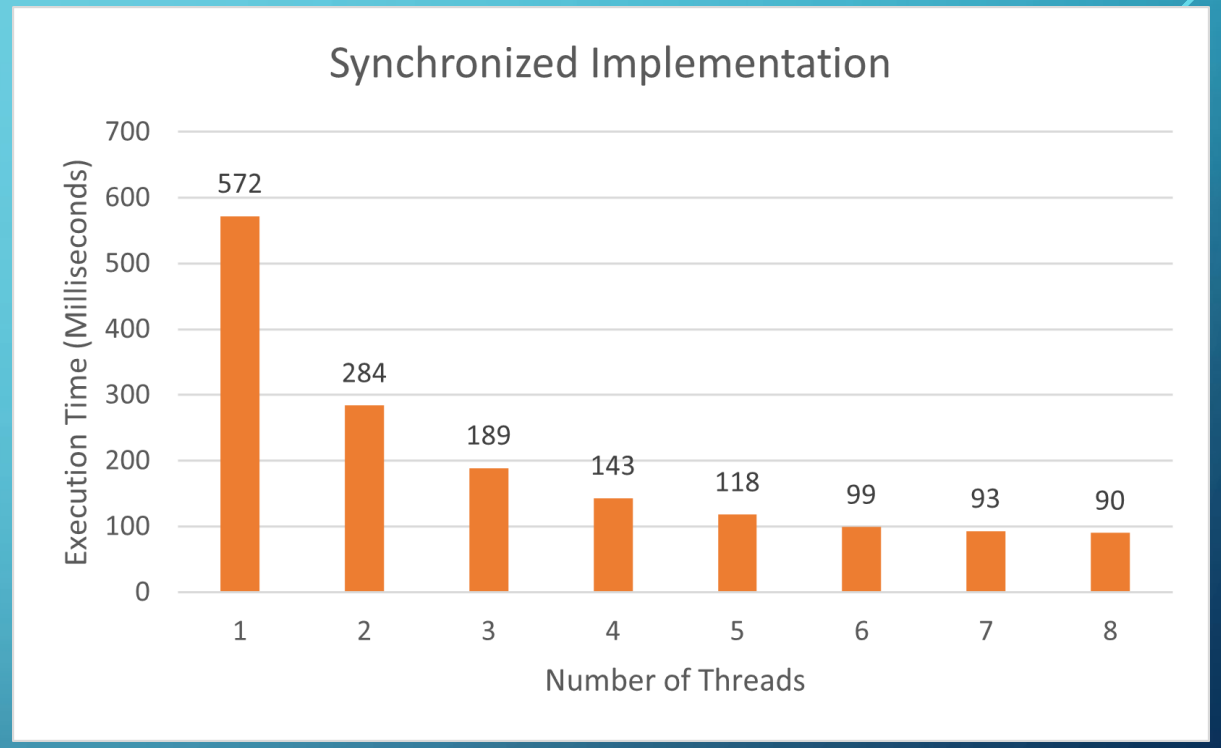
DRAWBACKS WITH PADDING



- must know cache line size to accurately pad the array
- wasted memory space used for padding
- hogging valuable cache space

SYNCHRONIZATION & THREAD LOCAL VARIABLES

```
#pragma omp parallel
{
    int id, num_threads;
    double x, sum = 0.0f;

    id = omp_get_thread_num();
    num_threads = omp_get_num_threads();

    // get master thread to return how many threads were actually created
    if (id == 0)
    {
        actual_thread_count = num_threads;
    }

    // each thread is responsible for calculating the area of a specific set of sections underneath the curve
    for (int i = id; i < n; i = i + num_threads)
    {
        x = ((double)i + 0.5f) * step;
        sum += 1.0f / (1.0f + x * x);
    }

    #pragma omp critical
    {
        // sum up each calculation to get approximation of pi
        pi += 4 * sum * step;
    }
}
```



CONCLUSION

False sharing is a sneaky problem requiring detail code inspection.

When left unchecked, false sharing drastically affects a program's performance and scalability.