# Lab 5: Parallel processors & parallel programming

Hung-Wei Tseng

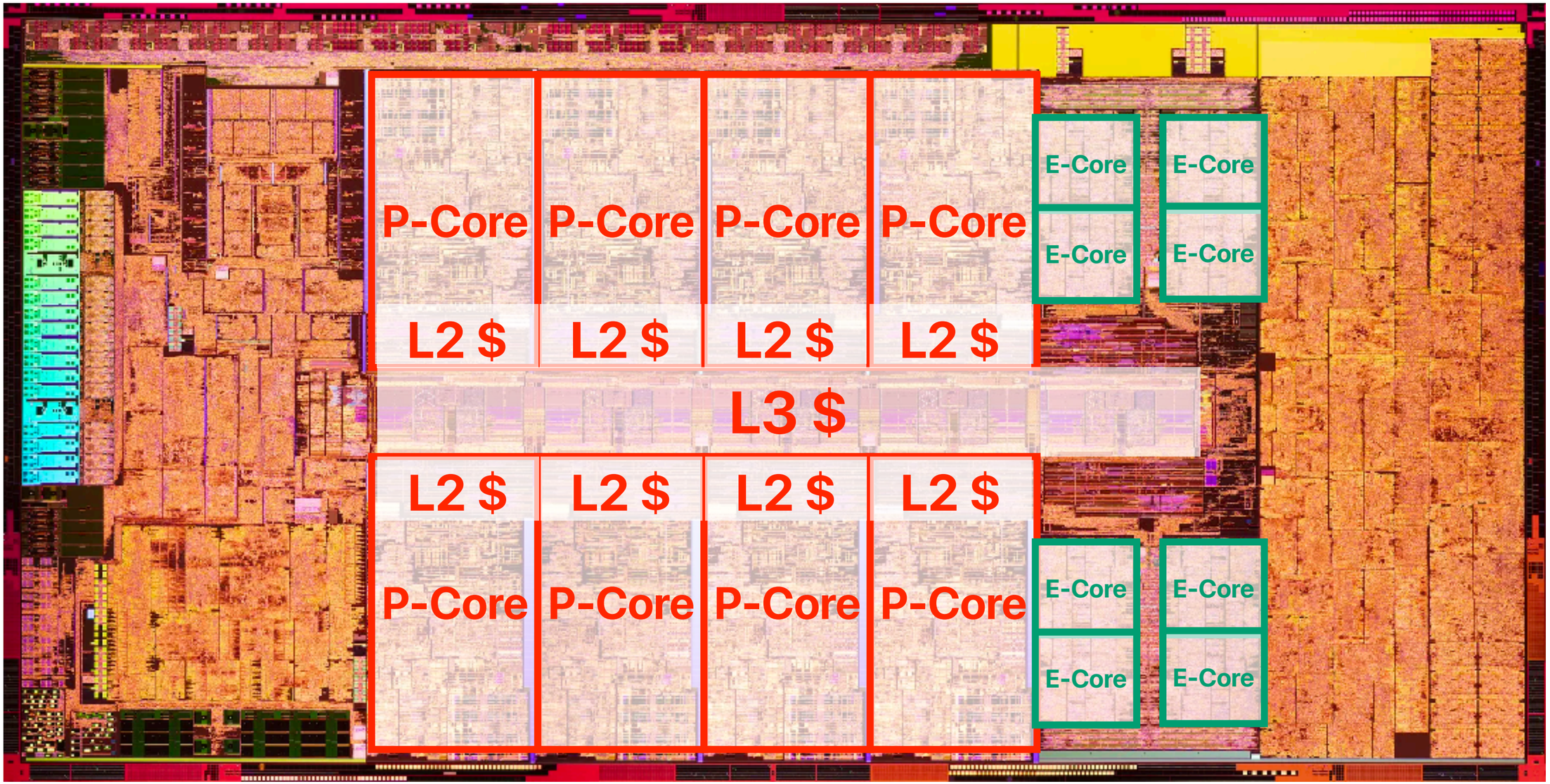
# Parallelism in modern computers

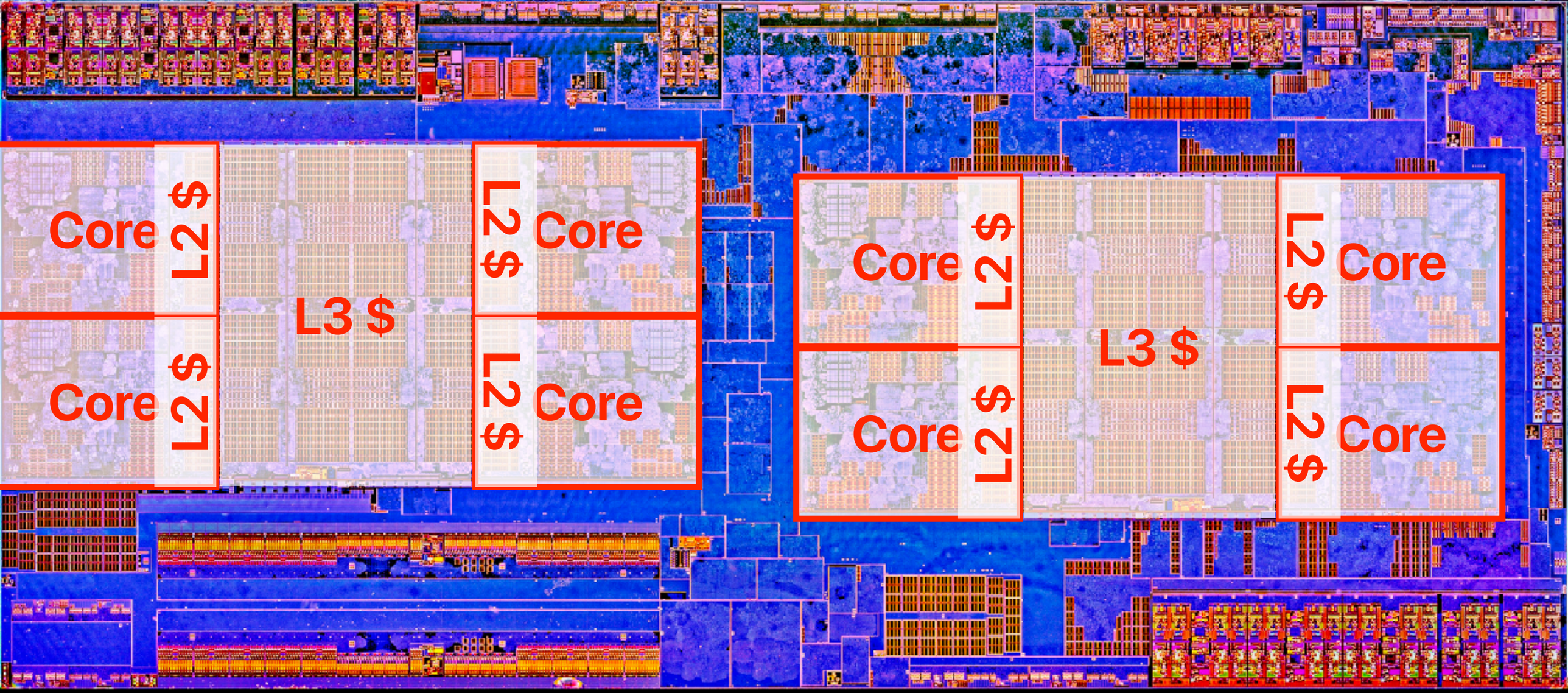
- Instruction-level parallelism
  - The ability to execute multiple instructions concurrently
- Thread-level parallelism
  - The ability to execute multiple program instances concurrently
- Data-level parallelism
  - The ability to process data concurrently

# Processing models

- SISD — single instruction stream, single data
  - Pipelining instructions within a single program
  - Superscalar
- SIMD — single instruction stream, multiple data
  - Vector instructions
  - GPUs
- MIMD — multiple instruction stream (e.g. multiple threads, multiple processes), multiple data
  - Multicore processors
  - Multiple processors
  - Simultaneous multithreading

# Intel Alder Lake

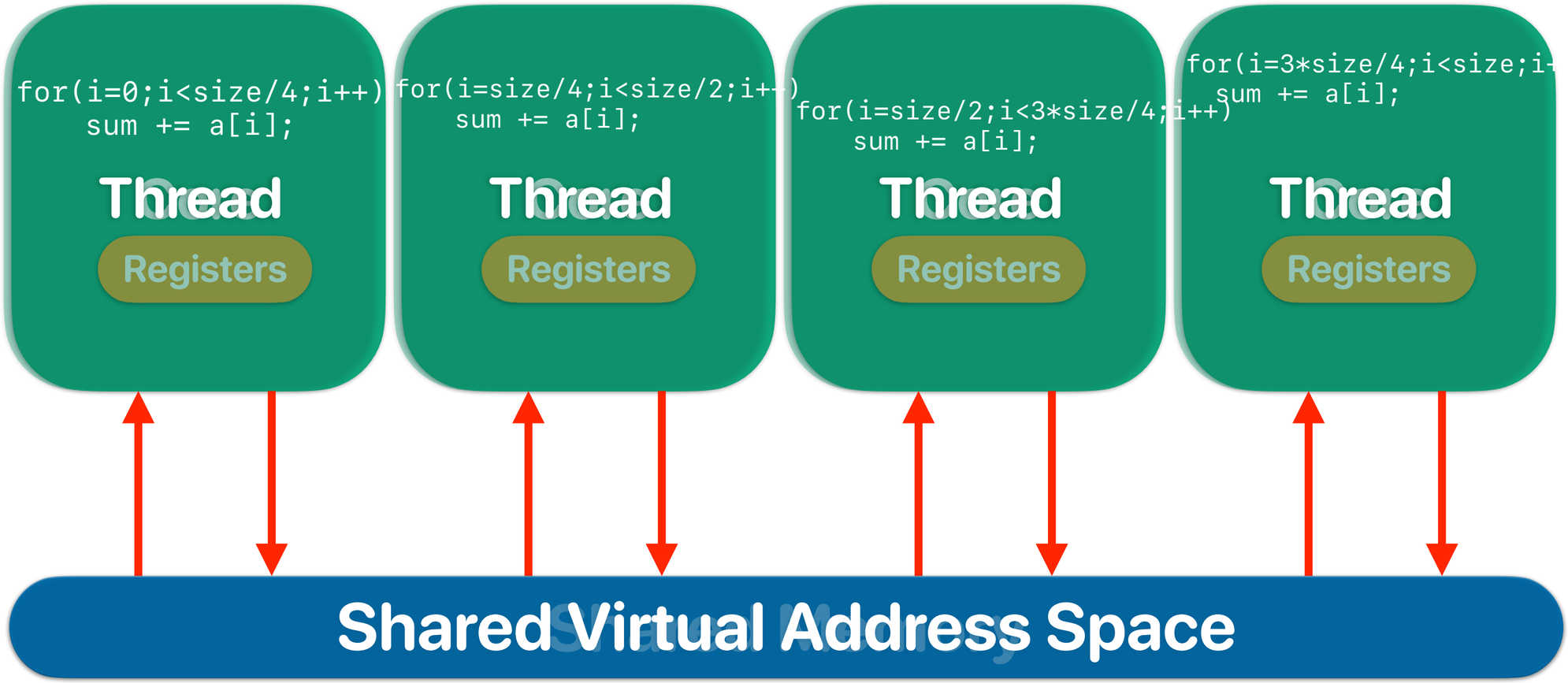



Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core
Core
L2 $
L3 $
L2 $
Core
Core
L2 $
L2 $
Core
AMD
RYZEN

# Thread programming

# pthread

# The “abstracted” multithreading machine



# Thread programming

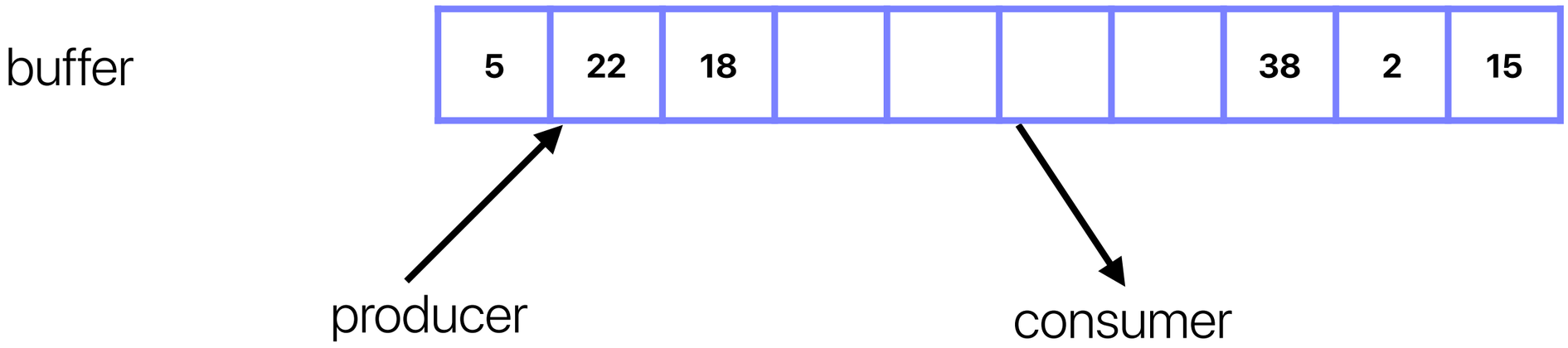
- pthread_t — thread descriptor
- pthread_init() — init a thread descriptor
- pthread_create() — create a thread running a “start_routine” function with “arg” as the argument

  ```
  int pthread_create(pthread_t *restrict thread,
                     const pthread_attr_t *restrict attr,
                     void *(*start_routine)(void *),
                     void *restrict arg);
  ```

  - The thread could be any function you like that fulfills the interface requirements
  - Once created, the thread will run as a separate identity in the OS
    - The OS can schedule it separately from the main thread
    - The thread can share/access/modifies the content of the main’s threads’ virtual memory space
- pthread_join() — synchronize thread execution
  - The main thread will wait for the thread descriptor in the argument to finish
- pthread_mutex_lock, pthread_mutex_unlock — managing a lock

# Bounded-Buffer Problem

- Also referred to as "producer-consumer" problem
- Producer places items in shared buffer
- Consumer removes items from shared buffer



# Q2: will this kind of code working?

```
int buffer[BUFF_SIZE]; // shared global
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;

        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;

    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {

        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;

        // do something w/ item
    }
    return NULL;
}
```

**There is no guarantee who is going/running faster**

# We need to control accesses to the buffer!

```
int buffer[BUFF_SIZE]; // shared global
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;

        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;

    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {

        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;

        // do something w/ item
    }
    return NULL;
}
```

**There is no guarantee who is going/running faster**

# Solving the "Critical Section Problem"

1. Mutual exclusion — at most one process/thread in its critical section
2. Progress — a thread outside of its critical section cannot block another thread from entering its critical section
3. Fairness — a thread cannot be postponed indefinitely from entering its critical section
4. Accommodate nondeterminism — the solution should work regardless the speed of executing threads and the number of processors

# Q3: Use locks

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

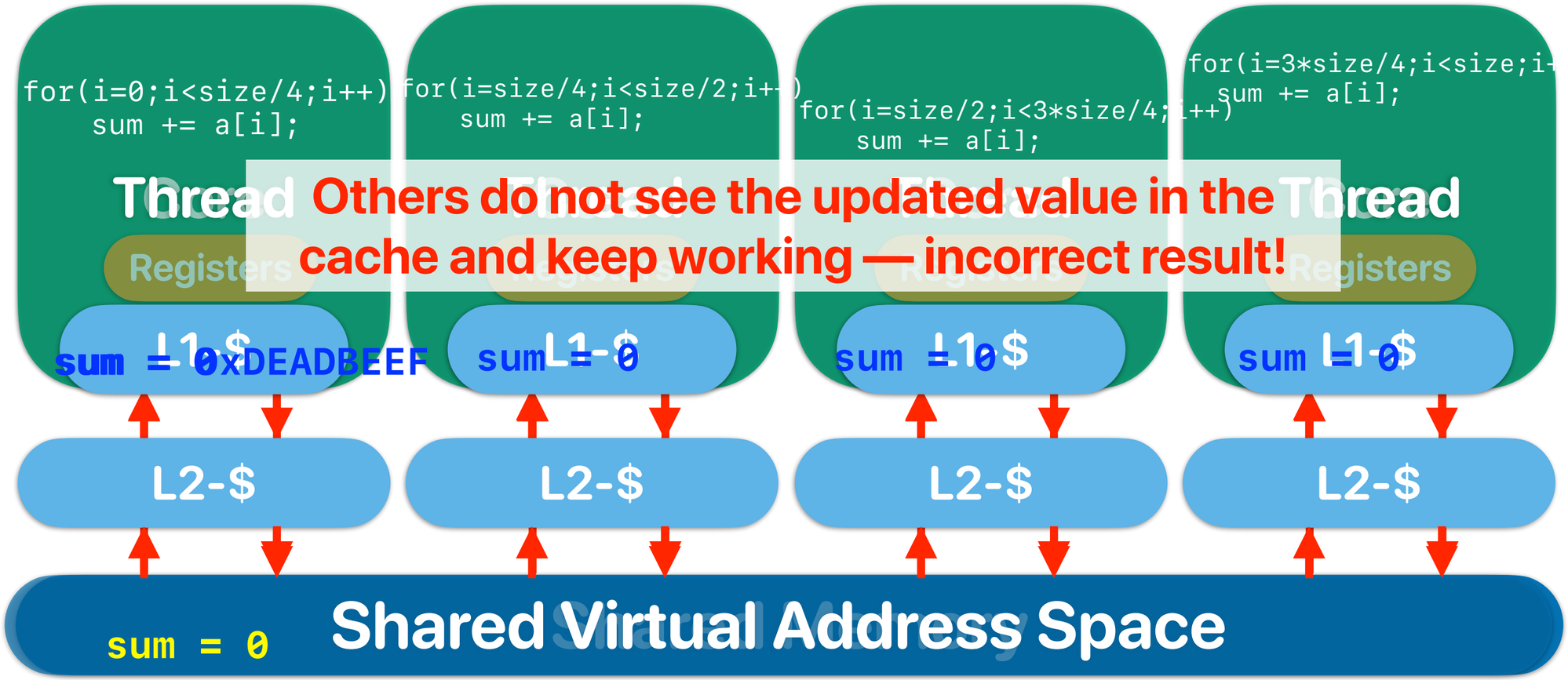
```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

# Q4 and Q8: No free-lunch — lock has overhead!

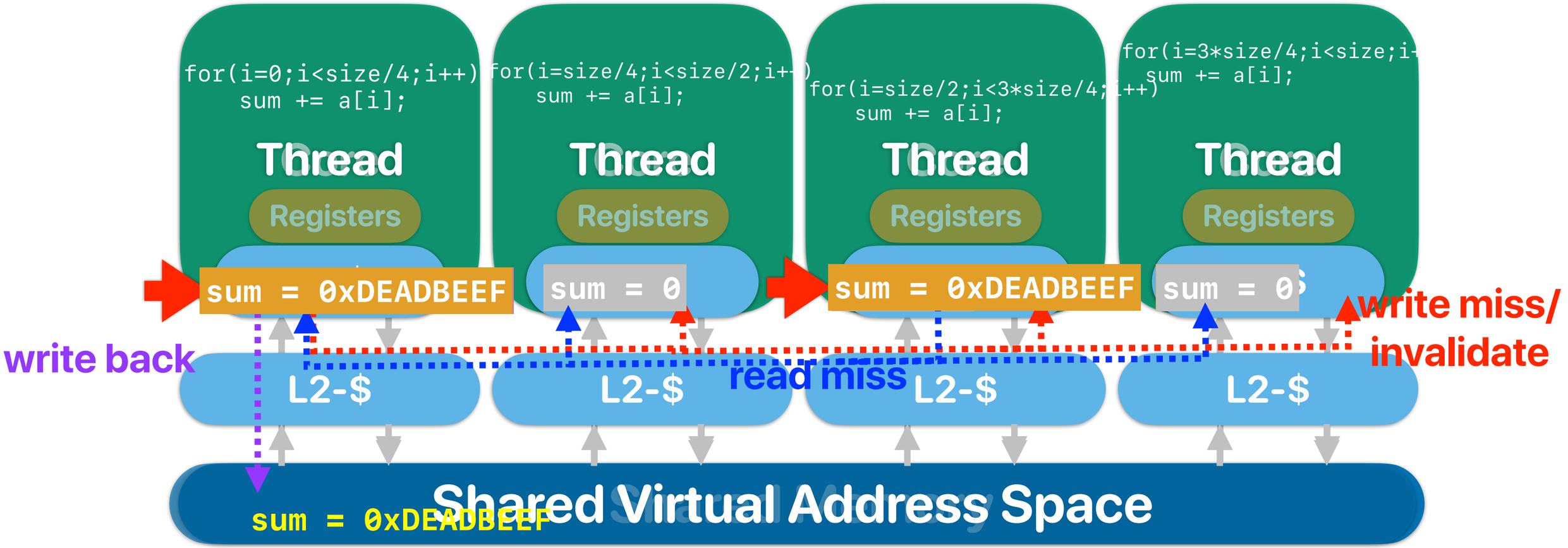
- How can you measure the overhead of adding locks?
  - Run an experiment of baseline that doesn't use lock, no threads
  - Run another experiment of an implementation use locks, no threads
    - A fair experiment should change just one thing at a time
  - How much more time we spent per lock/unlock?

# What software thinks about “multiprogramming” hardware

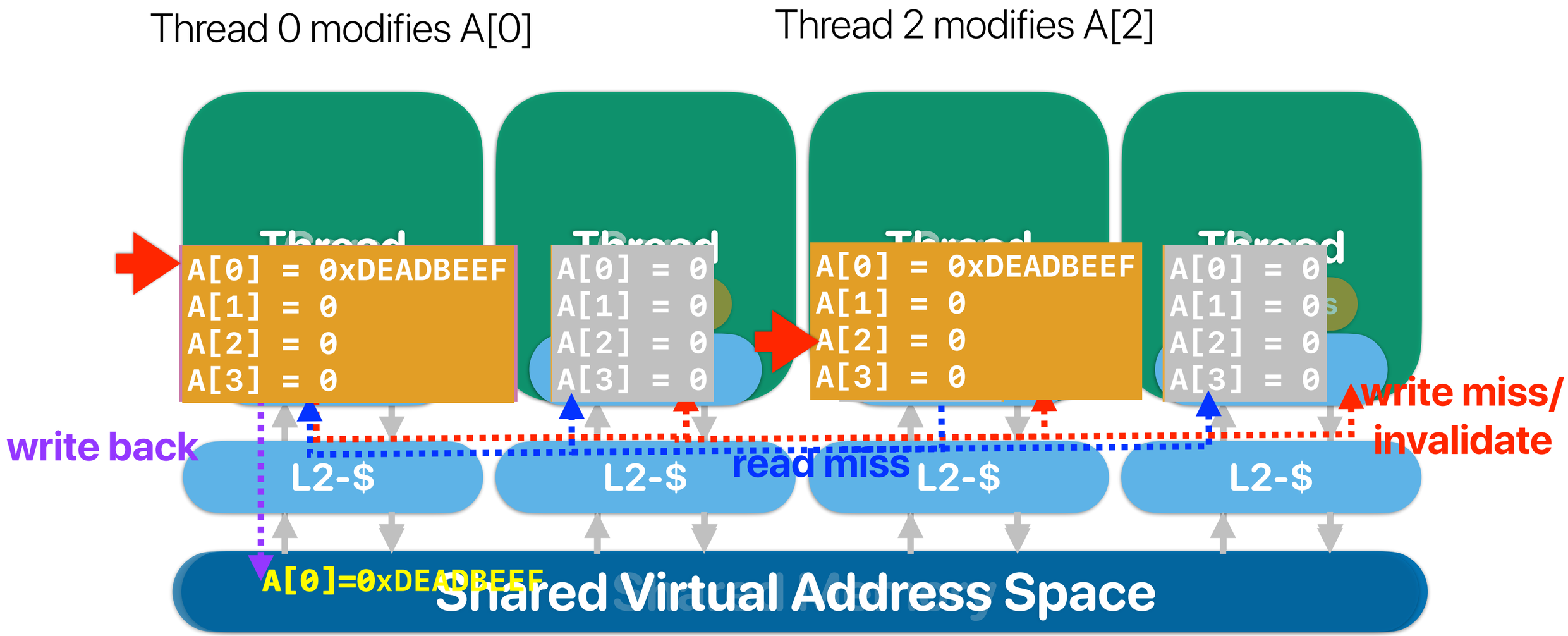



Others do not see the updated value in the cache and keep working — incorrect result!

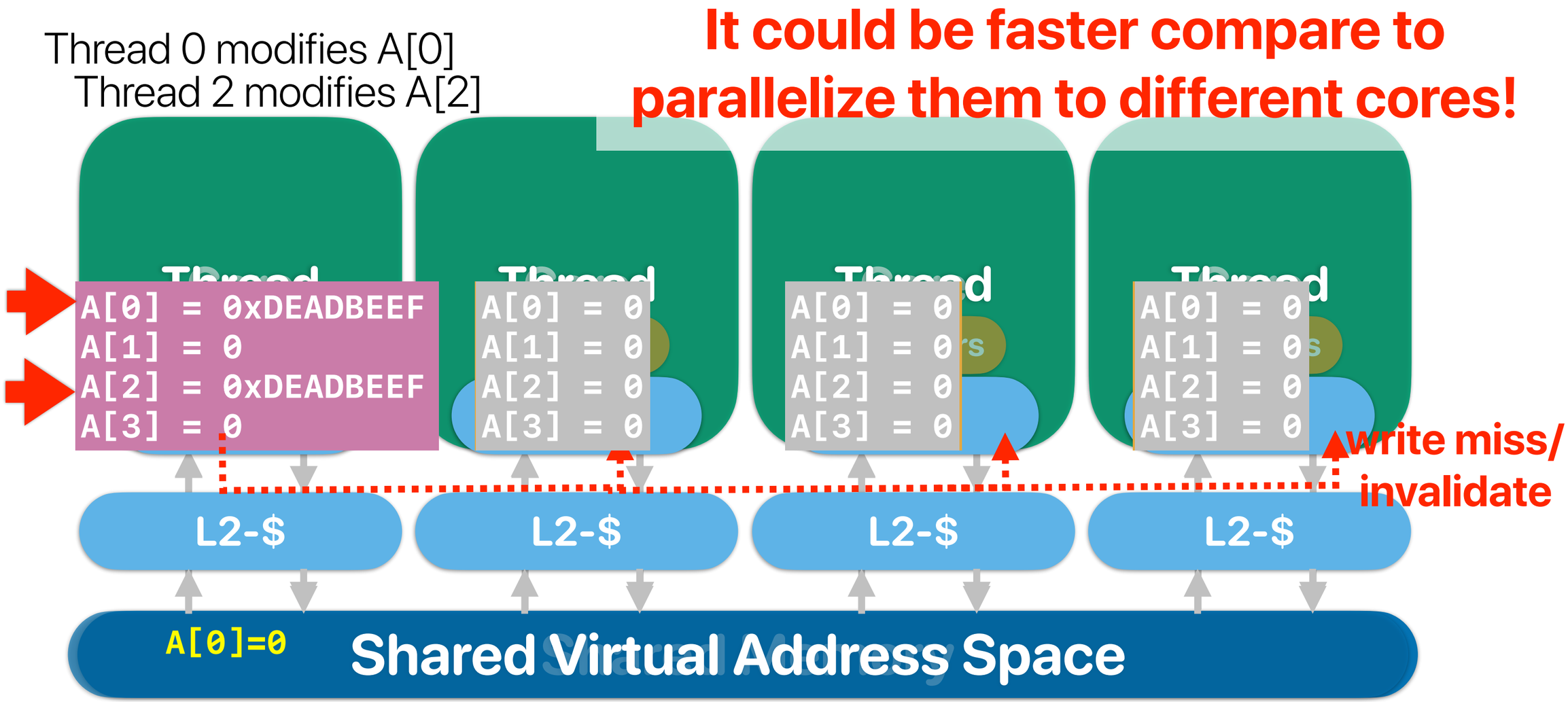
# What happens when we write in coherence caches?



# Remember, cache manages data in “blocks”



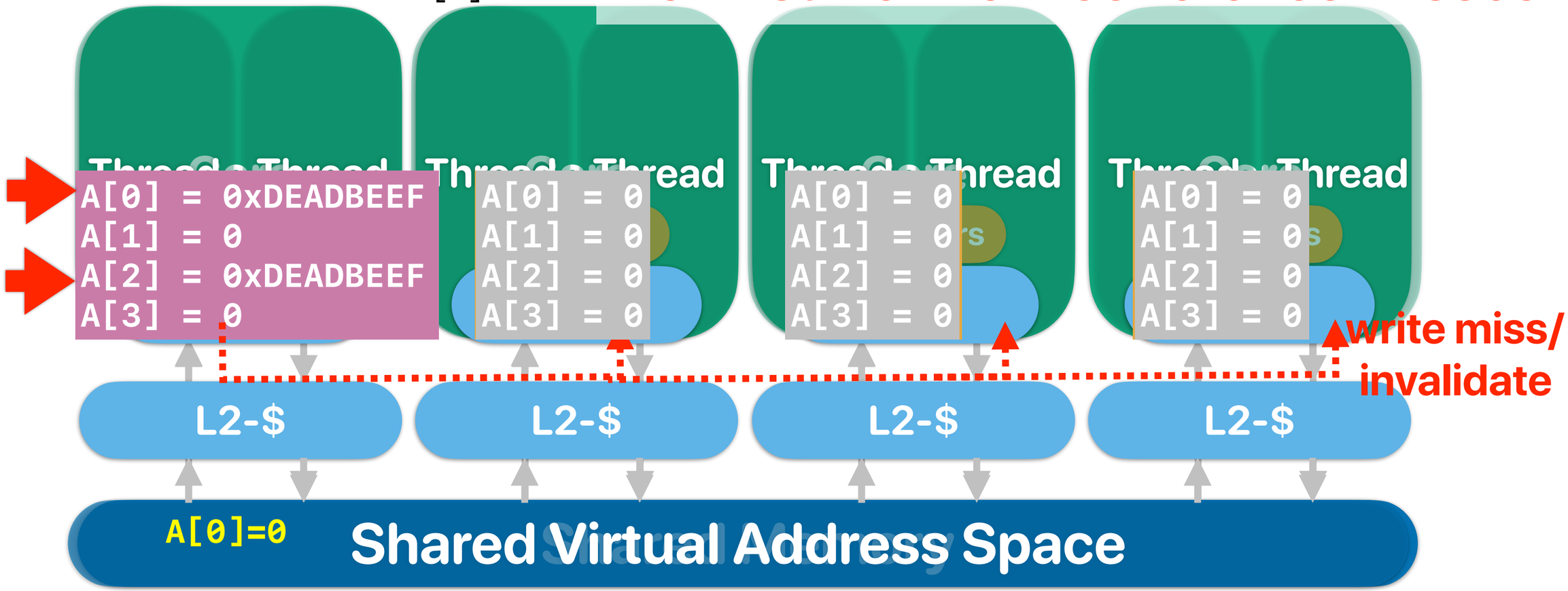
# Q5: If we have both threads on the same core…



# Interesting question — what if we have SMT

Thread 0 modifies A[0]
Thread 2 modifies A[2]

**Threads on the same physical core won't suffer from coherence misses**



# Q11: How can we fix the tasks/threads to specific cores?

- Linux command: taskset
- C/Linux API: the following will fix the task to either core id 0 or 2

```
#define _GNU_SOURCE
#include <sched.h>

cpu_set_t  mask;
CPU_ZERO(&mask);
CPU_SET(0, &mask);
CPU_SET(2, &mask);
int result = sched_setaffinity(0, sizeof(mask), &mask);
```

**Which set is the best?**

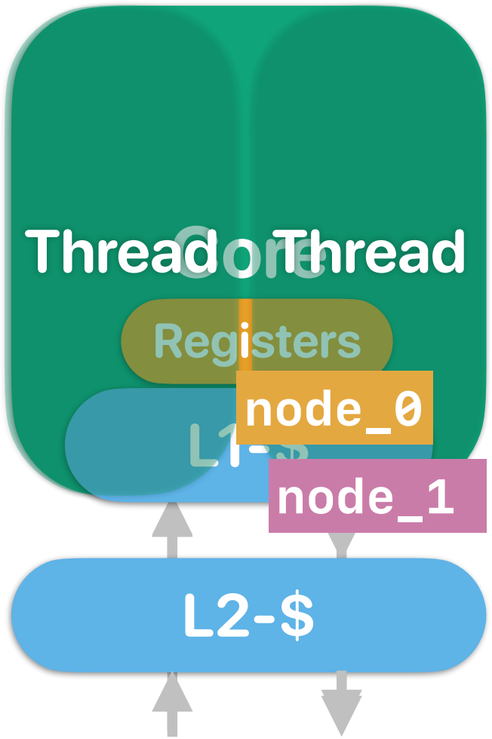
# Q6: How can you avoid false sharing?

- Make sure the partition of your tasks does not overlap on the same block
- Make sure the variables that each thread accesses are allocated in different blocks
  - Padding the data structure

# Can you think about a clever way to use SMT?

```
do {
    current = current->next;
} while ( current != NULL );
```

```
do {
    number_of_nodes++;
    current = current->next;
} while ( current != NULL );
```




```
do {
    number_of_nodes++;
    current = current->next;
} while ( current != NULL );
```

**Not parallelizable due to dependencies**

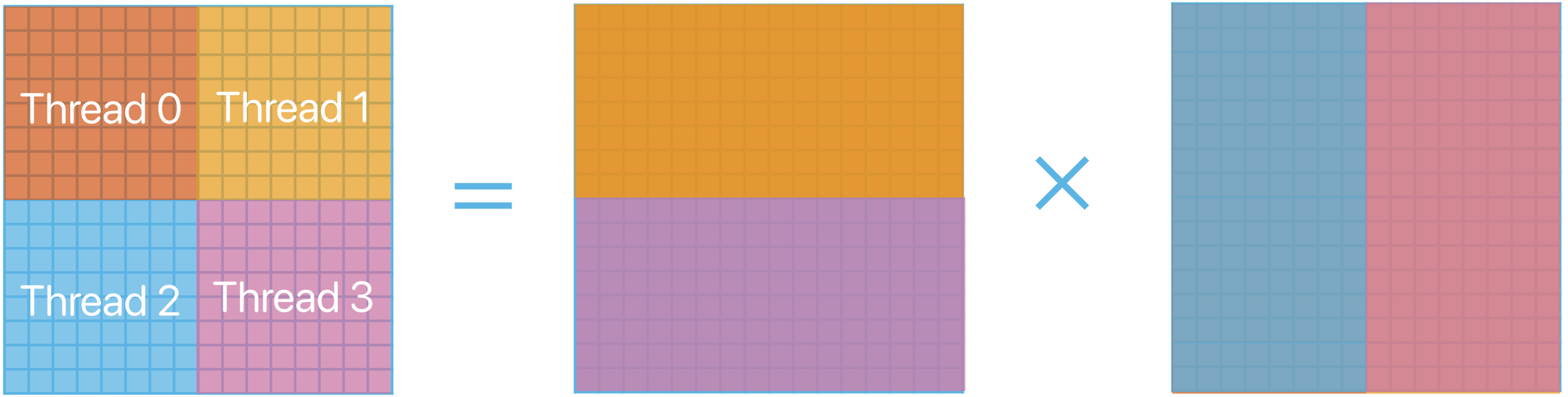
**Can we use SMT to make this faster?**

# Tips of performance thread programming

- Reduce sharing: coherence misses
- Avoid fine-grained locks: serialization of execution
- Avoid short threads: thread spawning overhead

# How are you going to parallelize “matrix multiplications”

# Is tiling still a good idea?




**No sharing among threads!**

**What if the tile size creates false sharing?**

**They're read only — not false sharing**

**We reduced the footprint of each thread**

# Retrospects: Amdahl's Law

# Recap: Amdahl's Law on Multicore Architectures

- Symmetric multicore processor with $n$ cores (if we assume the processor performance scales perfectly)

$$Speedup_{parallel}(f_{parallelizable}, n) = \frac{1}{(1 - f_{parallelizable}) + \frac{f\_parallelizable}{n}}$$

# Demo — merge sort v.s. bitonic sort on GPUs

## Merge Sort

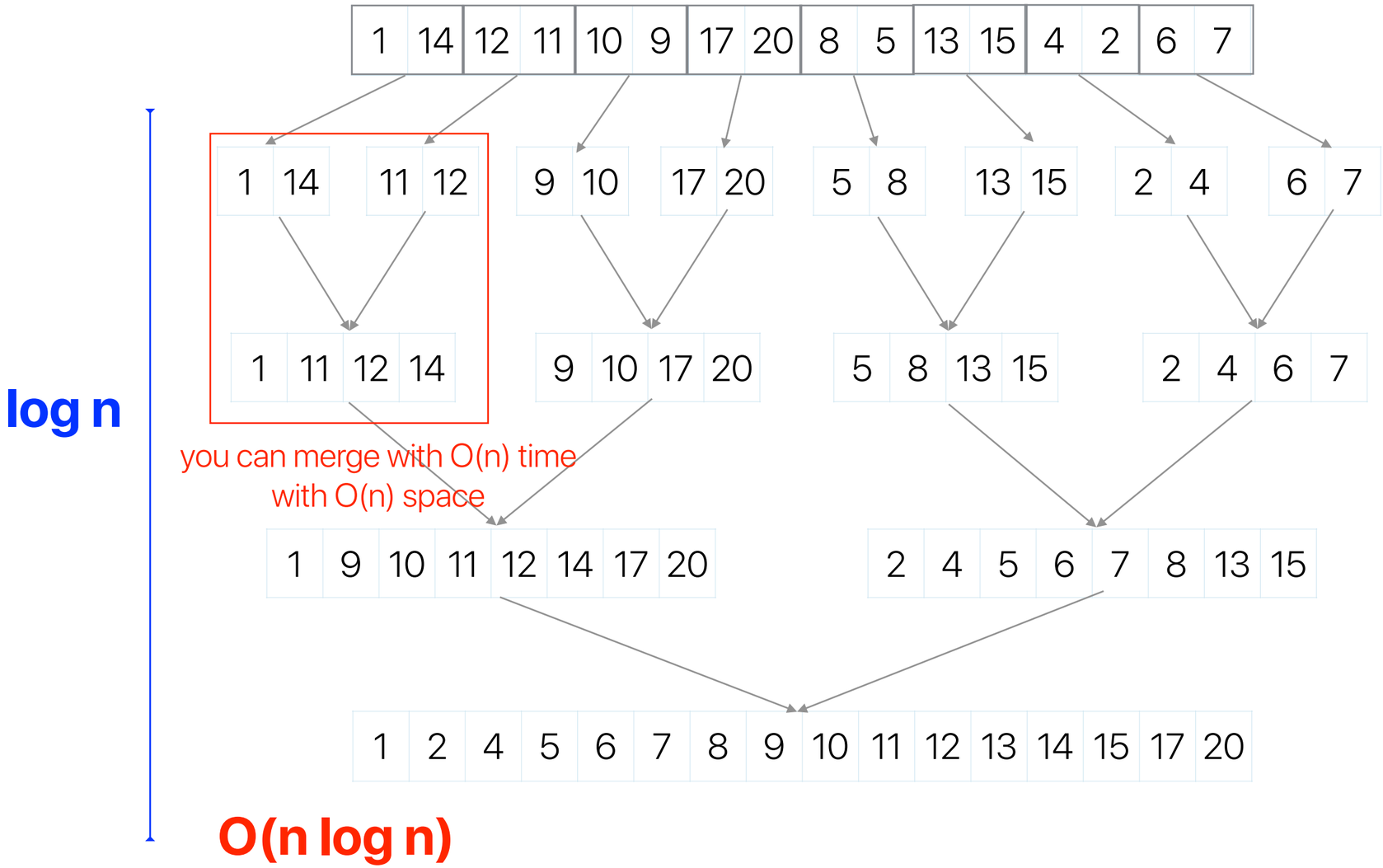
$O(nlog_2n)$

## Bitonic Sort

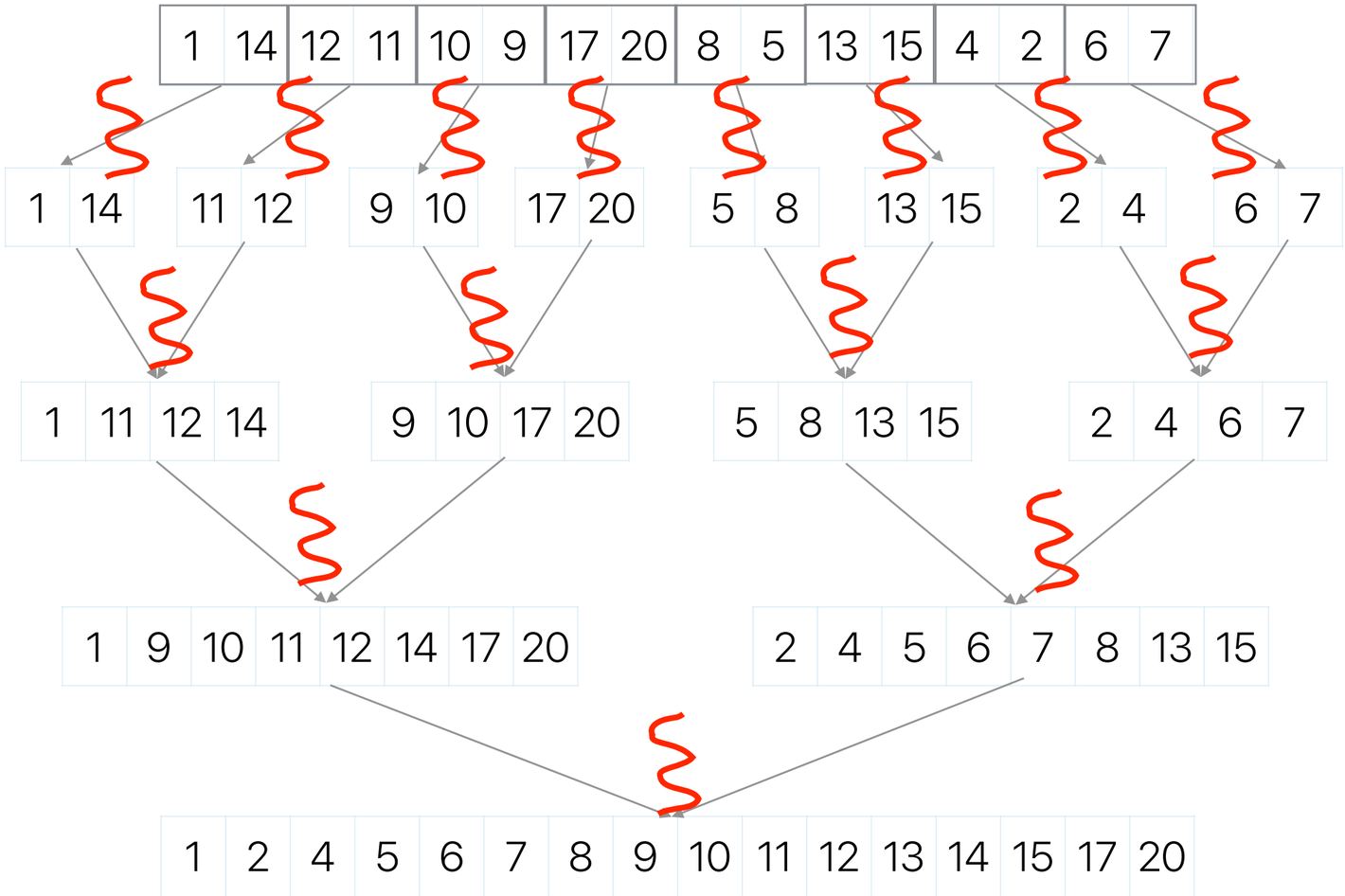
$O(nlog_2^2n)$

```
void BitonicSort() {

    int i,j,k;

    for (k=2; k<=N; k=2*k) {
        for (j=k>>1; j>0; j=j>>1) {
            for (i=0; i<N; i++) {
                int ij=i^j;
                if ((ij)>i) {
                    if ((i&k)==0 && a[i] > a[ij])
                        exchange(i,ij);
                    if ((i&k)!=0 && a[i] < a[ij])
                        exchange(i,ij);
                }
            }
        }
    }
}
```

# Merge sort


1 14 12 11 10 9 17 20 8 5 13 15 4 2 6 7
1 14 11 12 9 10 17 20 5 8 13 15 2 4 6 7
1 11 12 14 9 10 17 20 5 8 13 15 2 4 6 7
you can merge with O(n) time
with O(n) space
1 9 10 11 12 14 17 20 2 4 5 6 7 8 13 15
1 2 4 5 6 7 8 9 10 11 12 13 14 15 17 20
log n
O(n log n)

# Parallel merge sort



# What's the speedup of merge sort using Amdahl's Law

The degree of parallelism is $1, 2, 4, \ldots, \frac{n}{2}$

at step $1, 2, 3, \ldots, log_2(n)$

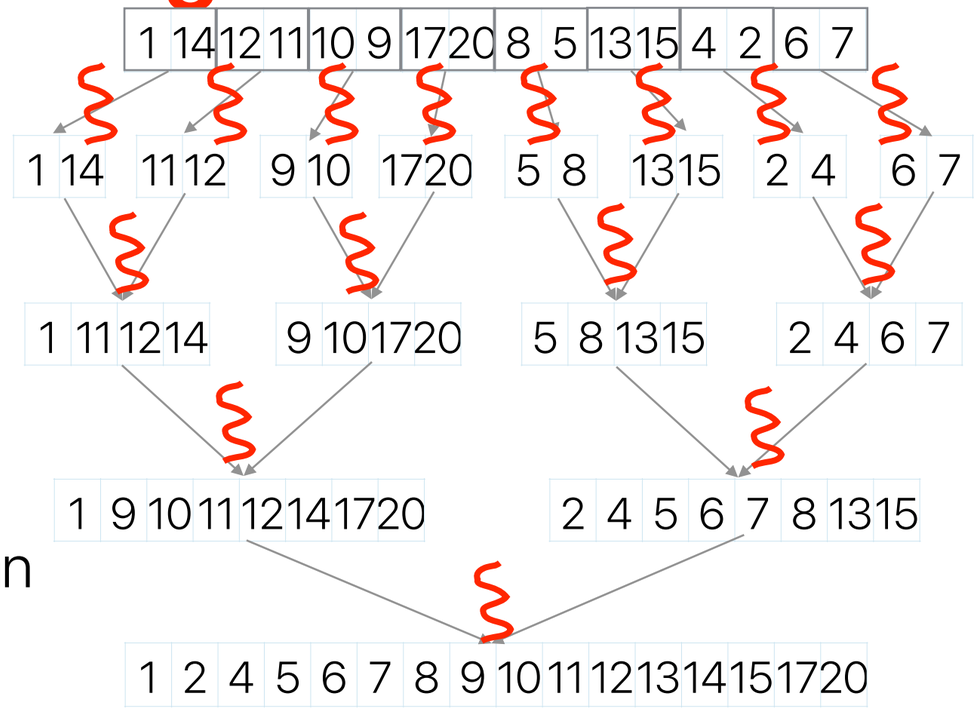
The ideal speedup of each step is $1, 2, 4, \ldots, \frac{n}{2}$ or say

$1, 2, 4, \ldots, 2^{log_2(n)-1}$ if we have **unlimited** parallelism

if we assume equal amount of time in each step in the baseline,

each step is going to take $\frac{1}{lg(n)}$ portion of time in the baseline, an

the first $\frac{1}{lg(n)}$ is not parallelizable (i.e., $(1-x) = \frac{1}{lg(n)}$)

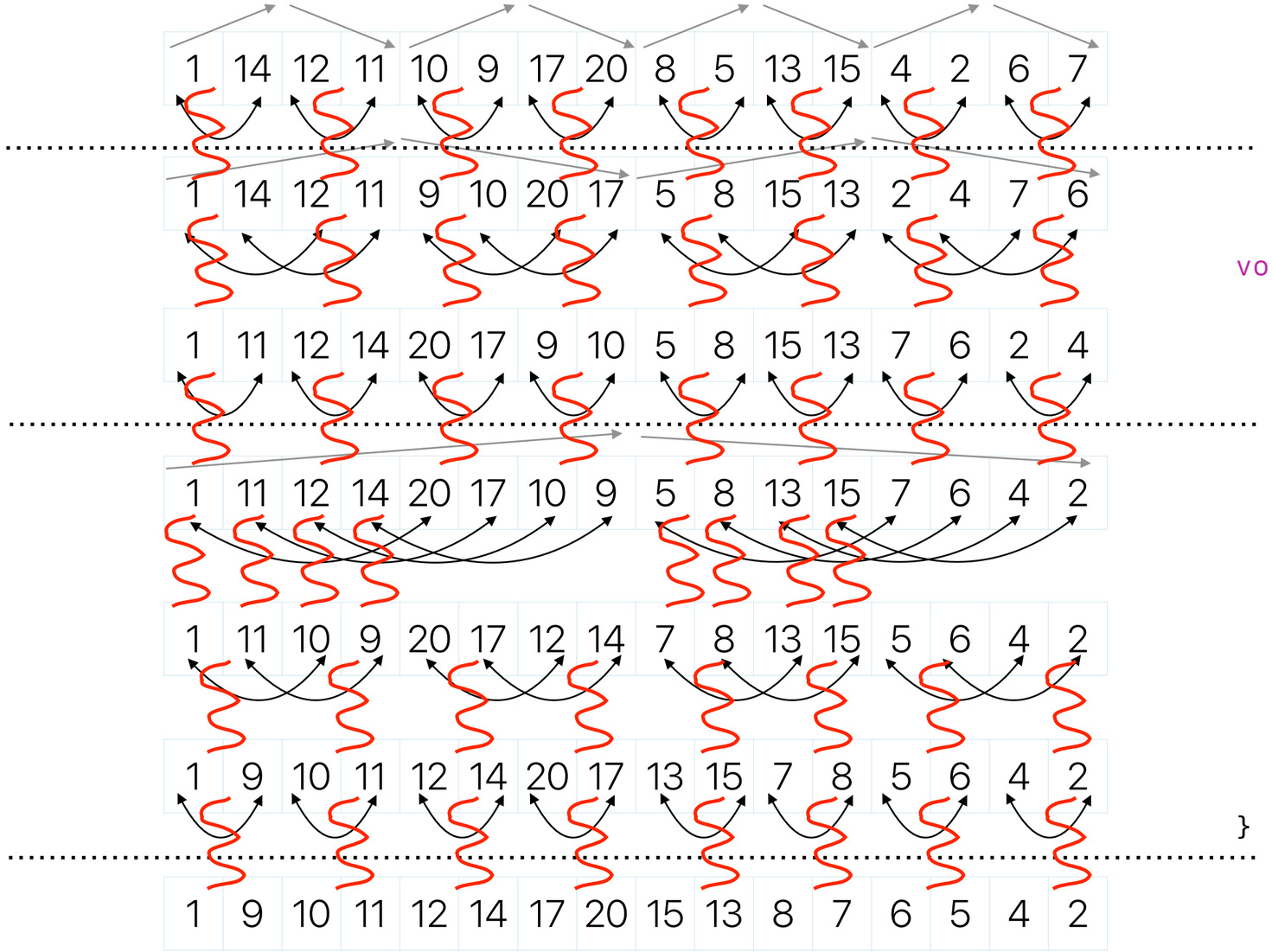



So the Amdahl's Law's evaluation will become

$$\frac{1}{\frac{1}{lg(n)} + \frac{1}{lg(n)\times 2} + \frac{1}{lg(n)\times 4} + \ldots + \frac{1}{lg(n)\times 2^{lg(n)-1}}} = \frac{1}{\frac{1}{lg(n)}(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{lg(n)-1}})}$$

$$= \frac{1}{\frac{1}{lg(n)}(1 + 1 - \frac{1}{2^{lg(n)-1}})} = \frac{1}{\frac{2}{lg(n)}} = \frac{lg(n)}{2}$$

# Bitonic sort

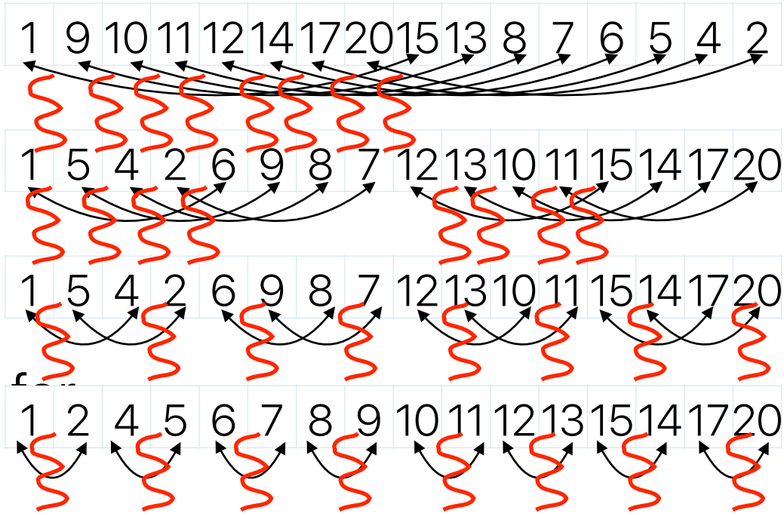



```
void BitonicSort() {

    int i,j,k;

    for (k=2; k<=N; k=2*k) {
        for (j=k>>1; j>0; j=j>>1) {
            for (i=0; i<N; i++) {
                int ij=i^j;
                if ((ij)>i) {
                    if ((i&k)==0 && a[i] > a[ij])
                        exchange(i,ij);
                    if ((i&k)!=0 && a[i] < a[ij])
                        exchange(i,ij);
                }
            }
        }
    }
}
```

# What's the speedup of bitonic sort using Amdahl's Law

The degree of parallelism is always $\frac{n}{2}$

at step $1,\ 2,\ 3,\ \ldots,\ log_2^2(n)$




The ideal speedup of each step is $\frac{n}{2}$ if we have **unlimited** parallelism & for each step of bitonic sort. However, bitonic sort will have lg(n)x more steps than merge sort.

If the baseline is merge sort — the speedup of using bitonic sort is

$$\frac{1}{\frac{1 \times lg(n)}{\frac{n}{2}}} = \frac{n}{2lg(n)} > \frac{lg(n)}{2}$$

# Q10: What if we have only p processors?

For **bitonic sort**,The degree of parallelism is always $\frac{n}{2}$

at step $1,\ 2,\ 3,\ \ldots,\ log_2^2(n)$, but we have only p processors...

The theoretical speedup of each step is $max(\frac{n}{2}, p) = p$ since n is very likely to be larger than **p** & assume equal amount of time in each step in the baseline

So the Amdahl's Law's evaluation will become $\frac{1}{\frac{p}{lg(n)}} = \frac{p}{lg(n)} = \frac{1024}{30} = 34.1333333$

**What about merge sort? (Q10)** $= \frac{1}{\frac{1}{lg(n)}(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{p})}$

What if p=1024, n=1M=$2^{30}$? $= \frac{1}{\frac{1}{30}(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{1024} + 20 \times \frac{1}{30})} = 14.862119$

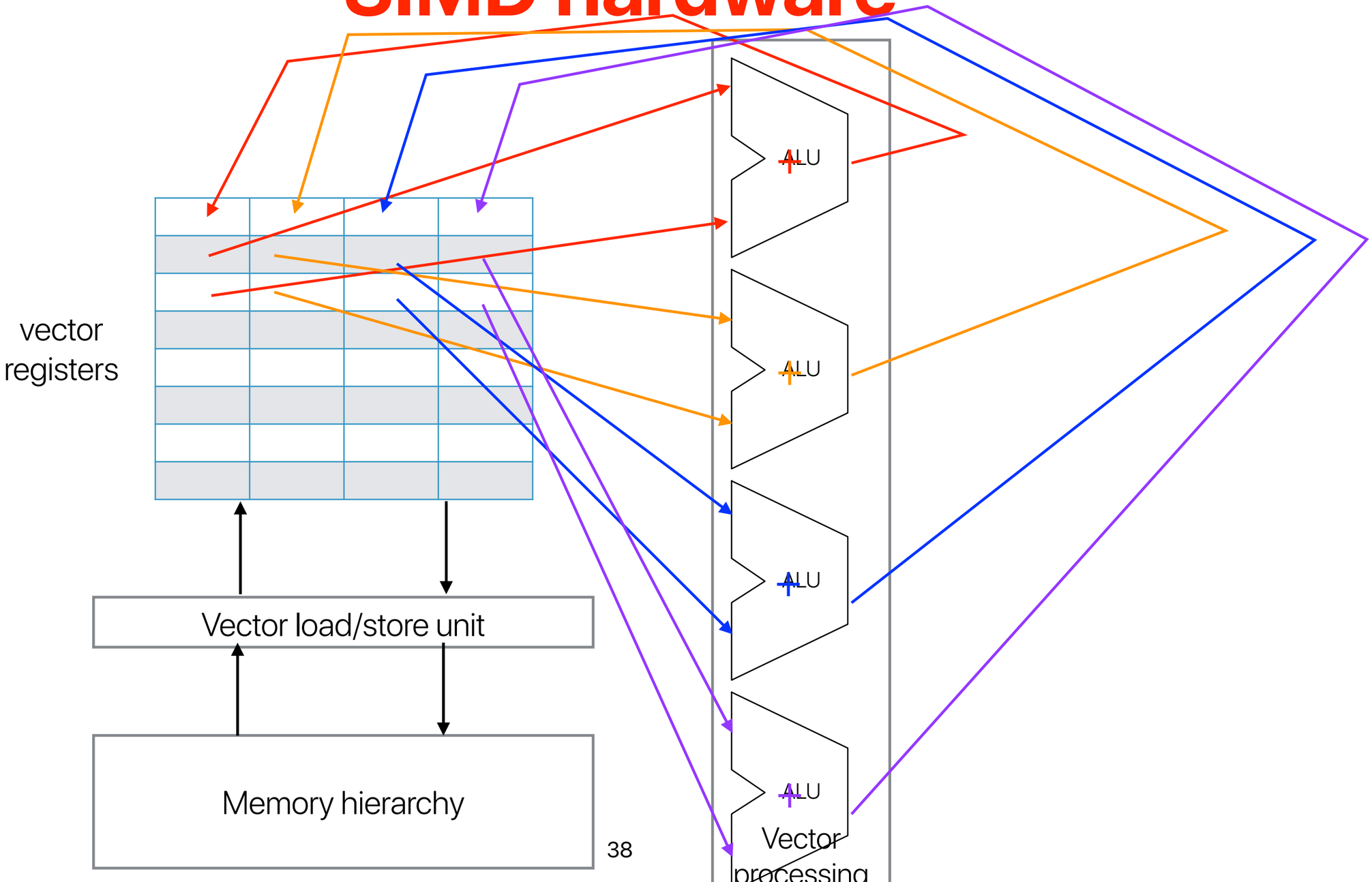
# Data level parallelism

# SIMD in processors

- SIMD: Single instruction, multiple data
  - Each instruction can perform operations on several datasets concurrently
- Streaming SIMD Extensions (SSE) that allows x86 processor architectures to support “SIMD” execution model
- ARM’s NEON

$$\begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{bmatrix} + \begin{bmatrix} 5.0 \\ 6.0 \\ 7.0 \\ 8.0 \end{bmatrix} = \begin{bmatrix} 6.0 \\ 8.0 \\ 10.0 \\ 12.0 \end{bmatrix}$$

# SIMD hardware



# Streaming SIMD Extensions (SSE)

- SSE, introduced by Intel in 1999 with the Pentium III, creates eight new 128-bit registers
  - Added 8 128-bit registers — XMM0-XMM7
  - You may use each register to store
    - 2 double-precision floating point numbers
    - 4 single-precision floating point numbers
  - Extended since introduced — SSE2, SSE3, SSE4, SSE4.1, SSE4.2, SSE4a
- They are processor-dependent instructions
  - AMD RyZen supports SSE4a, SSE4.1, SSE4.2
  - intel Core i7 doesn't support SSE4a
  - VIA Nano only support SSE4.1

# Matrix multiplication with SSE4

```
void vector_blockmm(double **a, double **b, double **c)
{
  int i,j,k, ii, jj, kk, x;
  __m256d va, vb, vc; // compiler would allocate a register as long as these variables can fit
  for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
      for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
          for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
              for(ii = i; ii < i+(ARRAY_SIZE/n); ii++) {
                  for(jj = j; jj < j+(ARRAY_SIZE/n); jj+=VECTOR_WIDTH) {
                      vc = _mm256_load_pd(&c[ii][jj]);   // load values into a vector register

                      for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
                      {
                          va = _mm256_broadcast_sd(&a[ii][kk]);  // load one value & fill the vector register
                          vb = _mm256_load_pd(&b[kk][jj]);          // load values into a vector register
                          vc = _mm256_add_pd(vc,_mm256_mul_pd(va,vb));
                                                                    // vector multiplication
                      }
                      _mm256_store_pd(&c[ii][jj],vc);
                  }                                                 // store values into a vector register
              }
          }
      }
  }
}
```

# Using OpenMP for both TLP/DLP

# Most of time, we want to parallelize just iterations in a loop: OpenMP

- They all start with "`#pragma omp`"
- "#pragma ..." is a way to add arbitrary extension to C/C++
- #pragma omp parallel for
  - Run the following for loop with multiple threads.
  - The loop needs to be pretty simple.
  - Something like "for(int i = C; i < K; i+=B)"
    - K and B need to fixed for the execution of the loop
    - Otherwise, the compiler will ignore the #pragma
  - Apply to one loop
    - Nesting is not productive
    - Use an outer loop – Bigger chunks of work for the threads.

# Parallelism OpenMP exploits for you

- MIMD parallelism
  - `#pragma omp parallel` for for parallelizing loops.
  - `int omp_get_thread_num(void)` can help you know which thread you're
- SIMD parallelism
  - `#pragma omp simd` for vectorizing loops
- No parallelism
  - `#pragma omp critical` for critical section — serialized execution.

# Histogram in OpenMP

```
extern "C"
uint64_t* run_openmp_histogram(uint64_t thread_count, uint64_t * data,
uint64_t size, uint64_t arg1, uint64_t arg2, uint64_t arg3) {

   for(int i =0; i < 256;i++) {
      histogram[i] = 0;
   }

#pragma omp parallel for
   for(uint64_t i = 0; i < size; i++) {
      for(int k = 0; k < 64; k+=8) {
         uint8_t b = (data[i] >> k)& 0xff;
#pragma omp critical
         histogram[b]++;
      }
   }
   return data;
}
```

# Continue from PA 2

- Same problem as last time.
- Now with threads and vectors
- Speedup targets — 24x on Gradescope
- You’re supposed to use OpenMP
  - You may try other available tools on Gradescope/datahub, we won’t provide technical support.

Computer
Science &
Engineering

142L

つづく