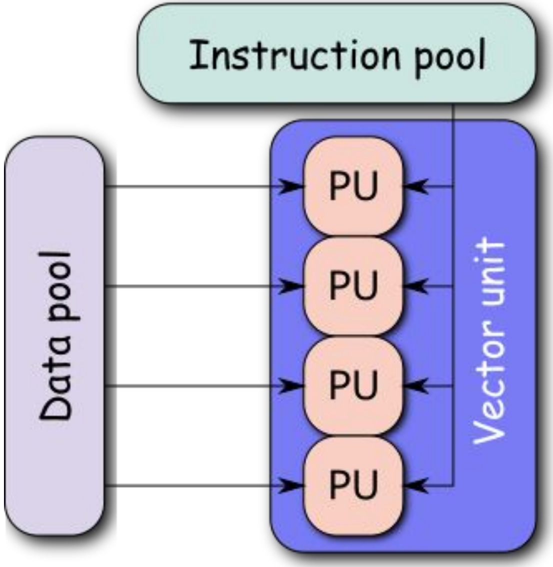
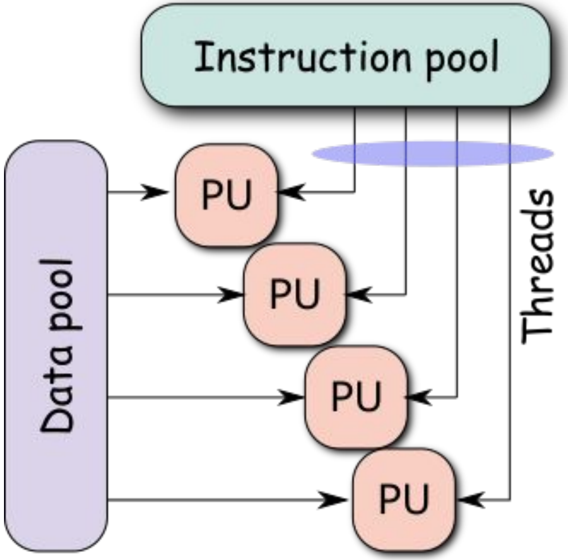
Lab 9

CS61C

Data-level (Lab 8) vs Thread-level (Lab 9) Parallelism



- 1 core, parallel ALUs



- >1 thread, 1 ALU/thread
- Threads can run on different cores

https://en.wikipedia.org/wiki/Single_instruction,_multiple_data

OpenMP

- **O**pen specification for **m**ulti**p**rocessing
- Enables us to easily parallelize code
- Invoked using compiler directives

OMP Example

declares that the directive is for OpenMP

says that the following block should be executed in parallel by different threads

Tells the compiler that this is a compiler directive

```
int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("hello world from thread %d\n", thread_id);
    }
}
```

Every single thread is going to execute this block!

Vector Addition

```
void v_add(double* x, double* y, double* z) {
    #pragma omp parallel
    {
        for(int i=0; i<ARRAY_SIZE; i++)
        {
            z[i] = x[i] + y[i];
        }
    }
}
```

Every single thread is going to execute this loop!

This is not what we want - we want the threads to split up the work of the loop

Vector Addition

```
void v_add(double* x, double* y, double* z) {
    #pragma omp parallel for
    for(int i=0; i<ARRAY_SIZE; i++)
    {
        z[i] = x[i] + y[i];
    }
}
```

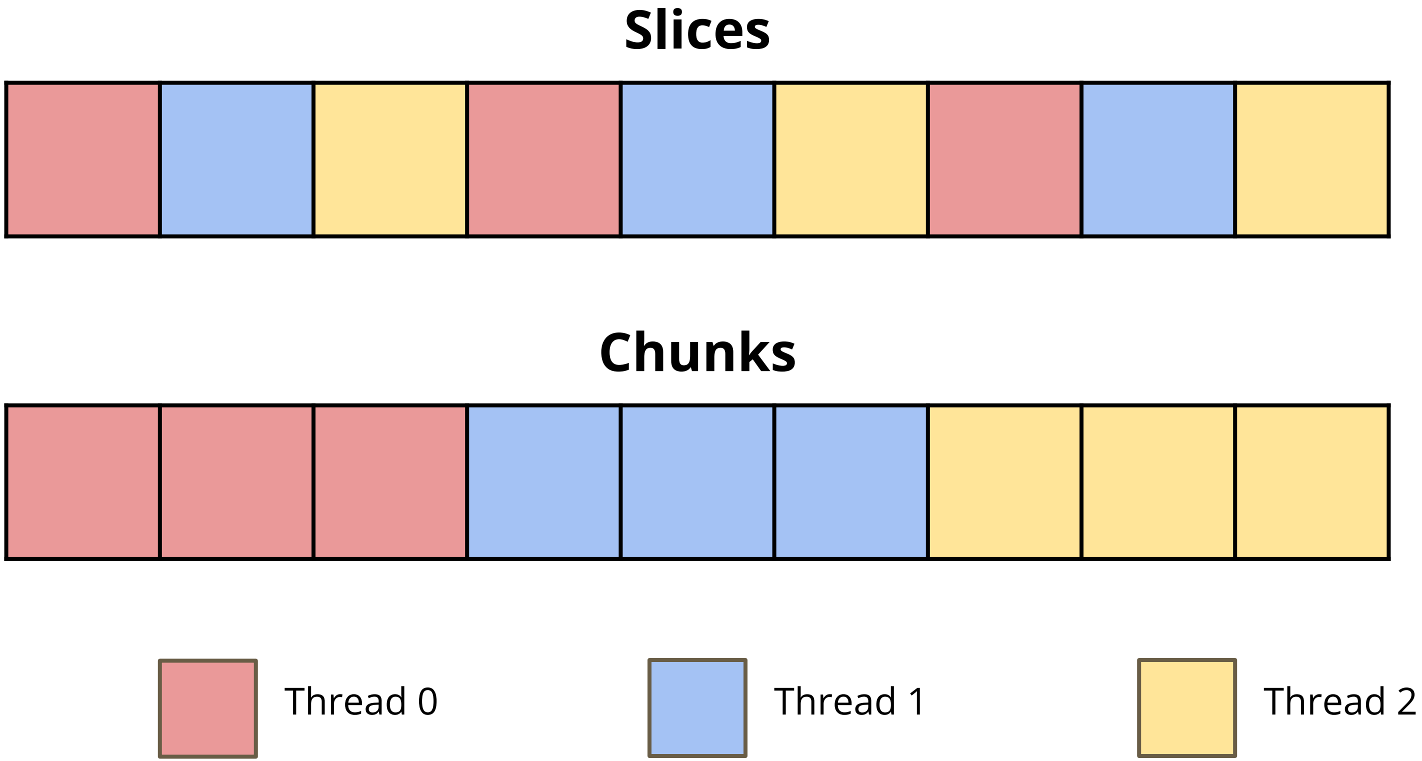
This will split up the loop for us

Very convenient but let's not use it for now :)

Exercise 2

- You will be manually parallelizing the for loop using omp
  - You can use `#pragma omp parallel`
  - You cannot use `#pragma omp parallel for`
- Useful functions
  - `int omp_get_num_threads()` - returns the current total number of OpenMP threads. Note that the number of threads will be 1 outside of an OpenMP parallel section
  - `int omp_get_thread_num()` - returns the thread number of the current thread, commonly used as thread ID

Exercise 2



Do Exercise 2

Using #pragma omp parallel for

```
void v_add(double* x, double* y, double* z) {
    #pragma omp parallel for
    for(int i=0; i<ARRAY_SIZE; i++)
    {
        z[i] = x[i] + y[i];
    }
}
```

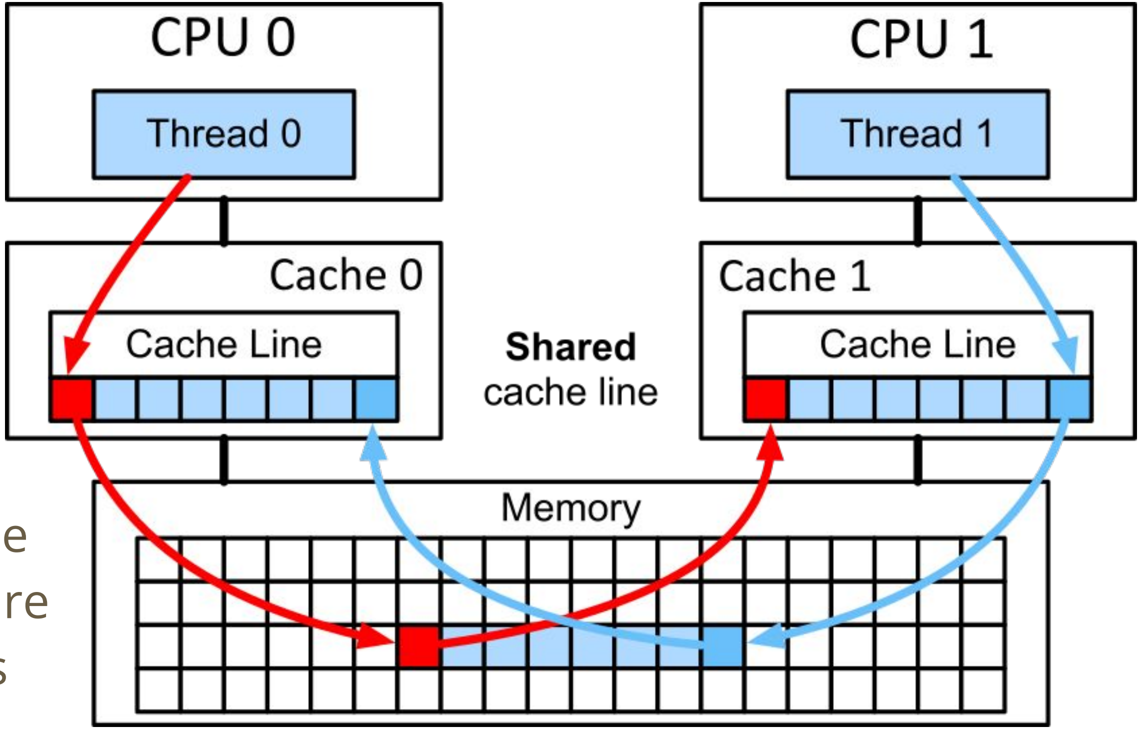
- That code actually divides the for loop into <u>chunks</u> (part 2 of exercise 2).
- For large array sizes, chunking minimizes/avoids false sharing = faster
  - Array_size > num_threads*cache_block_size

False sharing

Thread 0 writing to cache 0 invalidates cache 1 copy for thread 1 (coherence miss).

Thread 1 writing to cache 1 invalidates cache 0 copy for thread 0 (coherence miss).

Both threads had to reload the cache line even though they are modifying different addresses



Synchronization

- Sometimes our threads need to write to the same location
- If multiple threads try to write to the same location at the same time, it will lead to a **data race**
  - The order of accesses is non-deterministic which can lead to different results each time you execute the program

```
double dotp_race(double* x, double* y, int arr_size) {
    double global_sum = 0.0;
    #pragma omp parallel for
        for (int i = 0; i < arr_size; i++) {
            global_sum += x[i] * y[i];
        }
    return global_sum;
}
```

What's the problem here?

Each spawned thread can overwrite the global_sum values written by other threads

Return value will be wrong!

Synchronization

- OMP provides two methods to deal with this
  - `#pragma omp critical`
    - only one thread can execute this section at a time
  - `#pragma omp for reduction(+:var_name)`
    - Whenever you execute this operation on the given variable, make sure that only one thread can execute it at one time

Synchronization

```
double dotp_naive(double* x, double* y, int arr_size) {
    double global_sum = 0.0;
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < arr_size; i++)
            #pragma omp critical
            global_sum += x[i] * y[i];
    }
    return global_sum;
}
```

This is equivalent to
#pragma omp parallel for

Separating them like this allows for local variables (per thread) to be declared

The output is correct but this will be too slow! Why?

Synchronization

```c
double dotp_race(double* x, double* y, int arr_size) {
    double global_sum = 0.0;
    #pragma omp parallel for
    for (int i = 0; i < arr_size; i++){
        #pragma omp critical
        global_sum += x[i] * y[i];
    }
    return global_sum;
}
```

Each thread will use the critical section one at a time!

Execution of the critical section is serial, no parallelism :(

Exercise 3

- Optimize the naive implementation while still using **#pragma omp critical**
  - Does every thread need to update the global sum every iteration?
  - You can declare a local variable that can be used per thread!

- Optimize using **#pragma omp for reduction**
  - What should be the argument for the reduction keyword?

Do Exercise 3

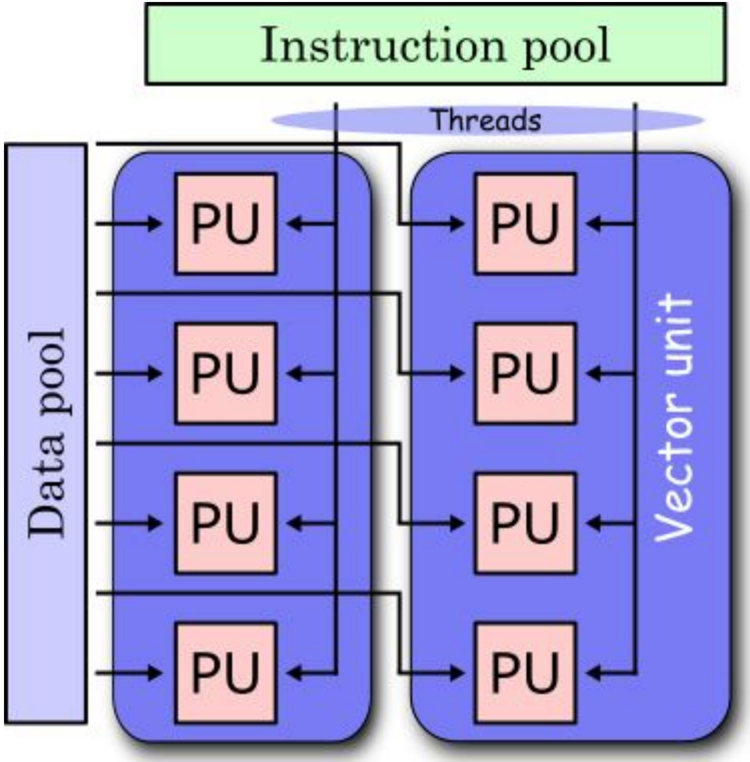
Bonus: Can we use SIMD?

```
double dotp_simd_omp(double* x, double* y, int arr_size) {
    double global_sum = 0.0;

      __m256d sum = _mm256_setzero_pd();
      for(unsigned int i = 0; i < arr_size / 4 * 4; i += 4) {
          __m256d x_vec = _mm256_loadu_pd((double *) (x + i));
          __m256d y_vec = _mm256_loadu_pd((double *) (y + i));
          __m256d temp = _mm256_mul_pd(x_vec,y_vec);
          sum = _mm256_add_pd(sum,temp);
      }

      double P[4] = {0, 0, 0, 0};
      _mm256_storeu_pd((double *) P, sum);
      global_sum += P[0] + P[1] + P[2] + P[3];
      for(unsigned int j = arr_size / 4 * 4; j < arr_size; j += 1) {
          global_sum += x[j] * y[j];
      }
    return global_sum;
}
```

Of course!

Since this is mainly an array-based problem, it can be done through vector computation

Bonus: Can we use BOTH SIMD + OMP?



If SIMD = 1 core + parallel ALUs   and

If OMP = multi-core/threads + 1 ALU

WHY NOT BOTH?

Project 4!