Parallel Computing with GPUs

OpenMP
Part 2 - Loops & Critical Sections

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- Parallelising Loops
  - Assign parallel section of code from loops to threads within OpenMP
- Critical Sections
  - Identify the potential for race conditions in parallel code
  - Examine a range of solutions for different race conditions

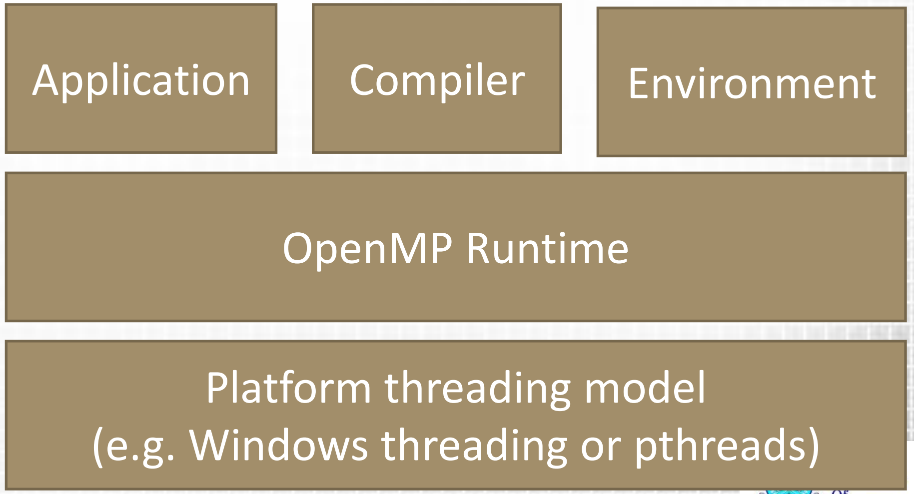
OpenMP Syntax

- Parallel region directive
  - `#pragma omp parallel [clause list] {structured block}`
  - Spawns a number of parallel threads
- Clauses
  - Are used to specify modifications to the parallel directive e.g.
    - Control scoping of variables in multiple threads
    - Dictate the number of parallel threads (example below)
    - Conditional parallelism

```
#pragma omp parallel num_threads(16)
    {
        int thread = omp_get_thread_num();
        int max_threads = omp_get_max_threads();
        printf("Hello World (Thread %d of %d)\n", thread, max_threads);
    }
```

num_threads()

- Without this clause `OMP_NUM_THREADS` will be used
  - This is an environment variable
  - Set to the number of cores (or hyperthreads) on your machine
  - This can be set globally by `omp_set_num_threads(int)`
  - Value can be queried by `int omp_get_num_threads();`
- `num_threads` takes precedence over the environment variable
- `num_threads()` does not guarantee that the number requested will be created
  - System limitations may prevent this
  - However: It almost always will



parallel for

- `#pragma omp for`
  - Assigns work units to the team
  - Divides loop iterations between threads
- For can be combined e.g. `#pragma omp parallel for`
  - Threads are spawned and then assigned to loop iterations

```
int n;
#pragma omp parallel for
for (n = 0; n < 8; n++){
    int thread = omp_get_thread_num();
    printf("Parallel thread %d \n", thread);
}
```

```
#pragma omp parallel
{
  int n;
  for (n = 0; n < 8; n++){
    int thread = omp_get_thread_num();
    printf("Parallel thread %d \n", thread);
  }
}
```

```
#pragma omp parallel
{
int n;
#pragma omp for
  for (n = 0; n < 8; n++){
    int thread = omp_get_thread_num();
    printf("Parallel thread %d \n", thread);
  }
}
```

Which is the odd one out?

parallel for

- `#pragma omp for`
  - Assigns work units to the team
  - Divides loop iterations between threa
- For can be combined e.g. `#pragm`
  - Threads are spawned and then assign

```
#pragma omp parallel
{
  int n;
  for (n = 0; n < 8; n++){
    int thread = omp_get_thread_num();
    printf("Parallel thread %d \n", thread);
  }
}
```

```
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 0
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 2
Parallel thread 5
Parallel thread 5
Parallel thread 5
Parallel thread 5
Parallel thread 4
Parallel thread 4
Parallel thread 3
Parallel thread 3
Parallel thread 1
…
```

What is wrong with this code?

- Consider a problem such as Taylor series expansion for *cos* function
  - $\cos(x) = \sum_{n=0}^{\infty} (-1)^{n-1} \dfrac{x^{2n-1}}{(2n)!}$
  - $\cos(x) = 1 - \dfrac{x^2}{2!} + \dfrac{x^4}{4!} + \dfrac{x^6}{6!} \ldots$

```
int n;
double result = 0.0;
double x = 1.0;

#pragma omp parallel for
for (n = 0; n < EXPANSION_STEPS; n++){
    double r = pow(-1, n - 1) * pow(x, 2 * n - 1) / fac(2 * n);
    result -= r;
}

printf("Approximation of x is %f, value is %f\n", result, cos(x));
```

Critical sections

- Consider a problem such as Taylor series expansion for *cos* function
  - $\cos(x) = \sum_{n=0}^{\infty}(-1)^{n-1}\ \frac{x^{2n-1}}{(2n)!}$
  - $\cos(x) = 1\ -\ \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} \ldots$

```
int n;
double result = 0.0;
double x = 1.0;

#pragma omp parallel for
for (n = 0; n < EXPANSION_STEPS; n++){
    double r = pow(-1, n - 1) * pow(x, 2 * n - 1) / fac(2 * n);
    result -= r;
}

printf("Approximation of x is %f, value is %f\n", result, cos(x));
```

Race Condition: Multiple threads try to write to the same value!
(undefined behaviour and unpredictable results)

Critical sections

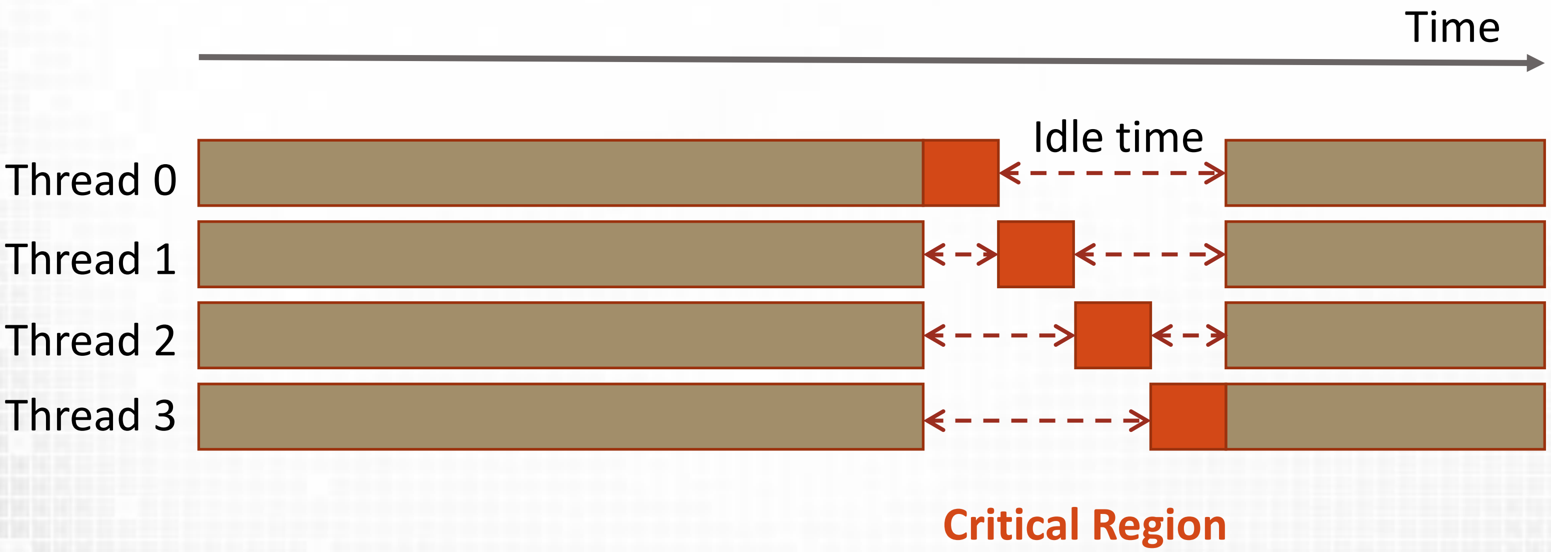
- Consider a problem such as Taylor series expansion for *cos* function
  - $\cos(x) = \sum_{n=0}^{\infty}(-1)^{n-1} \frac{x^{2n-1}}{(2n)!}$
  - $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} \dots$

```
int n;
double result = 0.0;
double x = 1.0;

#pragma omp parallel for
for (n = 0; n < EXPANSION_STEPS; n++){
    double r = pow(-1, n - 1) * pow(x, 2 * n - 1) / fac(2 * n);
    #pragma omp critical
    {
        result -= r;
    }
}

printf("Approximation of x is %f, value is %f\n", result, cos(x));
```

Solution: Define as a critical section

Critical sections

- `#pragma omp critical [name]`
  - Ensures mutual exclusions when accessing a shared value
  - Prevents race conditions
  - A thread will wait until no other thread is executing a critical region (with the same name) before beginning
  - Unnamed critical regions map to the same unspecified name



Atomics

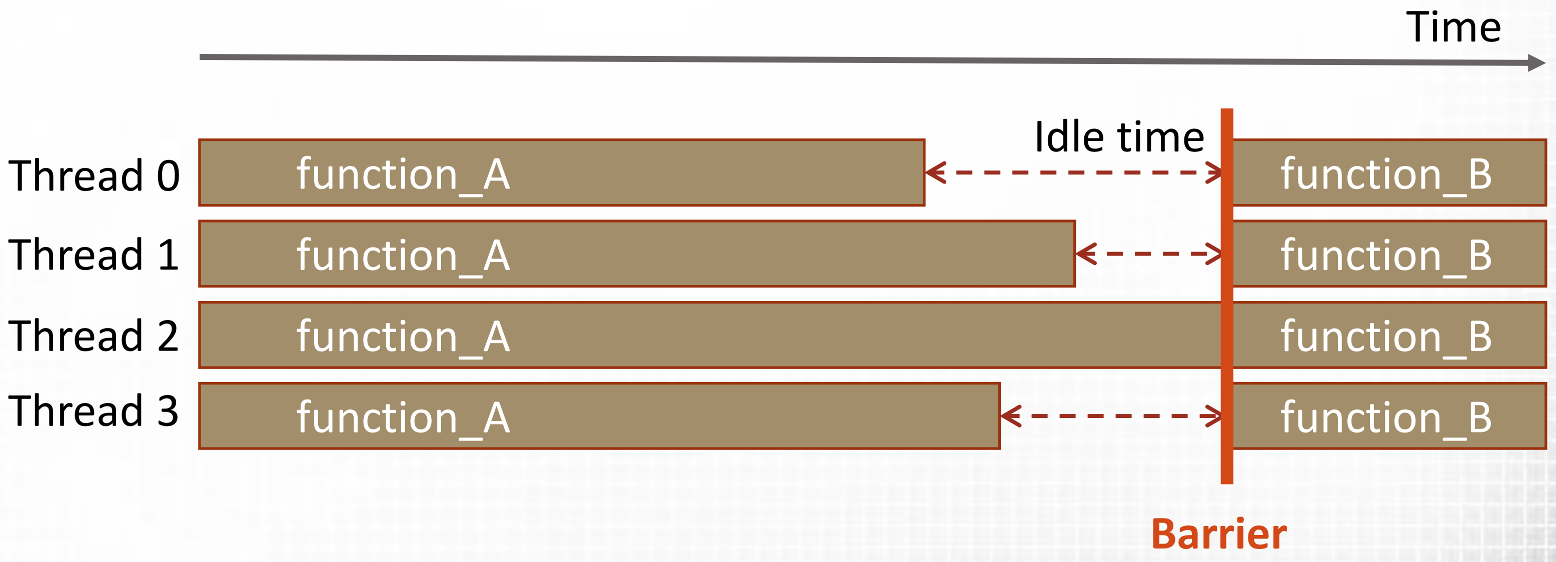
- Atomic operations can be used to safely increment a shared numeric value
  - For example summation
  - Atomics only apply to the immediate assignment
- Atomics are usually faster than critical sections (benchmark to confirm)
  - Critical sections can be applied to general blocks of code (atomics can not)
- Example
  - Compute histogram of random values for a given range
  - Random is an `int` array of size `NUM_VALUES` with random value within `0:RANGE`
  - Histogram is an `int` array of size `RANGE` with `0` values;

```
#pragma omp parallel
{
    int i;
#pragma omp for
    for (i = 0; i < NUM_VALUES; i++){
        int value = randoms[i];
#pragma omp atomic
        histogram[value]++;
    }
}
```

Barriers

- `#pragma omp barrier`
  - Synchronises threads at a barrier point
  - Parallel regions have an implicit barrier
  - Can be used to ensure execution of particular code is complete
    - E.g. data read by `function_B`

```
#pragma omp parallel
{
    function_A()
#pragma omp barrier
    function_B();
}
```



Single and Master Sections

- `#pragma omp single { … }`
  - Used to ensure that only a single thread executes a region of a structured block
  - Useful for I/O and initialisation
  - First available thread will execute the defined region
    - No control over which this is
  - Will cause an implicit barrier (after structured block) unless a `nowait` clause is used
    - E.g. `#pragma omp single nowait`
    - `nowait` will remove an implied barrier and can also be applied to parallel for loops
- `#pragma omp master { … }`
  - Similar to `single` but will always use the primary/master thread
  - Preferable to single (usually faster)
  - Does not have an implicit barrier

The University Of Sheffield.

Master example

```
int t, r;
int local_histogram[THREADS][RANGE];

zero_histogram(local_histogram);

#pragma omp parallel num_threads(THREADS)
  {
  int i;
#pragma omp for
  for (i = 0; i < NUM_VALUES; i++){
    int value = randoms[i];
    local_histogram[omp_get_thread_num()][value]++;
  }
#pragma omp barrier
#pragma omp master
  for (t = 0; t < THREADS; t++){
    for (r = 0; r < RANGE; r++){
      histogram[r] += local_histogram[t][r];
    }
  }
}
```

Same result as the atomic version

Benchmark to understand performance!

Summary

- Parallelising Loops
  - Assign parallel section of code from loops to threads within OpenMP
- Critical Sections
  - Identify the potential for race conditions in parallel code
  - Examine a range of solutions for different race conditions

- Next Lecture: Scoping and Tasks