Parallel Computing with GPUs

Optimisation
Part 3 – Memory Bound Code

The University Of Sheffield.

Dr Paul Richmond

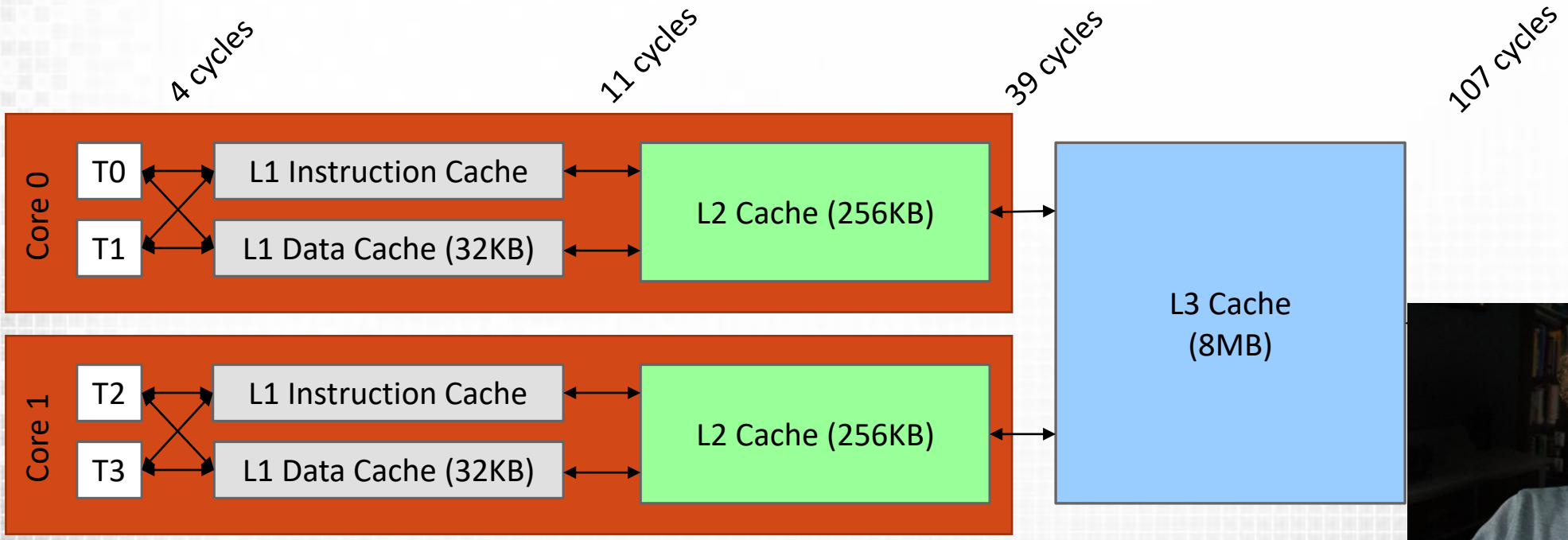
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- Memory Bound Code
  - Recognise the importance of data locality
  - Apply a series of approaches to improve memory bound code performance

Memory Bound: Optimisation

- Approach 1: Locality of data access
  - This is by far the most important consideration
  - CPU cache is small amount of very fast hierarchical memory
    - Holds contents of recently accessed memory locations
    - MUCH faster than main memory (orders of magnitude)



Memory Bound: Optimisation (Locality)

- Memory is read in cache lines of 64 bytes
  - Accessing a single bytes requires movement of the entire cache line
  - Reading patterns with common locality within cache lines reduced memory movement
    - Fewer wait (or idle) cycles
- Memory lines are pre-fetched
  - Predicable access patterns are good
  - Linear access patterns are **very** cache friendly (predictable and good locality)

Memory Bound: Optimisation

- Approach 2: Column major access
  - A special case of approach 1
    - Important for FORTRAN users.
  - Column major access has poor utilisation of cache lines
    - Despite predictability only a single value from each cache line is accessed
  - The alternative: row major access
    - Iterate the righter most index first
    - Good utilisation of the cache line

```
float array[N][M];
int i, j;

for (j = 0; j < M; j++){
    for (i = 0; i < N; i++){
        array[i][j] = 0.0f;
    }
}
```

Don’t do this!

Memory Bound: Optimisation

- Approach 3: Nice structures
  - Make your structures cache friendly
    - Multiples of cache size
    - Structures are padded: /Zp (Struct Member Alignment): default
      - **Any member whose size is less than 8 bytes will be at an offset that is a multiple of its own size based on the largest struct variable member size**
      - **any member whose size is 8 bytes or more will be at an offset that is a multiple of 8 bytes**
  - Reduce `struct` size as a result of padding
    - Arrange similar sized structure elements to avoid padding
  - Increase `struct` size to help padding
    - Add chars at the end of your structure to help it align with cache line size

What is the size of each `struct`?

```
struct sa{
    int a;
    char b;
    int c;
    char d;
};
```

```
struct sb{
    int a;
    int c;
    char b;
    char d;
};
```

Memory Bound: Optimisation

```
struct sa{        /* 16 bytes
total */
int  a;       /* 4 bytes */
char b;       /* 1 byte  */
char pad[3]; /* 3 bytes */
int  c;       /* 4 bytes */
char d;       /* 1 byte  */
char pad[3]; /* 3 bytes */
};
```

```
struct sa{
    int a;
    char b;
    int c;
    char d;
};
```

sizeof(): 16

```
struct sb{        /* 12 bytes
total */
int a;        /* 4 bytes */
int c;        /* 4 bytes */
char b;       /* 1 byte  */
char d;       /* 1 byte  */
char pad[2]; /* 2 bytes */
};
```

```
struct sb{
    int a;
    int c;
    char b;
    char d;
};
```

sizeof(): 12

Further Reading:
http://www.catb.

This Lecture (learning objectives)

- Memory Bound Code
  - Recognise the importance of data locality
  - Apply a series of approaches to improve memory bound code performance