Introduction to data parallelism

http://webloria.loria.fr/~slefebvr/teaching/pcomp

Sylvain Lefebvre
INRIA

Previously …

Execution model

- A compute device executes a *kernel* ...

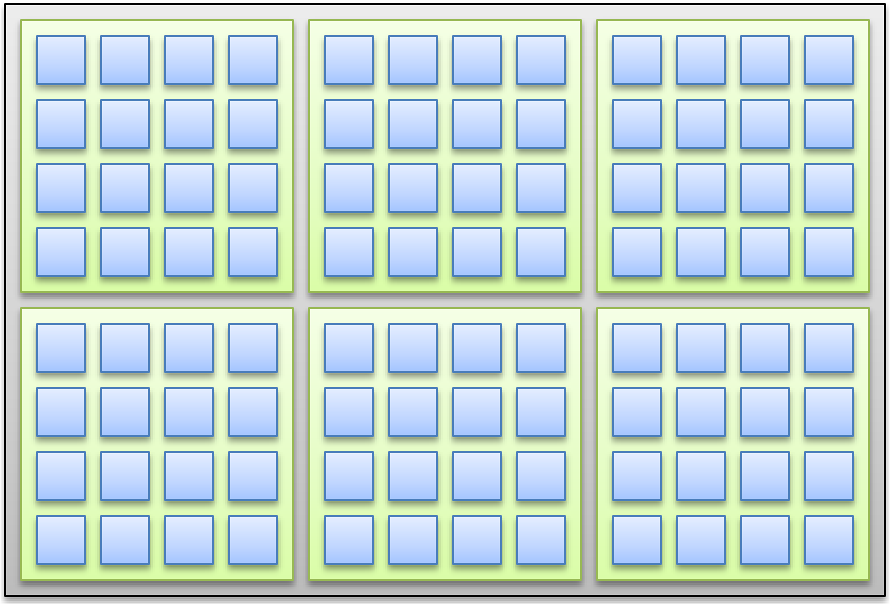
  ... in parallel on the processing elements

```
__kernel void mainKernel(
    __global const int *a,
    __global const int *b,
    __global int *c)
{
    id = get_global_id( 0 );
    c[ id ] = a[ id ] + b[ id ];
}
```

Execution model

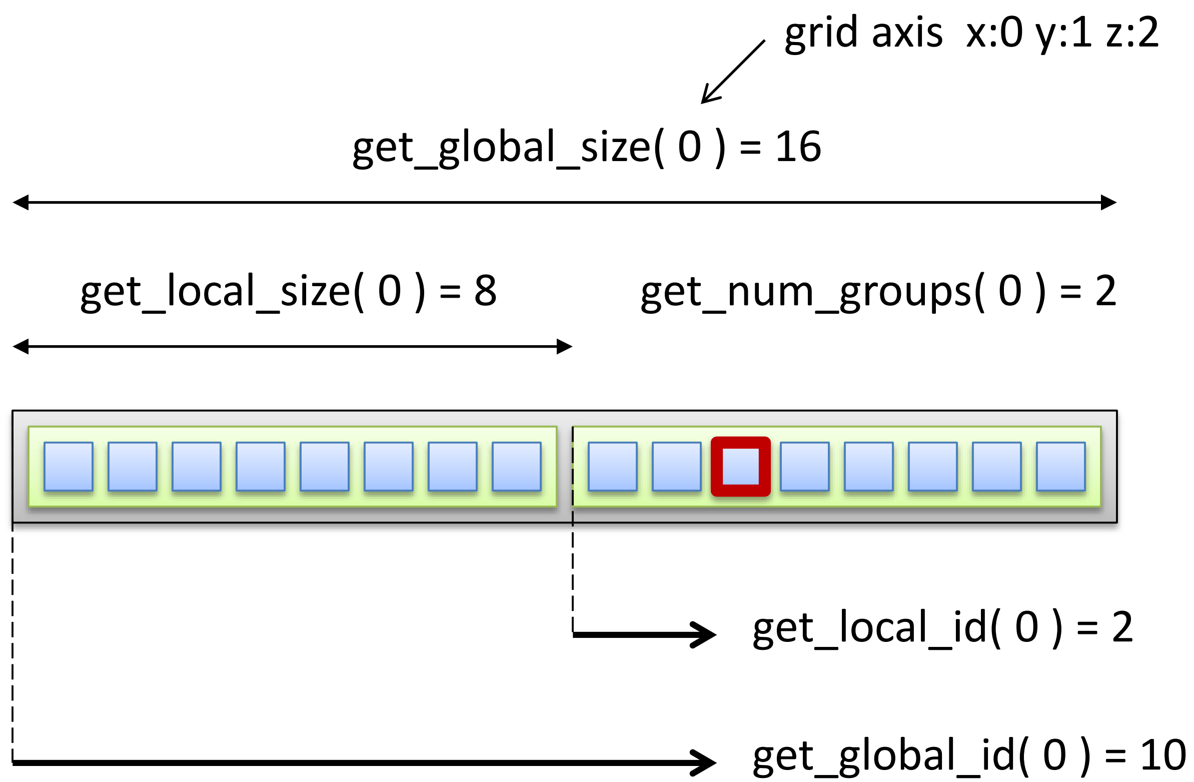
- Organized in a grid
  - 1D, 2D or 3D
- Work-items, work-groups
  - 12 x 8 work-items
  - 3 x 2 work-groups
  - Each group is 4x4 items

2D execution grid

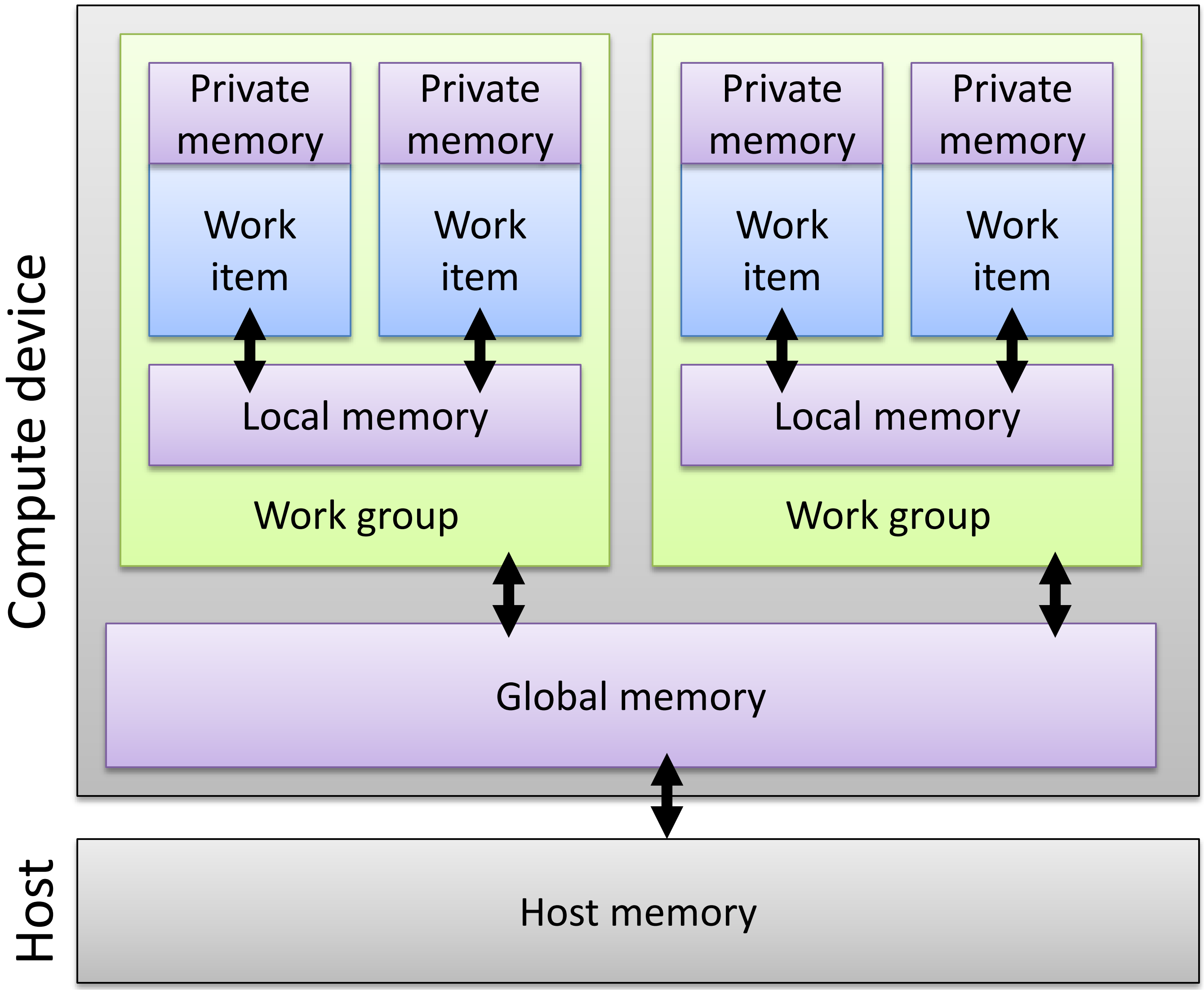


Execution kernel

- The kernel can request its location



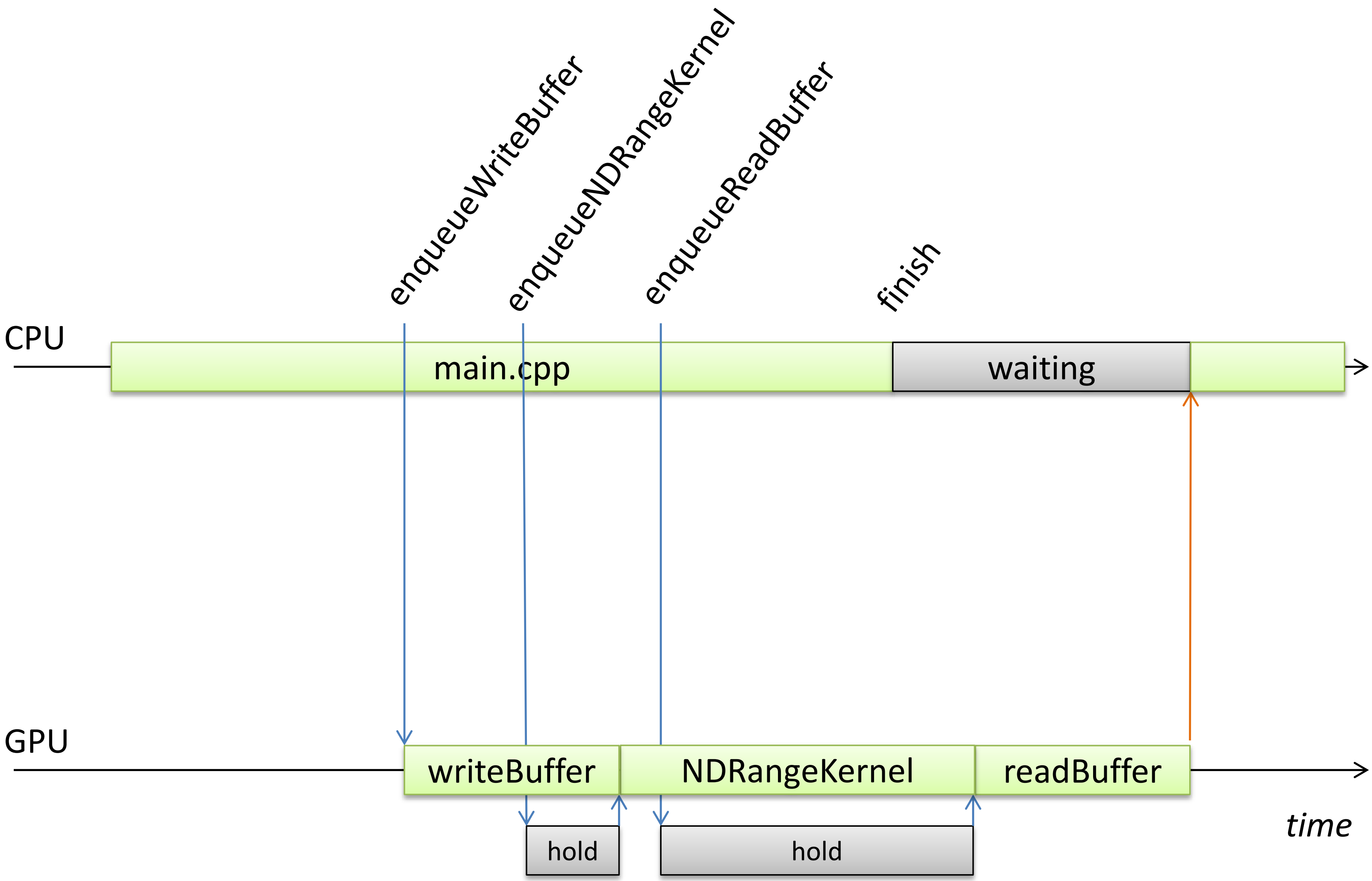
Memory model



Today

- Processing queue
- Synchronization
  - Barriers
  - Atomics
- Pre-fetch

Processing Queue



Synchronization

- So far, we avoided read/write conflicts
- Unavoidable in some cases
  - Example: Pre-fetch!

Synchronization mechanisms

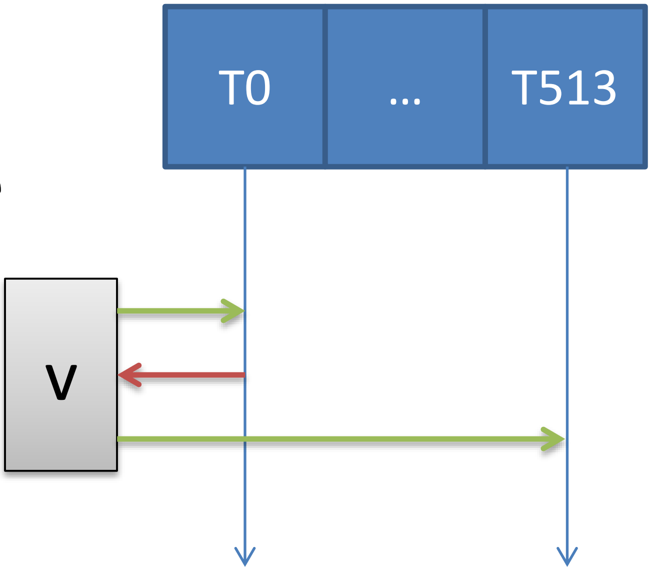
1. Global synchronization

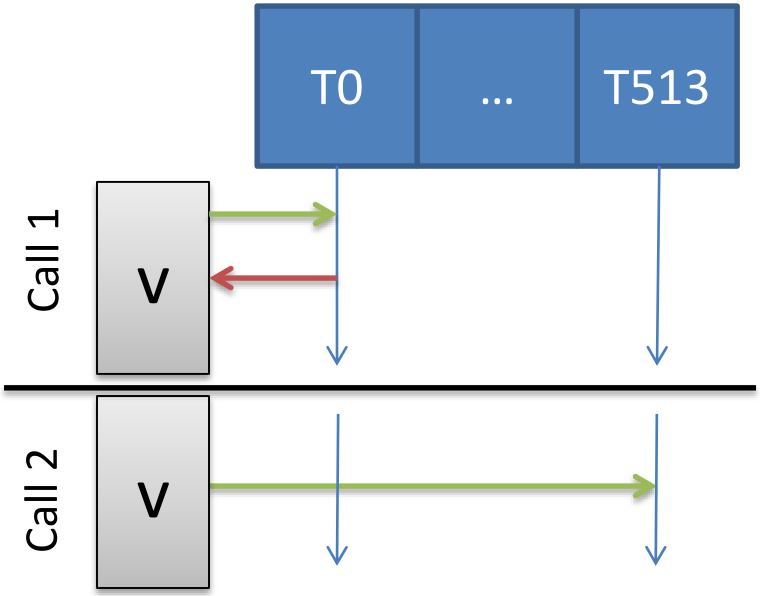
2. Memory barriers

3. Atomic operations

Global synchronization

- Typical situation:
  - loop with dependent read/write
- Synch all threads
  - Multiple kernel calls!
- Slow due to kernel restart
  - Avoid whenever possible





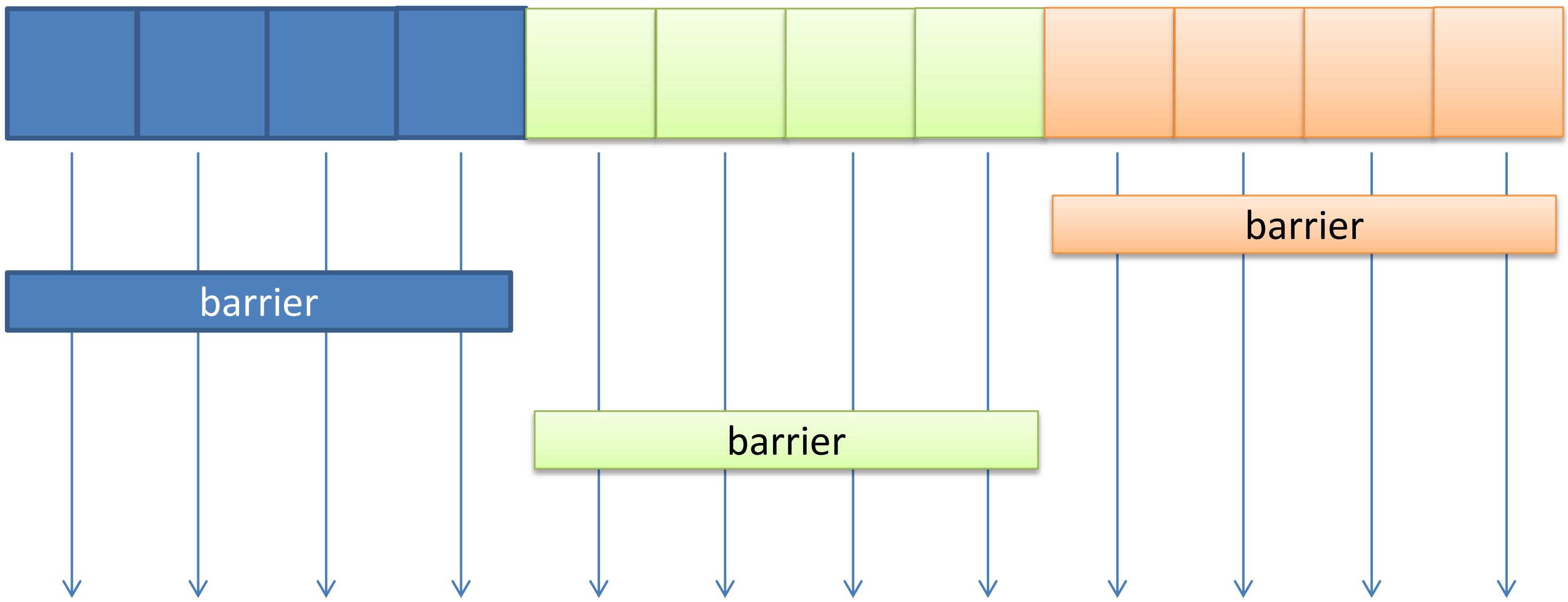
Barriers

- OpenCL instruction `barrier`
  - `barrier( CLK_LOCAL_MEM_FENCE );`
  - `barrier( CLK_GLOBAL_MEM_FENCE );`

- Guarantees that all memory operations ***within work group*** are completed

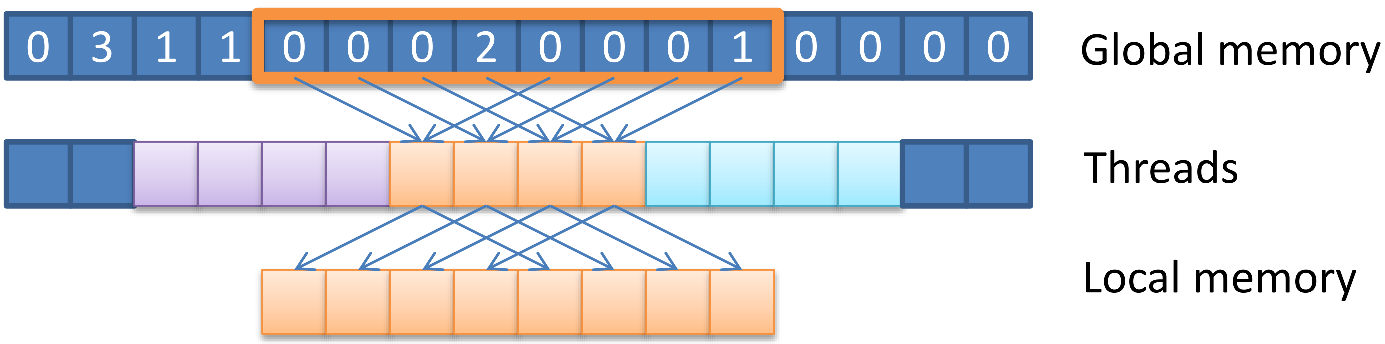
- Two flavors:
  - CLK_**LOCAL**_MEM_FENCE
  - CLK_**GLOBAL**_MEM_FENCE
  - Local synchronization is faster

Within work group!

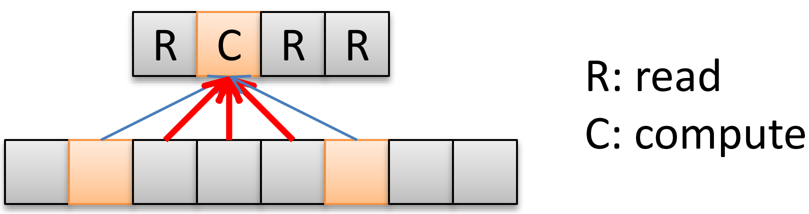


Pre-fetch and synchronization

1. Each thread in group reads some *global* data



2. Each thread computes from *local* data
   – Could start while others are not done reading



   – Use a barrier before!

Pre-fetch

```
__kernel void main(__global int *table,int N)
{
  int gid = get_global_id(0);
  int lid = get_local_id(0);
  // pre-fetch
  __local int shared[512];
  shared[lid] = table[gid];
  // sync all
  barrier(CLK_LOCAL_MEM_FENCE);
  //  compute and store in global
  // ...
  table[ gid ] = ...;
}
```

Warnings

- Execution stops until **all** thread in group reach it

- If (x) { barrier() } else { … }
  - ➔ Very likely to get stuck

- for (int i = 0 ; i < N ; i ++) { … barrier() … }
  - ➔ Ok iff N is a constant (no data dependency)

Atomic operations

- Synchronization at the hardware level
  - atomic_inc / dec
  - atomic_add / sub
  - atomic_and / or / xor
  - atomic_min / max
  - atomic_cmpxchg (compare and swap)
  - atomic_xchg

Atomic operations

- Avoids complex synchronization mechanisms
  - To be preferred whenever possible.

- No free lunch:
  - If two threads conflicts, a slow down results.
  - However, result is correct!

Final word

- Barriers
  - Avoid if possible.
  - Often required for correct results.
  - Rarely obvious, think twice!

- Atomic operations
  - Nice way to reduce synchronization.

Let’s practice!

Fix it (A)

```
__kernel void main(__global int *table,int N)
{
  int lid = get_local_id(0);
  table[ lid ] ++;
}
```

Fix it (B)

```
__kernel void main(__global int *table,int N)
{
  int id = get_global_id(0);
  for (int j = 0 ; j < 4 ; j ++) {
     int v0=0,v1=0;
     v0 = table[id];
     v1 = table[(id + N/2) % N];
     table[id] = v0 + v1;
  }
}
```

Fix it (C)

<u>Group size = 64</u>

```
__kernel void main(__global int *table,int N)
{
 int gid = get_global_id(0);
  int lid = get_local_id(0);
  int gsz = get_local_size(0);
  __local int tmp[64];
  tmp[lid] = table[ gid ];
  if (lid == 0) {
    for (int j = 0 ; j < gsz ; j ++ ) {
      tmp[lid] = tmp[lid] + table[ gid + j ];
        }
  }
  int tot = 0;
  for (int j = 0 ; j < gsz ; j ++ ) {
    tot = tot + tmp[ j ];
  }
  table[ gid ] = tot;
}
```

Fix it (D)

N <= 256

```
__kernel void main(__global int *table,int N)
{
  int gid = get_global_id(0);
  for (int j = 0 ; j < N ; j ++ ) {
      table[j] = max(table[j],gid);
  }
}
```