University of Leeds

Lecture 17: Synchroni

Previous lectures

Many of the previous lectures has mentioned **parallel synchronisation** in some form. However, there are many ways to synchronise:

- **reduction** *[Lecture 11]*.
- **Blocking communication**, which a synchronisation, in distributed memory
- ...

Also recall that GPU's have multiple memory types, some of which can be viewed as *shared* (`__global`), and some which can be viewed as *distributed* (`__local`) *[Lecture 16]*.

Today's lecture

Today's lecture will look at **synchronisation** on a GPU.

-
-

We will also see how the SIMD cores can potentially **improve** performance:

- Threads within a **subgroup** are **a** **d**.
- Threads performing different calculations can lead to **divergence** and reduced performance.

Reminder: Scalar product

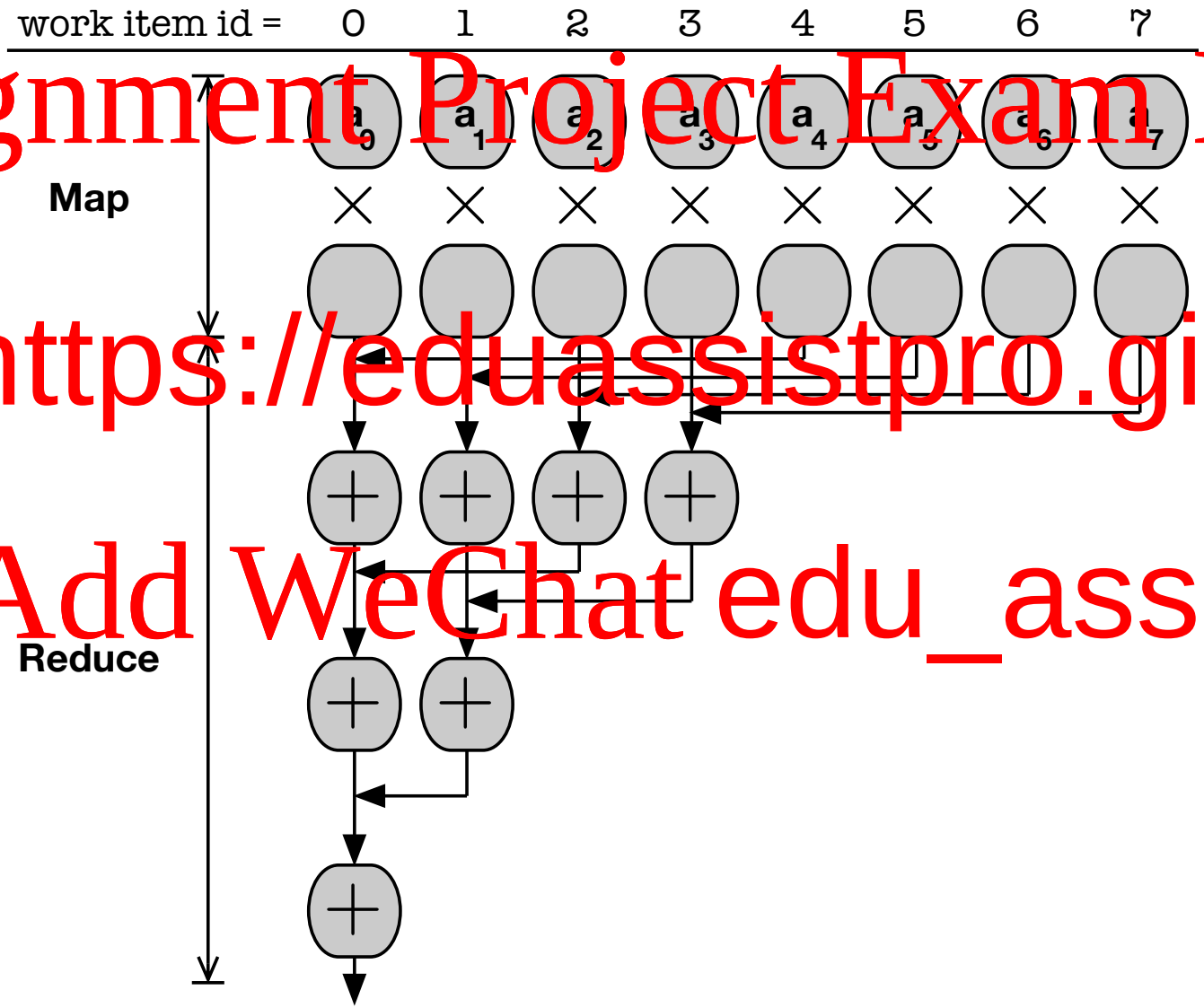
As an example, we will use the **vector product** between two n-vectors $\mathbf{a}$ and $\mathbf{b}$, as in Lecture 11.

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1} a_i b_i = a_1 b_1 + a$$

In serial CPU code *(indexing starting from 0)*:

```
1 float dot = 0.0f;
2 for( i=0; i<n; i++ ) dot += a[i] * b[i];
```

MapReduce pattern for $n = 8$



Reduction in local memory

First consider n equal to or less than the **work group size**.

Use **local memory** for the intermediate quantities *[Lecture 16]*.

- ithin a

Each work item copies `a[i]*b[i]` to local memory first.

- Reduce using the binary tree pattern on the p .

- Each addition performed by the work item w i.d.
- Final result in work item with i.d. $= 0$ copied to the answer (in global memory).

[1] Divide-by-two implemented by **compound bitwise right shift** operator '>>='.

Kernel code

Code on Minerva: `workGroupReduction.c`, `workGroupReduction.cl`, `helper.h`


```
1  __kernel
2  void reduceNoSync( __global float *device_a, __global
       float *device_b, __global float *dot, __local
3  {
4    int id
5                  = get_loc
6        groupSize = get_local_size(0);  //=work group
7
8    scratch[id] = device_a[id] * device_b[id];
9
10   for( stride=groupSize/2; stride>0; stride>>=1 )
11     if( id < stride )
12       scratch[id] += scratch[id+stride];
13
14   if(id==0) *dot = scratch[0];
15 }
```

Calling C-code


```
// float array of size 1 on device.
cl_mem device_dot = clCreateBuffer(...);

... // Set ke
clSet
  //NU

// Add to the command queue.
size_t indexSpaceSize[1]={N}, workGroupSize[1]={N};
clEnqueueNDRangeKernel(queue,kernel,1,NULL,
    indexSpaceSize,workGroupSize,0,NULL,NULL)

// Get the result back to host float 'dot'.
float dot;
clEnqueueReadBuffer(queue,device_dot,CL_TRUE,0,sizeof(
    float),&dot,0,NULL,NULL);
```

Barriers

Without synchronisation, this reduction is *not* guaranteed to work on *all* systems.

Reca
leave

-
- `MPI_Barrier()` in MPI.

In OpenCL[1], a barrier **within a work grou**

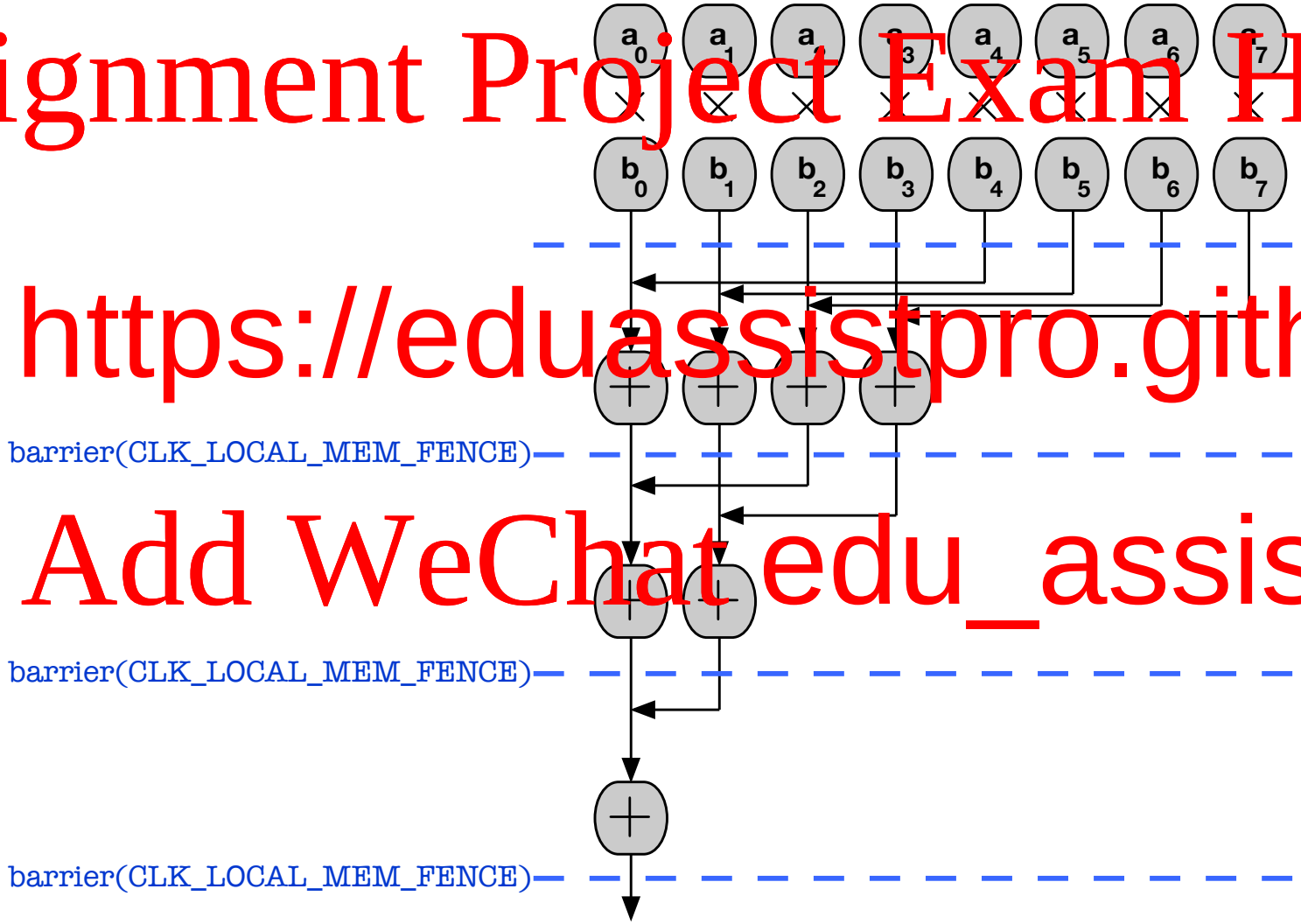
```
1 barrier(CLK_LOCAL_MEM_FENCE);
```

[1] In CUDA: `__syncthreads()` synchronises within a **thread block**=work group.

Reduction with synchronisation


```
1  void reduceWithSync(...)      // Same arguments.
2  {
3    int id=..., groupSize=..., stride; // As before.
4
5    sc
6    ba
7
8    for( stride=groupSize/2; stride>0; stride>>=1 )
9    {
10     if( id < stride )
11       scratch[id] += scratch[id+stride];
12
13     barrier(CLK_LOCAL_MEM_FENCE);     // Sync.
14   }
15
16   if(id==0) *dot = scratch[0];
17 }
```

Reduction with barrier(CLK_LOCAL_MEM_FENCE)



Problems larger than a single work group?

If we could synchronise **between** work groups, could use the same method as before:

1.
2.

How

GPUs cannot synchronise between work gro [2]

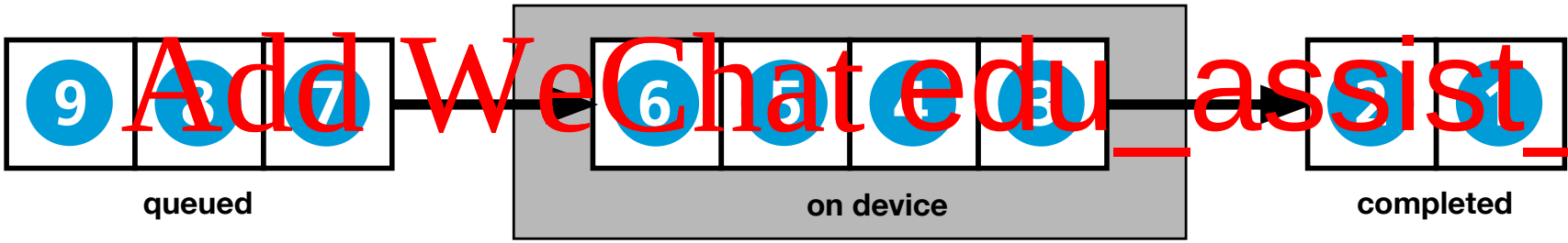
[1] `barrier(CLK_GLOBAL_MEM_FENCE)` *does* exist, but refers to *accesses* to global memory; it still only synchronises *within* a work group.
[2] Some modern GPUs support **cooperative groups** that allow synchronisation across multiple thread blocks; *e.g.* CUDA 9.0.

Warning!

You might see claims that it is possible to synchronise globally on any GPU by constantly **polling** a global memory location.

- This

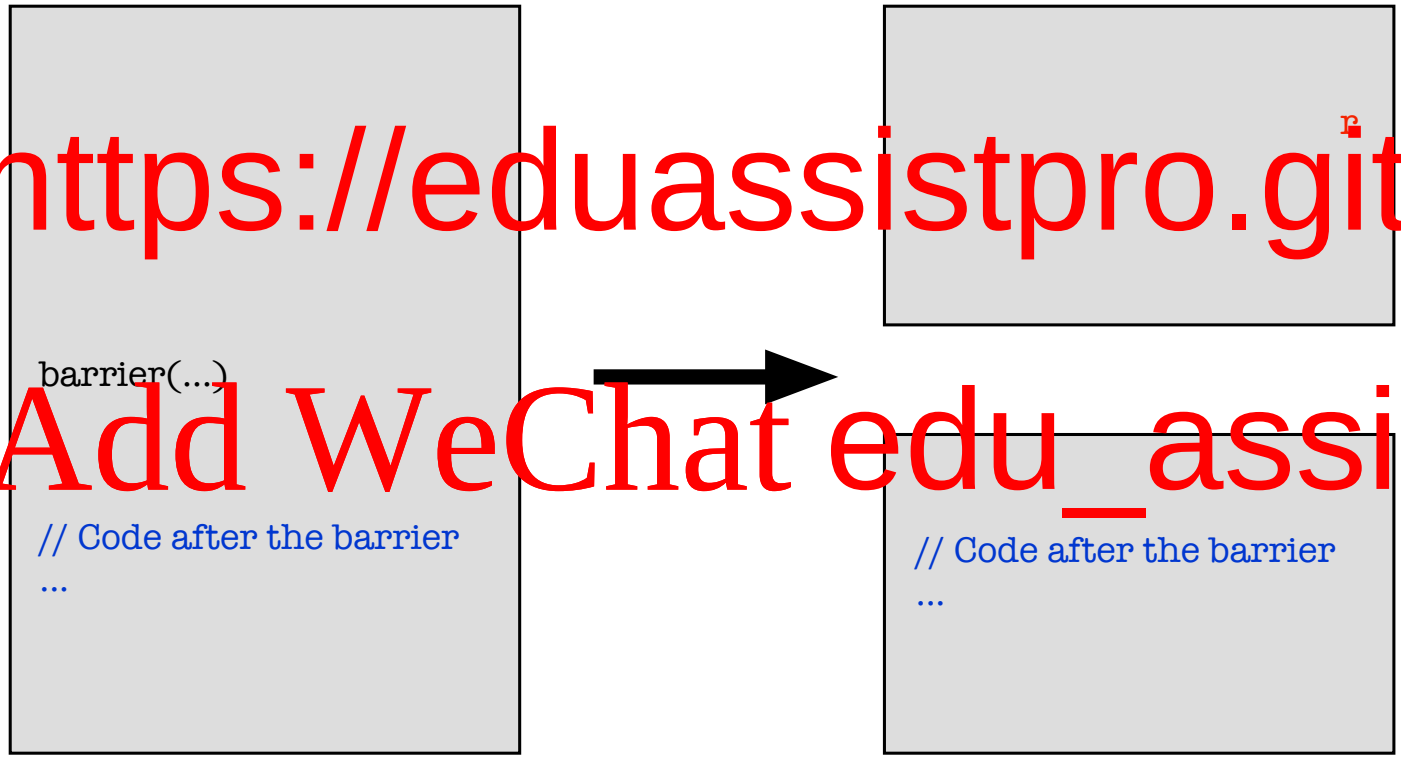
If there are too many work groups for the device, it m:



If they are not all on the device at the same time, it is **impossible to synchronise within one kernel** using this method.

Solution: Multiple kernels

The solution is to break the kernel at the barrier point into **multiple kernels** called consecutively:



This way kernel 1 completes before kernel 2 starts.

Reduction across work groups

It is possible to use this method for reduction[1]:

1. Repeatedly call kernel that reduces an array of **partial sums**
2. Final summation performed **on the**

It is sim

1. Each work group inserts its partial sum into a g

This is conceptually similar to an MPI program performing final calculations on rank 0.

[1] Wilt, *The CUDA handbook* (Addison-Wesley, 2013).

Subgroups (warp, wavefront, *etc.*)

Recall that GPUs are based on SIMD cores.

- Each core contains **multiple hardware threads** that perform

In Op

simultaneously on a single SIMD core is known as a

- **Smaller** than a work group.

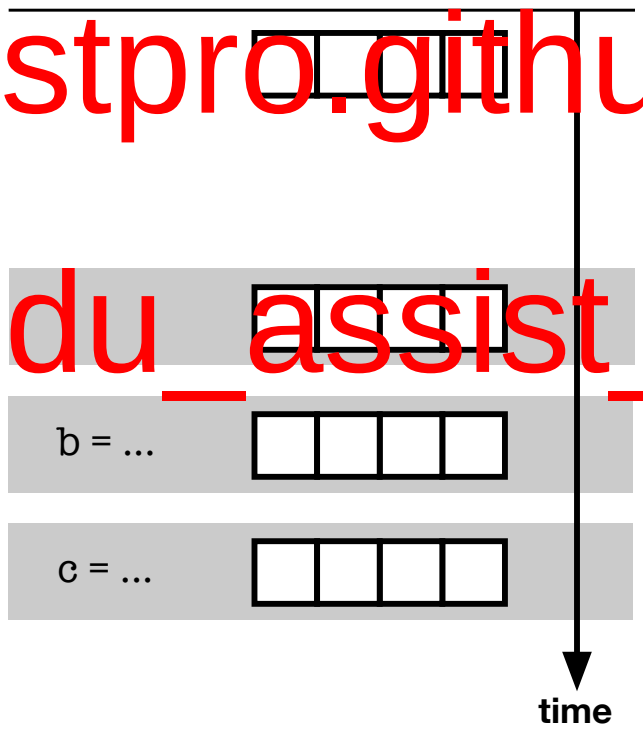
The actual size is vendor specific. For example:

- Nvidia call them **warps**, each of which has 32 threads.
- AMD have 64-thread **wavefronts**.

Lockstep

The SIMD core applies the **same** operation to all items in the subgroup **simultaneously**. We say it advances in **lockstep**.


```
1  __ker
2  void le
3  {
4    int id = get_global_id(0);
5    float a, b, c;
6
7    a = 4*array[id];
8
9    b = a*a;
10
11   c = b + a;
12 }
```

Reduction with a subgroup

For reduction, this means that once the problem has been reduced to the size of a subgroup, there is **no longer any need for explicit synchronisation**[1]:


```
1  __ker
2  void red
3  {
4    ...  // Start as before.
5
6    // Split the loop into two.
7    for(stride=group/2;stride>subgroup;stride>>
8      if(id<stride) scratch[id] += scratch[id+stride];
9      barrier(CLK_LOCAL_MEM_FENCE);      // Sync.
10   }
11   // See next slide ...
```


[1] Wilt, *The CUDA handbook* (Addison-Wesley, 2013).

Final reduction

For the final reduction, exploit **lockstepping** by removing the **synchronisation**:

```
1 for(;
2 {
3   if
4
5
6   // No barrier().
7 }
```


This avoids any overheads with calling `barrier()` (*i.e.* unnecessarily checking if all threads have reached this point, when we know they must have).

Divergence

Recall that SIMD cores can also be termed **SIMT** *[lecture 4]*.

- Single Instruction stream, Multiple **Threads**.

This
subg

- Only **one** distinct operation can be perf

This can lead to **serialisation** where op **one after the other**.

- Can lead to a severe performance penalty.

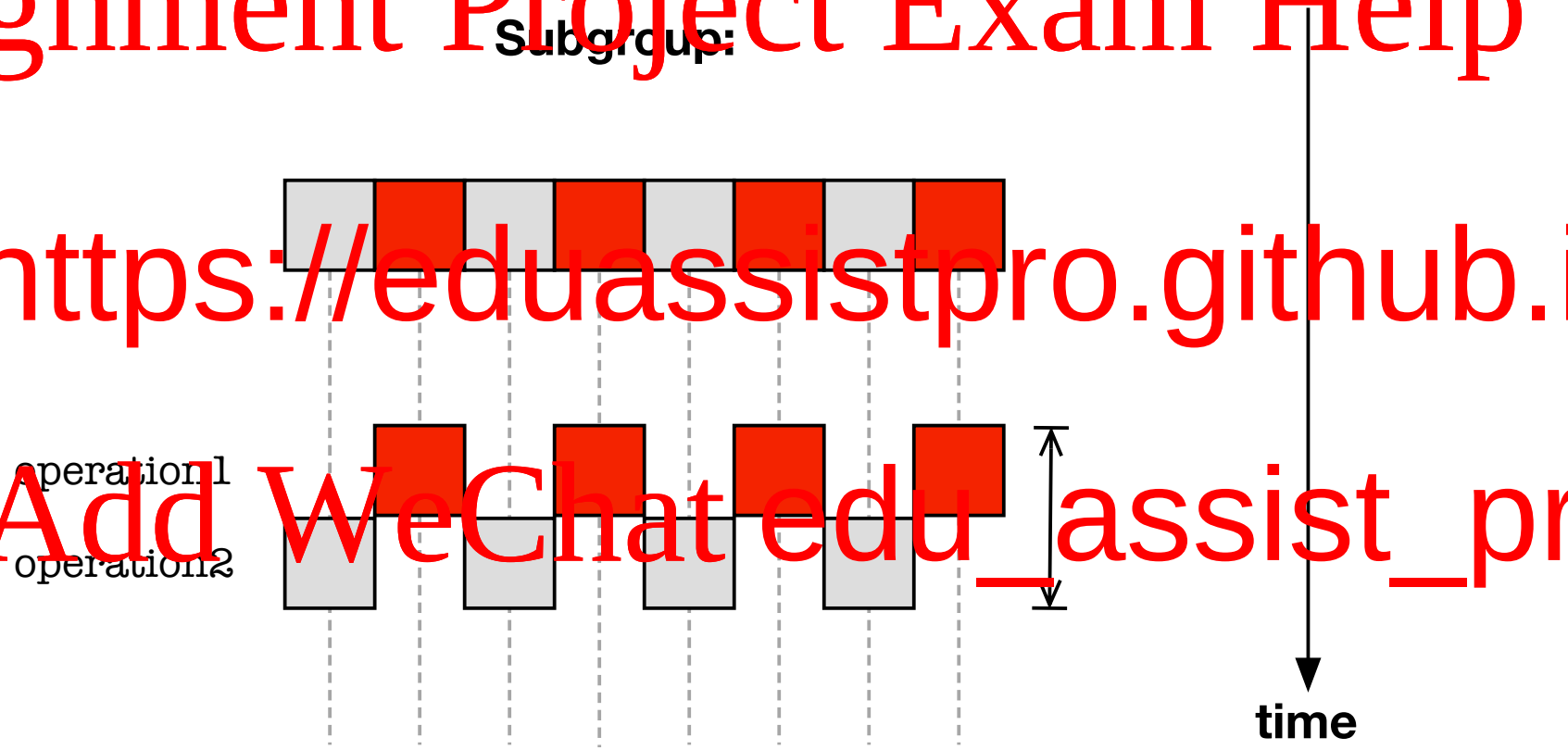
Code that leads to divergence

Suppose we want even-numbered work items to perform one operation, and odd-numbered items a different one[1]:

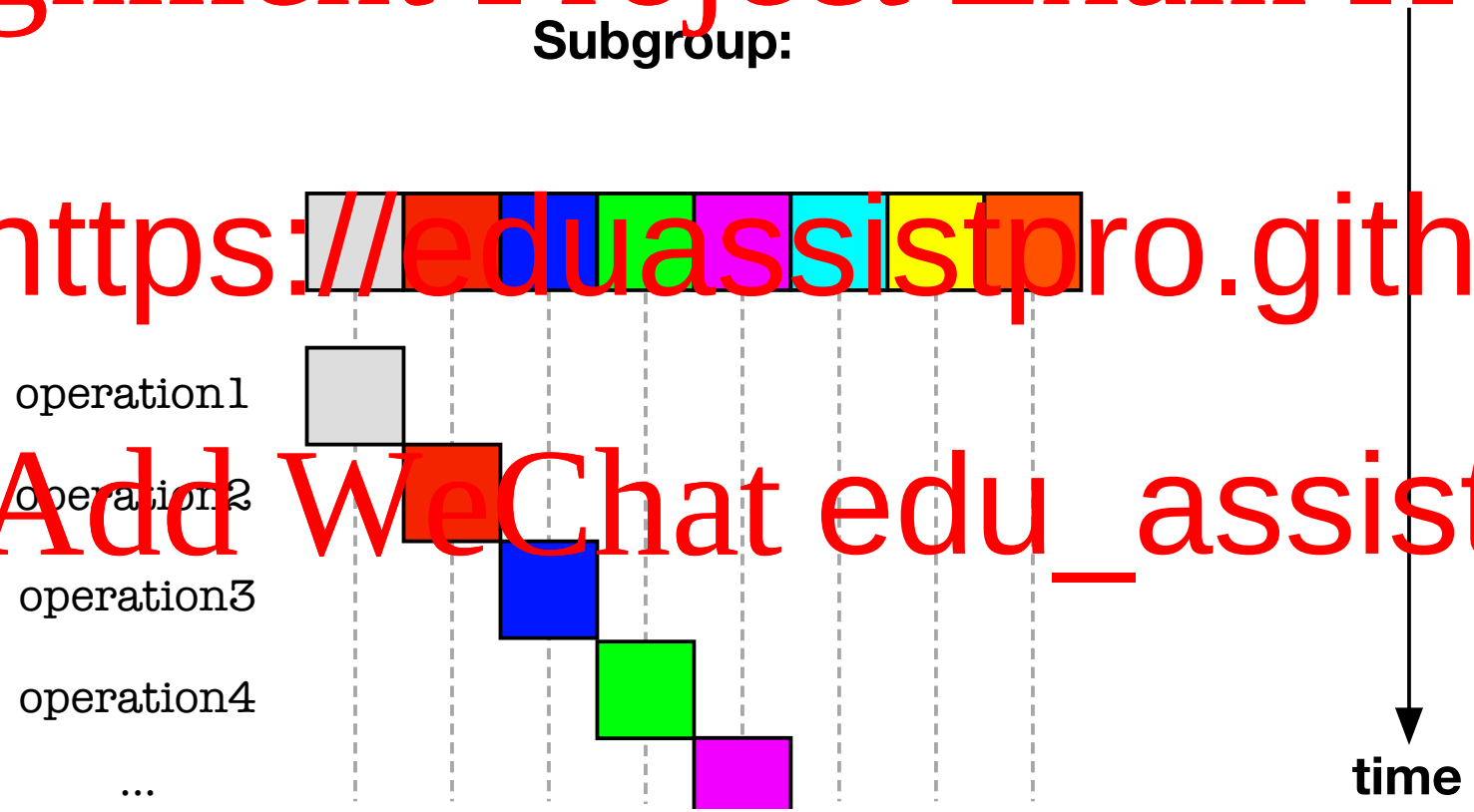

```
1  __ker
2  void ker
3  {
4    int id = get_global_id(0);
5
6    if( id%2 )
7      operation1;          // id odd.
8    else
9      operation2;          // id even.
10 }
```


[1] Recall `i%2==1` for `i` odd, `0` for `i` even.

In this example the execution time is **double** what was expected:



For more operations the execution time increases further, *e.g.* a switch-case clause where every thread performs a different operation:



This is true **serialisation**.

Summary and next lecture

Today we have looked at **synchronisation**, focussing on GPUs.

- **Barriers** can synchronise within a work group.
-
- lockstep.
  - No need for explicit synchronisation me
  - Can lead to **divergence** and redu

Next time we will look at **atomic** instructions, continuing what we started in Lecture 6.