Heterogeneous Computing using openCL lecture 3

F21DP Distributed and Parallel Technology

Sven-Bodo Scholz

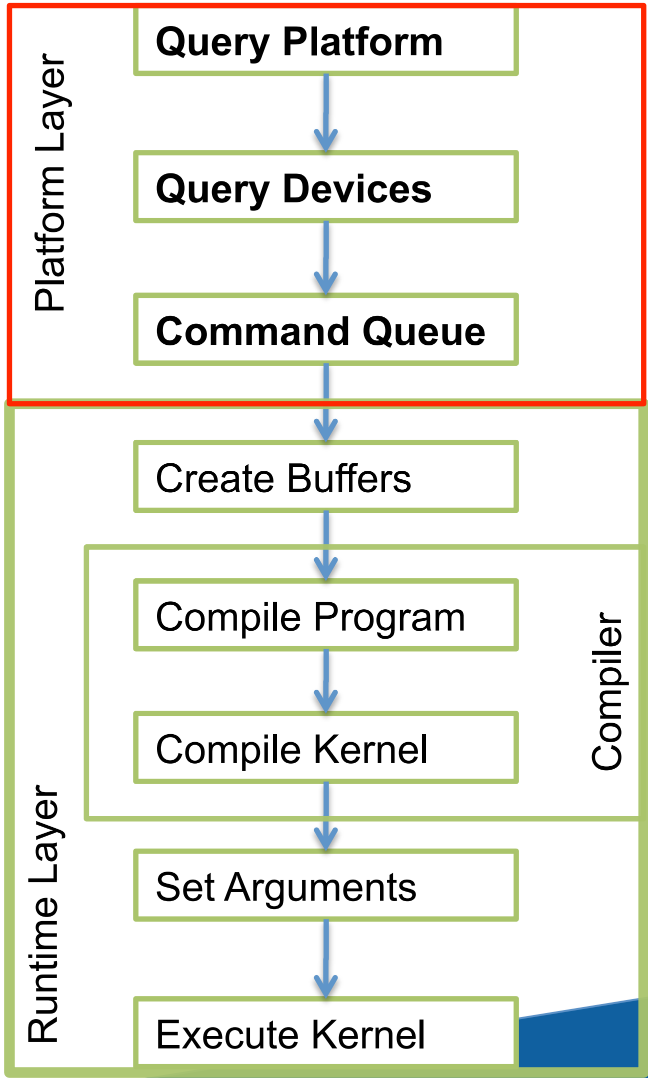
Recap: Initialise Device

- Declare context
- Choose a device from context
- Using device and context create a command queue

```
cl_context myctx = clCreateContextFromType (
                 0, CL_DEVICE_TYPE_GPU,
                 NULL, NULL, &ciErrNum);
```

```
ciErrNum = clGetDeviceIDs (0,
               CL_DEVICE_TYPE_GPU,
               1, &device,  cl_uint *num_devices)
```

```
cl_commandqueue myqueue ;
myqueue = clCreateCommandQueue(
               myctx, device, 0, &ciErrNum);
```



Recap: Create Buffers

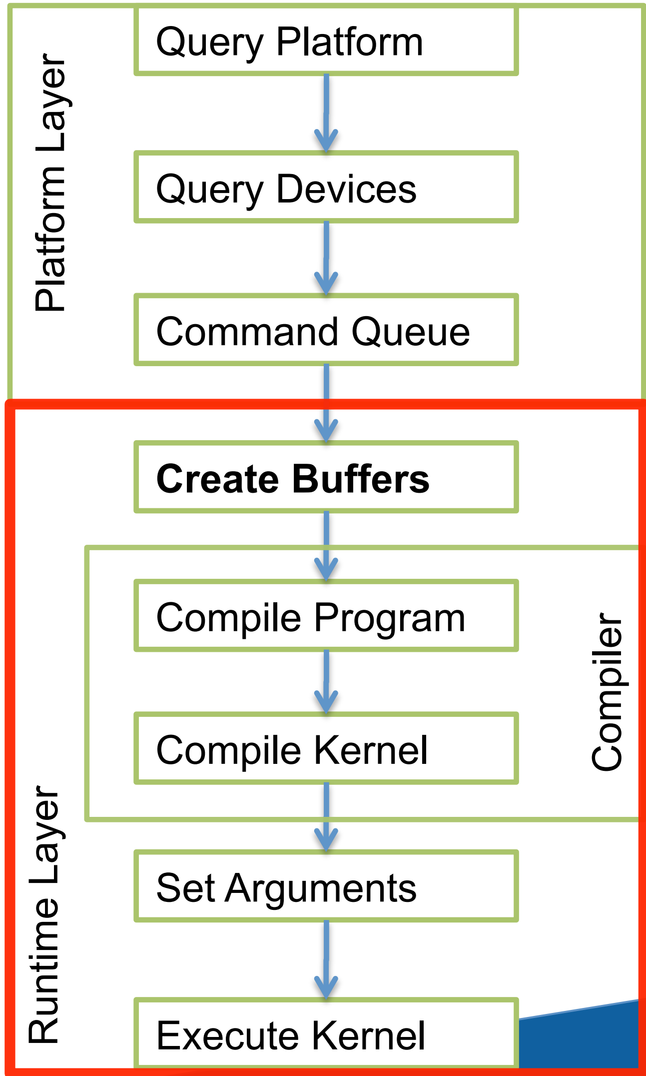
- Create buffers on device
  - Input data is read-only
  - Output data is write-only

```
cl_mem d_ip = clCreateBuffer(
                  myctx, CL_MEM_READ_WRITE,
                  mem_size,
                  NULL, &ciErrNum);
```

```
cl_mem d_op = clCreateBuffer(
                  myctx, CL_MEM_READ_WRITE,
                  mem_size,
                  NULL, &ciErrNum);
```

- Transfer input data to the device

```
ciErrNum = clEnqueueWriteBuffer (
              myqueue , d_ip, CL_TRUE,
              0, mem_size, (void *)src_vector,
              0, NULL,  NULL)
```

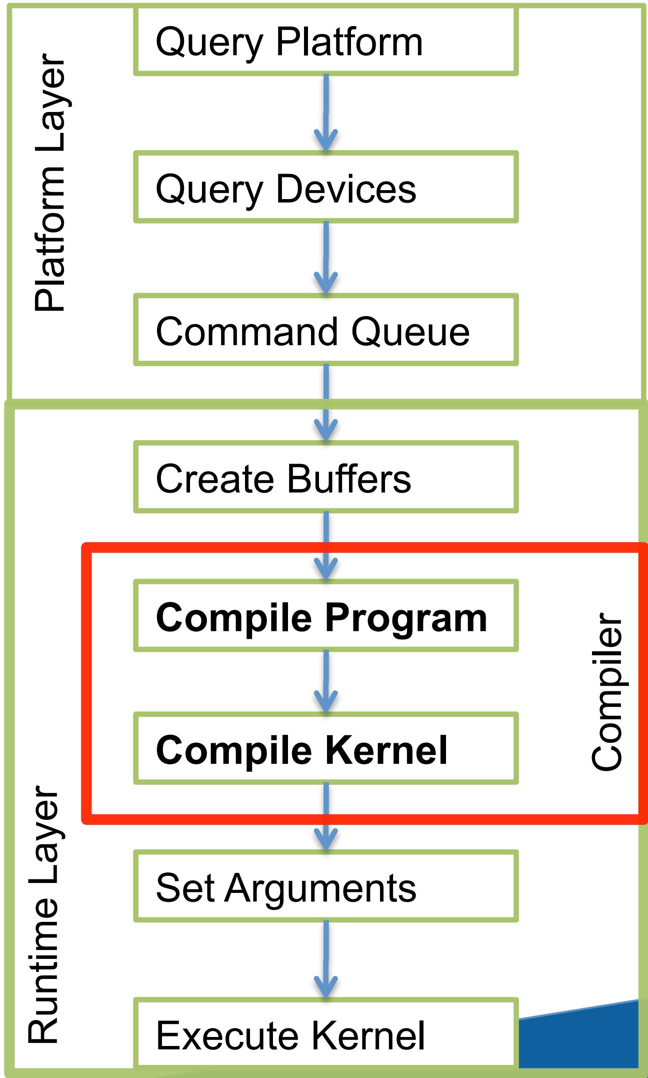


Recap: Build Program, Select Kernel

```
// create the program
cl_program myprog = clCreateProgramWithSource
                    ( myctx,1, (const char **)&source,
                    &program_length, &ciErrNum);
```

```
// build the program
ciErrNum = clBuildProgram( myprog, 0,
                    NULL, NULL, NULL, NULL);
```

```
//Use the “relax” function as the kernel
cl_kernel mykernel = clCreateKernel (
                    myprog , “relax” ,
                     error_code)
```

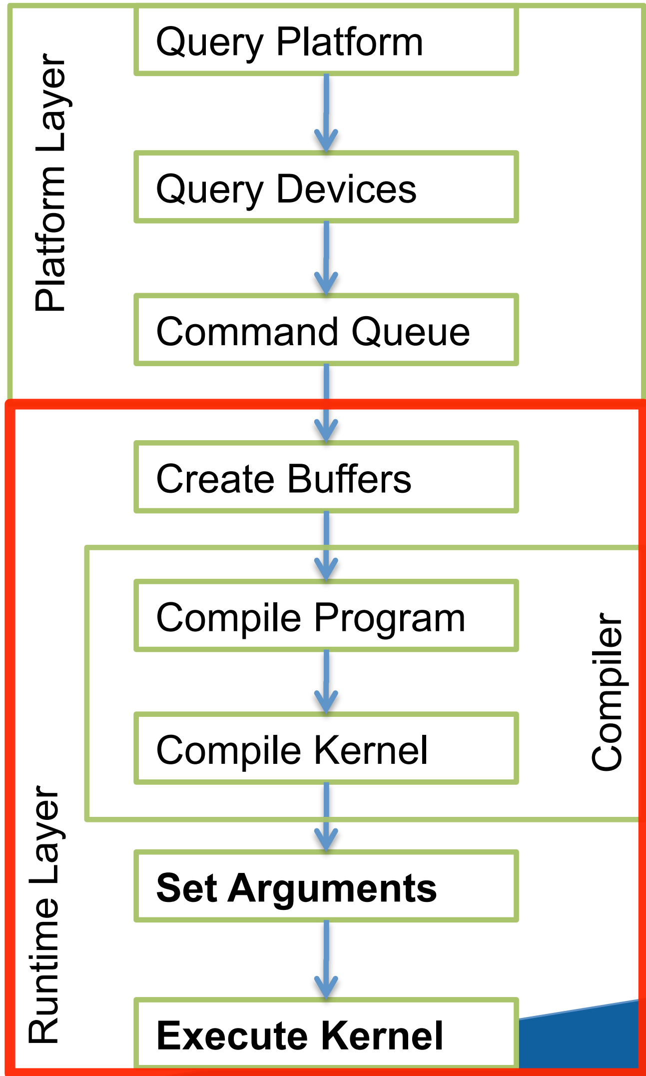


Recap: Set Arguments, Enqueue Kernel

```
// Set Arguments
clSetKernelArg(mykernel, 0, sizeof(cl_mem),
                 (void *)&d_ip);
clSetKernelArg(mykernel, 1, sizeof(cl_mem),
                (void *)&d_op);
 clSetKernelArg(mykernel, 2, sizeof(cl_int),
                 (void *)&len);
...
```

```
//Set local and global workgroup sizes
size_t localws[2] = {16} ;
size_t globalws[2] = {LEN};//Assume divisible by 16
```

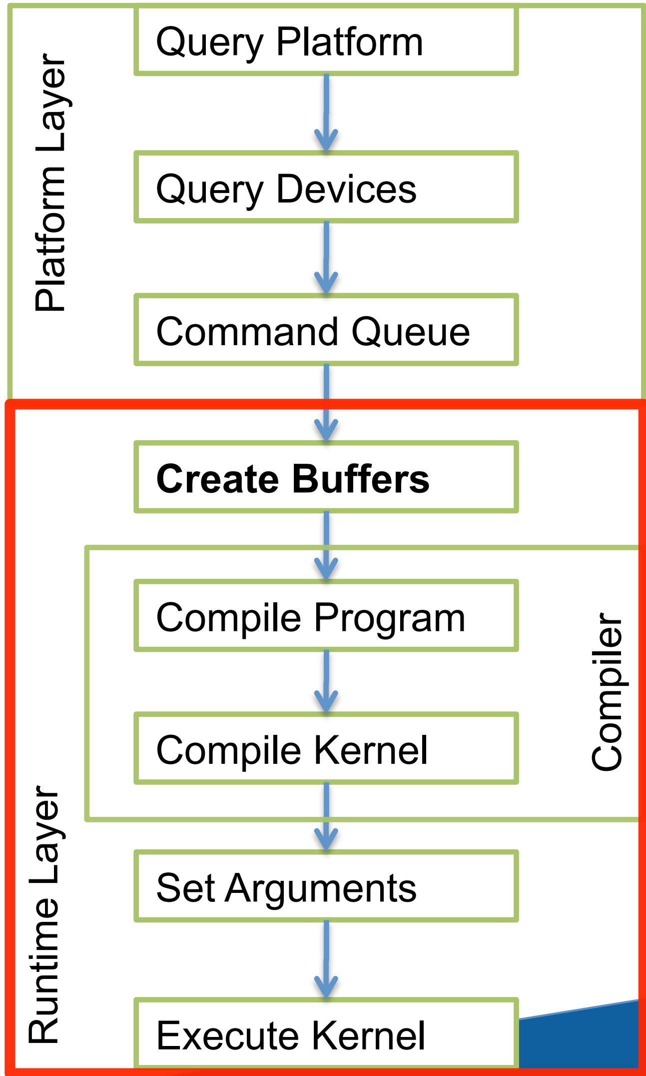
```
// execute kernel
clEnqueueNDRangeKernel(
                myqueue , myKernel,
                2, 0, globalws, localws,
                0, NULL, NULL);
```



Recap: Read Back Result

- Only necessary for data required on the host
- Data output from one kernel can be reused for another kernel
  - Avoid redundant host-device IO

```
// copy results from device back to host
clEnqueueReadBuffer(
        myctx, d_op,
        CL_TRUE,        //Blocking Read Back
        0, mem_size,  (void *) op_data,
        NULL, NULL, NULL);
```



The Big Picture

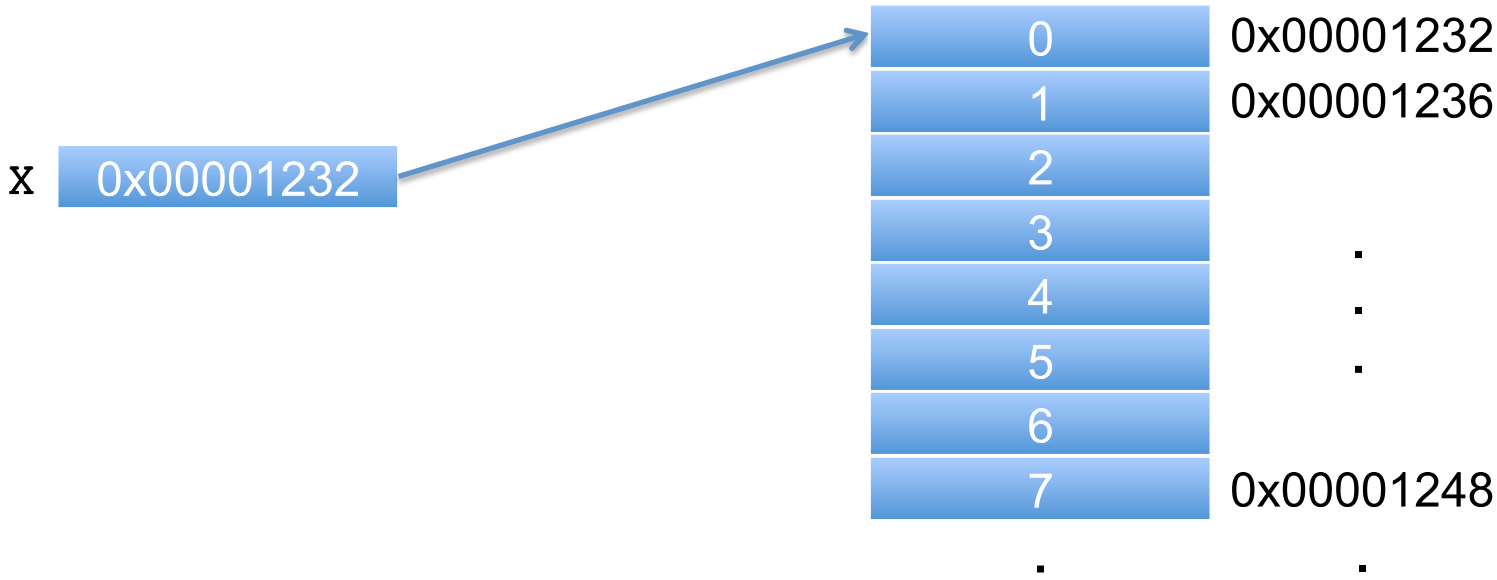
- Introduction to Heterogeneous Systems
- OpenCL Basics
- Memory Issues
- Scheduling

Aspects

- Introduction to GPU bus addressing
- Coalescing memory accesses (Global Memory)
- Synchronisation issues
- Local and Private Memory

Example

- Array X is a pointer to an array of integers (4-bytes each) located at address 0x00001232



- A thread wants to access the data at X[0]

```
int tmp = X[0];
```

Bus Addressing

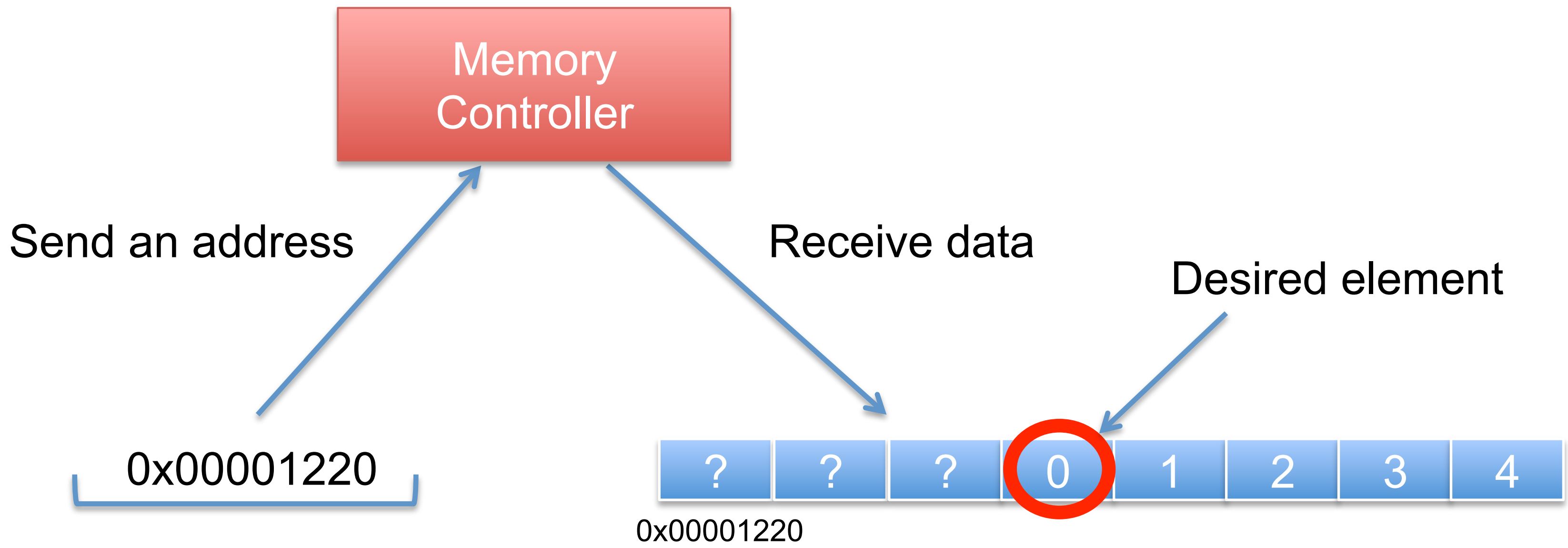
- Assume that the memory bus is 32-bytes (256-bits) wide
  - This is the width on a Radeon 5870 GPU
- The byte-addressable bus must make accesses that are aligned to the bus width, so the bottom 5 bits are masked off

Desired address:	`0x00001232`
Bus mask:	`0xFFFFFFE0`
Bus access:	`0x00001220`

- Any access in the range 0x00001220 to 0x0000123F will produce the address 0x00001220

Bus Addressing

- All data in the range 0x00001220 to 0x0000123F is returned on the bus
- In this case, 4 bytes are useful and 28 bytes are wasted



Coalescing Memory Accesses

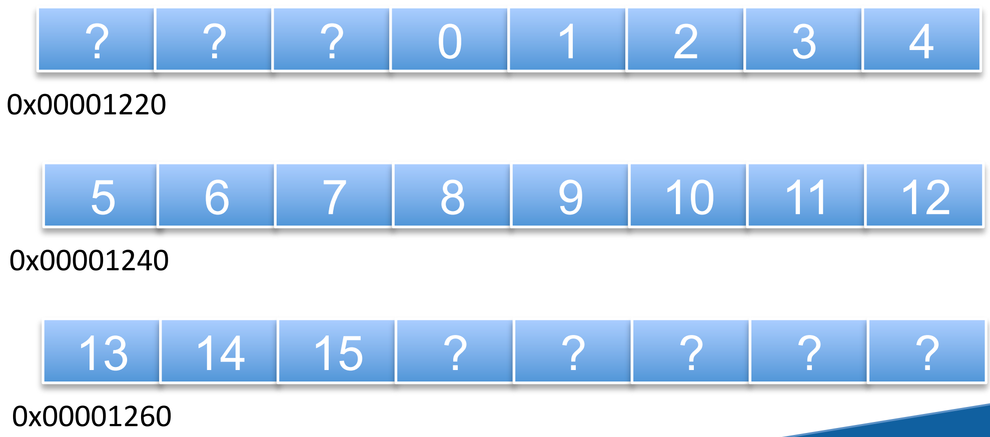
- To fully utilize the bus, GPUs combine the accesses of multiple threads into fewer requests when possible
- Consider the following OpenCL kernel code:

```
int tmp = X[get_global_id(0)];
```

- Assuming that array `X` is the same array from the example, the first 16 threads will access addresses 0x00001232 through 0x00001272
- If each request was sent out individually, there would be 16 accesses total, with 64 useful bytes of data and 448 wasted bytes
  - Notice that each access in the same 32-byte range would return exactly the same data

Coalescing Memory Accesses

- When GPU threads access data in the same 32-byte range, multiple accesses are combined so that each range is only accessed once
  - Combining accesses is called *coalescing*
- For this example, only 3 accesses are required
  - If the start of the array was 256-bit aligned, only two accesses would be required

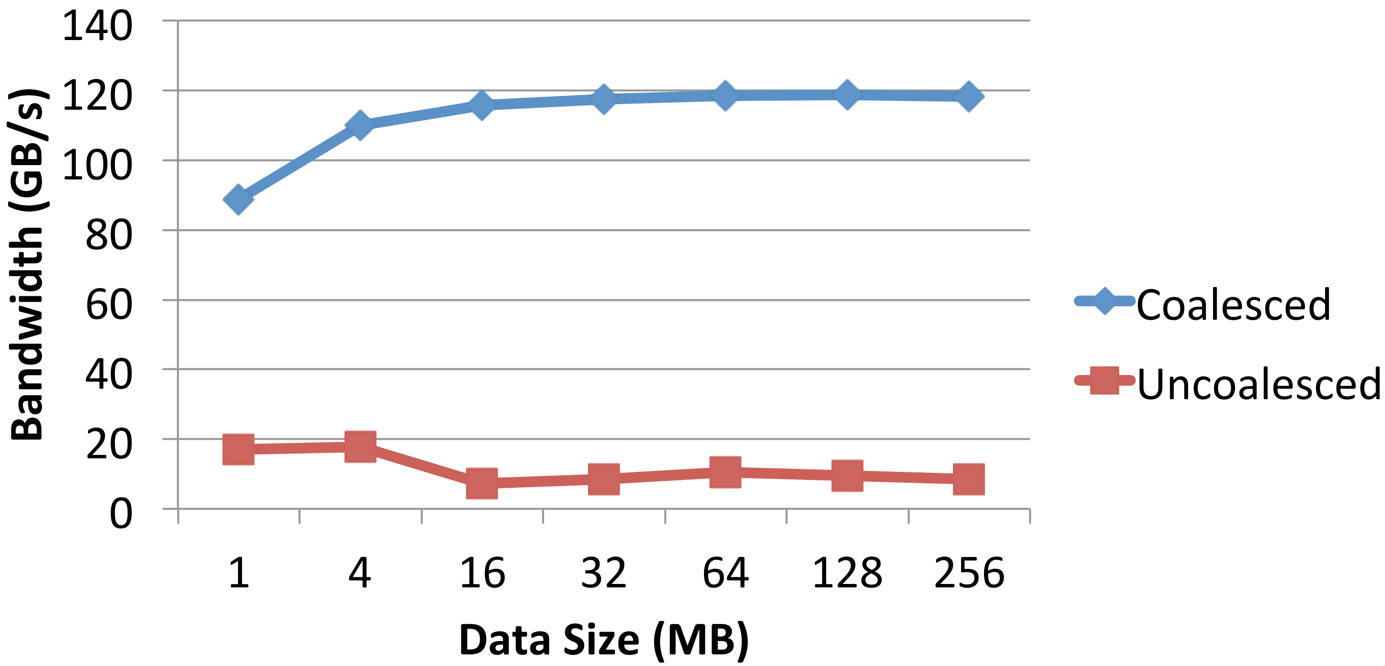


Coalescing Memory Accesses

- Recall that for AMD hardware, 64 threads are part of a wavefront and must execute the same instruction in a SIMD manner
- For the AMD 5870 GPU, memory accesses of 16 consecutive threads are evaluated together and can be coalesced to fully utilize the bus
  - This unit is called a quarter-wavefront and is the important hardware scheduling unit for memory accesses

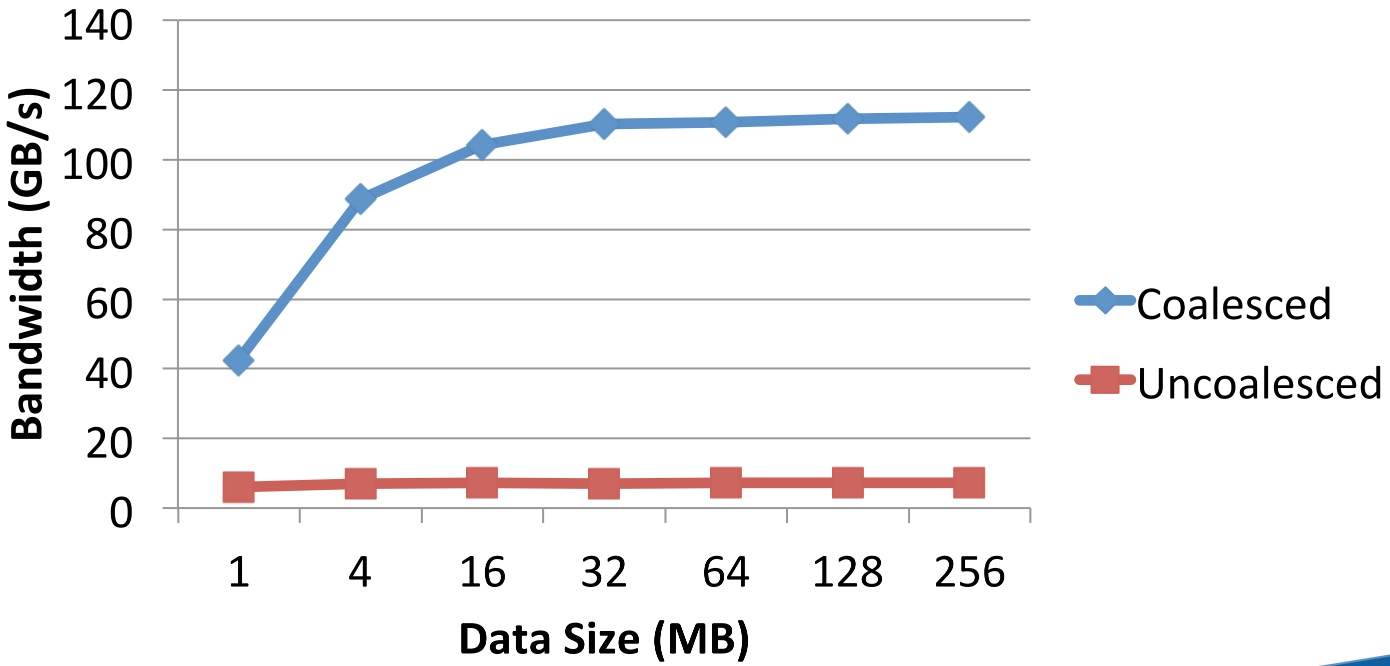
Coalescing Memory Accesses

- Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on an ATI Radeon 5870



Coalescing Memory Accesses

- Global memory performance for a simple data copying kernel of entirely coalesced and entirely non-coalesced accesses on an NVIDIA GTX 285



Working on Global Memory

global_id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
local_id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
__global x[16]	42 ?	42 ?	42 ?	42 ?	42	42	42	42	42	42	42	42	42	42	42	42
__global x2[16]																

kernel:

```
x[global_id] = 42;

x[3] = global_id;

x[local_id] = global_id;
```

To Sync or Not to Sync….

global_id	0	1	2	3	4	5	6	7	8	9	10	11	[illegible]	[illegible]	14	15
local_id	0	1	2	3	0	1	2	[illegible]	[illegible]	[illegible]	[illegible]	3	0	1	2	3
__global x[16]	0	2	4	[illegible]	[illegible]	10	12	14	16	18	20	22	24	26	28	30
__global x2[16]	4	6	8	10	12	14	16	18	20	22	24	26	28	30	0	2

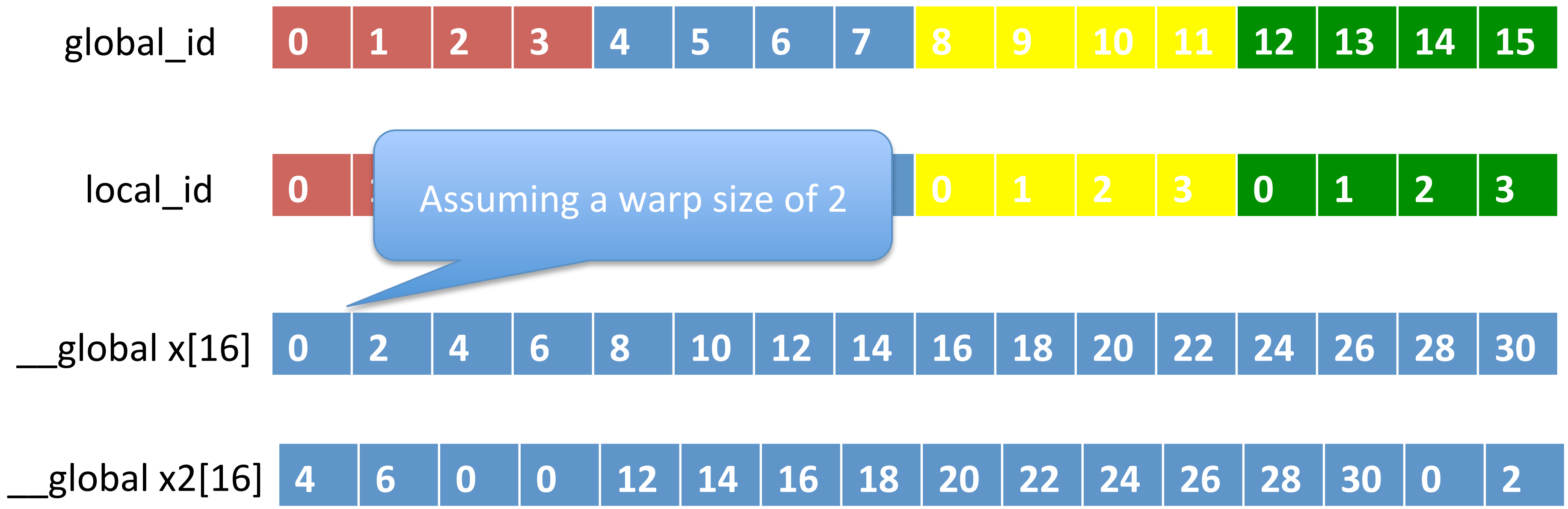
may happen

kernel: x[global_id] = foo(global_id);

Sync?

X2[global_id] = x[(global_id+2)%16];

But this can happen too!



kernel: x[global_id] = foo(global_id);

Sync?

X2[global_id] = x[(global_id+2)%16];

Enforcing Synchronisation

global_id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
local_id	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
__global x[16]	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
__global x2[16]	4	6	8	10	12	14	16	18	20	22	24	26	28	30	0	2

kernel:

```
x[global_id] = foo( global_id);

barrier(CLK_GLOBAL_MEM_FENCE);

X2[global_id] = x[(global_id+2)%16];
```

To Sync or Not to Sync II

global_id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
local_id	0								0	1	2	3	0	1	2	3
__global x[16]	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
__global x2[16]	4	6	8	10	12	14	16	18	20	22	24	26	28	30	0	2

Assuming a warp size of 2

kernel:

```
x[global_id] = foo( global_id);

~~barrier(CLK_GLOBAL_MEM_FENCE); ?????~~

X2[(global_id+14)%16] = x[global_id];
```

To Sync or Not to Sync III

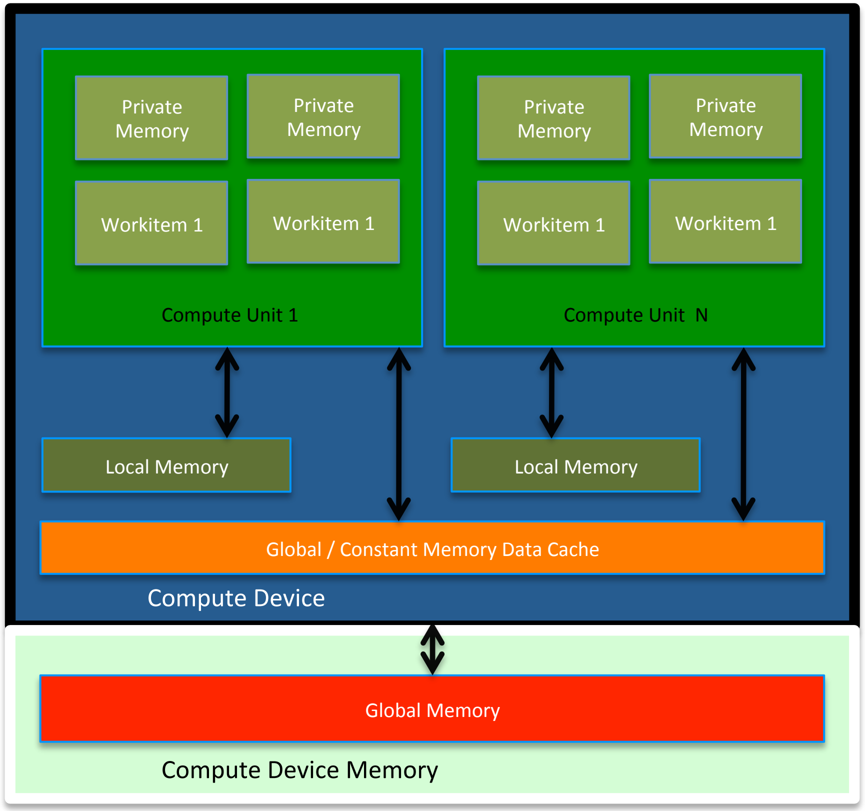
kernel: x[global_id] = foo(global_id);

~~barrier(CLK_GLOBAL_MEM_FENCE); ?????~~

X[(global_id+14)%16] = x[global_id];

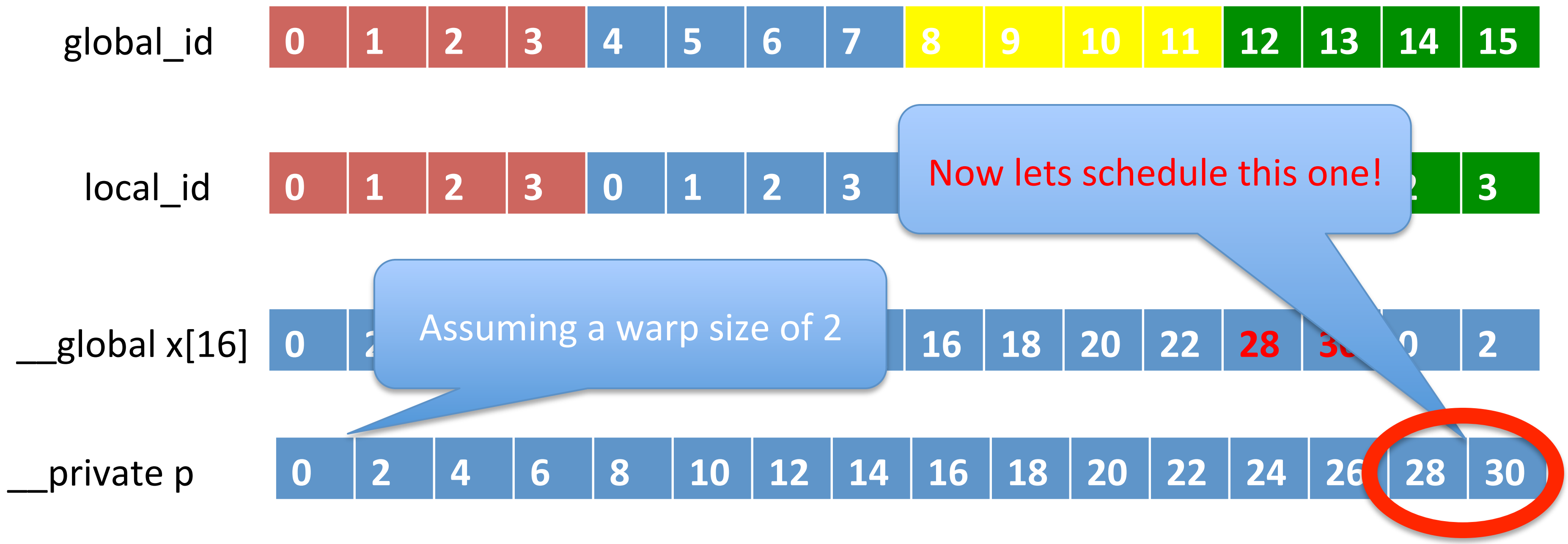
Does not help!!!

Memories….



- Like AMD, a subset of hardware memory exposed in OpenCL
- Configurable shared memory is usable as local memory
  - Local memory used to share data between items of a work group at lower latency than global memory
- Private memory utilizes registers per work item

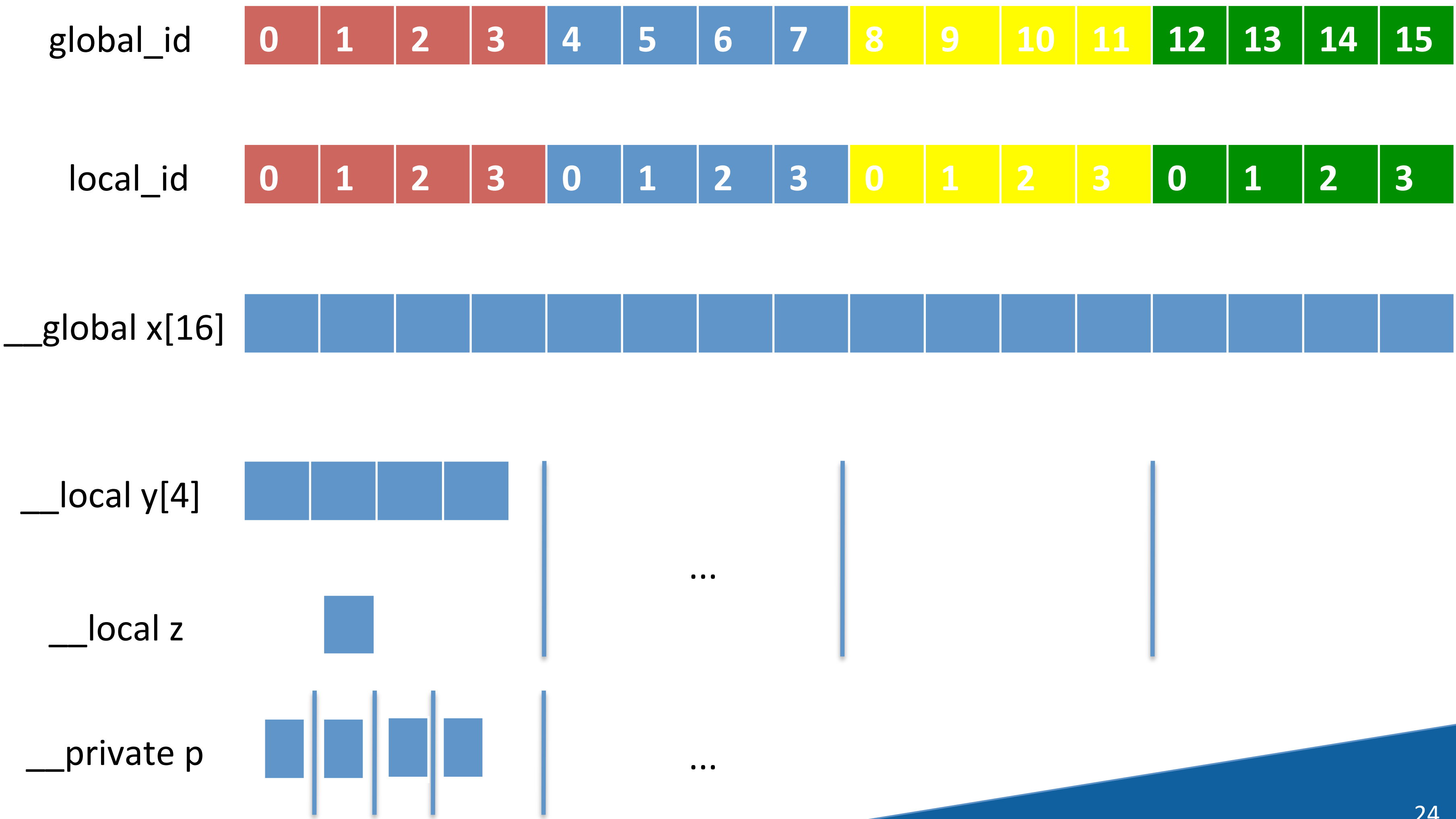
A solution



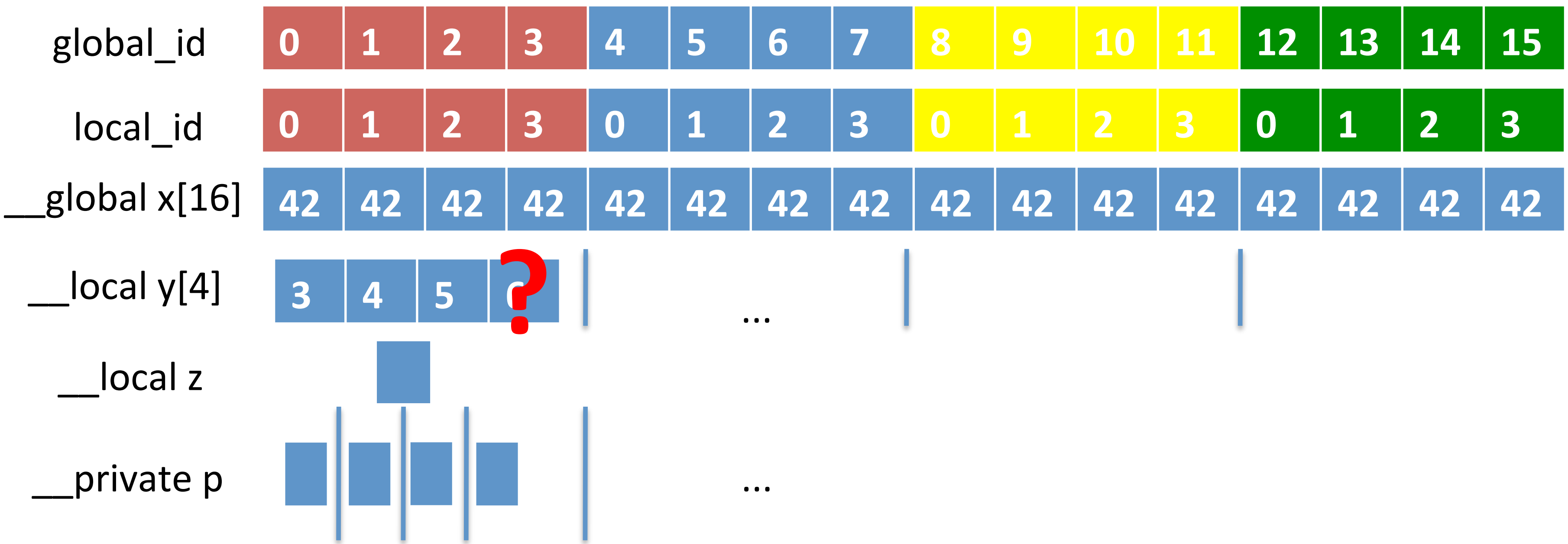
kernel:

```
x[global_id] = foo( global_id);
p = x[global_id];

barrier(CLK_GLOBAL_MEM_FENCE);

X[(global_id+14)%16] = p;
```

Work Groups, Threads, Local and Global Memory



Work Groups, Threads, Local and Global Memory



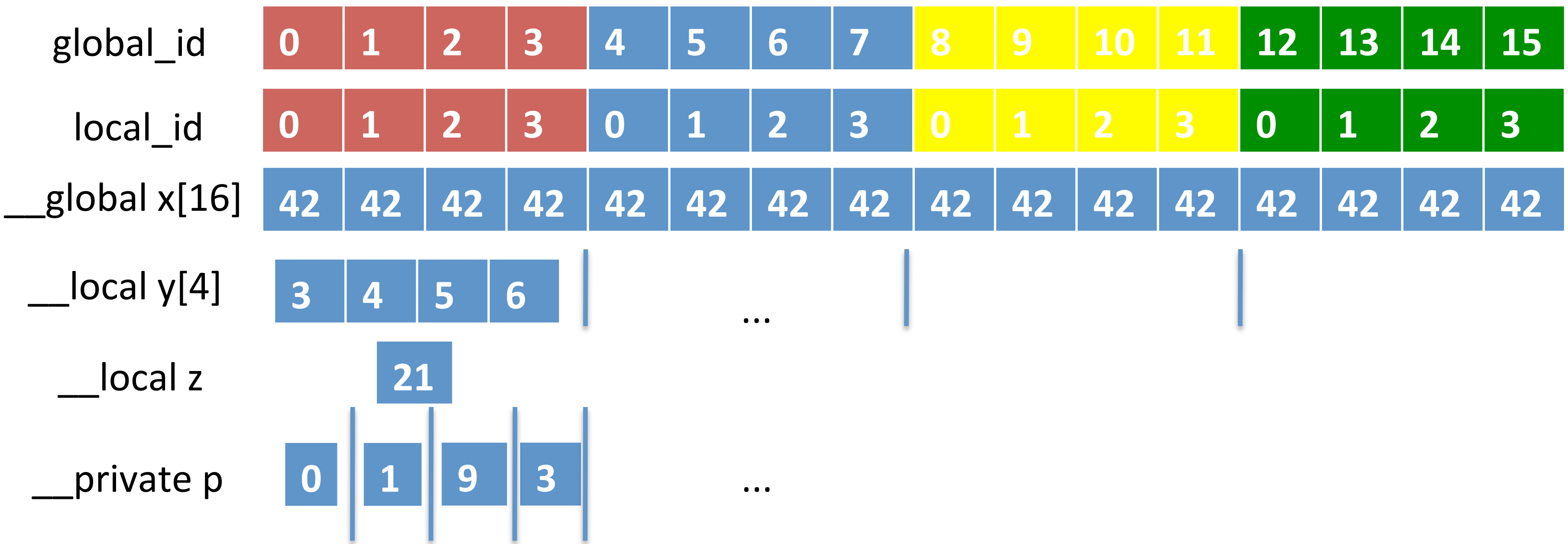
kernel: y[local_id] = global_id+3;

y[3] = local_id;

y[global_id] = local_id;

Out of bounds access!!

To Sync or Not to Sync?

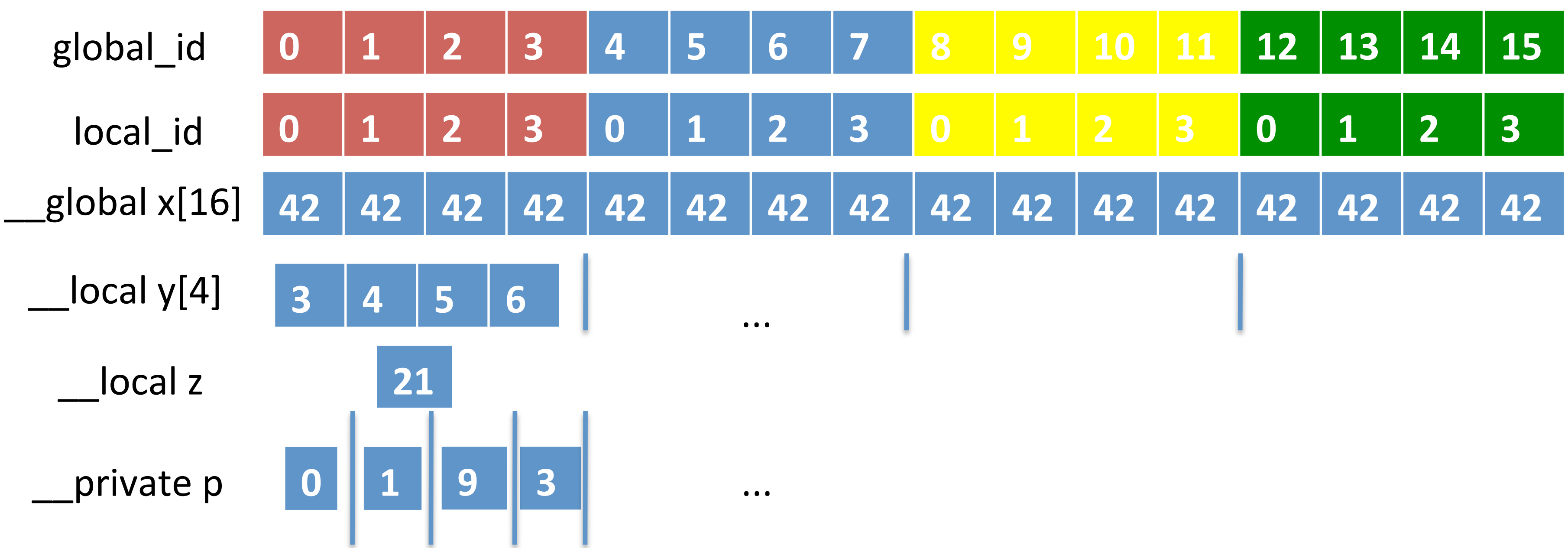


kernel: y[local_id] = foo(local_id);

~~barrier(CLK_LOCAL_MEM_FENCE);????~~

x[global_id+2 % 4] = y[local_id];

To Sync or Not to Sync?



kernel: y[(local_id+1) % 4] = foo(local_id);

barrier(CLK_LOCAL_MEM_FENCE);?!?!

X[global_id+2 % 4] = y[local_id];