University of Leeds

Lecture 16: GPU memory t

# Previous lectures



In the last lecture we saw how to program a simple GPU application:



- computation.
  - Performed by many **work items**
  - Arranged into **work groups** of pr
  - Can arrange work items in 1, 2 or 3 dimensio **nge** (=*n*-dimensional range).



- **Copy** the result back from the device to the host.

# Today's lecture



In the vector addition kernel, all of the arguments had the prefix `__global`:

```
1 __ker
2 void ker
3 {
4   // Ke
5 }
```



Today we will see what this means.



- The different **memory types** avai
- How and when to use them.
- Performance issues related to **register overflow**.

# GPU memory



GPUs are designed as **high throughput devices**.

- **Many** threads (100's, 1000's, ...) execute simultaneously.
- s, *i.e.*



To maximise throughput, GPUs have multiple

- Architecture varies greatly between
- Performance would ideally be optimised f which the code may be deployed.



[1] Although modern GPUs increasingly also have memory caches.

# Shared virtual, or 'unified', memory



- In this module we treat CPU and GPU memory as **separate**
  - Typical of early GPU architectures.

Incre
prog



- Integrated GPUs may share **physi**
- Supported from OpenCL 2.0, CUDA 4.0



We will not consider unified memory in this module.

[1] See *e.g.* Wilt, *The CUDA Handbook* (Addison-Wesley, 2013); Han and Sharma, *Learn CUDA Programming* (Packt, 2019).

# Memory coalescing



When copying from *e.g.* global to local memory, **adjacent** threads will often access **adjacent** memory locations.

GPUs detect and optimise for this by **memory coalescing**:



- nearby memory locations wherever possi
- For 2D/3D data can use a technique known



You are not expected to optimise your code for memory coalescence in this module.

[1] Rauber and Rünger, *Parallel Programming* (Springer, 2012). Tiling is also used to optimise cache access in CPUs.

# Memory types



GPUs typically contain 4 different types of memory:

**Global** Accessible to **all** work items in **all** work groups.

**L**



**Pr**

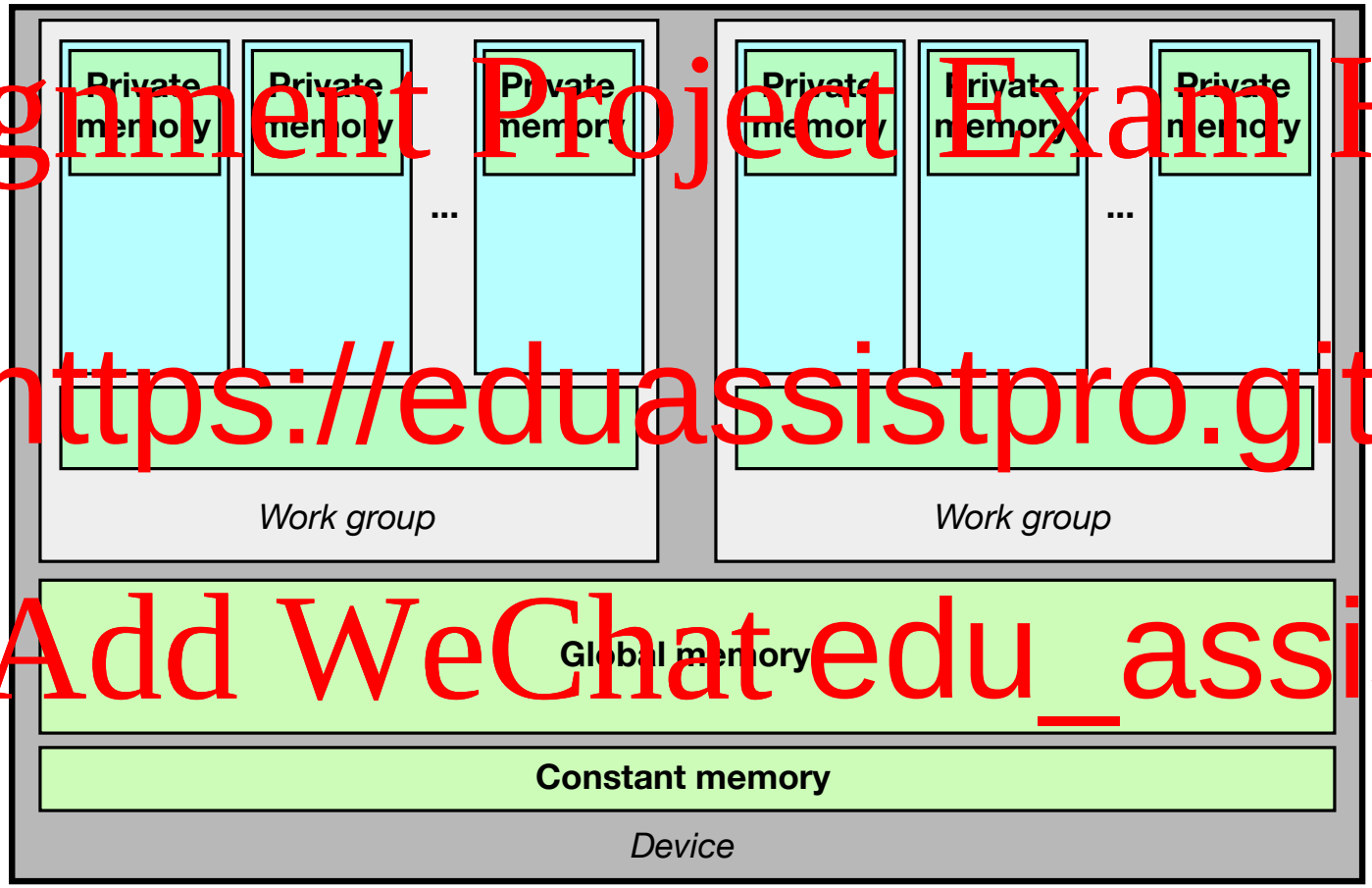
**Constant** Global memory optimised for - tions. Not available on all GPU



These are **disjoint** - it is **not** allowed to c another.

[1] In CUDA and Nvidia devices, *local* memory is referred to as *shared*.

# GPU memory types[1]






[1] After Kaeli *et al.*, *Heterogeneous computing with OpenCL 2.0* (Morgan-Kauffman, 2015).

# Analogy with CPU memory



Notice there is a (loose) analogy with CPU memory types:

| CPU | GPU | Similarity |
|---|---|---|
| Sha | | |
| me | | (GPU)] |
| Dis | | **process** |
| memory | memory | (CPU); **work group** (GPU). |





To send data between work groups, must use glob the host); a form of **communication**.

Cannot **directly** send data between the local memory of different work groups.

- The analogy with distributed memory CPUs breaks down.

# Allocating device memory



When allocating buffer memory on a device, it is placed in **global** memory.



For ex ay
capa

```
1 cl_mem device_array= clCreateBuffer(context,
      CL_MEM_READ_ONLY,N*sizeof(float),...);
```



Note that `CL_MEM_READ_ONLY` does **not** make it constant memory.

- Still allocated in the device's **global** memory.

# Read and write buffer flags



For OpenCL the buffer type should be specified as read-only, write-only, or read-write, by using one of the following flags:



`CL_`
`CL_`
`CL_MEM_READ_WRITE` Both read and write allowed.



- Hint to allow **optimisations** by the r
- The default is `CL_MEM_READ_WRITE`.
- These refer to the **device** accessibility, *i.e.* **from inside kernels**, *not* from the host.

# Memory type 1. Global memory



## Global memory



This is the memory type we have used already



- Convenient from a programming perspe
- Generally poor performance, although typically still faster than host-device communication.

# Using global memory in OpenCL[1]



Allocate global memory using clCreateBuffer():

```
cl_mem device_buffer = clCreateBuffer(context,flags,
    size,...);
```



In the kernel:

```
__kernel
void vectorAdd( __global float *a, __global float *b,
    __global float *c )
{
  int gid = get_global_id(0);
  c[gid] = a[gid] + b[gid];
}
```



[1] In CUDA: clCreateBuffer()→cudaMalloc(); no __global specifier.

# Memory type 2. Local memory



## Local memory

A tween



Typically used as a **scratch space** for c
than one work-item in a group.



- Use for intermediate calculations.
- Place final answer in global memory.

In practice, this also requires **synchronisation** which is next lecture's topic, so will see an example of local memory then.

# Static local arrays



To use local memory in a kernel, simply place `__local` before the variable[1]:

```
__kernel
void ker
{
  __
  // Ca
  ...
  // Place final answer in device_array.
}
```



However, this is **static allocation** - the si
known **when the kernel is built**.

[1] In CUDA: `__local` → `__shared__`.

# Dynamic local arrays



To create **dynamic local** arrays, declare it as a **kernel argument** with the specifier `__local`:

```
__ker
void ker
{
  // Ca
  ...
  // Place final answer in device_a.
}
```



Then when setting kernel arguments, specify the size **but set the pointer to** `NULL`:

```
clSetKernelArg(kernel,0,N*sizeof(int),NULL);
```

# Memory type 3. Private memory



## Private memory



In pra

hardware as **registers**:



- Small amount of memory that can be access
- Faster access than even local memory.
- **Automatic storage duration**, *i.e.* deallocated at the end of the kernel (or enclosing code block).

# Using private memory



- There is no specifier for private memory (*i.e.* no `__private`).



For in

```
1  __kernel
2  void kernel(__global float *array)
3  {
4    int gid = get_global_id(0);
5    ...
6  }
```



. . . the variable `gid` is treated as private memory.

# Private kernel arguments



Kernel arguments without a specifier are also treated as private.

In this e

```
__ker
void ker
{
  // Calculations involving N.
  ...
}
```



The corresponding call to `setKernelArg()` would be:

```
int N=...;
clSetKernelArg(kernel,1,sizeof(int),&N);
```

# Register overflow

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



Devices have a **fixed** amount of register memory. What happens if this is exceeded is device-dependent, but is usually one of:

1. 
2. 

Either mechanism reduces performance.



**Guidance**

Kernels should be **small** (in the sense of low register usage) to limit the risk of register overflow.

# Memory type 4. Constant memory



**Constant memory**

Accessible by all work items and work groups, but read only (by a ker



GPU
read fr

- Known as **constant** memory.
- Can still be written to **by the host**.
- Originally to accelerate the mapping of s ('texture' memory[1]).
- Typically much smaller than global memory, even if it exists.



[1] CUDA treats texture memory separately to other constant memory.

# Using constant memory[1]



For kernel arguments, use the __constant specifier:

```
__kernel
void ker
{ ... }
```



Initia
argu

Variables within kernels can also be __
specified at compile time:



```
__constant float pi = 3.1415926;
```

[1] In CUDA: Device data specified __constant__, and copy from host using cudaMemcpyToSymbol().

# Summary and next lecture



Today we have look at the different **memory types** in GPUs:





The main use of **local** memory is to coordi
between work items in a group. We will see an example next time when we look at **synchronisation** in GPUs.