XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 16: GPU memory types

Previous lectures

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

- **Allocate** memory on the **device** (the GPU).
- **Copy host** (CPU) data to the device.
- Build and execute a **kernel** on the device that performs the computation.
  - Performed by many **work items** *(threads)*.
  - Arranged into **work groups** of programmable size.
  - Can arrange work items in 1, 2 or 3 dimensions; the **NDRange** (=*n*-dimensional range).
- **Copy** the result back from the device to the host.

This lecture

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

```
__kernel
void kernel(__global float *a,__global float *b,...)
{
  // Kernel body.
}
```

In this lecture we will see what this means:

- The different **memory types** available to a GPU.
- 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.
- By contrast, CPUs are **latency reducing** architectures, *i.e.* fast memory access by use of caches[1], instruction level parallelism *[cf. Lectures 1, 2]*, *etc.*

To maximise throughput, GPUs have multiple **memory types**:

- Architecture varies greatly between, *and within*, vendors.
- Performance would ideally be optimised for **each** GPU on 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.

Increasingly, CPU and GPU memory are presented to the programmer as **unified**[1].

- API decides whether CPU or GPU memory is allocated.
- Integrated GPUs may share **physical** memory with the CPU.
- 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**:

- **Coalesces** multiple small memory accesses into a single large memory access — much faster.
- To exploit this, programmers can ensure nearby threads access nearby memory locations wherever possible.
- For 2D/3D data, can use a technique known as **tiling**[1].

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

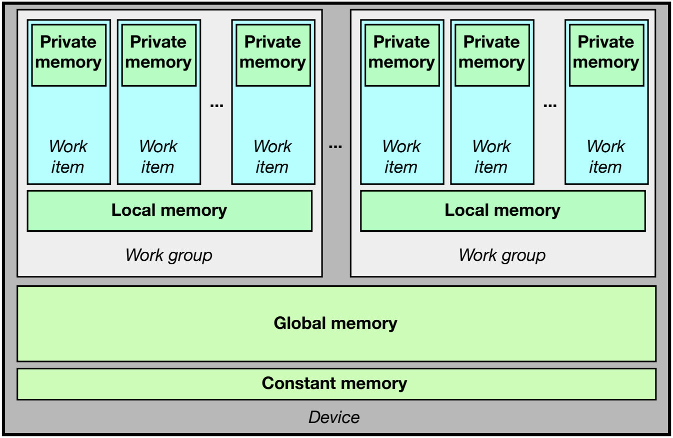
GPUs typically contain 4 different types of memory.

Global	Accessible to **all** work items in **all** work groups.
Local[1]	Shared by work items within **one work group only**.
Private	Accessible to a **single work item only**.
Constant	Global memory optimised for **read-only** operations. Not available on all GPUs.

These are **disjoint** - it is **not** allowed to cast one address space to 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**
Shared memory	Global memory	Accessible by all processing units [**threads** (CPU); **work items** (GPU)]
Distributed memory	Local memory	Only accessible to one **process** (CPU); **work group** (GPU).

To send data between work groups, must use global memory (or 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 example, to allocate a global buffer called `device_array` capable of storing `N` floats:

```
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_MEM_READ_ONLY`	The buffer should only be read from.
`CL_MEM_WRITE_ONLY`	The buffer should only be written to.
`CL_MEM_READ_WRITE`	Both read and write allowed.

- Hint to allow **optimisations** by the runtime system.
- 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

Accessible to all processing units, but bandwidth is **slower** compared to the other memory types. Typically cached in modern GPUs, but not in older devices.

This is the memory type we have used already (*cf.* Lecture 15).

- Convenient from a programming perspective.
- 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, prepend `__global` before the pointer(s):

```
__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

Accessible to all work items in a work group, but **not** between groups. Much faster than global memory.

Typically used as a **scratch space** for calculations involving more 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 kernel(__global int *device_array)
{
  __local int temp[128];
  // Calculations involving temp.
  ...
  // Place final answer in device_array.
}
```

However, this is **static allocation** - the size of the array must be 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`:

```
__kernel
void kernel(__local int *temp,__global int *device_a)
{
  // Calculations involving temp.
  ...
  // 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

Only accessible to each work item. *Very* fast access.

In practice, private memory is almost always implemented in hardware as **registers**:

- Small amount of memory that can be accessed quickly.
- 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`).

- Variables declared within a kernel **default** to private.

For instance, in this kernel . . .

```
__kernel
void kernel(__global float *array)
{
  int gid = get_global_id(0);
  ...
}
```

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

Private kernel arguments

Kernel arguments without a specifier are also treated as private.

In this example, `N` is treated as a private variable:

```
__kernel
void kernel(__global float *array,int N)
{
  // 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. Private memory will 'spill over' into **global memory**.
2. Fewer work groups will be launched simultaneously, resulting in an **under–utilisation** of available processing units.

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 kernel). Faster than global memory. Not available on all GPUs.

GPUs often have memory that is global in scope, but can only be read from within kernels:

- Known as **constant** memory.
- Can still be written to **by the host**.
- Originally to accelerate the mapping of **textures** to polygons ('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 kernel(__constant float *a,...)
{ ... }
```

Initialise the array (*i.e.* copy from the host) and set the kernel argument as normal.

Variables within kernels can also be `__constant`, but must 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:

- **Global** (slow), **local** (faster), **private** (very fast).
- Possibly also **constant** (faster than global).
- Private memory can **overflow**, resulting in performance loss.

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