Memory Buffers

How to share of memory efficiently between a Mali-T600 series GPU and a CPU.

Introduction

When dealing with large amounts of data (as is typically the case in OpenCL applications) it is important to ensure that transfer of that memory between the host and the OpencL device is as efficient as possible.

We have already seen how to use memory buffers in the Hello World tutorial. The Hello World tutorial follows what we consider "best practices" for data sharing between the host and an OpenCL device. This tutorial explains those best practises.

Unless otherwise noted, all code snippets come from hello_world_vector.cpp.

Shared Memory Systems

OpenCL is typically run on systems where the application processor and the GPU have separate memories. To share data between the GPU and CPU on these systems you must allocate buffers and copy data to and from the separate memories.

Systems with Mali GPUs typically have a shared memory so you are not required to copy data. However, OpenCL assumes the memories are separated and buffer allocation involves memory copies. This is wasteful because copies take time and consume power.

Sharing Data Across Devices

To avoid the copies, use the OpenCL API to allocate memory buffers and use mapping operations. These operations enable both the application processor and the Mali GPU to access the data without any copying.

Note
: OpenCL must be initialised before allocating memory buffers.

1. Allocating memory buffers

   The application creates buffer objects that can pass data to and from the kernels by calling clCreateBuffer(). All memory allocated on a shared memory system is physically accessible by both CPU and GPU cores. However, only memory that is allocated by clCreateBuffer is mapped into both the CPU and GPU virtual memory spaces. So memory allocated using malloc(), etc, is only mapped onto the CPU (image 1).

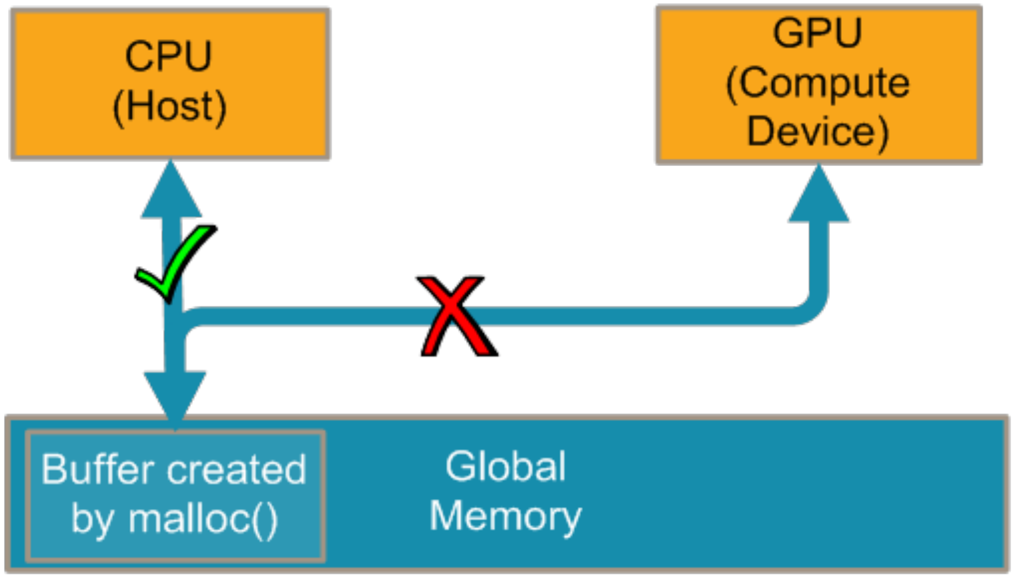

Image 1: Buffers created by user (malloc) are not mapped into the GPU memory space

So calling clCreateBuffer() with CL_MEM_USE_HOST_PTR and passing in a user created buffer requires the driver to create a new buffer and copy the data (identical to CL_MEM_COPY_HOST_PTR). This copy reduces performance (image 2).

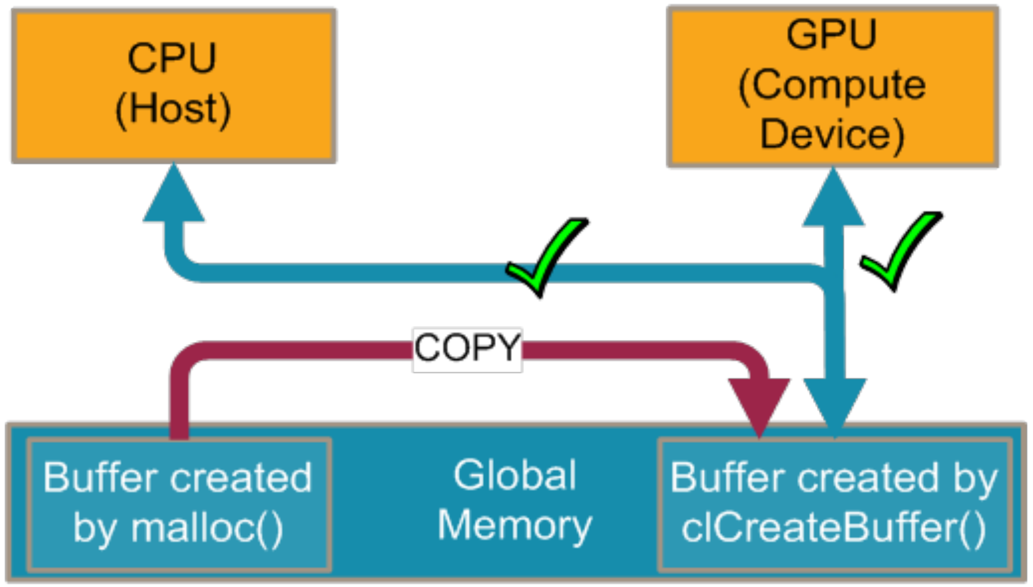
CPU (Host)
GPU (Compute Device)
COPY
Buffer created by malloc()
Global Memory
Buffer created by clCreateBuffer()

Image 2: clCreateBuffer(CL_MEM_USE_HOST_PTR) creates a new buffer and copies the data over (but the copy operations are expensive)

So where possible always use CL_MEM_ALLOC_HOST_PTR. This allocates memory that both CPU and GPU can use without a copy (image 3).

```
memoryObjects[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR, bufferSize, NULL, &errorNumber);
createMemoryObjectsSuccess &= checkSuccess(errorNumber);

memoryObjects[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR, bufferSize, NULL, &errorNumber);
createMemoryObjectsSuccess &= checkSuccess(errorNumber);

memoryObjects[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_ALLOC_HOST_PTR, bufferSize, NULL, &errorNumber);
createMemoryObjectsSuccess &= checkSuccess(errorNumber);
```

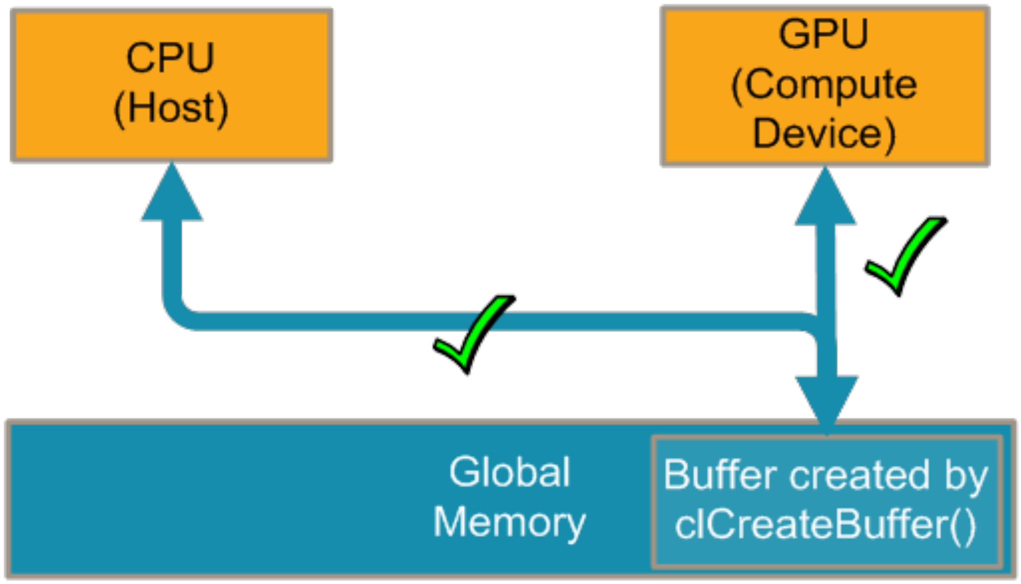

Image 3: clCreateBuffer(CL_MEM_ALLOC_HOST_PTR) creates a buffer visible by both GPU and CPU

2. Map memory objects

   As mentioned before, we map the memory buffers created by the OpenCL implementation to pointers so we can access them on the CPU.

```
cl_int* inputA = (cl_int*)clEnqueueMapBuffer(commandQueue, memoryObjects[0], CL_TRUE, CL_MAP_WRITE, 0, bufferSize, 0, NULL, NULL,
        &errorNumber);
mapMemoryObjectsSuccess &= checkSuccess(errorNumber);

cl_int* inputB = (cl_int*)clEnqueueMapBuffer(commandQueue, memoryObjects[1], CL_TRUE, CL_MAP_WRITE, 0, bufferSize, 0, NULL, NULL,
        &errorNumber);
mapMemoryObjectsSuccess &= checkSuccess(errorNumber);
```

3. Initialize data using the C pointers

   Once the buffers have been mapped to a *cl_int* pointer, they can be used as a normal C pointer to initialize the data.

```
for (int i = 0; i < arraySize; i++)
{
  inputA[i] = i;
  inputB[i] = i;
```

```
    }
```

4. Unmap memory objects

   When we are finished using the memory objects, we unmap them from the CPU side. We unmap the memory because otherwise:

   - Reads and writes to that memory from inside a kernel on the OpenCL side are undefined,
   - The OpenCL implementation cannot free the memory when it is finished.

```
if (!checkSuccess(clEnqueueUnmapMemObject(commandQueue, memoryObjects[0], inputA, 0, NULL, NULL)))
{
   cleanUpOpenCL(context, commandQueue, program, kernel, memoryObjects, numberOfMemoryObjects);
   cerr << "Unmapping memory objects failed " << __FILE__ << ":"<< __LINE__ << endl;
   return 1;
}

if (!checkSuccess(clEnqueueUnmapMemObject(commandQueue, memoryObjects[1], inputB, 0, NULL, NULL)))
{
   cleanUpOpenCL(context, commandQueue, program, kernel, memoryObjects, numberOfMemoryObjects);
   cerr << "Unmapping memory objects failed " << __FILE__ << ":"<< __LINE__ << endl;
   return 1;
}
```

Additional Recommendations

In addition to the above tutorial, we recommend the following:

- Do not use private or local memory. There is no performance gain using these on the Mali-T600 Series GPUs.
- In the event that your kernels are memory bandwidth limited, use a formula to compute variables instead of reading from memory.

More Information

For more information have a look at the Hello World tutorial.

Find solutions for Common Issues.