Lab 5 - CUDA Memory

COM4521/COM6521 - Parallel Computing with Graphical Processing Units (GPUs)

Code

- Starting Code
- Solution

Learning Outcomes

- How to query CUDA device properties
- Understanding how to observe the difference between theoretical and measure memory bandwidth
- Understanding an observing the difference between the constant cache and read-only cache
- Understanding how to use texture binding for problems which naturally map to 2D domains

Exercise 1

In exercise one we are going to extend our vector addition kernel. The code is provided in `exercie01.cu`. Complete the following changes:

1. Modify the example to use statically defined global variables (e.g. where the size is declared at compile time and you do not need to use `cudaMalloc()`). *Note: A device symbol (statically defined CUDA memory) is not the same as a device address in the host code. Passing a symbol as an argument to the kernel launch will cause invalid memory accesses in the kernel.*
2. Modify the code to record timing data of the kernel execution. Print this data to the console.
3. We would like to query the device properties so that we can calculate the theoretical memory bandwidth of the device. The formula for theoretical bandwidth is given by Equation 1.

   Using `cudaDeviceProp` query the two values from the first `cudaDevice` available and multiply these by two (as DDR memory is double pumped,

hence the name) to calculate the theoretical bandwidth. Print the theoretical bandwidth to the console in GB/s (Giga Bytes per second).

$$theoreticalBW = memoryClockRate \times memoryBusWidth \quad (1)$$

Note: Equation 1 will calculate the result in kilobits/second (as `memoryClockRate` *is measured in kilohertz and* `memoryBusWidth` *is measured in bits). You will need to convert to the memory clock rate to Gb/s (Gigabits per second) and then convert this to GB/s.*

4. Theoretical bandwidth is the maximum bandwidth we could achieve in ideal conditions. We will learn more about improving bandwidth in later lectures. For now we would like to calculate the measure bandwidth of the `vectorAdd()` kernel. Measure bandwidth is given by the formula in 2:

$$measuredBW = \frac{R_{Bytes} + W_{Bytes}}{t} \quad (2)$$

Where `R_Bytes` is the number of bytes read and `W_Bytes` is the number of bytes written by the kernel. You can calculate these values by considering how many bytes the kernel reads and writes and multiplying it by the number of threads that are launched. The value `t` is given by your timing data in ms you will need to convert this to seconds to give the bandwidth in GB\s. Print the value to the console so that you can compare it with the theoretical bandwidth.

Note: Don't forget to switch to Release mode to profile your code execution times.

Exercise 2

In the last lecture we learned about the different types of memory and caches which are available on a GPU device. For this exercise we are going to optimise a simple ray tracer application by changing the memory types which are used. Download the starting code (example02.cu) and take a look at it. The ray tracer is a simple ray casting algorithm which casts a raw for each pixel into a scene consisting of sphere objects. The ray checks for intersections with the spheres, where there is an intersection a colour value for the pixel is generated based on the intersection position of the ray on the sphere (giving an impression of forward facing lighting). For more information on the ray tracing technique read Chapter 6 of the CUDA by Example book which this exercise is based on. Try executing the starting code an examining the output image (`output.ppm`) using GIMP or Adobe Photoshop.

The initial code places the spheres in GPU global memory. We know that there are two options for improving this in the form of constant memory and texture/read only memory. Implement the following changes.

The following exercise should be completed by building the device code for the latest supported version of the GPU you are using. For the Diamond computers this is `compute_61,sm_61`. See the notes on "CUDA compilation with Visual Studio" for details on how to set this.

1. Create a modified version of ray tracing kernel which uses the read-only data cache (`ray_trace_read_only()`). You should implement this by using the `const` and `__restrict__` qualifiers. Calculate the execution time of the new version alongside the old version so that they can be directly compared: You will need to also create a modified version of the sphere intersect function (`sphere_intersect_read_only()`).
2. Create a modified version of ray tracing kernel which uses the constant data cache (`ray_trace_const()`). Calculate the execution time of the new version alongside the two other versions so that they can be directly compared.
3. How does the performance compare? Is this what you expected and why? Modify the number of spheres to complete the following table. For an extra challenge try to do this automatically so that you loop over all the sphere count sizes in the table and record the timing results in a 2D array.

Sphere Count	Normal	Read-only cache	Constant cache
16 32 64			
128 256			
1024 2048			

Exercise 3

In exercise 3 we are going to experiment with using texture memory. In this example we are going to explicitly use texture objects rather than using qualifiers to force memory loads through the read-only cache. There are good reasons for doing this when dealing with problems which relate to images or with problems decompose naturally to 2D layouts.[1] The example that we will be working with is an image blur. The code (exercise03.cu) is provided along with an image `input.ppm` (an image of a dog relaxing). Build and execute the code to see the result of executing the image blur kernel. You can modify the macro `SAMPLE_SIZE` to increase the scale of the blur.

Take a look at the kernel code and ensure that you understand it. We will now implement texture sampling by performing the following:

1. Duplicate the `image_blur()` kernel (naming it `image_blur_texture1D`). Create a 1 dimensional texture object with `cudaReadModeElementType`. Modify the new kernel to perform a texture lookup using `tex1Dfetch()`.

[1] E.g. Improved caching, address wrapping and filtering.

Modify the host code to execute the texture1D version of the kernel after the first version saving the timing value to the `y` component of the variable `ms`. You will need to add appropriate host code to bind and unbind the texture before and after the kernel execution respectively.

2. Duplicate the `image_blur()` kernel (naming it `image_blur_texture2D`). Declare a 2 dimensional texture object with `cudaReadModeElementType`, this will require copying the image data to a `cudaArray_t` allocated with `cudaMallocArray()`. Modify the new kernel to perform a texture lookup using `tex2D`. Modify the host code to execute the texture2D version of the kernel after the first version saving the timing value to the `z` component of the variable ms. You will need to add appropriate host code to bind and unbind the texture before and after the kernel execution respectively.
3. In the case of the 2D version it is possible to perform wrapping of the index values without explicitly checking the x and y offset values. To do this remove the checks from your kernel and set the `addressMode[0]` and `addressMode[1]` structure member of your 2D texture description to `cudaAddressModeWrap`.

Compare the performance metrics. There is unlikely to be much difference between the 1D and 2D texture version, however the code is more concise with the 2D version.