Com 4521 Parallel Computing with GPUs: Lab 09 (CUDA libraries and Streams)

Spring Semester 2022

Author: Dr Paul Richmond

Lab Assistants: Robert Chisholm, Jack Ashurst, John Charlton

Department of Computer Science, University of Sheffield

Learning Outcomes

- Understand how to use the Thrust library to perform sorting of key value pairs
- Understand how to use the Thrust library to simplify the scan algorithm
- Understand how parallel primitives can be used within a complex problem (particle-particle interactions)
- Understand how CUDA streams can be used to overlap memory copies and kernel execution
- Demonstrate performance differences of scheduling stream work via two distinct methods.

Exercise 01

For this exercise we are revisiting some code from the very first lab where we implemented a command driven calculator. The exercise has been extended to create a parallel calculator which will apply a fixed program of calculations to a set of input data using a SPMD model. A GPU implementation has already been provided. The performance critical (and hence timed) part of the implementation requires copying the input data to the device, performing the calculations and finally reading the data back form the device. The provided implementation is strictly synchronous in execution and uses only the default CUDA stream. It is possible to improve the performance by breaking the task down into smaller independent parts and using streams to ensure that the copy engines(s) and compute engine are kept busy with independent work. We will implement 2 different streaming versions and compare the performance of each.

The first asynchronous version that we implement will loop through each stream and perform a copy to the device, kernel execution and copy back from the device. Each stream will be responsible for a subset of the data (of size `SAMPLES` divided by `NUM_STREAMS`). Implement this version by completing the following tasks which require you to modify the `cudaCalculatorNStream1` function;

1.1 Allocate the host and device memory so that it can be used for asynchronous copying.
1.2 Create `NUM_STREAMS` streams in the streams array declared within the function.
1.3 Create a loop to iterate the streams. For each stream launch stages 1, 2 and 3 from the synchronous version so that the stream operates on only a subset of the data. Test your code.
1.4 Destroy each of the CUDA streams.

For the second asynchronous version you can copy the allocation, stream creation and destruction code from your `cudaCalculatorNStream1` function to `cudaCalculatorNStream2`. For this exercise we will make a subtle change to the way work within streams is scheduled.

1.5 Loop through the streams and schedule each stream to perform a host to device memory copy on a suitable subset of the data.

1.6 Loop through the streams and schedule each stream to perform the kernel execution on a suitable subset of the data.
1.7 Loop through the streams and schedule each stream to perform a device to host memory copy on a suitable subset of the data. Hence copying back the result.
1.8 Test and Benchmark your code. Modify the `NUM_STREAMS` value and complete the following table of results. Do you understand why the two different stream versions differ?

Version	Streams	GPU timing (s)
`cudaCalculatorDefaultStream`	NA	0.575488
`cudaCalculatorNStream1`	2	0.180224
`cudaCalculatorNStream1`	3	0.161792
`cudaCalculatorNStream1`	6	0.167936
`cudaCalculatorNStream2`	2	0.165984
`cudaCalculatorNStream2`	3	0.150528
`cudaCalculatorNStream2`	6	0.147456

Exercise 02

In exercise two we are going to migrate an implementation of the simple particle type simulation which was described in the previous lecture. The implementation calculates each particle's nearest neighbour within a fixed range. It does this through the following steps.

1. Generate key value pairs: where the key is the bucket that the particle belongs to and the value is the index of the particle.
2. Sorting the key value pairs: the key value pairs are sorted according to their key
3. Re-ordering particles: The particles are reordered from the sorted key values pairs. Using the value a scattered write can move the location of the individual particles.
4. Calculate a histogram of the particles: For each bucket that discretises the environment a count of the number of particles is generated.
5. Prefix sum of buckets: Given the bucket counts a prefix sum is used to determine where the starting index of each bucket is located within the sorted list of particles
6. Calculate the nearest neighbour for each particle by considering all neighbours within a fixed range.

The implementation provided uses the CPU for most of these stages (other than the nearest neighbour calculation). To improve this code we are going to re-implement the CPU functions on the GPU by completing a number of steps. In the `particlesGPU` function perform the following;

1.1 Start by copying the initialised participle data from the host (`h_particles`) to the device (`d_particles`).
1.2 Complete the kernel which calculates the key value pairs.
1.3 Use Thrust sort by key to sort the key value pairs (`d_key_values`).
1.4 Implement a kernel for reordering particles given the old positions (`d_particles`) and the key value pairs. Save the results in `d_particles_sorted`.
1.5 Implement a kernel to calculate the counts for each bucket in the environment. Store these in the `d_env` structure. Hint: You will need to use atomics if you want to increment device global values.
1.6 Use Thrust to perform an exclusive scan on the environment counts. The scan output should go in `d_env->start_index`. Hint: You will need to cast the arrays within `d_env` using device pointer casts.

1.7 Ensure you code produces the correct result and compare the CPU and GPU versions. Note that our timing code does not use CUDA events as we are timing both CPU and GPU code execution.