F21DP Distributed and Parallel Technologies.
Coursework 2: Sum Totient in OpenCL and parallel Haskell.

Summary

For coursework 2 (CWK2) you will develop and measure parallel versions of a program using one of the following parallel programming technologies: OpenCL on a GPU or parallel Haskell on a CPU. You will compare the performance and programming models of these technologies. The parallel machines are nodes in the Robotarium high-performance cluster at Heriot-Watt.

Robotarium cluster website: `https://ecr-cluster.github.io`

Submission instructions

There are two parts to the submission process.

1. **Your source code implementation.** Push your code to the Heriot-Watt GitLab server:
   `https://gitlab-student.macs.hw.ac.uk`
   - Either create 1 GitLab project for each team member (i.e. 2 GitLab projects for your submission), or create 1 GitLab project with you both as Members (Settings → Members when looking at the project page on GitLab) i.e. just 1 GitLab project for your submission. Choose whichever option works best for you as a group.
   - Include the URLs to the GitLab project in your report.
2. **A report**. A PDF should be submitted on Vision (sub-menu Assessment).

Marks will be lost if your report does not following the structure outlined below or does not containing all of the specified results. The deliverable is due on 23^{rd} April 2020.

Learning Objectives of coursework 2

The Learning Outcomes (LO) of this F21DP course are listed at the end of this document.
The learning objectives of coursework 2 is tied to many of these Learning Outcomes:

- Imperative OpenCL and parallel functional programming skills (LO2, LO8).
- Ability to benchmark parallel performance: *efficiency*, *scalability* and *speedup* (LO3, LO6, LO7).
- Compare the OpenMP, OpenCL and Haskell parallel programming models (LO1, LO10).
- Reflect on the programming models that OpenCL and parallel Haskell offer (LO10).
- Peer collaboration on the development of OpenCL and parallel Haskell implementation and comparison (LO9).

Sequential Program

The program that should be parallelised is the computation of the *sum of Euler totient computations* over a range of integer values.

The *Euler totient function* computes, for a given integer n, the number of positive integers smaller than n and relatively prime to n, i.e. $\Phi(n) \equiv\mid \{m \mid m \in \{1, \ldots, n-1\} \wedge m \perp n\} \mid$. Two numbers m and n are relatively prime, if the only integer number that divides both is 1. To test this, it is sufficient to establish that their greatest common divisor is 1, i.e. $m \perp n \equiv \text{gcd}\ m\ n = 1$. Thus, the task for this program is: for a given integer n, compute $\Sigma_{i=1}^{n}\Phi(n)$.

OpenCL reference code

The following C code is a direct implementation of the above specification, as starting point for the parallel algorithm is on GitLab. Your job is to adapt the reference C implementation into an OpenCL kernel and OpenCL program to execute that kernel on a GPU:
`https://gitlab-student.macs.hw.ac.uk/f20dp-2019-20/totient-range`

Haskell reference code

There is also a sequential Haskell reference implementation, as a starting point for parallel implementations. Your job is to use evaluation strategies or algorithmic skeletons to parallelise this code:
`https://gitlab-student.macs.hw.ac.uk/f20dp-2019-20/f20dp-2019-20-haskell-totient`

Organisation

The **assessed coursework work is to be carried out in pairs**, and you should choose your own partner. If you struggle to find a partner to team up with, I suggest you stay in the classroom after one of the lectures to meet other students that are in the same position.

Each member of the team chooses either OpenCL or parallel Haskell, and implements a parallel version of *totient range* in this technology and evaluates the performance of this parallel version:

- OpenCL for GPUs – develop multiple versions using the OpenCL memory and execution models in different ways;
- Parallel Haskell for shared memory multicore CPU parallelism.

Together you can share ideas about how to parallelise the code and how to tune the parallel performance. However, it needs to be clearly stated who implemented the parallel version using which technology. *Together* you should prepare a comparative report using the structure below. It is relatively easy to produce a simple parallelisation of both programs, however *additional marks are available for thoughtful sequential and parallel performance tuning.*

Tools that can help you, and have been discussed, are OpenCL profiling functions and Threadscope for Haskell. You are welcome to use other tools, if you find them useful. In each case you should motivate your choice of tool and reflect how useful it was for tuning performance. For plotting parallel performance results `gnuplot`, `R`, `Excel` or one of many Python libraries are recommended. For Haskell benchmarks, the `criterion` library is recommended – see the README on the GitLab `f20dp-2019-20-haskell-totient` repository.

A video has been created to describe the Sum Totient algorithm to be implemented. The video also gives suggestions on how to measure, plot and analyse your performance results:
`https://web.microsoftstream.com/video/39738d62-57ad-4669-8d97-5de387b81471`

Structure of the report

1. (4 points) Section 1: **Introduction**

   This should give a short summary of the task to implement and parallelise, describe the software and hardware environments it is performed in, the parallel technologies used, and the learning objective of this coursework.

2. (4 points) Section 2: **Sequential Performance Measurements**

   Run the sequential version of the programs and analyse the performance, for an extra mark using a profiling tool such as those mentioned above. Discuss the sequential performance of, and possible improvements to, these programs. Of interest are in particular code on the critical path (hotspots) in the program. *max 1 A4 page*

3. Section 3: **Comparative Parallel Performance Measurements**

   You should measure and record the following results in numbered sections of your report. The measurements are based on these inputs:

   - DS1: calculating the sum of totients between 1 and 15000.
   - DS2: calculating the sum of totients between 1 and 30000.
   - DS3: calculating the sum of totients between 1 and 100000.

   For each of these inputs, measure:

   - SEQ: the sequential runtime (Haskell only) on one of the AMD nodes on the cluster.
   - Parallel Haskell: the runtime of the parallel Haskell implementation one 1-64 cores on an AMD node on the cluster.
   - OpenCL: the runtime on a Nvidia K20 GPU on one of the cluster nodes using different OpenCL configurations e.g. workgroup sizes, dimensions in the NDRange and use of different memory regions.

   Runtime measurements should be repeated 3 times each, or how ever many runs that `criterion` performs for Haskell.

   (a) (6 points) Section 3a: **Runtime Table**

   Present your measurements in a table containing the median (OpenCL) and mean (Haskell) averages of the runtimes as well as difference between the mean and the extreme values. Use a suitable unit to ensure readability.

   (b) (2 points) Section 3b: **Runtime Graphs**

   Present your runtimes in **three** graphs, one for each problem size. For parallel Haskell, The x-axis should be the parallel degree (CPU cores), the y-axis runtime. For OpenCL, you should plot the runtimes for the different OpenCL configurations you have implemented – possibly as a bar graph.

   (c) (2 points) Section 3c: **Speedups: Haskell**

   Plot graphs for parallel Haskell, showing speedup graphs corresponding to the runtime results for DS1, DS2 and DS3. Include the ideal speedup as a line as well. Also show a table with the raw data for Haskell: the sequential performance and the average parallel runtimes using different numbers of a CPU cores.

   (d) (3 points) Section 3d: **Efficiency**

   Plot the corresponding parallelism efficiency graph for parallel Haskell. Analyse the efficiency performance of your Haskell and OpenCL implementations, speculate on the reasons for the efficiency performance.

   (e) (2 points) Section 3e: **Hardware Utilisation**

   OpenCL only: Figure out how many operations on `long` values are being performed for the three different inputs. Then compute the number of operations per second that you achieved in your implementations for the different inputs and for the fixed number of CUDA cores (OpenCL). Plot these findings as graphs and explain how you obtained the number of operations.

(f) (2 points) Section 3f: **OpenCL Implementations**

Describe your various OpenCL implementations. Explain how they differ in terms of how you have mapped the Totient Range algorithm to the OpenCL memory and execution models in different ways.

(g) (5 points) Section 3g: **Discussion**

A discussion of the comparative performance of scalability, efficiency and speedups for your parallel implementations. Try to identify a suitable platform-independent measure of performance and derive those figures from your experiments. Discuss the impact of the shared-memory for parallel Haskell and the OpenCL memory respectively, and how these models might explain your results. Also discuss how Amdahl's law impacts parallel performance of the Totient Range algorithm. *max 1 A4 page*

4. (4 points) Section 4: **Programming Model Comparison**

   An evaluation of the parallel programming models, specifically for implementing the Totient Range application. You should indicate any challenges you encountered in constructing and parallelising your programs and discuss situations/algorithms where each technology may usefully be applied.

5. (12 points) Section 5: **Reflection on Programming Models**

   This section should draw on your experiences in using the different programming models and discuss their suitability for parallelisation of applications in general. The discussion should include aspects such as performance expectations, programmability, advantages/disadvantages, debugging support, performance tuning challenges. You should also *compare the parallel programming models of parallel Haskell with OpenMP* from coursework 1, since they are both primarily designed for multicore CPUs. This discussion needs to be general, but can draw on the experience you gained in using the models on the Totient application. *max 1 A4 page.*

6. (20 points) **Appendix A and B**

   For each parallel implementation, the appendix should include a GitLab URL for your parallel Totient programs. Each implementation should clearly labelled with the single author's name in the report and in the GitLab source code as a comment. Include a paragraph in the appendix, and possibly diagram(s), identifying the *parallel paradigm* used, and *performance tuning approaches* used.

7. (4 points (bonus)) Additional marks are available for thoughtful sequential and parallel performance tuning throughout the report.

The total coursework mark (bonus points are included in your total mark) is calculated as follows:

Question:	1	2	3	4	5	6	7	Total
Points:	4	4	22	4	12	20	0	66
Bonus Points:	0	0	0	0	0	0	4	4

Notes

- Complete the OpenCL and parallel Haskell Lab exercises before starting the coursework.
- Graphs and tables must have appropriate captions, and the axes must have appropriate labels.
- The Robotarium cluster uses a batch-job system to give you exclusive access to the cluster when you need it for measurements. Follow this link to familiarise yourself with the batch-job system.

Learning Outcomes of this Course

The course aims and learning outcomes of this course are available course descriptor online:
`https://www.hw.ac.uk/documents/pams/201920/F21DP_201920.pdf`

The Learning Outcomes of this F21DP course are:

LO1 Understanding of foundational concepts of distributed and parallel software.

LO2 Knowledge and application of contemporary techniques for constructing practical distributed and parallel systems using both declarative and imperative languages.

LO3 Parallel performance tuning using appropriate tools and methodologies.

LO4 Understand the role of control and data abstraction in software design and implementation.

LO5 Appreciation of relationship between imperative and declarative models of parallelism.

LO6 Critically analyse parallel and distributed problems.

LO7 Generate, interpret and evaluate parallel performance graphs.

LO8 Develop original and creative parallel problem solutions.

LO9 Showing initiative, creativity and team working skills in shared distributed and parallel application development.

L10 Demonstrate critical reflection, e.g. understanding of applicability of, and limitations to, parallel and distributed systems.