RHODES UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

EXAMINATION: NOVEMBER 2020

COMPUTER SCIENCE HONOURS
PAPER 1 – GPU PROGRAMMING

Internal Examiner: Prof. K. Bradshaw **MARKS**: 120

External Examiner: Prof. I. Sanders **DURATION**: 2 hours

GENERAL INSTRUCTIONS TO CANDIDATES

1. This paper consists of 7 questions on 8 pages, including the Appendix. ***Please ensure that you have a complete paper.***

2. Answer all questions and please number your answers clearly.

3. State any assumptions and show all workings. Calculators may be used.

4. Diagrams are encouraged and should be labelled.

5. Provide answers that are concise, legible and clearly numbered.

6. To assist in determining the amount of detail to include in an answer, please note the mark allocation. This paper has been set on the basis of *1 mark per 1 min*.

7. Answer all questions in the answer book provided or on the computer or both. Please save all answers on the computer into a single pdf file (`ExamNov2020.pdf`).

8. The Concise Oxford English Dictionary may be used during this examination.

PLEASE DO NOT TURN OVER THIS PAGE UNTIL TOLD TO DO SO

Question 1 (16 marks)

In the context of OpenACC and CUDA, provide a brief definition (note the 2 marks per answer) of each of the following terms:

a) Memory latency
b) Amdahl's law
c) heterogeneous computing
d) compute capability (NVIDIA's context)
e) streaming multiprocessor
f) coalescence
g) gang and vector
h) warp

Question 2 (4+4+4+6 = 18 marks)

a) What is OpenACC and how is it used to parallelize sequential code?

b) How would the execution of the following two snippets of code differ, assuming that each snippet is part of a larger program that compiles?

```
// Code snippet 1
#pragma acc parallel num_gangs(1024)
{
for (int i=0; i<2048; i++) {
…
}
}

// Code snippet 2
#pragma acc parallel num_gangs(1024)
{
#pragma acc loop gang
for (int i=0; i<2048; i++) {
…
}
}
```

c) What are the two major differences between the OpenACC parallel construct and the kernels construct?

d) When profiling an OpenACC program (targeted at a GPU) with *pgprof*, you notice that on each iteration of the main loop in the program, data is transferred between the host and device. Using your knowledge of GPU computing, explain why this is not conducive to good speedup and what pragma(s) you might consider using to eliminate any inefficiency. Be sure to include in your explanation how each of the pragmas work.

Question 3 (8+2+6+6+4 = 26 marks)

a) Accelerate the vector addition code given below for parallel execution on a GPU. You should include as many OpenACC directives as possible to achieve the best optimization.

```
int main () {
    float * input1, * input2, * output;
    int inputLength = 100, j, i;

     input1 = (float *) malloc(inputLength * sizeof(float));
     input2 = (float *) malloc(inputLength * sizeof(float));
     output = (float *) malloc(inputLength * sizeof(float));

      // initialise input
      // Readin in values for input1 and input2 arrays

    for (i = 0; i < inputLength; ++i)
      // summation for-loop
       { output[i] = input1[i] + input2[i]; }

      // Release memory
    free(input1);
    free(input2);
    free(output);

    return 0;
}
```

b) What would need to change in the code or elsewhere if you wished to execute this OpenACC program on a multi-processor CPU?

c) Write a CUDA kernel for a 1-dimensional grid that implements the summation for-loop in the code in part (a). You may assume that each thread sums only one element.

d) What additional changes would you need to make to the code given in part (a) in order to execute the kernel written in part (c) on a CUDA-enabled GPU? You may either rewrite the code or explain the changes/additions that would need to be made.

e) When compiling for a CUDA target, the OpenACC compiler (*pgcc*) uses the various pragmas included in the source code to create a kernel for execution on the GPU. Based on your answer to part (d), explain how the *pgcc* compiler makes decisions regarding this necessary boilerplate code based on the various pragmas that have (or have not) been included.

Question 4 (2+5+3 = 10 marks)

Consider a vector addition kernel written in CUDA C/C++ (such as that created in Question 3c), with the input vector length set as 8000 and the block size as 1024 threads.

a) Assuming that each thread calculates only one output element and that the programmer wishes to configure the kernel launch to have a minimal number of thread blocks to cover all output elements, how many threads will be created in a 1-dimensional grid?

b) Now assume that the programmer wants to use each thread to calculate two output elements in the vector addition. Assume that variable `i` should be initialized with the index of the first element to be processed by each thread. How would you initialize `i` (i.e. fill in the missing expression in the code snippet below) to allow correct, coalesced memory accesses to these first elements by the statements that follow within the kernel? Explain how your answer achieves coalesced access.

```
int i = ____________________ ;

if(i<n) C[i] = A[i] + B[i];
```

c) Continuing from part (b), what would be the correct statement in the kernel for each thread to process the second element?

Question 5 (6+8 = 14 marks)

a) Briefly explain why computer architectures supporting general purpose GPU programming (GPGPU) are typically classed as SIMT rather than SIMD architectures. In your explanation be sure to explain the distinction between SIMD and SIMT architectures.

b) Four common metrics that need to be considered when parallelising a serial computation (especially for execution on a GPU) are the following:

i. occupancy
ii. latency
iii. throughput
iv. scalability

For each metric, give a definition thereof and explain its importance and relevance in creating efficient GPU kernels.

Question 6 (10 marks)

Branch divergence can be very detrimental to the efficient execution of parallel programs. Explain how you could manipulate thread or warp indices to alleviate some divergence. You are encouraged to use an example to clarify your answer.

Question 7 ((6+2+4+2)+6+6 = 26 marks)

Consider the naïve CUDA program for matching 4-char strings in a large text file, the code for which was given to you 24 hours prior to this exam with the suggestion that you profile it. A clean copy of the GPU code is attached as an appendix.

a) Explain what can be done to optimize the naïve code by changing the way GPU memory is used. Be sure to include in your explanation:
   i. exactly what your recommended change entails (you don't have to provide the actual code, but there must be sufficient detail to persuade the examiner that you know what you are doing);
   ii. what aspect of the profiling suggested that this change might be suitable;
   iii. a justification why you feel that this change will result in shorter execution times, by noting, for example, which metrics would be improved, and/or what benefits of the memory model are being leveraged; and
   iv. what limitations this change might impose on other aspects of the code.

b) Suggest one optimization to the execution model to improve speedup of the GPU code. Justify why your suggested optimization might improve speedup by noting which metrics would be improved.

c) Suggest one improvement to increase the amount of parallelism available. Justify why your suggested improvement might improve speedup by noting which metrics would be improved.

END OF EXAMINATION

Appendix: CUDA source code for use with Question 7

```
/* Naive kernel for matching 4-char test strings to strings in a large text */

// Pattern-matching program – unoptimized GPU Oct 2020

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdlib.h>
#include <stdio.h>

void CheckErrMsg(cudaError_t const err, char const* const msg)
{  if (err)
    {
        printf("CUDA Error [%d]: %s\n %s\n",err,cudaGetErrorString(err),msg);
        exit(1);
    }
}

////////////////////////////////////////////////////////////////////////
// define matching routine
////////////////////////////////////////////////////////////////////////

__global__ void gpu_match(unsigned int *d_text, unsigned int *d_words,
       unsigned int *d_outputmatches, int nwords, int length)

{      unsigned int word, found;
       int k = threadIdx.x + blockDim.x * blockIdx.x;
       if (k < length)
        {     d_outputmatches[k] = 0;  // no match initially

// Loop: so that each thread checks for matches starting at
// each of the 4 chars stored in the unsigned int assigned to the thread
             for (int offset = 0; offset < 4; offset++) {
               if (offset == 0) word = d_text[k];
               else
                word = (d_text[k] >> (8 * offset)) +
                                  (d_text[k + 1] << (32 - 8 * offset));

             found = d_outputmatches[k];
              // loop: check all 4 words for matches
             for (int w = 0; w < nwords; w++) {
                    found = found << 8;
                    if (word == d_words[w]) {
                    //set the correct byte in the 4-byte unsigned int to 255
                      found = (found | 255);
                      }
             }
             d_outputmatches[k] = d_outputmatches[k] | found;
       }
   }
}

////////////////////////////////////////////////////////////////////////
// Convert match output to individual totals for each matched word
////////////////////////////////////////////////////////////////////////

void convertoutput(unsigned int *output, int matches[], int nwords, int len)
{
       int temp;
       for (int l = 0; l < len; l++) {
```

```
                temp = output[l];
                for (int w = nwords - 1; w >= 0; w--)
                {
                        if (temp & 255) matches[w]++;
                        temp = temp >> 8;
                }
        }
        return;
}

////////////////////////////////////////////////////////////////////////
// Main program
////////////////////////////////////////////////////////////////////////

int main(int argc, const char **argv) {

        cudaError_t err = cudaSuccess;   // keep track of errors
        char *ctext, *cwords[] = { "TTTT", "TACT", "TAAC", "ACTA" };
        unsigned int  *text, *words, *h_matchoutput,
                      *d_matchoutput, *d_text, *d_words;

        int   length, len, nwords = 4, matches[4] = { 0, 0, 0, 0 },
              blocks, threads;

        err = cudaSetDevice(0);
        CheckErrMsg(err, "cudaSetDevice failed! Do you have CUDA-capable GPU?");

        // read in text for processing
        FILE *fp;
        if (!(fp = fopen("data.txt", "r")))
                { printf("Error reading input file \n"); return 10; };

        length = 0;
        while (getc(fp) != EOF) length++;
        if (length == 0) { printf("Error reading input file \n"); return 10; }

        ctext = (char *)malloc(length + 4);
        err = cudaMalloc((void **)&d_text, length + 4);
        CheckErrMsg(err, "cudaMalloc failed!  text");

        rewind(fp);

        for (int l = 0; l < length; l++) ctext[l] = getc(fp);
// padding for final word check
        for (int l = length; l < length + 4; l++) ctext[l] = ' ';
        fclose(fp);

// define number of words of text, and set pointers
        len = length / 4;
        text = (unsigned int *)ctext;


// create output array to record what matches are found at each location
        int outsize = len * sizeof(unsigned int);

// Unsigned int has the 4 match words in order from left to right
        h_matchoutput = (unsigned int *)malloc(outsize);
        err = cudaMalloc((void **)&d_matchoutput, outsize);
        CheckErrMsg(err, "cudaMalloc failed!  matchout");

        // define words for matching
        words = (unsigned int *)malloc(nwords * sizeof(unsigned int));
        err = cudaMalloc((void **)&d_words, nwords * sizeof(unsigned int));
```

```
	CheckErrMsg(err, "cudaMalloc failed!  words");

// change each word for matching into single unsigned int to facilitate
comparison
// i.e. don't have to compare each of the 4 letters of the words for matching
separately.
	for (int w = 0; w < nwords; w++) {
		words[w] = ((unsigned int)cwords[w][0])
			+ ((unsigned int)cwords[w][1]) * 256
			+ ((unsigned int)cwords[w][2]) * 256 * 256
			+ ((unsigned int)cwords[w][3]) * 256 * 256 * 256;
	}

	threads = 512;
	blocks = ((len + threads -1)/threads);

	err =  cudaMemcpy(d_text, ctext, length + 4, cudaMemcpyHostToDevice);
	CheckErrMsg(err, "cudaMemcpy failed!  text");
	err = cudaMemcpy(d_words, words, nwords * sizeof(unsigned int),
	cudaMemcpyHostToDevice);
	CheckErrMsg(err, "cudaMalloc failed!  words");

	gpu_match<<<blocks, threads>>>(d_text, d_words, d_matchoutput, nwords,
len);
	err = cudaGetLastError();
	CheckErrMsg(err, "kernel failed!");

	cudaDeviceSynchronize();
	err = cudaMemcpy(h_matchoutput, d_matchoutput, outsize,
	cudaMemcpyDeviceToHost);
	CheckErrMsg(err, "cudaMemcpy failed!  matchoutput");

	convertoutput(h_matchoutput, matches, nwords, len);
	printf(" GPU matches = %d %d %d %d \n",
		matches[0], matches[1], matches[2], matches[3]);

	// Release GPU and CPU memory
	cudaFree(d_matchoutput);
	cudaFree(d_text);
	cudaFree(d_words);
	free(ctext);
	free(words);
	free(h_matchoutput);
}
```