RHODES UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

EXAMINATION: JUNE 2018

COMPUTER SCIENCE HONOURS
PAPER 1 – GPU PROGRAMMING

Internal Examiner: Prof. K. Bradshaw

MARKS: 120
DURATION: 2 hours

External Examiners: Prof. M. Kuttel

GENERAL INSTRUCTIONS TO CANDIDATES

1. This paper consists of 8 questions and 2 Appendices on 11 pages. ***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. Provide answers that are concise, legible and clearly numbered.
5. Answer all questions in the answer book provided or in your preferred word processor, and SAVE regularly to your exam folder.
6. Diagrams are encouraged and should be clearly and neatly labelled. Any diagrams in the examination book must be clearly referenced.
7. 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) pragma
b) gang and vector
c) occupancy
d) parallel reduction
e) warp
f) bank conflict
g) redundant computation
h) padding

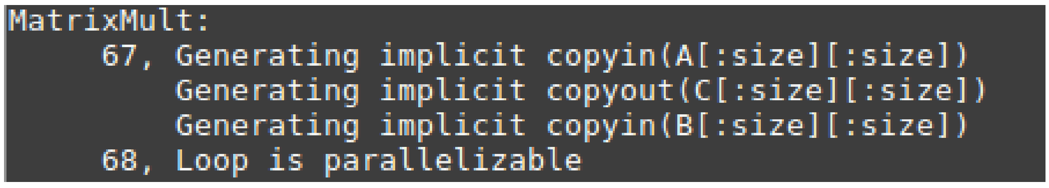
Question 2 **(5+4+5=14 marks)**

a) Code for a simple `for` loop is given below. Parallelise this loop using the OpenACC parallel and loop directives. Give the new version of the code and explain what the effect is of any changes you made.

```
for (i=0; i<N; i++)
     { a[i] = a[i] + b[(i+1) % N]; }  // % means modulo
```

b) What are the advantages and disadvantages of using the kernels pragma to achieve the parallelisation in (a) above?

c) Consider the screenshot below showing the output from the pgcc compiler while compiling a matrix multiplication OpenACC program. Explain what each line of output means.



```
MatrixMult:
     67, Generating implicit copyin(A[:size][:size])
         Generating implicit copyout(C[:size][:size])
         Generating implicit copyin(B[:size][:size])
     68, Loop is parallelizable
```

Question 3 **(5+6=11 marks)**

Consider Figure 1 in Appendix A, which was taken from a pgprof profiling session of an OpenACC program.

a) Explain in as much detail as possible, what inefficiencies are clearly obvious from the information provided in this profiling screen dump.

b) Using your knowledge of GPU computing, explain what pragmas you could use to optimize the source code to eliminate the inefficiency identified in (a). Be sure to include in your explanation how these pragmas work.

Question 4 **(12+4=16 marks)**

a) Flynn's taxonomy is a widely used scheme for classifying computer architectures. Explain the classification types in this scheme, giving a current architectural example of each (if possible).

b) Define Amdahl's law and explain its relevance to parallel processing.

Question 5 **(3+9+2=14 marks)**

GPUs can provide significant speedup when processing very large scale datasets as long as global memory does not become a limiting factor.

a) Define global load efficiency and state why this metric is important in GPU programming.

b) There are various access patterns for global memory loads that need to be considered when trying to optimize global load efficiency. Briefly describe three different patterns and also state what the ideal scenario of load addresses is for each pattern.

c) What two GPU hardware limitations affect global load and store efficiency?

Question 6 **(6+3+2+4=15 marks)**

a) Explain how modern GPUs handle branch divergence, e.g. that arising from the execution of an if-else statement.

b) Explain how the metric for branch divergence is calculated.

c) What effect does high branch divergence have on the throughput metric?

d) What could a programmer consider doing to reduce branch divergence from if-else statements?

Question 7 **(4+6=10 marks)**

a) Describe what a CUDA stream is.

b) Explain how streams can be used to hide the latency of various operations involving the host and device to achieve greater parallelism.

Question 8 **(8+8+8=24 marks)**

Consider the four CUDA kernels for copying and transposing matrices, the code for which was handed to you 24 hours prior to this exam with the suggestion that you profile it (with L1 cache disabled) to understand the execution efficiencies. The code is attached as Appendix B to this exam paper.

To answer the questions below, you will also need to refer to the screen dumps (in Appendix A) of various profiling sessions using the CUDA kernels.

a) When profiling copyRow (Figure 2), the nvvp profiler reports that "Kernel performance is bound by memory bandwidth". Explain what this means and what the programmer should consider to remove this limitation.

b) When profiling copyCol (Figure 3), the nvvp profiler reported that "Kernel performance is bound by instruction and memory latency". Explain what this means and what optimizations the programmer should consider to improve performance.

c) Consider the metrics obtained by the profiler for the two transpose kernels (i.e., transposeNaiveCol and transposeNaiveRow) given in the table below. With reference to the source code for these kernels, explain why the metric values differ.

Name	**transposeNaiveRow (float*, float*, int, int)**	**transposeNaiveCol (float*, float*, int, int)**
Duration (ns)	2124096	902679
Achieved Occupancy	0.848	0.893
Global Memory Load Efficiency (%)	100	25
Global Memory Store Efficiency (%)	12.5	100
Device Memory Read Throughput (bytes/sec)	9541033304	18749652511
Instructions Executed	4063232	4063232
Device Memory Write Throughput (bytes/sec)	7847116828	18634918758
Device Memory Utilization	Low (3)	Mid (5)

END OF EXAMINATION

Appendix A: Profiler Screen Dumps

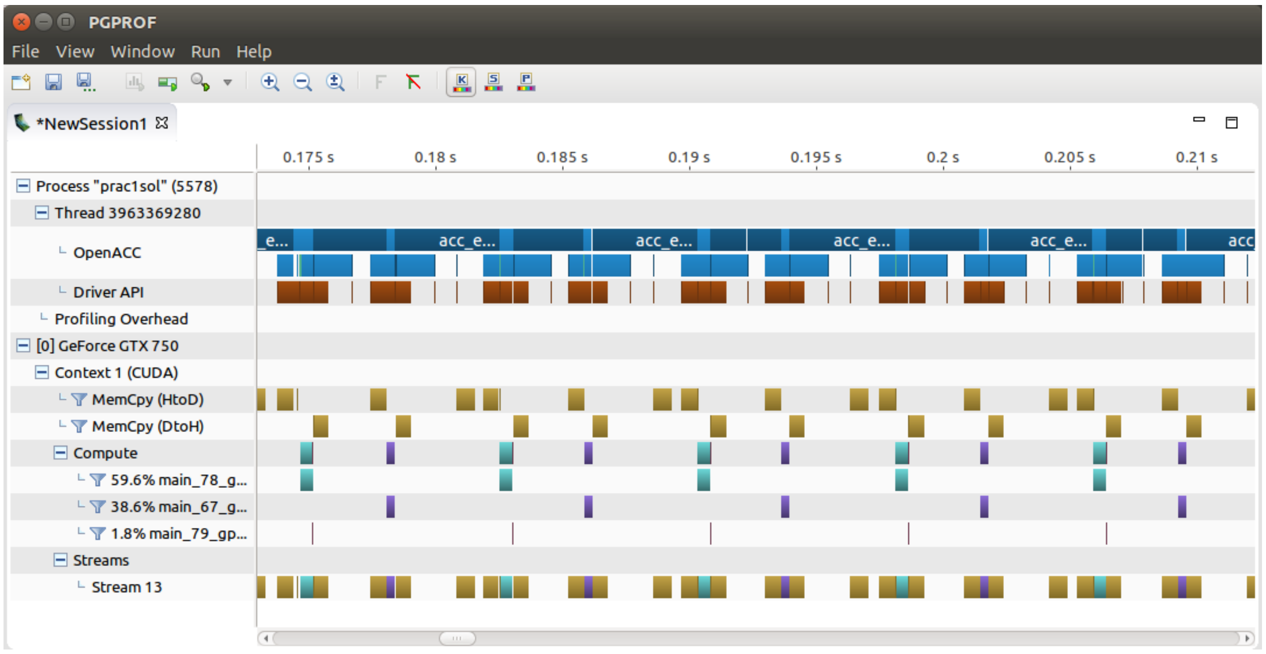
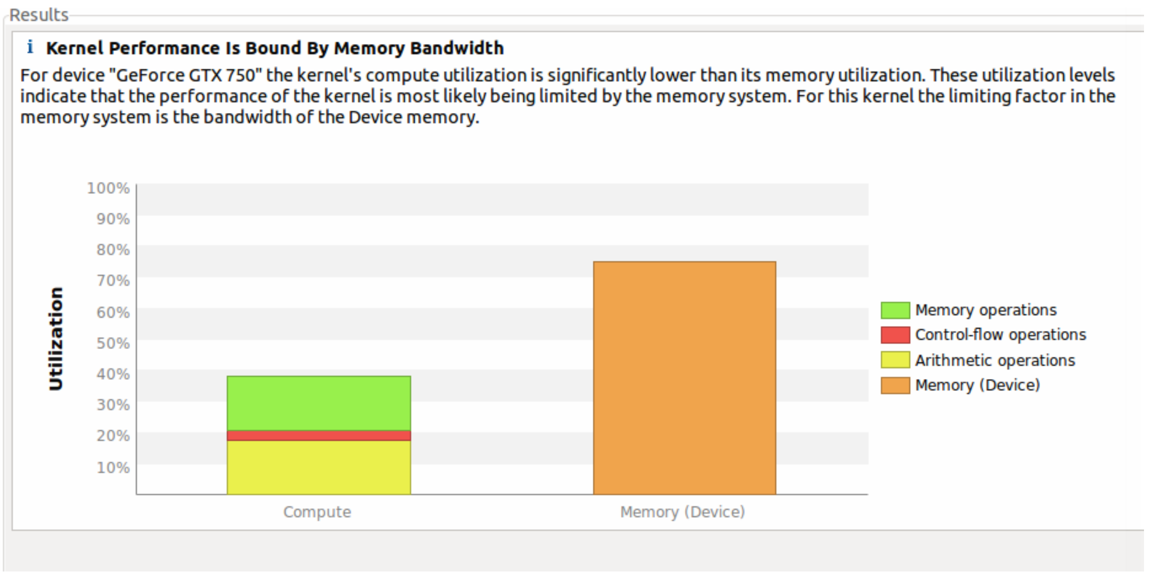


Figure 1: PGPROF profiling session: for use with Question 3

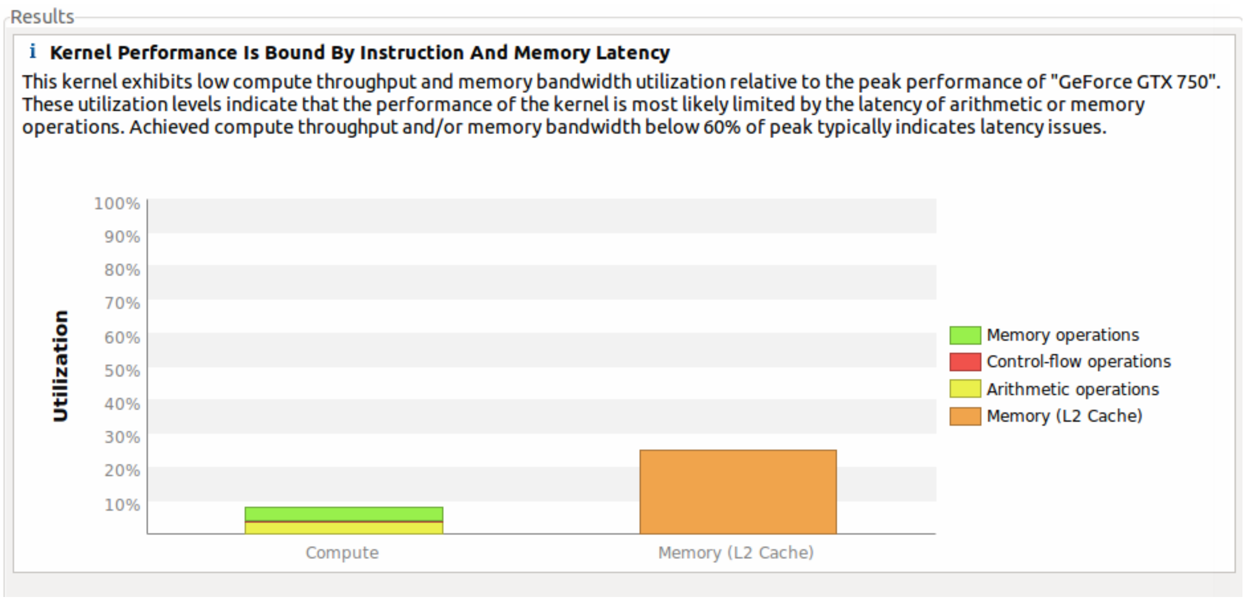


Results

Kernel Performance Is Bound By Memory Bandwidth

For device "GeForce GTX 750" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Device memory.

Figure 2: NVVP profiling session of copyRow(): for use with Question 8(a)



Results

Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 750". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

Figure 3: NVVP profiling session of copyCol(): for use with Question 8(b)

Appendix B: CUDA source code for use with Question 8

```
/* Naive kernels for copying and transposing a rectangular host matrix */

#include <cuda_runtime.h>
#include <stdio.h>

#define CHECK(call)                                                         \
{                                                                           \
    const cudaError_t error = call;                                         \
    if (error != cudaSuccess)                                               \
    {                                                                       \
        fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);              \
        fprintf(stderr, "code: %d, reason: %s\n", error,                    \
                cudaGetErrorString(error));                                 \
        exit(1);                                                            \
    }                                                                       \
}


void initialData(float *in,  const int size)
{  // initialise matrix
    for (int i = 0; i < size; i++)
        {   in[i] = (float)(rand() & 0xFF) / 10.0f;    }
    return;
}

void printData(float *in,  const int size)
{  // print matrix
    for (int i = 0; i < size; i++)
        {   printf("%3.0f ", in[i]);     }
    printf("\n");
    return;
}

void checkResult(float *hostRef, float *gpuRef, int rows, int cols)
{  // check that transposed matrix is correct
    double epsilon = 1.0E-8;
    bool match = 1;

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            int index = i*cols + j;
            if (abs(hostRef[index] - gpuRef[index]) > epsilon) {
                match = 0;
                printf("different on (%d, %d) (offset=%d) element in transposed
matrix: host %f gpu %f\n", i, j,  index, hostRef[index], gpuRef[index]);
                break;
            }
        }
        if (!match) break;
    }

    if (!match)  printf("Arrays do not match.\n\n");
}
```

```
void transposeHost(float *out, float *in, const int nrows, const int ncols)
{  // transpose using CPU
    for (int iy = 0; iy < ncols; ++iy)
    {
     for (int ix = 0; ix < nrows; ++ix)
        {   out[ix * ncols + iy] = in[iy * nrows + ix];     }
    }
}

// case 0 copy kernel: access data in rows (both reads and writes)
__global__ void copyRow(float *out, float *in, const int nrows, const int ncols)
{   // routine to copy data from one matrix to another -- no transposition done
    // get matrix coordinate (ix,iy)
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

     // copy data as is with boundary test
       if (ix < nrows && iy < ncols)
         {  out[iy * nrows + ix] = in[iy * nrows + ix];   }
}

// case 1 copy kernel: access data in columns (both reads and writes)
__global__ void copyCol(float *out, float *in, const int nrows, const int ncols)
{    // routine to copy data from one matrix to another -- no transposition done
    // get matrix coordinate (ix,iy)
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    // copy data as is with boundary test
       if (ix < nrows && iy < ncols)
         { out[ix * ncols + iy] = in[ix * ncols + iy];  }
}

// case 2 transpose kernel: read in rows and write out columns
__global__ void transposeNaiveRow(float *out, float *in, const int nrows,
                                  const int ncols)
{   // naive routine to transpose a matrix -- no optimisations considered
    // get matrix coordinate (ix,iy)
    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    // transpose with boundary test
       if (ix < nrows && iy < ncols)
       {  out[ix * ncols + iy] = in[iy * nrows + ix]; }
}

// case 3 transpose kernel: read in columns and write out rows
__global__ void transposeNaiveCol(float *out, float *in, const int nrows,
                                  const int ncols)
{   // naive routine to transpose a matrix -- no optimisations considered
    // get matrix coordinate (ix,iy)

    unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;

    // transpose with boundary test
       if (ix < nrows && iy < ncols)
        {  out[iy * nrows + ix] = in[ix * ncols + iy];  }
}
```

```
int main(int argc, char **argv)
{
    // set up device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("%s starting transpose at ", argv[0]);
    printf("device %d: %s ", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // initialise CUDA timing
    float milli;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    bool iprint = 0;

    // set up default array size 2048
    int nrows = 1 << 11;
    int ncols = 1 << 11;

    // select block size
    int blockx = 16;
    int blocky = 16;

    // interpret command line arguments if present
    if (argc > 1) iprint = atoi(argv[1]);   // whether to print (1) or not (0)
    if (argc > 2) blockx  = atoi(argv[2]);  // Block x-dim
    if (argc > 3) blocky  = atoi(argv[3]);  // Block y-dim
    if (argc > 4) nrows  = atoi(argv[4]);   // Rectangular matrix num of rows
    if (argc > 5) ncols  = atoi(argv[5]);   // Rectangular matrix num of cols

    printf(" with matrix nrows %d ncols %d \n", nrows, ncols);
    size_t ncells = nrows * ncols;
    size_t nBytes = ncells * sizeof(float);

    // execution configuration
    dim3 block (blockx, blocky);
    dim3 grid  ((nrows + block.x - 1) / block.x, (ncols + block.y - 1) / block.y);

    // allocate host memory
    float *h_A = (float *)malloc(nBytes);
    float *hostRef = (float *)malloc(nBytes);
    float *gpuRef  = (float *)malloc(nBytes);

    //  initialize host array
    initialData(h_A, nrows * ncols);

    //  transpose at host side
    transposeHost(hostRef, h_A, nrows, ncols);

    // allocate device memory
    float *d_A, *d_C;
    CHECK(cudaMalloc((float**)&d_A, nBytes));
    CHECK(cudaMalloc((float**)&d_C, nBytes));

    // copy data from host to device
    CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
    // execute copy kernels
    CHECK(cudaMemset(d_C, 0, nBytes));
    memset(gpuRef, 0, nBytes);

    cudaEventRecord(start); // start timing
    int iKernel = 0;
    copyRow<<<grid, block>>>(d_C, d_A, nrows, ncols); // execute this to run kernel 0
       // OR these statements to run kernel 1
       // iKernel = 1; copyCol<<<grid, block>>>(d_C, d_A, nrows, ncols);
    CHECK(cudaGetLastError());
    CHECK(cudaDeviceSynchronize());
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&milli, start, stop);  // stop timing actual kernel execution

    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprint) printData(gpuRef, nrows * ncols);

    float ibnd = 2 * ncells * sizeof(float) / (1024.0 * 1024.0 * 1024.0) /
(milli/1000);  // convert bytes and millisec to GB/sec
    printf("Copy kernel %d elapsed %f msec <<< grid (%d,%d) block (%d,%d)>>> effective
bandwidth %f GB/s\n", iKernel, milli, grid.x, grid.y, block.x, block.y, ibnd);

    // execute naive transpose kernels
    CHECK(cudaMemset(d_C, 0, nBytes));
    memset(gpuRef, 0, nBytes);

    cudaEventRecord(start); // start timing
       // execute these statements to run kernel 2
       iKernel = 2; transposeNaiveRow<<<grid, block>>>(d_C, d_A, nrows, ncols);
       // OR these statements to run kernel 3
       // iKernel = 3; transposeNaiveCol<<<grid, block>>>(d_C, d_A, nrows, ncols);

    CHECK(cudaGetLastError());
    CHECK(cudaDeviceSynchronize());
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&milli, start, stop);  // stop timing actual kernel execution

    CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));

    if(iprint) printData(gpuRef, ncells);

    checkResult(hostRef, gpuRef, ncols, nrows);
    ibnd = 2 * ncells * sizeof(float) / (1024.0 * 1024.0 * 1024.0) / (milli/1000);

    printf("Transpose naive kernel %d elapsed %f msec <<< grid (%d,%d) block
(%d,%d)>>> effective bandwidth %f GB/s\n", iKernel, milli, grid.x, grid.y, block.x,
block.y, ibnd);

    // free host and device memory and reset device
    CHECK(cudaFree(d_A));  CHECK(cudaFree(d_C));
    free(h_A);             free(hostRef);
    free(gpuRef);
    CHECK(cudaDeviceReset());
    return EXIT_SUCCESS;
}
```