ZJU-UIUC Institute
First Exam, ECE 408

Tuesday 9 April 2019

Name (pinyin and Hanzi):

Student ID: SOLUTION

- **Be sure that your exam booklet has EIGHT pages.**
- **Write your name and Student ID on the first page.**
- **Do not tear the exam apart.**
- **This is a closed book exam. You may not use a calculator.**
- **You are allowed one handwritten A4 sheet of notes (both sides).**
- **Absolutely no interaction between students is allowed.**
- **Show all work, and clearly indicate any assumptions that you make.**
- **Don't panic, and good luck!**

Problem 1	28 points	_____
Problem 2	30 points	_____
Problem 3	42 points	_____
Total	100 points	_____

Problem 1 (28 points): Short Answer Questions

1. **(4 points)** Consider the problem of dithering. Given an image in which pixels are encoded using a large number of colors (say 24-bit RGB), the goal is to recolor each pixel using a smaller number of colors (called the palette, with perhaps 256 colors) in such a way that humans continue to perceive the original image.

   A popular sequential dithering algorithm (Floyd-Steinberg) works as follows: starting with the upper-left pixel, choose the palette color that is 'closest' to the original RGB color, calculate the difference between these two colors as an error vector, and linearly spread the error vector amongst pixels to the right and below the current pixel (about ½ of the error goes to the pixel to the right, and the remainder goes to pixels in the next row below the current pixel). Continue to the right across the current row, adjusting RGB values using error information, selecting palette colors, calculating error vectors, and spreading them across pixels to the right and below. When one row is done, proceed to the leftmost pixel in the next row down, until the entire image has been dithered.

   Is this algorithm more suitable to a CPU or a GPU? Explain your answer.

   CPU, because of the sequential dependences across columns and from row to row.
   (A parallel wavefront is possible, but complex.)

2. **(4 points)** Prof. Lumetta wants to push his autograding technology to new levels and plans to make use of GPUs to accelerate the computation. When grading an assignment, correct answers are easy to verify, but incorrect answers require much more work: classification of the students' error, mapping to the rubric for the problem, and assignment of partial credit. Prof. Lumetta plans to assign one warp of GPU threads to each student to avoid control divergence. What other high-level challenge with parallel execution might be an issue for Prof. Lumetta's approach? Explain your answer.

   Load imbalance: threads have highly variable timing, and the scheduler waits for the last to finish (same with syncthreads, but larger pool of threads, so even worse).

3. **(4 points)** A friend is trying to map a linear filter onto GPUs. The friend has written the code below to compute `N_d` based on `M_d`, but `N_d` always seems to be filled with 0s after the kernel finishes executing.

```
cudaMalloc ((void **)&M_d, 1048576 * sizeof (float));
cudaMalloc ((void **)&N_d, 1048676 * sizeof (float));

// Read from input file f into M_d.
fread (M_d, sizeof (float), 1048576, f);

executeKernel<<<1024, 1024>>> (M_d, N_d, 1048576);

// Make use of N_d.
// ...
```

   Explain what your friend is doing wrong.

   Must use CudaMemcpy to copy host memory to GPU memory. fread cannot use a GPU address directly. (Also need to copy back before using on host.)

Problem 1, continued:

4. **(4 points)** A friend in ECE408 is having problems calculating the coordinate values for his 3D voxel input data from the thread and block indices of the CUDA threads. Each thread operates on one voxel from the input `V`. Your friend has launched the kernel as follows:

```
dim3 one (10, 20, 15);
dim3 two (18, 12, 24);
doMyKernel<<<one, two>>> (V, Q);
```

The X, Y, and Z dimensions of the thread blocks are aligned with the X, Y, and Z dimensions of the original input array `V`. Write CUDA expressions to calculate the three coordinates within `V` as a function of the `gridDim`, `blockDim`, `blockIdx`, and `threadIdx` variables provided to each thread.

`x` = `blockDim.x * blockIdx.x + threadIdx.x;` ;

`y` = `blockDim.y * blockIdx.y + threadIdx.y;` ;

`z` = `blockDim.z * blockIdx.z + threadIdx.z;` ;

5. **(4 points)** In question 3, the voxel array `V` is organized as a 3-dimensional array using the standard C approach. In particular, as a 3D array, it would appear as `[Z size][Y size][X size]`. As is often the case in CUDA code, however, `V` is instead simply a pointer to a single voxel. Use the variables `x`, `y`, and `z` that you calculated in question 3 along with information in the kernel launch parameters to compute an index into `V` for the given thread (you need not perform the multiplications yourself—just write them as C expressions):

`V_index` = `(z * 20*12 + y) * 10*18 + x` ;

6. **(4 points)** Recall the forward computation process for a CNN, and in particular the computation of subsampled/pooled results by averaging over N×N tiles of the convolution output Y before applying the sigmoid function.

Explain why a GPU kernel that couples calculation of Y with subsampling is likely to produce better performance than is possible using two separate GPU kernels executed in sequence.

Subsampling requires only 1 FLOP per value, so the performance is memory-bandwidth bound. The combined kernel avoids the added overhead of reading Y back from global memory.

7. **(4 points)** Explain why the sign function (sign(x) = 1 for x > 0, sign (0) = 0, and sign (x) = -1 for x < 0) is not used with DNNs.

sign(x) is not differentiable, so using the chain rule to backpropagate errors for gradient descent doesn't work.

the derivative is defined and has derivative 0 wherever it

Problem 2 (30 points): Operating on Vectors

A friend comes to you for advice about GPU programming. The friend needs to calculate a function F on each of N points in a three-dimensional dataset. A point has X, Y, and Z coordinates, and each point is used only once (to calculate the function for that point).

1. **(4 points)** Your friend first asks: is it better to organize the dataset as `float data[N][3]` or as `float data[3][N]`? Explain your answer.

float data[3][N] to maximize the GPU threads' ability to coalesce memory accesses when reading coordinates. with data[N][3], any coordinate read obtains at most 1/3 of peak bandwidth!

2. **(2 points)** Each SM in your friend's GPU supports up to 1,024 threads and up to 4 thread blocks. How many threads per block are needed to maximize use of both resources? Show your work.

$\frac{1024}{4} = 256$ threads/block

3. **(4 points)** Assuming the thread block size that you gave in **part (2)** above, how many warps will experience control divergence, assuming that N = 2,000 and that warp size is 32. Explain your answer.

only the warp crossing the boundary (threads 2084 to 2115) experiences control divergence. Other warps are all active or all inactive.

$\lceil \frac{2000}{256} \rceil \cdot 256 = 2048$ threads

4. **(2 points)** Your friend is considering buying a new GPU in which each SM supports up to 1,536 threads and up to 8 thread blocks. How many threads per block are needed to maximize use of both resources? Show your work.

$\frac{1536}{8} = 192$ threads/block

5. **(2 points)** Assuming the thread block size that you gave in **part (4)** above, how many total threads will be launched if N = 1,000? Show your work.

$\lceil \frac{1000}{192} \rceil \cdot 192 = 1152$ threads

Problem 2, continued:

6. **(4 points)** Can you help your friend to write code that operates effectively on both GPUs, or does your friend need to modify the code after purchasing the new GPU? If you can help, explain how.

Use CUDA device query to obtain GPU-specific limits and calculate block size in host code.

7. **(4 points)** Your friend's GPU offers a peak computational performance of 1.28 TFLOPS (1.28×10^{12} single-precision floating-point operations per second) and has a memory bandwidth of 160 GB/s. Assuming that both resources are used with perfect efficiency, how many floating-point operations are needed to calculate F for one point in order to maximize use of both resources. Show your work.

Read 3 coordinates, write one value out. All floats. So 16 B / element.

$$\frac{1.28 \times 10^{12} \text{ FLOP/s}}{160 \times 10^{9} \text{ B/s} \;/\; 16 \text{ B/element}} = \frac{1280}{160/16} \frac{\text{FLOP}}{\text{element}} = 128 \text{ floating-point operations for each point (!)}$$

8. **(8 points)** Your friend has started to write the GPU code, but has made a couple of mistakes. Correct the code and write the kernel launch using the block size that you gave in **part (4)** on the previous page.

```
// data array of N 3-d points (float*) already populated
// result array of N function values (float*) already allocated
float* data_d;    // data in GPU memory
float* result_d;  // result in GPU memory


// allocate space for data and result in GPU memory
cudaMalloc ((void**) &data_d, sizeof (float) * N * 3);
cudaMalloc ((void**) &result_d, sizeof (float) * N);


// copy into GPU memory
memcpy (data_d, data, sizeof (float) * N * 3);   // host to device
```

cudaMemcpy

cudaMemcpyHostToDevice

```
// what else goes here?  CUDA is so confusing!
theKernel<<< ceil(N/192.0), 192>>> (data_d, result_d, N);


// copy results back to host memory
memcpy (result, result_d, sizeof (float) * N);   //device to host
```

cudaMemcpy

cudaMemcpyDeviceToHost

// could free device memory here

```
// later in the file, the kernel ... parameters needed?
int __global__ void
theKernel (float* data_d, float* result_d, int N)
{
    // assume that this code has been written correctly
}
```

Problem 3 (42 points): Convoluted Code

At your first internship, you are asked to complete a former co-worker's 2D convolution code. The application for which this code is being written convolves a mask with an input `P` and a second mask with an input `Q`, then adds the two convolutions together to produce the final `result`. `P` and `Q` both have dimension (`y_dim` × `x_dim`), as does the `result`, and the two masks are defined by the code. Both masks have the same size. In the convolutions, values beyond the boundaries of `P` and `Q` should be treated as 0.

The code should use "strategy 2," in which all threads load an input tile to shared memory and a subset of threads compute the output.

1. **(4 points)** Fill in the input tile size definitions (see "`(1)`" in the code below).

```
#define OUT_TILE_X 8
#define OUT_TILE_Y 8
#define MASK_X     5
#define MASK_Y     7

#define IN_TILE_X  (OUT_TILE_X + MASK_X - 1)   // (1)

#define IN_TILE_Y  (OUT_TILE_Y + MASK_Y - 1)   // (1)

__constant__ float mask_P[MASK_Y][MASK_X];
__constant__ float mask_Q[MASK_Y][MASK_X];

__global__
void sumOfConv (float* P, float* Q, float* result,
                const int y_dim, const int x_dim)
{
    __shared__ tile_P[IN_TILE_Y][IN_TILE_X];
    __shared__ tile_Q[IN_TILE_Y][IN_TILE_X];

    int tx = threadIdx.x, ty = threadIdx.y;
    int bx = blockIdx.x,  by = blockIdx.y;

    int x_out = bx * OUT_TILE_X + tx, x_in = x_out - MASK_X / 2;
    int y_out = by * OUT_TILE_Y + ty, y_in = y_out - MASK_Y / 2;

    // (2): Is the tile-reading code correct?
    __syncthreads ();
    tile_P[ty][tx] = P[y_in * grimDim.x * OUT_TILE_X + x_in];
    __syncthreads ();
    tile_Q[ty][tx] = Q[y_in * gridDim.x * OUT_TILE_X + x_in];
    __syncthreads ();

    float sum = 0.0f;
    // (3): Which threads should compute a sum?
    for (int ym = 0; MASK_Y > ym; ym++) {
        for (int xm = 0; MASK_X > xm; xm++) {
            sum += mask_P[ym][xm] * tile_P[ty + ym][tx + xm];
            sum += mask_Q[ym][xm] * tile_Q[ty + ym][tx + xm];
        }
    }

    // (4): Need to synchronize here?

    // (5): Which threads should write output?
    res[y_out * x_dim + x_out] = sum;
}
```

Problem 3, continued:

2. **(10 points)** Rewrite the block of code to read the two tiles into shared memory so that it works correctly and with minimal synchronization (see "`(2)`" in the code).

```
if (0 <= x_in && x_dim > x_in && 0 <= y_in && y_dim > y_in) {
    tile_P[ty][tx] = P[y_in * x_dim + x_in];
    tile_Q[ty][tx] = Q[y_in * x_dim + x_in];
} else {
    tile_P[ty][tx] = tile_Q[ty][tx] = 0.0f;
}
__syncthreads();
```

3. **(4 points)** Should all threads in a block compute a sum? If not, write an `if` condition that can be used to wrap the summation (convolution) code (see "`(3)`" in the code).

(No)

```
tx < OUT_TILE_X && ty < OUT_TILE_Y
```

(can include extra conditions from #5 below here, too)

4. **(4 points)** Do threads in a block need to synchronize after computing the convolution (see "`(4)`" in the code)? Explain your answer.

No. shared memory is not reused (re-written) after the computation, nor does any thread need the sum produced by any other thread.

5. **(4 points)** Should all threads write a value back to `result`? If not, write an `if` condition that can be used to wrap the global memory write operation (see "`(5)`" in the code). If your condition should include the condition that you gave for **part (3)** above, you may simply write "**CONDITION (3)**" instead of writing that part of the condition again, but be sure to connect it with an appropriate C operator.

(No)

```
CONDITION(3) && x_dim > x_out && y_dim > y_out
```

Problem 3, continued:

6. **(6 points)** On average across the tile, how many times is each element of `mask_P` read during the computation of the convolution? *No need to calculate a decimal value—just write the appropriate fraction.* Show your work.

oops! meant tile_P; mask_P is 8×8

reads values $\frac{\text{OUT\_TILE\_X} * \text{OUT\_TILE\_Y} * \text{MASK\_X} * \text{MASK\_Y}}{\text{IN\_TILE\_X} * \text{IN\_TILE\_Y}} = \frac{8 \cdot 8 \cdot 5 \cdot 7}{12 \cdot 14}$ ← can stop here

$= \frac{40}{3} = 13\frac{1}{3}$

don't need to carry this far

7. **(6 points)** Using the defined constants from the code and the input dimensions (`y_dim` × `x_dim`), write code to dimension the grid and blocks for the kernel launch below.

```
dim3 dimGrid(ceil(x_dim/(float)OUT_TILE_X), ceil(y_dim/(float)OUT_TILE_Y), 1);
dim3 dimBlock(IN_TILE_X, IN_TILE_Y, 1);

sumOfConv<<dimGrid, dimBlock>> (P_d, Q_d, result_d, y_dim, x_dim);
```

8. **(4 points)** Your manager finds that while your code produces the correct results, its performance is limited by the shared memory available in the company's GPUs. Can you suggest a modification to improve the performance? *Hint: You will not earn credit for saying that the company should use a single convolution and add the two results afterward.*

Reuse one tile of shared memory, adding barriers as necessary...

read P to shared mem.
syncthreads
convolve with P (retain sum)
syncthreads
read Q to shared mem.
syncthreads
convolve with Q, adding to known sum of P