ECE 408/CS 483/CSE 408: Applied Parallel Programming

Fall 2023 – Midterm Exam 1

October 10, 2023

1. This is a closed book exam except for 1 sheet of hand-written notes.
2. You may not use any personal electronic devices except for a simple calculator.
3. Absolutely no interaction between students is allowed.
4. Illegible answers will likely be graded as incorrect.

Good Luck!

Name: ____

NetID: ____

UIN: ____

Exam Room: __________________

Question 1 (20 points): ____________________

Question 2 (20 points): ____________________

Question 3 (30 points): ____________________

Question 4 (30 points): ____________________

Total Score: ______________________________

Problem 1 (20 points): CUDA Basics

Choose the proper response, and if multiple responses are correct, choose all. No partial credit will be provided if the answer is partially correct, or wrong.

1. If we want to allocate an array of `n` floating point numbers in the GPU global memory and have a pointer variable called `device_array` point to this array, what is the correct call to `cudaMalloc()` on the host side? Circle the correct answer.

```
1. cudaMalloc(device_array, n*sizeof(float));
2. cudaMalloc((void *)device_array, n*sizeof(float));
3. cudaMalloc((void)&device_array, n*sizeof(float));
4. cudaMalloc((void **)&device_array, n*sizeof(float));
```
5. None of these answers are correct.

2. If we want to copy an array, `host`, of 5 floating-point numbers from the host memory to the GPU constant memory and have a pointer variable called `device_array` point to this constant memory array, what is the correct call on the host side? Circle the correct answer.

```
1. cudaMemcpy(device_array, host, 5*sizeof(float), cudaMemcpyH2D);
2. cudaMemcpyToSymbol(device_array, host, 5*sizeof(float));
3. cudaMemcpy(host, device_array, 5*sizeof(float));
4. cudaMemcpyToSymbol(host, device_array, 5*sizeof(float));
```
5. None of these answers are correct.

3. How many CUDA threads are in each block as the result of the following kernel call?

```
#define VECTOR_N 1024
#define ELEMENT_N 256
...
scalarProd<<VECTOR_N, ELEMENT_N>>>(d_C, d_A, d_B, ELEMENT_N);
```

1. 1024*256
2. 1024
3. 256
4. 1024+256

4. How do we declare a 5x5 constant memory floating point array `Mc`?

1. `constant float[5][5] Mc;`
2. `constant float Mc[5][5];`
3. `__constant__ float[5][5] Mc;`
4. `__constant__ float Mc[5][5];`
5. None of these answers are correct.

5. For a vector addition, assume that the vector length is 8000, each thread calculates 8 output elements, and the thread block size is 512 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements. How many threads will be in the grid?

1. 1000
2. 8196
3. 8192
4. 1024
5. 8200
6. None of these answers are correct

6. What are the possible values of `dst[0]` after this kernel execution?

```
__global__ void kernel(char *dst) {
    dst[0] = blockIdx.x;
}

// dst is a pointer to an array of one char allocated on
// the device and initialized to value of 3, e.g., dst[0]=3
kernel<<<2,1>>>(dst);
```

1. 0
2. 0 or 1
3. 3
4. 0 or 1 or 3
5. 1

7. A particular CUDA device's streaming multiprocessor (SM) can take up to 1536 threads and up to 4 thread blocks. Which of the following block configurations could guarantee the SM be fully utilized?

1. 256 threads per block
2. 384 threads per block
3. 576 threads per block
4. 1024 threads per block
5. None of these answers are correct

8. Consider a kernel where for every floating-point operation (FLOP) performed, it accesses 8 bytes from global memory. What can you infer about this kernel's performance characteristics based on its bytes/flops ratio assuming a GPU with global memory bandwidth of 150 GB/s and 1,000 GFLOP/s of compute power?

1. The kernel is likely compute-bound since it performs more computations relative to memory accesses.
2. The kernel is likely memory-bound since it performs fewer computations relative to memory accesses.
3. The kernel optimally balances between memory accesses and computations, ensuring maximum GPU utilization.
4. The bytes/flops ratio is unrelated to performance bottlenecks and provides no useful insight.

9. For the two cases below, which type of memory will be most suitable (fastest accesses) for their purpose?

A. Suppose your kernel code requires certain threads to read data items written by other threads in the same thread block.

Answer: ____

B. Suppose your kernel code requires all threads to share the data items and those data items remain same throughout the execution.

Answer: ____

10. Consider a convolutional neural network that takes 100x200 images with three color channels (red, green, blue). The first layer of this network generates ten output feature maps using 9x9 filters, where all channels are combined in each output feature map. Assuming all convolutions are performed in floating point, and considering only the convolutional layer (e.g., no pooling, thresholding, non-linearity, etc.), how many floating-point operations (both multiplications and additions) are required to generate all the output feature maps in a single forward pass? Remember: output feature maps are smaller than input maps because only pixels without ghost elements are generated. Your answer does not need to be a single number, you can write the expression to compute it. In fact, we would prefer the expression so we can see if you are on the right path in case we need to give you a partial credit for your solution. Also, feel free to draw a picture below to help you to solve the problem.

Answer: ____

Problem 2 (20 points): Performance Analysis

Choose the proper response, and if multiple responses are correct, choose all. No partial credit will be provided if the answer is partially correct, or wrong.

1. Consider a DRAM system with a burst size of 512 bytes and a peak bandwidth of 240 GB/s. Assume a thread block size of 1024 and warp size of 32 and that `A` is a floating-point array in the global memory. What is the maximal memory data access throughput we can hope to achieve in the following access to `A`?

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float temp = A[4*i] + A[4*i+1] + A[4*i+2];
```

1. 240 GB/s
2. 180 GB/s
3. 120 GB/s
4. 60 GB/s
5. None of them are correct.

2. Consider a 3D video filtering (convolution) code in CUDA with a 5x5x7 mask, which is stored in constant memory. Shared memory is used to fully store the input tile required for an 16x16x16 output tile (for example, using strategy 2). What is the ratio of total global memory read operations to shared memory accesses for one output tile? For this question, only consider interior tiles with no ghost elements.

1. 16*16*16 to 5*5*7*16*16*16
2. 12*12*10 to 16*16*16
3. 12*12*10 to 5*5*7*16*16*16
4. 20*20*22 to 5*5*7*16*16*16
5. None of these answers are correct.

3. For a 4x16 2-dimensional array M, elements are placed into the linearly addressed memory space according to the row major convention. Assume that we are using 4×4 blocks and that the warp size is 4, k is the number of loading iteration. Which of the following array accesses in the kernel code has coalesced memory access?

```
A: M[k*Width+blockIdx.x*blockDim.x+threadIdx.x]
B: M[(blockIdx.y*blockDim.y+threadIdx.y)*Width+k]
```

1. A
2. B
3. Both
4. Neither

4. Given the following kernel:

```
__global__ void Kernel(float* d_Pin, float* d_Pout, int n, int m)
{
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;

    if((row < m) && (col < n)) {
        d_Pout[row*n+col] = 2*d_Pin[row*n+col];
    }
}
```

We use 16x16 thread blocks, i.e. each block is organized as a 2D 16x16 array of threads.

A. Assume we are processing a picture with 800x600 pixels (800 pixels in the x-direction, 600 pixels in the y-direction, or n=800 and m=600). How many warps will have control divergence?

Answer: ____

B. Now assume we are processing a picture with 600x800 pixels (n=600 and m=800). How many warps will have control divergence?

Answer: ____

Name and NetID: ____

Problem 3 (30 points): Kernel Code Completion

Consider the following tiled 3D convolution implementation. Fill in the missing lines in the GPU kernel:

```
#define MASK_WIDTH 3
#define MASK_RADIUS MASK_WIDTH / 2
#define TILE_SIZE 4
#define BLOCK_SIZE TILE_SIZE + MASK_WIDTH - 1


__constant__ float deviceKernel[MASK_WIDTH][MASK_WIDTH][MASK_WIDTH];


int main()
{
  /* skipped pre-kernel launch code */

  /* kernel grid configuration */
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(ceil(((float)x_size) / TILE_SIZE),
               ceil(((float)y_size) / TILE_SIZE),
               ceil(((float)z_size) / TILE_SIZE));

  /* kernel execution */
  conv3d<<<dimGrid, dimBlock>>>(deviceInput, deviceOutput,
                                z_size, y_size, x_size);


  /* skipped post-kernel launch code */
}
```

(see next page)

```
__global__ void conv3d(float *A, float *B, const int z_size,
                        const int y_size, const int x_size)
{
  __shared__ float A_s[BLOCK_SIZE][BLOCK_SIZE][BLOCK_SIZE];
  const int out_z = blockIdx.z * TILE_SIZE + threadIdx.z;
  const int out_y = blockIdx.y * TILE_SIZE + threadIdx.y;
  const int out_x = blockIdx.x * TILE_SIZE + threadIdx.x;
  float output = 0.0f;

  const int in_z = ____________________________________;               // #1
  const int in_y = ____________________________________;               // #2
  const int in_x = ____________________________________;               // #3

  if (_____________________________________________________________

      _____________________________________________________________)   // #4
    A_s[threadIdx.z][threadIdx.y][threadIdx.x] =

      A[___________________________________________________________];  // #5
  else
      A_s[threadIdx.z][threadIdx.y][threadIdx.x] = 0.0f;

  _________________________________________________________________;   // #6

  if (_____________________________________________________________

      _____________________________________________________________) { // #7
    for (int z = 0; z < MASK_WIDTH; ++z)
      for (int y = 0; y < MASK_WIDTH; ++y)
        for (int x = 0; x < MASK_WIDTH; ++x)

          output += _________________________________________________

          ___________________________________________________________; // #8

    if (___________________________________________________________)   // #9

      ____________________________________________________ = output;  // #10
  }
}
```

Problem 4 (30 points): CUDA Programming

During your inaugural internship in CUDA Kernel Optimization, you are asked to implement a specific matrix operation: `A * B + C`, where `A`, `B`, and `C` are `n×n` matrices and `n` is divisible by 64. Learning that you have taken ECE 408 from UIUC, your manager has outlined the following explicit expectations for your task:

1. The block dimensions must be fixed at (16, 32, 1).
2. You need to employ a tiled version utilizing shared memory for your implementation.
3. Each thread within your kernel should compute precisely **two adjacent elements** in the resulting output matrix.

The provided kernel function prototype is as follows:

```
__global__ void MAC(float *A, float *B, float *C, float *output, int n);
```

To complement your implementation, your manager requests an illustrative diagram that delineates the organization of each tile in your solution. Also, the manager is generous enough to provide you with the following grading rubric to help you understand his expectations:

- Host code in `main()` function (GPU memory allocation/deallocation, data movement, grid configuration, kernel call, etc.) - 10 points
- CUDA kernel correctness (your code produces correct answer) - 8 points
- Correct *tiled* kernel implementation (you are implementing a tiled version correctly; a non-tiled version will not receive these points) - 10 points
- An illustrative diagram (to show how your tiled implementation is organized) - 2 points
- **You do not need to get the syntax exactly correct to get the full points.**

The rubric shows that even if you do not implement the tiled version, you may still earn up to 20 out of 30 points for this problem.

Please write your code on the next two pages and make your illustrative diagram below:

Write your **CUDA host** code on this page. **You do not need to** write code for error checking on CUDA API calls.

```
#define n 128  // as an example
int main() {
  float *A, *B, *C, *output;
  A = (float *)malloc(n * n * sizeof(float));
  B = (float *)malloc(n * n * sizeof(float));
  C = (float *)malloc(n * n * sizeof(float));
  output = (float *)malloc(n * n * sizeof(float));
  getData(A, B, C, n);   // this function fills in data for matrixes A, B, and C









  free(A); free(B); free(C); free(output);
  return 0;}
```

Write your **CUDA kernel** code on this page.

```
__global__ void MAC(float *A, float *B, float *C, float *output, int n)
{
```