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

Spring 2024 – Midterm Exam 1

March 5, 2024

1. This is a closed book exam; **no hand-written notes permitted**.

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 (24 points): _____

Question 4 (36 points): _____

Total Score: _____

Problem 1 (20 points): Warm-up!

Provide the proper response to the questions below.

1. **True** or **False**? Circle the correct answer.

- GPUs are commonly categorized as latency-oriented devices.

  Answer: True, False

- CPU execution and GPU execution could overlap with each other.

  Answer: True, False

- Shared memory is private per thread block.

  Answer: True, False

- Global memory is a separate hardware unit from the GPU core.

  Answer: True, False

- The "grid-block-thread" hierarchy is a software interface and does not describe the GPU's physical architecture.

  Answer: True, False

- In recent NVIDIA GPU architecture, a warp consists of 32 threads that execute the same instruction at a time.

  Answer: True, False

2. After doing Lab 4, Ze has mastered the usage of constant memory. While reviewing his Lab 1 for his midterm preparation, he suggests copying two input vectors to constant memory. Evaluate whether this strategy is beneficial by briefly stating one benefit and one drawback (each in a phrase for no more than SIX words) of using constant memory for this purpose.

Benefit: _____

Drawback: _____

3. Daksh has written a C program that takes 40 seconds to execute on his CPU. After taking ECE 408, he realizes that his application is a perfect fit for his NVIDIA GPU. Subsequently, he develops a CUDA version of the program and compares the performance of both the original C version and the CUDA variant. After proper profiling, he discovers that 10% of the workload cannot offload to GPU, and the overall execution time is reduced to 10 seconds, with 4 seconds of GPU kernel execution. Assume CPU and GPU execution are sequential.

(a) Determine the speedup of the part of the program that is parallelizable and executed on the GPU, compared to its performance on the CPU. Show your work and clearly indicate your final answer.

(b) Calculate the combined duration of the kernel launch overhead and memory transfer in the CUDA implementation. Show your work and clearly indicate your final answer.

4. Yuxiang is developing a convolutional neural network that takes 32 x 32 images with three color channels (red, green, blue). The first layer of this network generates five output feature maps using 3 x 3 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 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): Prof. Kindratenko's Lab Workshop

Choose the proper response, and if multiple responses are correct, choose all. **There is no partial credit.**

1. Which factor(s) is/are important for achieving optimal global memory coalescing in CUDA?

a) Making each thread within a warp accesses consecutive memory address.
b) Maximizing the number of global memory accesses per kernel launch.
c) Using constant memory for all data accesses.
d) Aligning kernel execution to the size of the L2 cache.

2. Which of the following is a correct consideration about the function call `__syncthreads()`?

a) It ensures global synchronization across all threads in all blocks executing a kernel.
b) It must be used at the beginning of a kernel to synchronize thread execution.
c) It synchronizes all threads within a block but using it excessively can lead to deadlocks if not all threads reach the synchronization point.
d) It is optional and can be omitted for kernels that do not access shared memory.

3. What is the grid size and block size, respectively, that will be produced as a result of the following kernel call?

```
scalarProd<<<1024, 256>>>(d_C, d_A, d_B, ELEMENT_N);
```

a) 256, 1024
b) 1024, 256
c) 256, 1024+256
d) 1024, 1024+256

4. Which line(s) below don't have a memory-coalesced pattern? Assume that array `g` is in global memory and `BLOCK_WIDTH = 32`.

a) `int a = g[threadIdx.x];`
b) `int a = g[threadIdx.x * BLOCK_WIDTH + blockIdx.x];`
c) `int a = g[BLOCK_WIDTH / 2 + threadIdx.x];`
d) `int a = g[blockIdx.x * BLOCK_WIDTH / 2 + threadIdx.x];`

5. For a tiled-matrix multiplication kernel, if we use a 16 x 16 tile, what is the reduction of memory bandwidth usage for input matrices A and B?

a) 1/8 of the original usage
b) 1/16 of the original usage
c) 1/32 of the original usage
d) 1/64 of the original usage

For Questions 6 and 7, Assume a tile size of 32 x 32 and use the following dimensions to do your calculations. `numARows`: 100, `numAColumns`: 200, `numBRows`: 200, `numBColumns`: 300.

6. How many floating-point operations are performed in a tiled matrix multiply kernel WITH consideration of halo cells? Meaning, halo cells in shared memory are still part of your calculation.

Answer: _____ floating point operations.

7. How many global memory data reads in TOTAL are performed in a tiled matrix multiply kernel?

Answer: _____ global memory reads.

For questions 8 and 9, assume a CUDA 3D convolution implementation using strategy 2 in which each thread loads one value from input `N`. The output tile size is 8 x 8 x 8 and the mask size is 1 x 3 x 5.

8. How many thread blocks (on average) access any particular input element? Do not consider accesses to the mask, only input `N`. Assume that `N` is very large.

Answer: _____ thread blocks.

9. Consider an internal tile, what is the average number of times any particular value loaded into shared memory is used?

Answer: _____ average use.

10. Is there a possibility of thread divergence in this code? Explain in no more than FIFTEEN words. Assume 1D grid configuration and thread block size is a multiple of 32.

```
__global__ void do_work(int *A) {
    int index = threadIdx.x + blockDim.x * blockIdx.x;
    if (threadIdx.x <= 127) {
        A[index] *= 2;
    } else if (threadIdx.x > 255) {
        A[index] /= -2;
    } else {
        A[index] += 1;
    }
}
```

Answer: Yes / No

Explanation: _____

_____

Problem 3 (24 points): Help Howie!

Howie writes the code below to accelerate a modified normalization application. This normalization is designed to normalize inputs in groups of 1024 consecutive elements, with any remaining elements less than 1024 comprising the final group. For instance: out[0:1023] = in[0:1023]/max(in[0:1023]), out[1024:2047] = in[1024:2047]/max(in[1024:2047])...

```
#define ELEM_THREAD     8
#define BLOCK_ELEMENTS  1024
#define BLOCK_DIM       ________            // part (a)

__global__ void normalization(float *in, float *out, int size) {
    __shared__ float max;   max = 0;
    float local[ELEM_THREAD];
    int index = (threadIdx.x + BLOCK_DIM * blockIdx.x) * ELEM_THREAD;
    // step 1: load the inputs              // part (c)
    for (int i = 0; i < ELEM_THREAD; ++i) {
        if (index + i < size) {
            local[i] = in[index + i];
        } else {
            local[i] = 0;
        }
    }
    __syncthreads();
    // step 2: find the max
    for (int i = 0; i < ELEM_THREAD; ++i) {
        if (local[i] > max) {
            max = local[i];
        }
    }
    __syncthreads();
    // step 3: output
    for (int i = 0; i < ELEM_THREAD; ++i) { // part (e) line A
        if (index + i < size) {             // part (e) line B
            out[index + i] = local[i] / max;
        }
    }
}
```

Howie's idea is that each block of threads will handle a group of 1024 consecutive elements. The kernel performs the operations as follows:

- Each thread loads 8 elements as a `local` copy.
- Determine the maximum value within the entire segment.
- Output the normalized values.

(a) Howie is not good at counting. He needs assistance in determining the block dimension. Write your answer in the blank marked part (a).

(b) Identify the GPU memory types of the following variables or arrays (the choices are *global, constant, register, shared*):

`in`	`max`	`local`	`index`

(c) After Huili reviews Howie's code, she points out the memory access pattern in step 1 and step 3 is not optimized. Is Huili correct? If she is correct, rewrite the code in step 1 ONLY below to better utilize memory throughput; if she is not correct, explain why in no more than FORTY words.

(d) Control divergence is so confusing to Howie. Using no more than THIRTY words total, define control divergence by specifying the conditions under which it occurs **and** how the GPU hardware executes the code.

(e) Determine whether control divergence is likely to occur at lines A or B of the provided code snippet, and if so, identify the maximum possible number of warps that will have control divergence for that line throughout the kernel execution.

Line A: Yes / No Number of divergent warps: _____

Line B: Yes / No Number of divergent warps: _____

(f) Andy tells Howie that his step 2 implementation sometimes incorrectly computes the max value. Concisely describe the cause of this error in no more than TWENTY words. We are not looking for the solution, just the cause of the error.

Problem 4 (36 points): Thomas' Image Blurring

During his internship, Thomas is asked to blur an image. He decides to apply tiling and adopts Strategy 3 discussed in our lectures. First, each thread loads a single element from global memory into shared memory. Subsequently, every thread computes one output element. If a thread requires an element beyond its tile's boundary, it will fetch it directly from global memory.

Here are the considerations:
- The dimensions of the image are **`row`** by **`col`**, and for simplicity, assume the image is grayscale with only one channel.
- The blur radius is defined by the variable **`blur_r`**.
- Each thread takes elements within the blur radius from the input image, computes the average, and writes the result to the output image.
- Yifei tells Thomas that the x-dimension should not be utilized. Instead, to employ the y-dimension to represent **`col`** and the z-dimension to represent **`row`**.
- We want to **maximize** the number of threads per block (max 1024 threads per block).
- Blocks must be square.

Consider an example with a 4 x 3 image and a blur radius of 1:

```
3.00    4.00    5.00
4.00    6.00    8.00
5.00    8.00   11.00
6.00   10.00   14.00
```

The expected output should be:

```
4.25    5.00    5.75
5.00    6.00    7.00
6.50    8.00    9.50
7.25    9.00   10.75
```

Observe that the value at the top-left corner is computed using the following expression: (3.00+4.00+4.00+6.00)/4 = 4.25.

(a) Define your `TILE_WIDTH`

```
#define TILE_WIDTH _____
```

(b) We provide a skeleton code of the computing step. Complete the **CUDA kernel** code below:

```
__global__ void blurKernel(float *in, float *out,
                           int z_size, int y_size, int blur_r) {
  int tz = threadIdx.z;  int ty = threadIdx.y;
  int bz = blockIdx.z;   int by = blockIdx.y;
  int index_z = blockIdx.z * blockDim.z + threadIdx.z;
  int index_y = blockIdx.y * blockDim.y + threadIdx.y;

  // variable initialization and shared memory declaration




  // load in shared memory







  // calculate the sum around threads location + the count
  float sum = 0.0f;
  for (int z = index_z - blur_r; _______________________________; ++z) {

    for (int y = index_y - blur_r; _______________________________; ++y) {

      if (_____________________________________________________________) {

        count--;

      } else if (_____________________________________________________

                  _____________________________________________________) {

        sum += tile[_____________________________][_____________________________];

      } else {

        _____________________________________________________________________;

      }
    }
  }

  // write back into global memory




}
```

(c) Write your **host** code here:

```
void blur(float *in, float *out, int row, int col, int blur_r) {

}
```

(d) Refer to your kernel code in part (b). Notice that you created thread barriers by calling **`__syncthreads()`**. Explain the necessity of each call using less than THIRTY words respectively.

(e) To save GPU memory, Prof. Patel suggests allocating a single device memory copy for the image to read and write into. Explain if this idea will work using less than FORTY words.

CUDA API

Function/variable Type Qualifiers

`__global__`
Executed on the device. Callable from the host or from the device for devices of compute capability3.x or higher. Must have void return type.

`__host__`
Executed on the host. Callable from the host only (equivalent to declaring the function without any qualifiers).

`__device__`
Executed on the call device. Callable from the device only.

`__constant__`
Variable in constant memory.

`__shared__`
Variable in shared memory.

Built-in Variables

`gridDim`
Dimensions of the grid (`dim3`).

`blockIdx`
Block index within the grid (`uint3`).

`blockDim`
Dimensions of the block (`dim3`).

`threadIdx`
Thread index within the block (`uint3`).

Synchronization

`__syncthreads();`
All threads in a block will be blocked at the calling location until each of them reaches the location.

`cudaDeviceSynchronize()`
Blocks host code until all operations on the device are completed.

Kernel Configuration and Kernel launch

```
dim3 dimGrid(x_size, y_size, z_size);
dim3 dimBlock(x_size, y_size, z_size);
kernelName<<<dimGrid, dimBlock>>>(input parameters);
```

Device Memory Management

`cudaError_t` **`cudaMalloc`**`(void** devPtr, size_t size);`
Allocates `size` bytes of linear memory on the device and points `devPtr` to the allocated memory.

```
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count,
                       cudaMemcpyKind kind);
```
Copies `count` bytes of data from the memory area pointed to by `src` to the memory area pointed to by `dst`. The direction of copy is specified by kind, and is one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`.

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, const void* src,
                               size_t count, size_t offset=0,
                           cudaMemcpyKind kind=cudaMemcpyHostToDevice);
```
Copies count bytes from the memory area pointed to by `src` to the memory area pointed to by `offset` bytes from the start of symbol `symbol`. `symbol` can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. `kind` can be either `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`.

`cudaError_t` **`cudaFree`**`(void* devPtr);`
Frees the device memory space pointed to by `devPtr`.