Final Exam

Prof. Naga Kandasamy
ECE Department
Drexel University

June 4, 2020

This exam consists of eight questions for a total of 50 points. The exam is open book/notes. You must work on this exam by yourself.

1. **(5 points)** You have designed a multi-threaded program to calculate the sum of a million integers in parallel as per the following code snippet.

```
int thr_id[5], array_1[1000000], array_2[5];

for(int i = 0; i < 5; i++){
    pthread_create(&thr_id[i], NULL, add, NULL))
    pthread_join(thr_id[i], NULL);
}
/* Code to compute the final sum from the partial sums stored
in array_2 */
...
```

Each thread executes a function called `add`, whose skeleton is shown below, to generate and store the partial sum into `array_2`.

```
/* The add function executed by each thread */
void add(void *arg){
    /* Compute the partial sum and store in array_2 */
    ...
}
```

You create five threads, each of which calculates the partial sum of $200,000$ elements in `array_1` and writes the partial sum to `array_2`. For example, thread 0 adds elements $[0, 199, 999]$ of `array_1` and writes this partial sum to `array_2[0]`, thread 1 adds elements $[200,000, 399, 999]$ of `array_1` and writes this partial sum to `array_2[1]`, and so on. The threads use the function `add` to obtain these partial sums. You initially execute the code on a uniprocessor machine with a single core and measure the execution time. Next, you execute the code on a processor with four cores and to your surprise the execution time shows no speedup over the single CPU case. Identify the problem with the above code that is preventing the speedup on the multi-core machine, and fix it. Write the corrected code.

2. (**5 points**) Consider the situation shown by the code listing below for two concurrent threads `A` and `B`, and two resources that are protected by semaphores.

```
semaphore resource_1 = 1;
semaphore resource_2 = 1;

void thread_A(void)
{
    probe_semaphore(resource_1);
    probe_semaphore(resource_2);

    use_both_resources();

    signal_semaphore(resource_2);
    signal_semaphore(resource_1);
}

void thread_B(void)
{
    probe_semaphore(resource_2);
    probe_semaphore(resource_1);

    use_both_resources();

    signal_semaphore(resource_1);
    signal_sempahore(resource_2);
}
```

Examine the code listing for a potential deadlock. If you believe the code is prone to a deadlock, describe the scenario that may lead to a deadlock and propose a fix.

3. **(5 points)** If a CUDA device's SM can take up to 1536 threads and up to four thread blocks, which of the following block configurations would result in the most number of threads in the SM: (a) 128 threads per block; (b) 256 threads per block; (c) 512 threads per block; or (d) 1024 threads per block? Explain your answer clearly.

4. **(5 points)** A CUDA programmer says that if they launch a kernel with only 32 threads per block, thay can leave out the *__syncthreads()* instruction whenever barrier synchronization is needed. Do you think this is a good Idea? Explain.

5. **(5 points)** A kernel performs 36 floating-point operations and seven 32-bit word global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute or memory bound: (a) Peak FLOPS = 200 GFLOPS, peak memory bandwidth = 100 GB/s; (b) Peak FLOPS = 300 GFLOPS, peak memory bandwidth = 250 GB/s. Explain your answer clearly.

6. **(5 points)** A graduate student has written a CUDA kernel to reduce a large floating-point array by summing all its elements. The array is always sorted in ascending order, from the smallest values to the largest values. To avoid branch divergence, the student has decided to implement the following reduction algorithm within the GPU kernel:

```
__shared float partialSum[];
unsigned int t = threadIdx.x;
int stride;
for(stride = blockDim.x >> 2; stride > 0; stride = stride >> 1){
     if(t < stride)
        partialSum[t] += partialSum[t + stride];
     __syncthreads():
}
```

Explain why this design can reduce the accuracy of the results.

7. **(5 points)** Consider performing a 1D tiled convolution using the GPU kernel shown below on an array of size n with a mask of size m using a tile of size t. Answer the following questions: (a) how many blocks are needed?; (b) how many threads per block are needed? (c) how much shared memory is needed in total?

```
__global__ void convolution_kernel(float *N, float *result,
                                   int num_elements, int kernel_width)
{
    __shared__ float N_s[THREAD_BLOCK_SIZE + MAX_KERNEL_WIDTH - 1];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int half_width = kernel_width/2;

    /* Load left halo elements from the previous tile */
    int left_halo_index;
    left_halo_index = (blockIdx.x - 1) * blockDim.x + threadIdx.x;
    if (threadIdx.x >= (blockDim.x - half_width)) {
        if (left_halo_index < 0)
            N_s[threadIdx.x - (blockDim.x - half_width)] = 0.0;
        else
            N_s[threadIdx.x - (blockDim.x - half_width)] =
                                                N[left_halo_index];
    }
    /* Load center elements for the tile */
    if (i < num_elements)
        N_s[half_width + threadIdx.x] = N[i];
    else
        N_s[half_width + threadIdx.x] = 0.0;
    /* Load right halo elements from the next tile */
    int right_halo_index;
    right_halo_index = (blockIdx.x + 1) * blockDim.x + threadIdx.x;
    if (threadIdx.x < half_width){
        if (right_halo_index >= num_elements)
            N_s[threadIdx.x + (blockDim.x + half_width)] = 0.0;
        else
            N_s[threadIdx.x + (blockDim.x + half_width)] =
                                                N[right_halo_index];
    }
    __syncthreads();

    float sum = 0.0;
    int j;
    for(j = 0; j < kernel_width; j++)
        sum += N_s[j + threadIdx.x]*kernel_c[j];

    result[i] = sum;
}
```

8. Given an input array

$$A = [a_0, a_1, a_2, \ldots, a_{n-1}],$$

where n is an arbitrary integer, write GPU code to generate the result array R in which each element of R is the square of the corresponding element in A. That is,

$$R = [a_0^2, a_1^2, a_2^2, \ldots, a_{n-1}^2].$$

Assume the following:

- The input array `A` in CPU memory has been copied over and stored in GPU memory starting at location `Ad`.
- The storage for the result array has been allocated on the GPU starting at location `Rd`.

Answer the following questions related to executing the kernel on the device.

(5 points) Complete the code below to set up the execution grid on the GPU. The number of elements in `A` is given to you via the variable `num_elements`.

```
dim3 thread_block_size =

int num_thread_blocks =

dim3 execution_grid =
```

When generating the grid, consider the trade-off between the number of thread blocks to be scheduled on the GPU and the granularity of the workload assigned to each thread.

(10 points) Complete the kernel below to generate the result array `Rd`. Your code should clearly show your strategy for mapping each CUDA thread to the data and the work performed by each thread. Do not worry about copying `Rd` back to the host.

```
compute_squares_kernel (float *Ad, float *Rd, int num_elements)
{









}
```