GPUs and Heterogeneous Systems – A.Y. 2023-24

Scuola di Ingegneria Industriale e dell'Informazione
Prof. Antonio Miele

POLITECNICO
MILANO 1863

Example of Exam - **FIRST PART OF THE EXAM**

Surname:	Name:	Personal Code:

Question	**1**	**2**	**3**	**4**	**5**	**OVERALL**
Max score	**3**	**3**	**3**	**3**	**3**	**15**
Score						

Instructions:

- This first part of the exam is "closed book". The students are not allowed to consult any course material and notes.
- No extra devices (e.g., phones, iPad) are allowed. Please, shut down and store any electronic device.
- Students are not allowed to communicate with any other ones.
- Students can write in pen or pencil, any color, but avoid writing in red.
- Any violation of the above rules will lead to the invalidation of the test.
- **Duration: 30 minutes**

Question 1

Consider the following histogram kernel exploiting privatization, shared memory, and thread coarsening using contiguous partitions. Let's assume that the kernel processes an input with 524,288 elements to produce a histogram with 5 bins and that it is configured with 1024 threads per block; what is the maximum number of atomic operations that the kernel may perform on global memory? Motivate the answer.

```
#define CHAR_PER_BIN  6
#define ALPHABET_SIZE 26
#define BIN_NUM ((ALPHABET_SIZE - 1) / CHAR_PER_BIN + 1) /* equal to 5 */
#define FIRST_CHAR 'a'

__global__ void histogram_kernel(char *data, int *histogram, int length) {
  int tid    = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  __shared__ int histo_s[BIN_NUM];
  for (int binIdx = threadIdx.x; binIdx < BIN_NUM; binIdx += blockDim.x)
    histo_s[binIdx] = 0;
  __syncthreads();

  for (int i = tid; i < length; i += stride) {
    int alphabet_position = data[i] - FIRST_CHAR;
    if (alphabet_position >= 0 && alphabet_position < ALPHABET_SIZE)
      atomicAdd(&(histo_s[alphabet_position / CHAR_PER_BIN]), 1);
  }
  __syncthreads();

  for (int binIdx = threadIdx.x; binIdx < BIN_NUM; binIdx += blockDim.x) {
    int binValue = histo_s[binIdx];
    if (binValue > 0) {
      atomicAdd(&(histogram[binIdx]), binValue);
    }
  }
}
```

The kernel performs at most 5 atomic writes per each block in the grid. These writes are performed concurrently by threads in the last for loop in the code; the if statement avoids performing the write when the counter is equal to 0 (for this reason, 5 is the upper bound). Based on the provided code, it is not possible to state how many blocks there are in the grid.

Question 2

Describe the benefits of the unified memory introduced in the Pascal architecture.

The unified memory introduces a unified virtual addressing on the overall CPU-GPU system. This offers the opportunity to share pointers (pointing allocated memory) between the host code and the device code. This increases considerably the programmability of the system. The programmer can pass a pointer declared in the host code as a parameter to a kernel; then, CUDA runtime automatically and transparently manages data movements between host memory and device one.

Question 3
Explain the benefits of introducing a unified shader core in Tesla architecture.

Introducing a unifier shader processor solves issues related to balancing the workload in the different graphics pipeline stages. In fact, in previous GPU architectures having separate shader processors for the various graphics pipeline stages, the saturation of computing resources in one shader processor would represent a bottleneck for GPU performance. From the Tesla architecture, the unifier shader processor executes all the graphics pipeline stages; this allows an optimal usage of processing resources, thus solving issues related to balancing the workload.

Question 4
Let's assume to run the following kernel on a Maxwell (or more recent) architecture and to size the grid with a single block of 32 threads; which is the efficiency of global load and store operations of the following CUDA kernel? Motivate the answer.

```
__global__ void foo(char* a, char* b){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  b[i+1] = a[(i+1)%blockDim.x];
}
```

READ: In the access to array `a`, 32 threads in the block read 32 bytes (1 char occupies 1 byte). Accessed data are contiguous and aligned in the memory; in fact, `%blockDim.x` operation modifies only the index of the last thread from 32 to 0. The result is a single transaction where all values are then used. The overall load efficiency is, therefore, 100%.

WRITE: in the access to array `b`, the 32 threads access 32 bytes in a non-aligned way; in fact, indexes used for accessing `b` spans from 1 to 32. This cause 2 different transactions (thus 64 bytes to be transmitted) where only 32 bytes are used. Therefore, the overall store efficiency is, therefore, 50%.

Question 5
In the following snippet of code using OpenACC pragmas, how many times will `foo()` and `bar()` be executed? Motivate the answer.

```
#pragma acc parallel num_gangs(64)
{
  foo();
  #pragma acc loop gang
  for (int i=0; i<n; i++) {
    bar(i);
  }
}
```

The first pragma generates 64 gangs, each one with a single worker and vector element, executing the subsequent code in a parallel redundant way. Then, the second pragma uses the generated gangs to parallelize the related loop. In conclusion, `foo` is called 64 times while `bar` is called `n` times.