COMPUTAÇÃO DE ALTO DESEMPENHO

2018/2019

Test 1

1. [2 values] Is a GPU a shared memory MIMD (Multiple Instruction Multiple Data) architecture? Justify your answer.

2. Consider the following sequential C function:

```c
int* some_func(int* a, int* b, int x, int y, int size) {
   int* result = malloc (size * sizeof(int));
   int z = sqrt (x * y);
   for (int i = 0; i < size; i++) result[i] = a[i] + b[i]*z;
}
```

   a. [1 value] Is there a potential hotspot in this algorithm? Where and why?
   b. [2 values] Parallelize the identified hotspot by providing a CUDA implementation (please provide only the CUDA kernel not the host code).
   c. [2 values] Provide a rewriting of function some_func that makes use of the kernel implemented in the previous question. You must include all memory allocation and transfers operations.

3. Consider a sequential implementation of an algorithm A with execution time of 100 time-units. Consider also a parallel implementation (PA) of that same algorithm with the following execution times:

Number of threads	**Execution time (in time-units)**
1	120
2	60
4	40
8	24
16	20
32	12

   a. [0.5 values] What is the speedup yield by PA when executed by 8 threads?
   b. [1.5 values] Roughly plot PA's efficiency curve. What is your conclusion regarding the PA's efficiency? Justify.
   c. [1 value] Is PA cost-optimal? Why?

4. Consider that you have the task of parallelizing, with CUDA, the search of an element in a huge array.

```c
int contains(int* array, size_t size, int elementToSearch);
```

   a. [1.5 values] Which would be your implementation strategy? In other words, how many kernels would you use, what would be their behavior (map, reduce, scan), and which operation/function would they apply?
   b. [1.5 values] Does any of your kernels benefit from the use of shared memory? Justify.
   c. [1.5 values] Consider now that you a have a framework that allows you to execute your kernel in multiple GPUs. Which modifications do you need to apply to your strategy? Justify.
   d. [1.5 values] What is the lower and upper of bound of the expected speedup of a N-GPU execution of your algorithm when compared against a 1-GPU execution? Consider that all N GPUs are of the same model.

5. You now have the task of parallelizing the following algorithm:

```
let a, b and c be arrays in
   array e = a + b
   e = (a*e – c*e)^4;
   for i in [1, (e.size()-2)/2] do f[i] = (e[i-1] + e[i] + e[i+1])/3
   for i in ](e.size()-2)/2, e.size()-2] do f[i] = (2*e[i-1] + e[i] + 2*e[i+1])/5
```

   a. [1.5 values] In the context of the BSP abstract computer model, how many super steps are needed to parallelize this algorithm? What would each of these super steps do?

b. [1 value] Can the solution you gave to the previous question be implemented on CUDA (or other GPGPU framework)? Justify your answer.
c. [1.5 values] Does your CUDA solution uses kernels that have thread divergence (for instance **ifs**)? If not, why? If so, how harmful will this divergence be in the performance of your kernel(s)?

Appendix

`shared` `a`

allocates variable a in shared (thread block local) memory

`syncthreads()`

synchronizes all threads in the current thread block

`kernel_function<<<NUMBER_BLOCKS, BLOCK_SIZE, LOCAL_MEMORY, STREAM>>>(arguments)`

signature of kernel launching in CUDA

```
cudaError_t cudaMalloc (void** dev_ptr, size_t count)

cudaError_t cudaFree (void* dev_ptr)

cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)

cudaError_t cudaMemcpyAsync (void* dst, const void* src,
                      size_t count, cudaMemcpyKind kind, cudaStream_t stream = 0 )
```

`cudaMemcpyKind:`

cudaMemcpyHostToHost	Host -> Host.
cudaMemcpyHostToDevice	Host -> Device.
cudaMemcpyDeviceToHost	Device -> Host.
cudaMemcpyDeviceToDevice	Device -> Device