COMPUTAÇÃO DE ALTO DESEMPENHO

2017/2018

Test 1

18/04/2018

1. Consider the following C/C++ code.

Version A	Version B

`void add(int * __restrict__ c, int * __restrict__ b, int * __restrict__ a) {`
 	
`  for (int i = 0; i < N; i++)`
`  c[i] = a[i] + b[i];`
`}`
	`void add(int * __restrict__ b, int * __restrict__ a) {`
	
	`  for (int i = 0; i < N; i++)`
	`  b[i] = b[i] + a[i];`
	`}`

   a. The **__restrict__** type qualifier informs the compiler that pointers a, b and c, point to different (non-overlapping) memory locations. This information is crucial for enabling vectorization, why?
   b. Is the for loop of both versions A and B vectorizable? Justify your answer.

2. What information gives the isoefficiency function of an algorithm about that same algorithm? **Note**: remember that the isoefficency is given by formula: $K \times T_{overhead}(p)$, where $K = \frac{E}{1-E}$ and p is the number of processors running the computation.

3. Consider the problem of generated an inverted array of a given array of integers:

   invert({ 1, 2, 3, 4, 5, 6}) → {6, 5, 4, 3, 2, 1}

   a. Provide the pseudo-code of a parallel version of the algorithm for shared memory architectures. Assume the existence of the constructs in appendix A.

   b. Did you use a recursive (divide-and-conquer) or a data parallel approach? Justify.

   c. Given that the complexity of the serial algorithm, for an array of size n, is $T_1 = \Theta(n)$, and assuming that the complexity of your parallel algorithm is $T_p = \Theta(n/p)$, for p processors
      i. What's the maximum speed-up of your algorithm for p processors?
      ii. What's the efficiency of your algorithm for p processors?
      iii. What's the isoefficiency of your algorithm for p processors? **Note**: if you used a data parallel approach, remember how *parallel for*s are implemented in shared memory parallel computing platforms.

   d. Provide the pseudo-code of a parallel version of the algorithm (kernel) for GPUs. Assume the existence of the constructs in appendix B.
      i. How many global memory reads and writes are being performed by your kernel? Explain

ii. How many local memory reads and writes are being performed by your kernel? Explain.

4. Consider the Successive Over-Relaxation (SOR) algorithm for solving linear equations, given below:

```
sor(double matrix[M][N], double omega, int n_inter) {
  for (int n = 0; n < n_iter; n++)
    for (int i=1; i < M; i++)
      for (int j=1; j < N; j++)
        matrix[i][j] = omega/4 * (matrix[i-1][j] + matrix[i+1][j] + matrix[i][j-1] +
            matrix[i][j+1]) + (1-omega) * matrix[i][j];
}
```

   a. Provide the pseudo-code of a parallel version the algorithm for shared memory architectures. Assume the existence of the constructs in appendix A.
   b. Provide the pseudo-code of a parallel version of the algorithm (kernel) for GPUs. Assume the existence of the constructs in appendix B.
      i. How many global memory reads and writes are being performed by your kernel? Explain
      ii. How many local memory reads and writes are being performed by your kernel? Explain.

5. Given the Dijkstra's (Single Source) Shortest Path (DSP) algorithm, that computes the shortest path between a given vertex v and all remainder vertices in the graph.

```
dsp(vertices V, edges E, weights w, source s) {
  VT = {s};
  l = [];
  foreach (v in (V-VT)) {
    if ( (s, v) exists ) l[v] = w(s, v);
    else l[v] = ∞ ;
  }
  while (VT != V) {
    find a vertex u such that l[u] == min{l[v]|v in (V - VT)};
    VT = VT ∪ {u};
    foreach (v in (V-VT))
      l[v] = min{l[v], l[u] + w(u, v)};
  }
}
```

   a. Which are the parallelization points of this algorithm?
   b. How can these be efficiently parallelized? Explain your solution. You may, but you do not need to provide code.

6. Consider the following excerpt of CUDA C/C++ program that processes an unbounded set of images, offloading the image processing computation to a GPU.

```
int *a, *b;
int *d_a, *d_b;
int size = IMAGE_SIZE * sizeof(float);

cudaMalloc((void **)&d_a, size); cudaMalloc((void **)&d_b, size);
a = (float *)malloc(size); b = (float *)malloc(size);

while (are_images_to_process()) {
  load_next_image(a, size);
  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
  image_process<<<…, …>>>(d_a, d_b, IMAGE_SIZE);
  cudaMemcpy(b, d_b, size, cudaMemcpyDeviceToHost);
}
```

a. How many images are being processed simultaneously in the GPU? Justify.
b. Provide a new version of the code that allows for more images to be processed simultaneously in the GPU. Your code does not have to be syntactically accurate, as long as the behavior is clear. Assume the existence of the constructs in Appendix B.

Appendix A

`spawn(function_name, arguments)`
creates a new task to run `function_name(arguments)`

`parallel for (initialization; booleanExpression; update) { codeblock }`
acts like a regular for loop, but the iteration space is divided among a predefined number (p) of processors

`barrier()`
synchronizes all the workers currently running a parallel section, e.g. a parallel for

Appendix B

`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

`cudaStream_t` s
declares a stream variable

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

asynchronous data transfer between host and GPU (or vice-versa)