Laboratory 6

(Due date: Nov. 5th)

OBJECTIVES

- Compile and execute C++ code using the TBB library in Ubuntu 12.04.4 using the Terasic DE2i-150 Development Kit.
- Execute applications using TBB: *parallel_for* and *parallel_reduce* (reducing group of arrays into an array)
- Implement image histogram with TBB.

REFERENCE MATERIAL

- Refer to the board website or the Tutorial: Embedded Intel for User Manuals and Guides.
- Refer to the Tutorial: High-Performance Embedded Programming with the Intel® Atom™ platform → *Tutorial 6* for associated examples.

ACTIVITIES

* You can alternatively complete these activities using a Linux laptop.

FIRST ACTIVITY: IMAGE HISTOGRAM COMPUTATION (100/100)

- Given a grayscale image I of $nrows$ by $ncols$, we want to get the histogram of I, represented by the vector $\vec{h}$ (of size nb)
  - ✓ We use $nb=256$ bins in this exercise. Fig. 1 depicts an example.

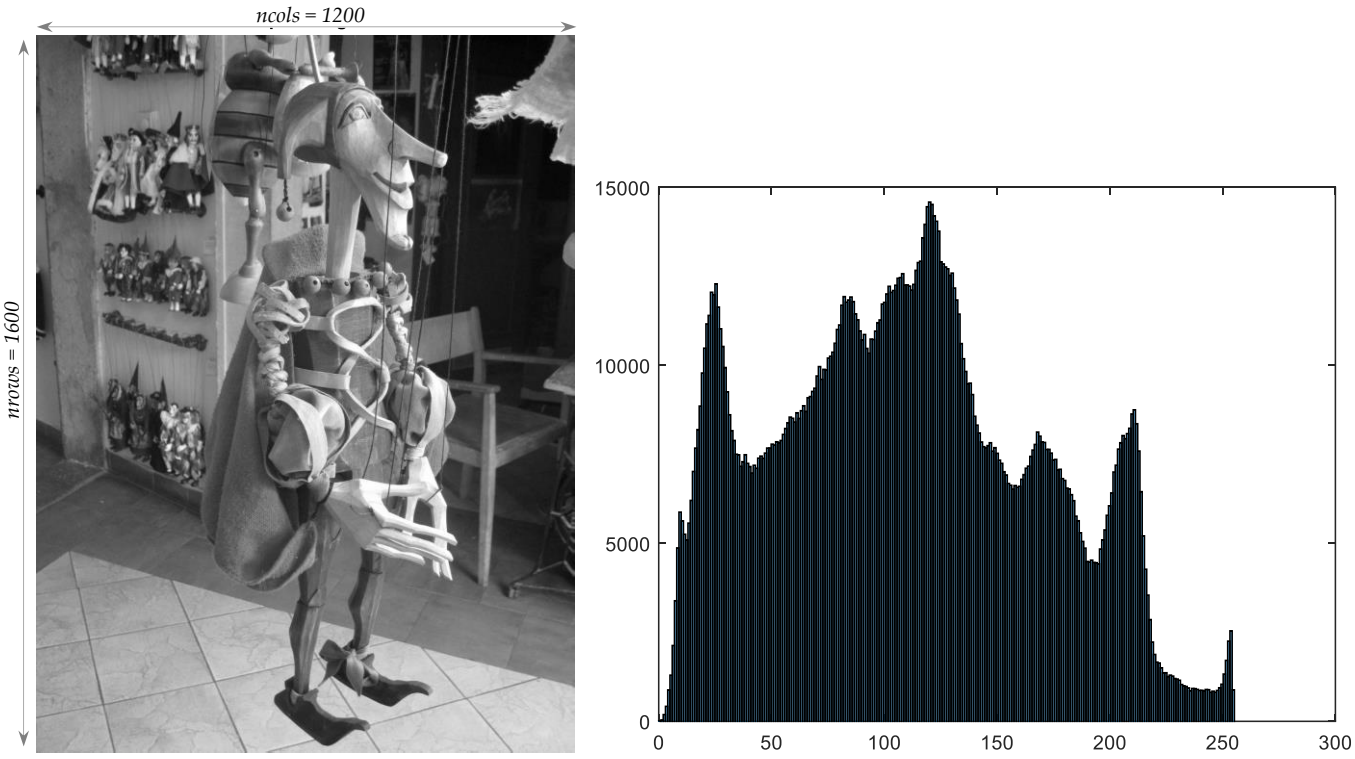

Figure 1. (a) Grayscale image of 1600x1200. (b) Histogram with 256 bins.

- **Serial approach**: $n = nrows \times ncols$.
  - ✓ Image I: represented as a n-element vector (image stored in a raster scan fashion).

Naïve serial approach	Optimized serial implementation

`for i = 0:255`
`  for j = 0:n-1`
`  if i = I[j]`
`  h[i] ← h[i]+1`
`  end`
`  end`
`end`
	`for j = 0:n-1`
	`  h[I[j]] ← h[I[j]]+1`
	`end`

  - ✓ It is very clear that the optimized serial implementation should be used.

- **Parallel approach**:
  - ✓ It seems that we can attempt to use the optimized serial implementation in parallel, so that `h[I[j]]` can be updated by multiple threads. Here, *parallel_for* can be used with the iteration space [0,n-1]. Example:

```
parallel_for(blocked_range<int>(0,n), [&] blocked_range<int> r) {
   for (int j = r.begin(); j!= r.end(); ++j)
       h[I[j]] = h[I[j]]+1;
}
```

    - However, there is a possibility that two or more threads update `h[I[j]` at the same time, causing a race condition.

  - ✓ A safe parallel implementation would look like this:
    - Divide the array I into nt groups (e.g.: $nt = 4$).
    - For each group, generate a histogram, called partial histogram `hp[i]`, `i=0:nt-1`. Note that `hp` has $nb=256$ elements.
      - Here, you use *parallel_for* with iteration space [0,nt-1]
    - Once the partial histograms are ready, add up all these vectors onto a vector $\vec{h}$ (of size $nb=256$).
      - Here, you use **only** *parallel_reduce* to generate the resulting 256-element vector.

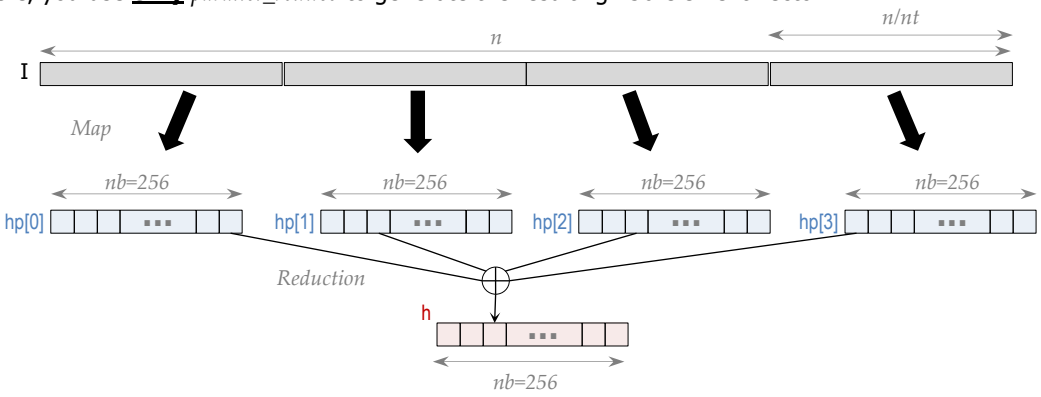

Figure 2. Safe parallel implementation of histogram computation. $nt = 4$

INSTRUCTIONS

- Write a `.cpp` program that reads a binary input file (`.bif`), computes the histogram, and stores the result (256 values) in a binary output file (`.bof`).
  - ✓ Your code should be parallelized via TBB *parallel_for* and *parallel_reduce* as per the approach illustrated in Fig. 2.
  - ✓ Your program should read in a parameter nt (number of groups in which the input image is being partitioned).

- **Considerations:**
  - ✓ Input matrix: Read from an input binary file (`.bif`). You can use the provided `puppet.bif` file that represents the 1600×1200 input image in Fig. 1(a). Each element is an unsigned 8-bit number (or `uint8`).
    - Some MATLAB/Octave versions may have a slightly different implementation of *rgb2gray* function. It is strongly suggested that you generate your own `puppet.bif` file with the provided MATLAB script (choose option '1'). Or you can use the provided `puppet.bif` file, but MATLAB must compute the histogram based off that `.bif` file.
    - You can use the function `read_binfile` from *Laboratory 3* to read data the image data (stored as a 1D array in a raster-scan fashion) (use `typ=0` since each element is of type `uint8`).
    - You can also use the read image function available in *Tutorial #2* (for image convolution).
  - ✓ Output histogram: Elements are of type `int` (32-bit signed integer), also referred as `int32`.
    - To store the `int` output array in a `.bof` file, you can use write image code available in *Tutorial #2*.

- **Output array verification**: You need to verify the generated `.bof` file. You can do this via the `lab6.m` script.
  - ✓ Once you place the `.bof` file (`puppet.bof`) in the same folder as the script, run the script. The script will display the input image.
  - ✓ When prompted to select an option, choose option '2'. This will compute the histogram and display it.
  - ✓ Then, when prompted to select an option, choose option '3'. Here, the MATLAB® script will read the `puppet.bof` file, plot the histogram generated by your C++ code (save this file as a .jpeg), and display the sum of differences between the MATLAB and C++-generated histograms. The result should be **0**.

- Compile the code and execute the application on the DE2i-150 Board. Complete Table I (use an average of 10 executions in order to get the computation time for each case).
  - ✓ Example: `./lab6 4` ↵
    - It will compute the application using $nt = 4$.

- Take a screenshot of the software running in the Terminal for nt=4. It should show the output histogram values (try to print out as many as you can on the screen) and the processing time.
  - ✓ Your code should measure the computation time (only the actual computation portion) in us.

- Provided files: `lab6.m, puppet.jpg, puppet.bif`.

TABLE I. COMPUTATION TIME (US) – PARALLEL IMPLEMENTATION WITH TBB PARALLEL_FOR AND PARALLEL_REDUCE

nt	**Computation Time (us)**
4	
10	
20	
50	
100	

SUBMISSION

- Demonstration: In this Lab 6, the requested screenshot of the software routine running in the Terminal suffices.
  - ✓ If you prefer, you can request a virtual session (Zoom) with the instructor and demo it.

- Submit to Moodle (an assignment will be created):
  - ✓ One .zip file:
    - 1st Activity: The .zip file must contain the source files (`.cpp, .h, Makefile`), the output binary file (`.bof`), the requested screenshot, and the plotted histogram (values generated by your C++ code) as a .jpeg file.
  - ✓ The lab sheet (a PDF file) with the completed Table I.

TA signature: _____ **Date:** _____