ICEN360-GPU GPU Architecture and Programming

Project 3 (25pts) Due: Oct 25, 2016 11:59PM

- **You will be writing this project on our GPU cluster. Your code is expected to work in the cluster.**
- **If you do not allocate/de-allocate memory properly, points will be taken off your program. So, your malloc() and cudaMalloc()'s must be used properly in your code.**
- **Each late day will subtract 2 points from your assignment points**

- **PART A (3 pts):** Develop a multi-threaded CPU code to take a large BMP image (e.g., Astranaut.bmp) and reduce its resolution by an (X,Y) ratio specified by the user. For example, if the user says (3,2), the image will be shrunk by 3x in the x direction, and 2x in the y direction.
- Use the simplest implementation of "throwing away" pixels as follows:
  - If you are going to shrink an image in the x direction by 2x, you will throw away every other pixel in every row. So, in each row: keep a pixel, throw away a pixel, keep, throw away, keep, ...
  - That way, if the image has 7000 horizontal pixels, 3500 of them will be kept (pixels 0, 2, 4, 6, ...) and 3500 pixels will be thrown away (pixels 1, 3, 5, ...). The image x dimension will shrink by 2x.
  - For the y direction, throw away every other row completely.
- Use the ImageStuff.h and ImageStuff.c code for your implementation.
- Submit your code as **imshrunk.c** through Blackboard, along with everything needed to compile it (ImageStuff etc.). Call it **Proj3PartA.zip**.
- The program is launched with the following command line parameters
  - **./imshrunk Infile.bmp Outfile.bmp xshrink yshrink**
  - For example, ./imshrunk Astranaut.bmp Ast23.bmp 2 3 will read the Astranaut.bmp file, shrink it by 2x in the x horizontal direction and 3x in the vertical direction and write the shrunk image as another file named Ast23.bmp. They are both 24b RGB files.
  - Use the *Floor* function when calculating the destination size (i.e., integer division with the remainder thrown away). So, a 3002 wide image shrunk down by 3x ends up being 1000 wide.

- **PART B (5 pts):** Implement the CUDA version of **imshrunk**, named **imshrunk.cu**. Reading the user parameters, etc. will be done on the CPU side and the image will be completely transferred to the GPU side and the shrinking will be performed on the GPU side. The shrunk image will be transferred back to the CPU side and written to disk.
- In other words, the C program is identical to the CU, except that the shrinking execution is done on the GPU for Part B.
- Submit all of your files as a completely separate submission, ready to compile, named **Proj3PartB.zip**.

UNIVERSITY AT ALBANY
State University of New York

- **PART C (5 pts):**
- In this part, you will write a CPU program that sorts a large array of numbers using Bubble and Merge sort. For Part C, write a program called **sort.c** that implements the following:
  - You will run the program with the following command line arguments:
  - **./sort HowMany Type ChunkSize**
  - **HowMany** is the size of the number array. It is specified in MEGA. For example, 32 means 32 Mega numbers (i.e., 32*1024*1024=33,554,432 values). Notice that, this is different than Million.
  - **ChunkSize** is the size of the small chunks you will sort individually before merging them. Allowed options are : 32, 64, 128, 256, 512, and 1024.
  - **Type** is the type of the numbers, specified as a single character. "I" means 32-bit integer, "L" means, 64-bit long integer, "F" means single precision 32-bit Floating point, and "D" means, double-precision 64-bit Floating point. Note : "I" and "L" are signed integer types, so, negative numbers are allowed.
  - In main(), write code to create an array of **HowMany** random numbers. For example, 32 means, 32M numbers, as described earlier. The type of the numbers you are creating will be specified by **Type**.
  - You will need to malloc() sufficient space for this array first. The easiest thing to do is to have a switch() statement based on the type of the numbers, and have a different malloc() etc. based on the type. However, the for() loop could be the same.
  - Call this input array InputArray[].
  - malloc() another array named SortedArray[]. Same size as InputArray[].
  - Have a loop (doesn't need to be multi-threaded) to create random numbers, spanning the entire range of the type. For example, integers range from -2 billion to +2 billion approximately.
  - Write a function called SortChunk() that sorts a small chunk of the InputArray[] at a given index. The size of the chunk is specified in the **ChunkSize** parameters that the user specifies. Use simple Bubble sort for this.
  - The sorted chunks may overwrite the InputArray[], exactly where the chunk is located.
  - For example, for 32M numbers and a ChunkSize of 1024, you have a total of 32K chunks (i.e., 32,768). This means that, you will call the same SortChunk() function 32,768 times, each of which will sort 1024 numbers.
  - In this example, these chunks are located at array indexes
    - Chunk 0 : InputArray[0] ... InputArray[1023]
    - Chunk 1 : InputArray[1024] ... InputArray[2047]
    - Chunk 2 : InputArray[2048] ...InputArray[3071]
    - Chunk 32767 : InputArray[33553408] ... InputArray[33554431]
  - When you loop through every single one of these chunks in a for() loop, every chunk is sorted locally. So, Chunk0 is perfectly sorted, Chunk1 is perfectly sorted, etc.
  - After this, you will write another function that merges every chunk into a single sorted array and writes the output to SortedArray[]. Now, this array contains the sorted version of every number. This is your final output.
  - Don't forget to free() everything you malloc'd at the end of main().
- Submit your code as **Proj3PartC.zip**.

- **PART D (12 pts):**
    - Now, implement only the sorting part in the GPU (i.e., merging is still on the CPU). Name your code **sortG.cu**. This code will run using the same command line parameters as the CPU code, plus an additional parameter for GPU block size. i.e. it should be run as:

      ```
      ./sort G HowMany Type ChunkSize BlockSize
      ```

- You will also include a report for the PART D regarding how different GPU parameters affected your performance. Name this report **Proj3PartDReport.pdf**.
- Same as before, every required file along with your report should be zipped in **Proj3PartD.zip** and submitted. Here is how you will modify the program:
    - After malloc()'ing the InputArray[] and SortedArray[] on the CPU (host) side, also CudaMalloc() them on the GPU (device) side. These GPU arrays will reside in the Global memory of the GPU. Call them InputArrayG[] and SortedArrayG[].
    - Don't forget to CudaFree() these GPU arrays, in addition to free()'ing the CPU arrays.
    - Your main() will still create the random arrays and fill them on the CPU side according to their type.
    - Right after cudaMalloc()'ing the GPU arrays and filling the CPU arrays, you will immediately transfer them from the InputArray[] to the InputArrayG[].
    - You will implement the SortChunk() functionality on the GPU side. Call the kernel SortChunkG().
    - The easiest thing to do is to write one kernel for each different type of number, such as SortChunkGI(),SortChunkGL(),SortChunkGF(),SortChunkGD(). They are for I integer, L integer, F and D float types, respectively. (Get one version working, then copy it.)
    - Your CPU-side for() loop to sort each chunk will be replaced with a kernel launch that sorts the array on the GPU side.
    - Each GPU thread in a launched GPU block will be responsible for sorting an entire chunk using Bubble sort. However, the *GPU kernel block size* is a different story.
    - So, to fully sort 32M elements in our earlier example, you need 32K kernels (threads) running to sort the entire 32M elements, if the ChunkSize was specified as 1024.
    - The question is: what should the GPU kernel block size be? In this part, you will try different block sizes and report your results and comment on the performance.
    - If you choose a kernel block size of, say, 128, then, the 32K kernels should be launched as 256 blocks with 128 threads in each block. Each thread is responsible for Bubble-sorting 1024 numbers.
    - Of course, the performance will change when you choose a different **ChunkSize** in the command line. Alternatively, the choice of the GPU kernel block size is totally up to you.
    - Modify **ChunkSize** in the command line from 32 to 1024 in 2x geometric steps (32, 64, 128, ...).
    - For each **ChunkSize**, try different GPU kernel block sizes that make sense to you. Good options are something like 64, 128, 256, 512, 1024. Don't go higher. If your blocks have more threads, you will need fewer blocks.
    - After sorting all chunks on the GPU side, completely transfer the sorted chunks back to the CPU InputArray[] as before.
    - Completely perform the merge function on the CPU side. Confirm that the results are correct (i.e., they match the CPU golden truth results). You may want to always seed your random number generator with the same value, to make sure you're always sorting the same list.

- Report your results in your **Proj3PartDReport.pdf** report and evaluate your results. Evaluation (understanding and describing the behavior) is very important.
    - We only care about the GPU kernel execution time and how it compares to the CPU version of the chunk-sorting time. Do not report the transfer times or image read/write times.
    - Plot the GPU chunk-sorting execution time with "Runtime" in the Y axis and ChunkSize in the X axis. Vary ChunkSize from 32 to 1024 in the X Axis.
    - Your plots should be only for 32M numbers, although the code should work for different amount of numbers.
    - You will need to provide four completely separate plots, one for each **type** of number.
    - Each plot will include 6 different lines (different colors/styles), when you launch this kernel with a block size of 32, 64, 128, 256, 512, and 1024. The number of blocks will be computed automatically, e.g. with 32M elements, ChunkSize=1024, and blockSize=256, numBlocks will be 128. This is a lot of data to plot – 144 trials – so you should probably generate the results using a small script to loop over data type, chunk size, and block size.