Low-Level Parallel Programming

Assignment 4

Group A 4
Jenny Olsson & Nevine Gouda
06-03-2018

Questions

A. *Describe the memory access patterns for each of the three "heatmap" creation steps. How well does the GPU handle these access patterns?*

In order to calculate the heatmap and update the values, we accomplish this by splitting the code into the following 3 steps:

1- Setting up the memory needed in the CPU and the GPU as well as copying the data needed from the host to device.

2- Launching the Kernel functions to update the heatmap.

3- Copying the data back from the device to the host so that the heatmap could be painted.

4- For each tick repeat steps 2 and 3.

Where our kernel functions that are performed by the GPU are the following:

1- intensifyHeat

Where this kernel function is responsible for fading the heatmap from previous steps and intensifying it. We implement this by creating B number of blocks, and N numbers of threads per block. Where B = N = size of heatmap = 1024. Thus each thread is responsible for calculating the heatmap value for each position in our 2D map. Then each thread is responsible to scale its respective position heatmap calculated value to pixels since each "position" in the 2D map consists of 25 pixels.

And this is done by allocating 4 arrays, 2 in each the CPU and GPU, as seen in ***Figure 1***. Where the heatmap values are stored in a one dimensional integer array (called Device_heatmapValues) and the data is accessed through a double pointer array to said 1D array. And both the CPU and GPU have their own copy of said values. And the communication between the CPU and GPU is done by copying data back and forth between Device_heatmapValues and Host_HeatmapValues. While accessing and updating the values is done by Device_heatmapAddr and Host_heatmapAddr. This approach attempts to take advantage of locality and that no two threads will access the same memory location.

And since the number of blocks is set to the number of rows we have in the map, the gpu has each block to be responsible for one row in the map, and each thread inside that block is responsible for its respective cell in that row.

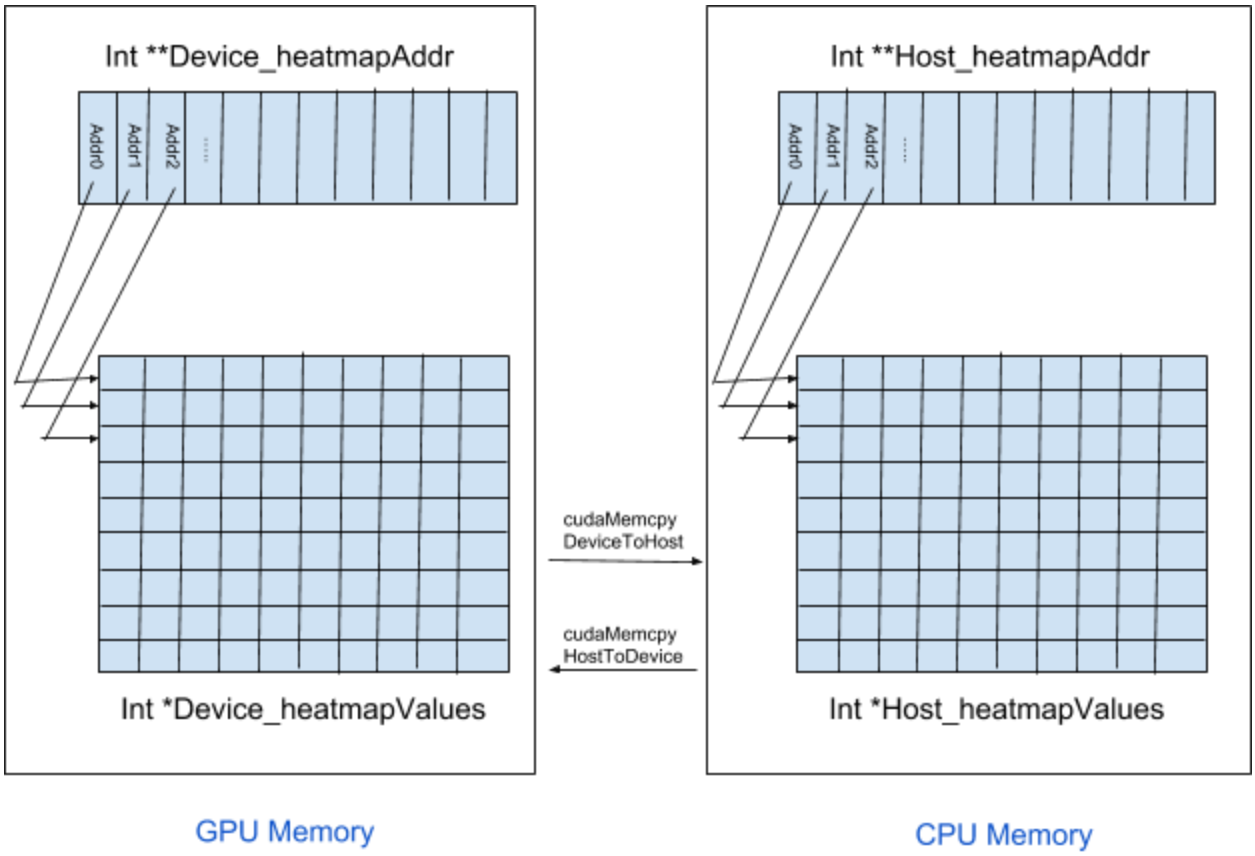


Figure 1

2- blurrHeatMapBlock

This kernel function is responsible for blurring the cells based on the calculated faded and intensified heatmap values. But since the algorithm is based on blurring the 25 (5x5) neighbouring pixels in order to blur the heatmap, there could be more than one thread wanting to blur the same pixel at the same time. Thus to fix this and have a speed up we did the following:

- Create B number of blocks, and N numbers of threads per block. Where B = N = 1000.
- Create a shared memory to be accessed and shared by all the threads within the same block. Where this shared memory is a 2 dimensional array of size 5 x 1000. Where we choose 5 because each thread needs 5 rows to blur all the surrounding pixels, and the 1000 is used to cover all the row so that no thread would need to

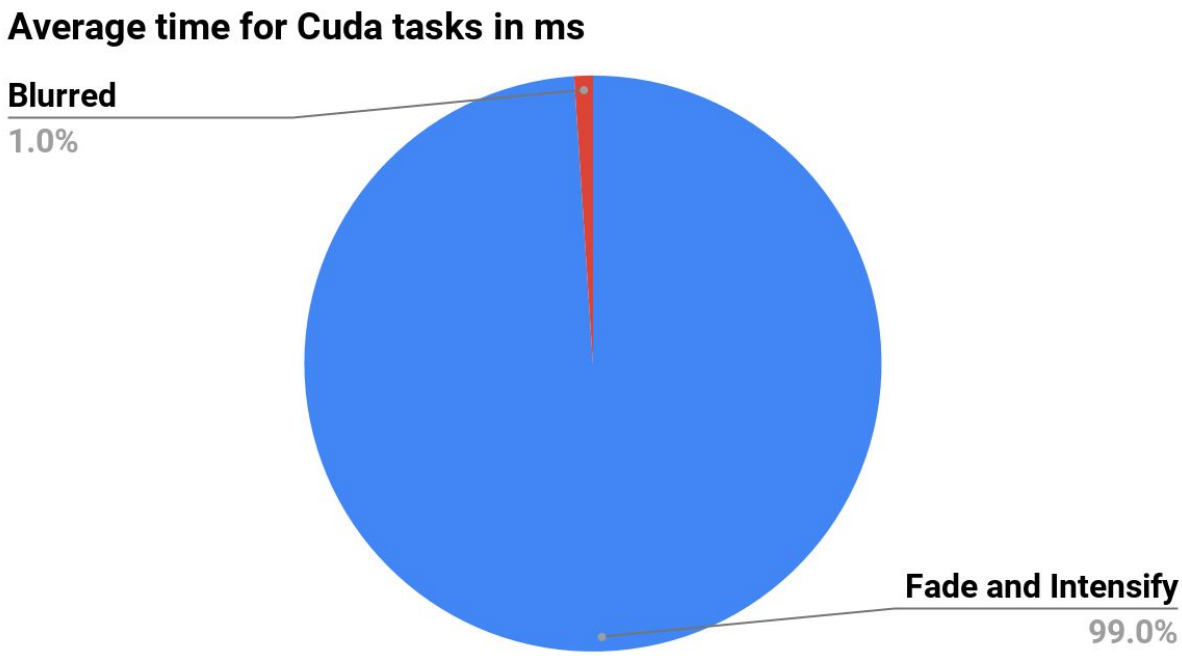
look outside the shared memory in case the pixel is not within its scoop.

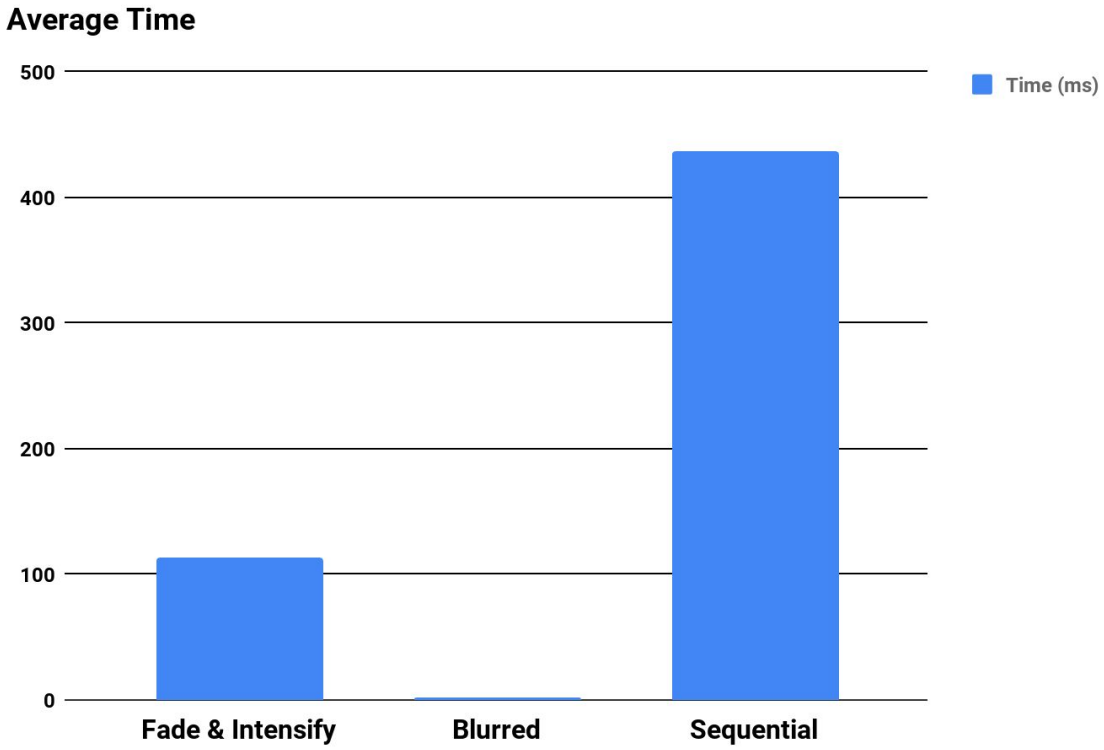
B. Plot and compare the performance of each of the heatmap steps. Discuss and explain the execution times. (Which type of plot is suitable?)

We merged the fading and intensifying steps into one kernel function and the time for all the threads to finish this kernel function in one tick has an average of 113 milliseconds. While the blurred kernel function takes 1.95 ms to finish. The fade & intensify takes more time as it contains more loops.

Number of Steps/ticks	**Average time for Scenario.xml**		
100	**Fade & Intensify**	**Blurred**	**Sequential**
	113 *ms*	1.95 *ms*	437 *ms*

Table 1





C. *What speed up do you obtain compared to the sequential CPU version? Given that you use N threads for the kernel, explain why you do not get N times speed up?*

Given that the kernel functions have synchronization between them, thus we can add both times to calculate the overall time for the parallel version. Thus we have a speed up of $437ms/115ms = 3.8$. And we don't get N times speed up because the threads have to synchronize and wait for the rest of the threads to finish thus, this time will increase as the number of threads increase.

D. How much data does your implementation copy to shared memory?

The shared memory created is a 2D array of size 5x1000x4 where 4 is the size for an integer, 1000 is the size of columns per row and 5 is the number of rows needed for each. Thus for each block the threads load around 5000 integers .

Specifications

The different versions can be chosen by passing an argument to the application as suggested in the assignment description:
Project Properties→Configuration Properties→Debugging→Command Arguments field
The acceptable inputs are *AB* where *A* is nothing or *--timing-mode* and
B is nothing or one of the following:

- *--OMP*
- *--PTHREAD*
- *--VECTOR*
- *--OMPMOVE*
- --*SEQ* (default)

Some examples for the command line arguments are:

1. *--timing-mode --OMP*
2. *--PTHREAD*
3. *--timing-mode --SEQ*
4. *--timing-mode --VECTOR*

We used a lab computer with the following processor:

1. Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, 3201 Mhz, 4 Core(s), 4 Logical Processor(s)

We used a lab computer with the following GPU:

1. **Name:** NVIDIA GeForce GT 630M
   Manufacturer: NVIDIA
   Chip Type: GeForce GT 630M
   DAC Type: Integrated RAMDAC
   Approx. Total Memory: 4095 MB
   Current Display Mode: 1920 x 1080 (32 bit) (60Hz)
   Monitor: Generic PnP Monitor