1) Control flow divergence is instructions that diverge from each other so that they prevent us from executing them according to the Single Instruction Multiple Data model and reducing the performance of our GPU.
2) http://stackoverflow.com/questions/5531247/allocating-shared-memory/5531640#5531640 Stackoverflow helped me a lot with this question in the coding part, as explained in the second answer: extern __shared__ int a[]; dynamically allocates an array and its size is already passed to the device function as the third argument when it was launched in the host.
3) https://www.microway.com/hpc-tech-tips/5-easy-first-steps-on-gpus-accelerating-your-applications/ Here's a satisfactory answer, by using shared variables for our threads we can accelerate our program.
4) I ran a sample program that can be found on Nvidia's CUDA documentation website http://devblogs.nvidia.com/parallelforall/how-query-device-properties-and-handle-errors-cuda-cc/ The result is this:

```
Device Number: 0
  Device name: Tesla K20c
  Memory Clock Rate (KHz): 2600000
  Memory Bus Width (bits): 320
  Peak Memory Bandwidth (GB/s): 208.000000

Device Number: 1
  Device name: GeForce GTX 480
  Memory Clock Rate (KHz): 1848000
  Memory Bus Width (bits): 384
  Peak Memory Bandwidth (GB/s): 177.408000
```

5) -arch=sm_20 This is what I found at https://devtalk.nvidia.com/default/topic/488327/easy-question-what-compile-flag-for-atomicadd-/ Apparently only compilers with version 2.0+ support atomic instructions and THANKS A LOT for talking about atomic instructions it solved my problem I kept getting different results than the CPU function's result.

Those were the answers to the question in part 2.

By the way, in addition to 15 expected runs we need to make with 3 different k values and 5 different number of threads, I repeat each run 4 times to get the average of the runtimes when I'll be plotting, so be aware of it when you run the bash script.

About my implementation details,

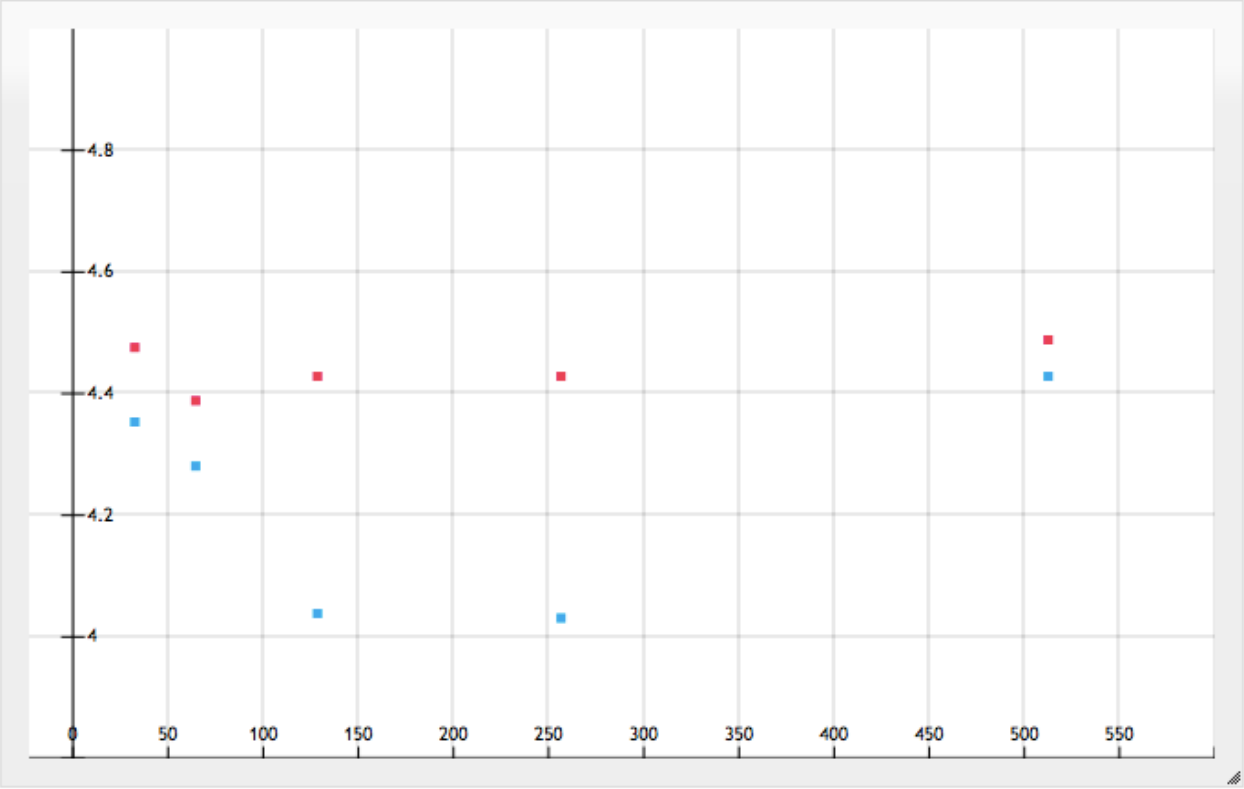
I'd like to say thank you for the questions in part 2, they were very nice clues on figuring out what to do in the Gpu function.

Here's some details about the implementation:

```
// threadIdx.x                                          -> offset inside the block
// blockIdx.x * blockDim.x                              -> offset of the block on its grid
// blockIdx.y * blockDim.x * gridDim.x  -> which grid we are on times the size of a grid
int index = threadIdx.x + blockIdx.x * blockDim.x + blockIdx.y * blockDim.x * gridDim.x;
```

This is what I had to determine the physical location of a thread when I needed to find out which index of the passed arrays I'll have to access. I'm using a shared memory to put the results of each product of the corresponding elements of the vectors. And all else goes down pretty smoothly after synchronizing the threads, however one important remark for the device function is I need to use

atomic add to sum the products inside blocks to find the overall sum, if I don't I get variable results which is not obviously good.

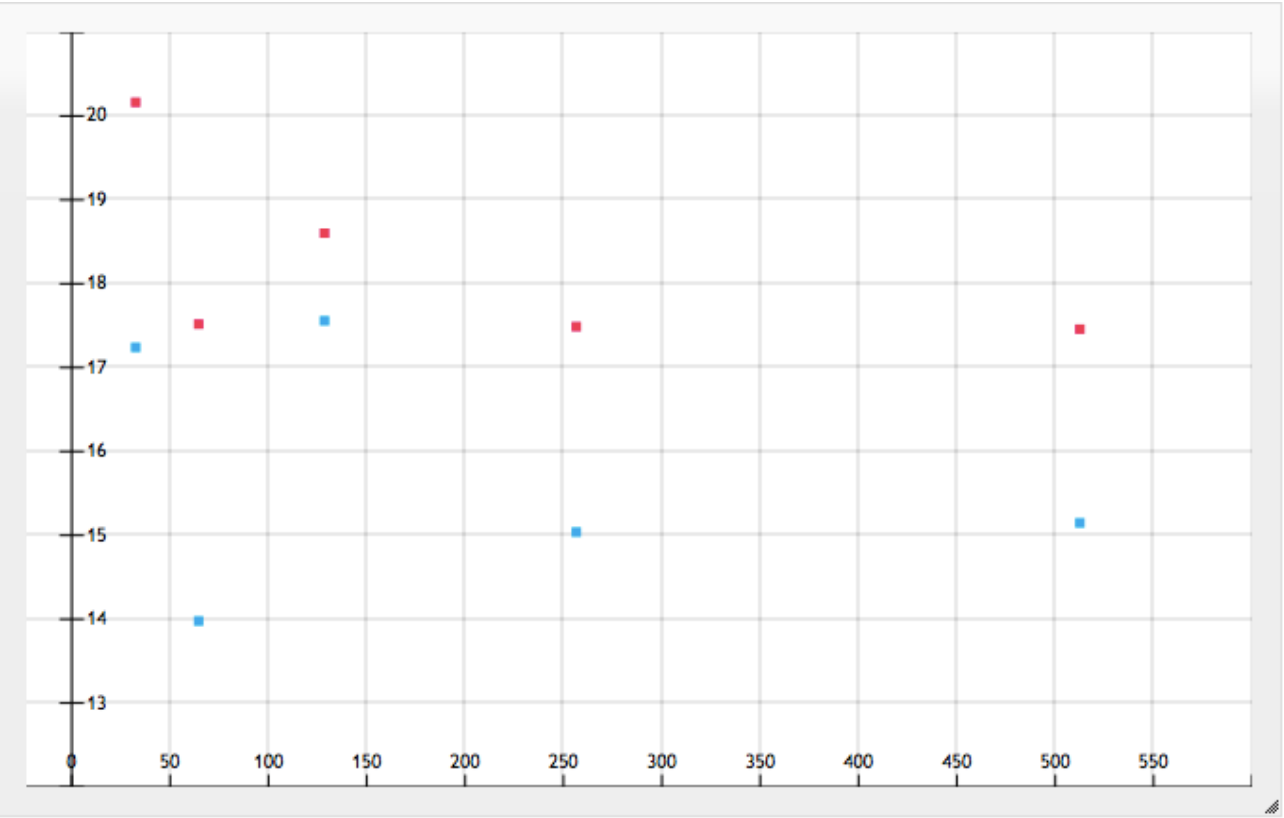


Note: The runtime results have been rounded to 2 digits after the decimal and then the average is calculated.

In this plot vector dimension is 1,000,000. I have time on the y-axis in miliseconds and number of threads per block on the x-axis with values 32, 64, 128, 256 and 512. Red points stand for CPU runtime and blue points stand for GPU runtime. There can be seen a clear decline for the runtime with increasing number of threads per block in the GPU compared to the CPU, and by the way these results are the average of 4 different runs of the program with the same parameters and the GPU performance may seem to have oddly decreased with 512 threads per block, however in that case the runtimes were as follows, 4.03 ms, 5.56ms, 4.09 ms and 4.06 ms, the median is actually 4.075~ ms, so 5.56 ms was probably due to something else,
(maybe another student was running the program too?, possible?)
but I still opted to include it in the plot.

On the next page is the plot for dimension = 5,000,000. And I follow the same concept as the previous plot, have time in ms on the y axis and varying number of threads on the x-axis 32, 64, 128, 256 and 512. Again, the red points stand for CPU runtimes and the blue points for the GPU runtimes. And I need to add again, I have odd results as the one before again: Just one of the runtimes may be very different than the other 3 but I still included them in the plot, although this time the increase of GPU performance with larger number of threads is not that clear because of the zigzags.

But more importantly we now have a sharper difference between the CPU versus GPU performance and GPU is clearly doing better in the order of the first comparison.



I shall not have a graph for varying vector dimensions with number of threads remaining constant since it is pointless and I didn't plot for dimension equals 10,000,000 since the results are very similar to the previous plots.

But I have a few comments to make in the overall, firstly the main bottleneck for the device performance is the transfer of data from host to device, whereas the kernel execution takes around 2 percent of the overall runtime. The performance