Questions
=========

hello_omp
----

* How many omp threads are reported as being available? Try increasing the number of cpus-per-task. Do you always get a corresponding number of omp threads? Is there a limit to how many omp threads you can request?

The number of reported available threads are same as the number of cpus-per-task. We could get a maximum of 39 threads available with 39 cpus-per-task. For 40 cpus-per-task the job is queued for a long time and is not completed.

* What is the reported hardware concurrency and available omp threads if you execute ompi_info.exe on the login node?

The hardware concurrency and available omp threads on the login node are 96.

norm
----

* What are the max Gflop/s reported when you run norm_parfor.exe with 8 cores? How much speedup is that over 1 core? How does that compare to what you had achieved with your laptop?

With 8 cores, the max performance is 14.039 GFlop/s for 8 threads. Similar performance is achieved by the sequential version- 14.237 GFlop/s.
With 1 core the max performance is 1.839 GFlop/s, which is about 8 times speed up with 8 cores.
The max performance on the laptop was 15.836 GFlop/s for 8 threads and for the sequential version the max performance was 2.51082 GFlop/s.

matvec
------

* What are the max Gflop/s reported when you run pmatvec.exe with 16 cores? How does that compare to what you had achieved with your laptop? What happens when you “oversubscribe”?

With 16 threads we get a maximum of 26.1734 Gflop/s for a problem size- N(Grid) = 512.
On the laptop the maximum performance was 8.438 GFlop/s with 8 threads. With 16 threads the maximum performance was 7.789 GFlop/s.
On oversubscribing(32 cpus), the performance is drastically reduced- 9.5 Gflop/s for N(Grid) = 512 and the program is killed for bigger problem sizes.

pagerank
----

* How much speedup (ratio of elapsed time for pagerank comparing 1 core with 8 cores) do you get when running on 8 cores?

The elapsed time for pagerank with 1 core is 3568 and with 8 cores is 1228, almost 3 times speedup.

cu_axpy
----

* How many more threads are run in version 2 compared to version 1? How much speedup might you expect as a result? How much speedup do you see in your plot?

Version 2 runs 256 threads as compared to 1 thread for version 1. We expect a 256 times speedup as a result assuming there is no overhead associated with accessing the partitioned vectors. We see about 50-60 times speedup in the plot.

* How many more threads are run in version 3 compared to version 2? How much speedup might you expect as a result? How much speedup do you see in your plot? (Hint: Is the speedup a function of the number of threads launched or the number of available cores, or both?)

The number of threads run by version 3 depends on the problem size N. Specifically, total number of threads is num_blocks*block_size = (N+blocksize-1)≈ N. Version 2 has 1 block with 256 threads.
There are a total of 4608 cores(72 SMs and 64 cores/SM)on Quadro RTx 6000, thus we expect the speedup (as compared to a single thread) to have an upper limit of 4608*number of threads that can be executed by each core (I was unable to find the info regarding the number of warps per SM for the Turing architecture).
In the plot we see about 2300 times speedup as compared with version 1 cu_axpy1 and about 35 times speedup as compared with version 2.

* (AMATH 583) The cu_axpy_3 also accepts as a second command line argument the size of the blocks to be used. Experiment with different block sizes with, a few different problem sizes (around :math:`2^{24}` plus or minus). What block size seems to give the best performance?

For problem sizes 2^22, 2^23, 2^24 we see performance increase going from blocksize=32,64,128,256, with max performance for blocksize = 512 and the plateau in performance for blocksize = 1024. For problem size 2^25, 2^26 we see increasing performance as we increase the blocksize with a maximum at 1024.

nvprof

* (AMATH 583) Looking at some of the metrics reported by nvprof, how do metrics such as occupancy and efficiency compare to the ratio of threads launched between versions 1, 2, and 3?

Striding

* Think about how we do strided partitioning for task-based parallelism (e.g., OpenMP or C++ tasks) with strided partitioning for GPU. Why is it bad in the former case but good (if it is) in the latter case?

We have massive number of threads available with the GPU and thus smaller number of partitions are needed this probably hides the latency issues with non-contiguous access of data from the gpu memory. CPU threads are fast but fewer in number and have to do many more non-contiguous load operations because the problem can be divided into fewer partitions with the CPU. Thus, each CPU thread operates on many more elements and loading them is costly if non-contiguous.

norm_cuda

* What is the max number of Gflop/s that you were able to achieve from the GPU? Overall (GPU vs CPU)?

We were able to achieve a maximum of 62.1378 GFlop/s with GPU for a problem size of 2^21, with cu_norm_4. The maximum performance with CPU is 51.5684 GFlop/s with 8 threads, for a problem size 2^20. The CPU seems to be quite fast and executes at ~6 GFlop/s and goes up to ~52 GFlop/s with 8 threads. It’s unclear how this performance gain is being achieved.

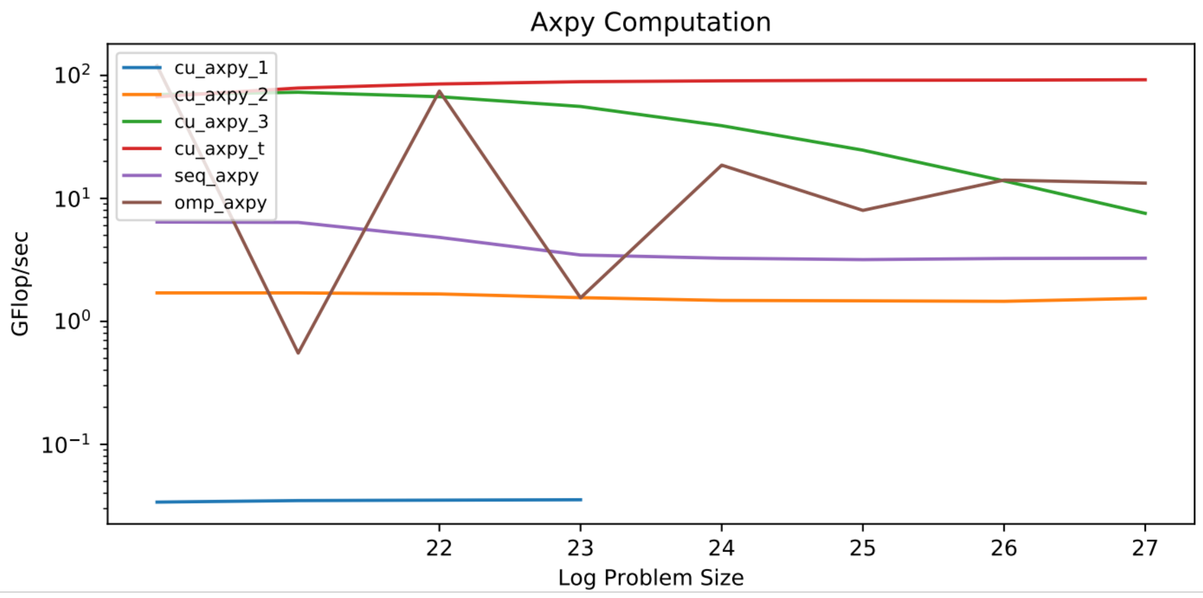
About ps7

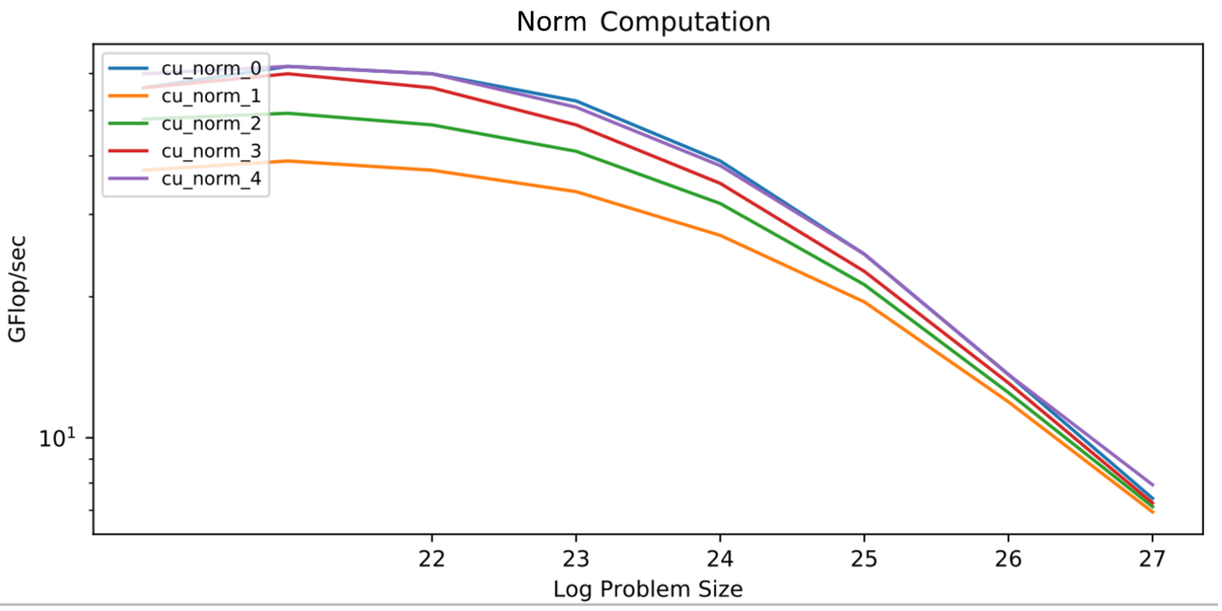
Answer the following questions (append to Questions.rst):
a) The most important thing I learned from this assignment was...

GPUs can hide non-contiguous memory access by running a large number of threads in parallel, which wasn’t the case with the CPU.

b) One thing I am still not clear on is...

Why does cu_norm_0 perform just as good as cu_norm_4 and slightly better for N = 10^23-10^24, and performs much better than other versions of cu_norm?


Axpy Computation
cu_axpy_1
cu_axpy_2
cu_axpy_3
cu_axpy_t
seq_axpy
omp_axpy
10^2
10^1
10^0
10^{-1}
GFlop/sec
Log Problem Size


Norm Computation
cu_norm_0
cu_norm_1
cu_norm_2
cu_norm_3
cu_norm_4
10^1
GFlop/sec
Log Problem Size