Questions

hello_omp

- Question 1: 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?

Answer: With "srun --time 5:00 -A niac --cpus-per-task 2 ./ompi_info.exe", we have 2 omp threads available. When I try to increase the number of cpus-per-task, I can get a corresponding number of omp threads when the number I tried is less or equal to 40. There is a limit of 40 as the max number of omp threads I can request. When I try to increase the number of cpus-per-task over 40 (e.g. 41), I would get an error.

- Question 2: What is the reported hardware concurrency and available omp threads if you execute ompi_info.exe on the login node?

Answer: The reported hardware concurrency and available omp threads by executing ompi_info.exe on the login node is 40.

norm

- Question 3: 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?

Answer: The max Gflop/s reported when running norm_parfor.exe with 8 cores is 12.4028 Gflop/s with 8 threads when N = 2097152. Compared with running norm_parfor.exe over 1 core, it's about 6.46 times speedup. When I did ps6 and ran norm_parfor.exe on my own laptop, my max Gflop/s reported is 19.9126 Gflop/s with 8 threads when N = 1048576 (my laptop also has 8 cores). So for my laptop, this performance compared with the 1 core performance achieved about 7.54 times speedup. We see that my laptop achieved better speedup in the performance compared to the performance when we run on the Hyak cluster.

matvec

- Question 4: 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?

Answer: The max Gflop/s reported when I run pmatvec.exe with 16 cores is 9.97083 Gflop/s for my CSC^T with 16 threads with N(Grid) = 512. The max Gflop/s reported when I run pmatvec.exe with my own laptop when I did ps6 was 13.5018 Gflop/s for my CSC^T with 8 threads with N(Grid) = 256. So, my laptop has achieved a better max Gflop/s result compared to the result when we run on the Hyak cluster.

pagerank

- Question 5: How much speedup (ratio of elapsed time for pagerank) do you get when running on 8 cores?

Answer: Elapsed time for pagerank for 1 core is 2609 ms, for 2 cores is 2652 ms, for 4 cores is 1937 ms, for 8 cores is 1406 ms. So, compared to running on 1 core, when running on 8 cores we get about 1.86 times speedup; compared to running on 2 cores, when running on 8 cores we get about 1.88 times speedup; compared to running on 4 cores, when running on 8 cores we get about 1.38 times speedup.

cu_axpy

- Question 6: 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?

Answer: Only 1 thread is running in version 1, and 256 threads are running in version 2. Version 2 also utilized partition so that the madd() function gets to know which thread it is at when running. Compared to version 1, version 2 should get around 50 times speedup in the performance. From my plot, version 1 gets about 0.03 Gflop/s, version 2 gets about 1.62 Gflop/s, so it's about 55 times speedup which fits my expectation.

- Question 7: 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?)

Answer: Version 2 and version 3 both are running on 256 threads. Version 3 utilized Blocks. The indexing in version 3 also taking into account blocks. The speedup should also be around 50 times for version 3 compared to version 2. From the plot, version 2 gets about 1.62 Gflop/s, version 3 gets about 70 Gflop/s (peak performance), so it's about 45 times speedup. This fits my expectation.

From the plot, when the log problem size increases, the performance of version 3 is dropped and slowly towards to the performance of version 2. The speedup for version 3 compared to version 2 as the log problem size inscreases are decreased. This might because when the problem size grows larger and larger, we need more cores to solve for version 3 and when there is not that many available, the performance would slow down gradually. Version 2 and version 3 both using the same number of threads, we still see about 45 times speedup (max speedup). Combining with question 6 version 2 uses more threads than version 1 and had about 55 times speedup, we can see that the speedup is a function of both the number of threads launched and the number of available cores. Both factors can influence how many speedup we can get.

- (AMATH 583) Question 8: The cu_axpy_t 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 plus or minus). What block size seems to give the best performance?

Answer: I am from Amath 483.

nvprof

- (AMATH 583) Question 9: 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?

Answer: I am from Amath 483.

Striding

- Question 10: 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?

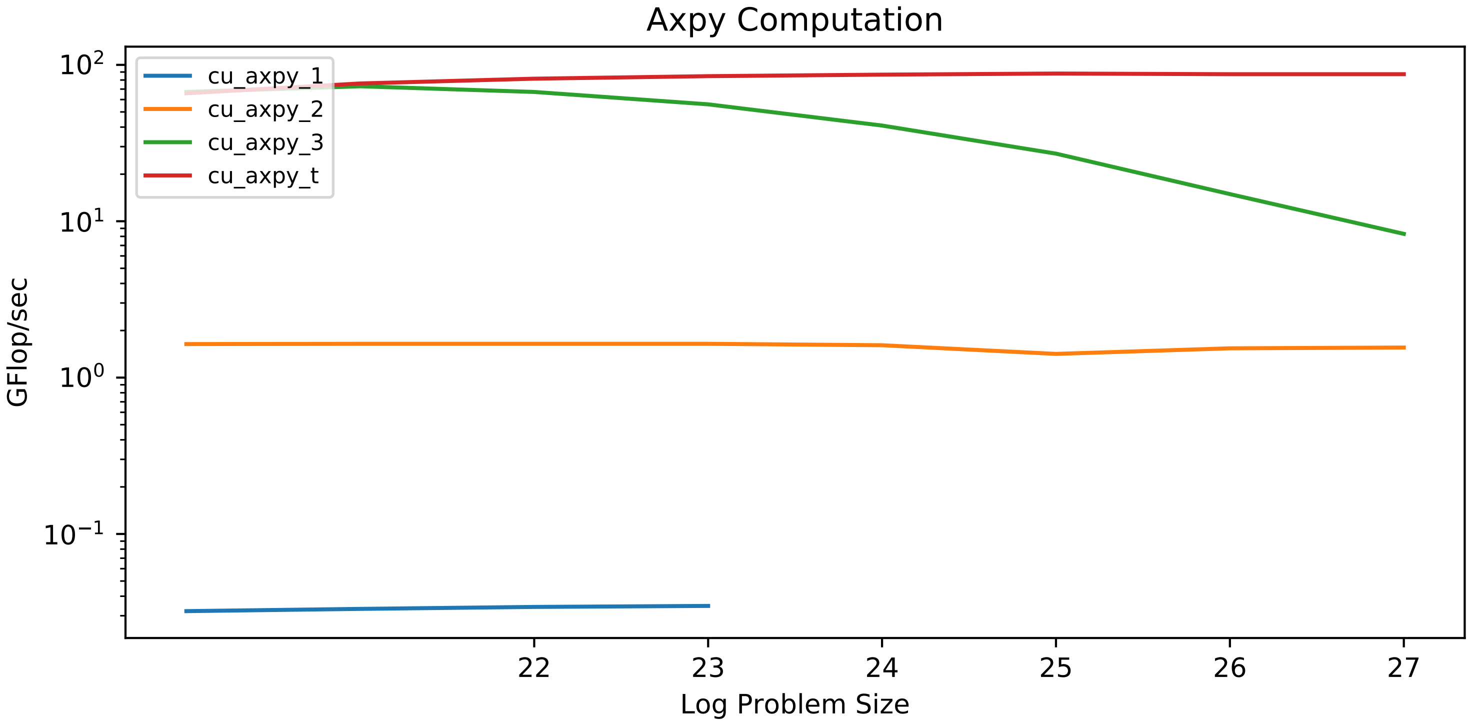
Answer: When we do strided partitioning for task-based parallelism, the element we try to access is not nearby each other in the memory so we don't have the benefit of quick accessing the data that already stored in the cache, instead we would need to fetch those separated elements every time we try to get them which can take a lot of time if summed together. This is why the cyclic methods in the previous assignments (e.g. cnorm) presents slower performance compared to the sequential performance. However, in this CUDA case, even it seems like we are still using strided partitioning, in a grid which we stored in the memory, we have blocks stored next to each other. Inside each block, we have the number of threads stored inside. So, with this special Grids and blocks logical organization, when we stride partition with blocks, the blocks we are trying to access are next to each other in the memory. Each thread loads one element from global to shared memory. For this case, we still utilizing the benefits of quick accessing the different blocks that already stored in memory. This is why it is bad in the former case but good in the latter case.

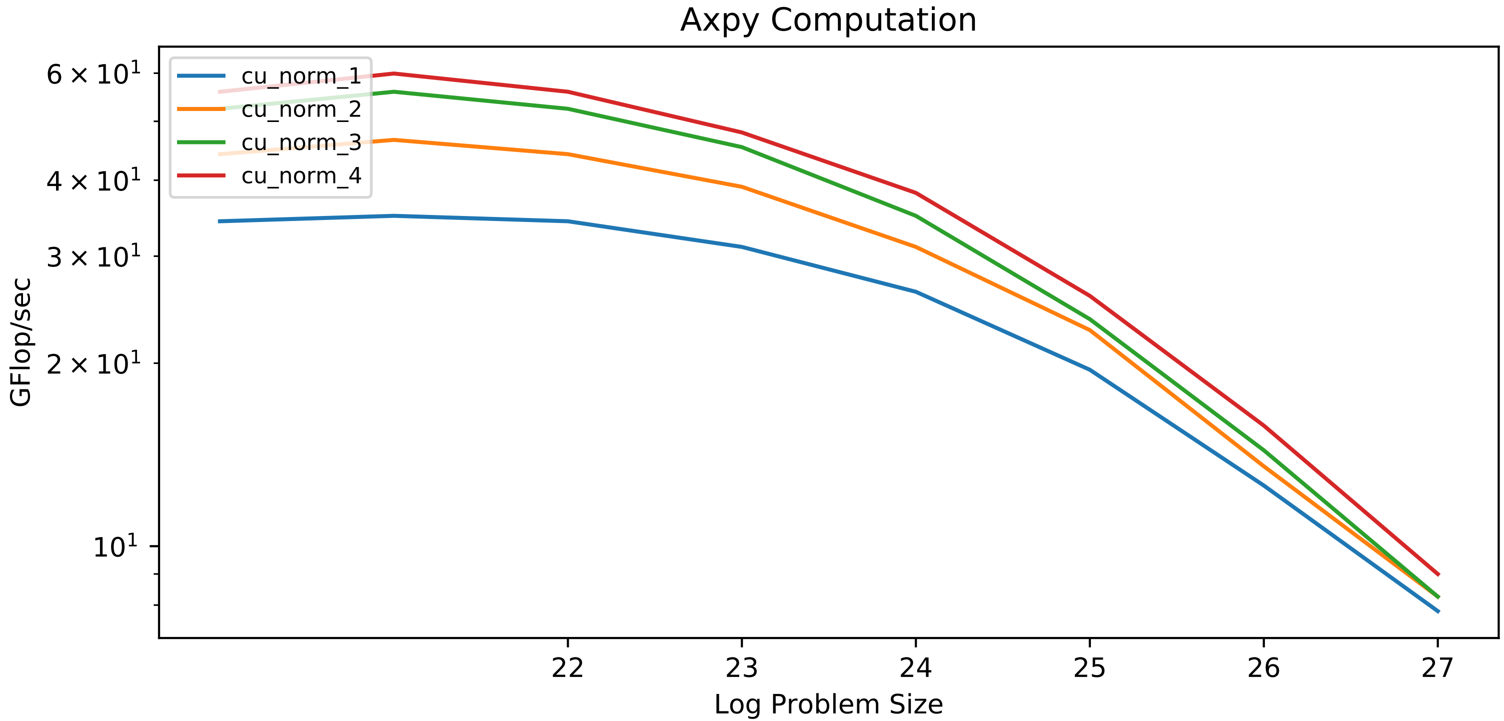
norm_cuda

- Question 11: What is the max number of Gflop/s that you were able to achieve from the GPU? Overall (GPU vs CPU)?

Answer: The max number of Gflop/s that I was able to achieve from the GPU was about 55.9241 Gflop/s with cu_norm_4.

Overall, the max number of Gflop/s that I was able to achieve from the CPU was about 78.0336 Gflop/s with omp_axpy with the number of --cpus-per-task being 8; the max number of Gflop/s that I was able to achieve from the CPU from the 4 cu_axpy_~ (~ = 1, 2, 3, t) cases was about 69.9051 Gflop/s with cu_axpy_3. So, overall the max number of Gflop/s I achieved was from the CPU with 78.0336 Gflop/s.


Axpy Computation
cu_axpy_1
cu_axpy_2
cu_axpy_3
cu_axpy_t
10^2
10^1
10^0
10^{-1}
GFlop/sec
22
23
24
25
26
27
Log Problem Size


Axpy Computation
GFlop/sec
Log Problem Size
6×10^1
4×10^1
3×10^1
2×10^1
10^1
22
23
24
25
26
27
cu_norm_1
cu_norm_2
cu_norm_3
cu_norm_4