Questions

Ben Francis

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?

For the case of –cpus-per-task being 2, we get 2 threads. This works up to 40, and then tells me I can't use more than 40.

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

96 cores, and 96 threads available.

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?

6-7X speedup, depending on problem size. This is better than my ~2-3x speedup I saw on my Mac.

matvec

(Question 4) * What are the max Gflop/s reported when you run pmatvec.exe with 16 cores?

10.65 GF/s

How does that compare to what you had achieved with your laptop?

4.5 GF/s (on macbook pro), so over twice as much better

pagerank

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

Pagerank time with 1 core: 10321 ms

Pagerank time with 8 cores: 1420 ms

This is a 7x speedup

cu_axpy

(Question 6) * How many more threads are run in version 2 compared to version 1?

256 threads in version 2 versus 1 thread in version 1.

How much speedup might you expect as a result?

We would expect 256x speedup.

How much speedup do you see in your plot?

It went from about ~0.03 to ~1.6, which is ~53x speedup.

(Question 7) * How many more threads are run in version 3 compared to version 2?

Now it runs blocks, with each block getting 256 threads. So it will run a proportionally more threads compared to blocks; the blocks are defined by (N + blockSize - 1) / blockSize, where blockSize = 256, so: N + 256 - 1 / blockSize (note it is int, so it will round down) N is 1 << 16, which is 2^16, which is ~65K. (65,536) (65536 + 256 - 1) / 256 = 256.99 –> 256 after rounding.

How much speedup might you expect as a result? 256x.

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?)

We see at best 45x speed up, and it degrades with larger problem size, hovering around 40 for the first few before really falling off. This is because our number of cores are limited; we can't actually each thread separately, due to constraints. Of course, the speedup is still proportional to the threads launched, but here we try to launch a ton of them so that isn't the constraining factor.

* (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 2^{24} plus or minus). What block size seems to give the best performance?

I experimented with many block sizes, ranging from size 8 to 256, and they all basically performed at 88 GF/s.

I expect that this is because it does not support thrust in some way.

nvprof

(Question 9) * (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?

On Piazza, Professor Lumsdaine said:

Unfortunately it looks like we won't be able to get the privileges needed to run nvprof (or equivalents) on hyak. Go ahead and skip this problem for this assignment. (I'll see if there is some way to do some profiling for the next assignment(s)).

As such, this question is skipped.

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?

In the case of OpenMP/C++, we had each thread access similar regions of memory, instead of neatly dividing them up among threads. In that case, we were using stride to assign our threads to the data.

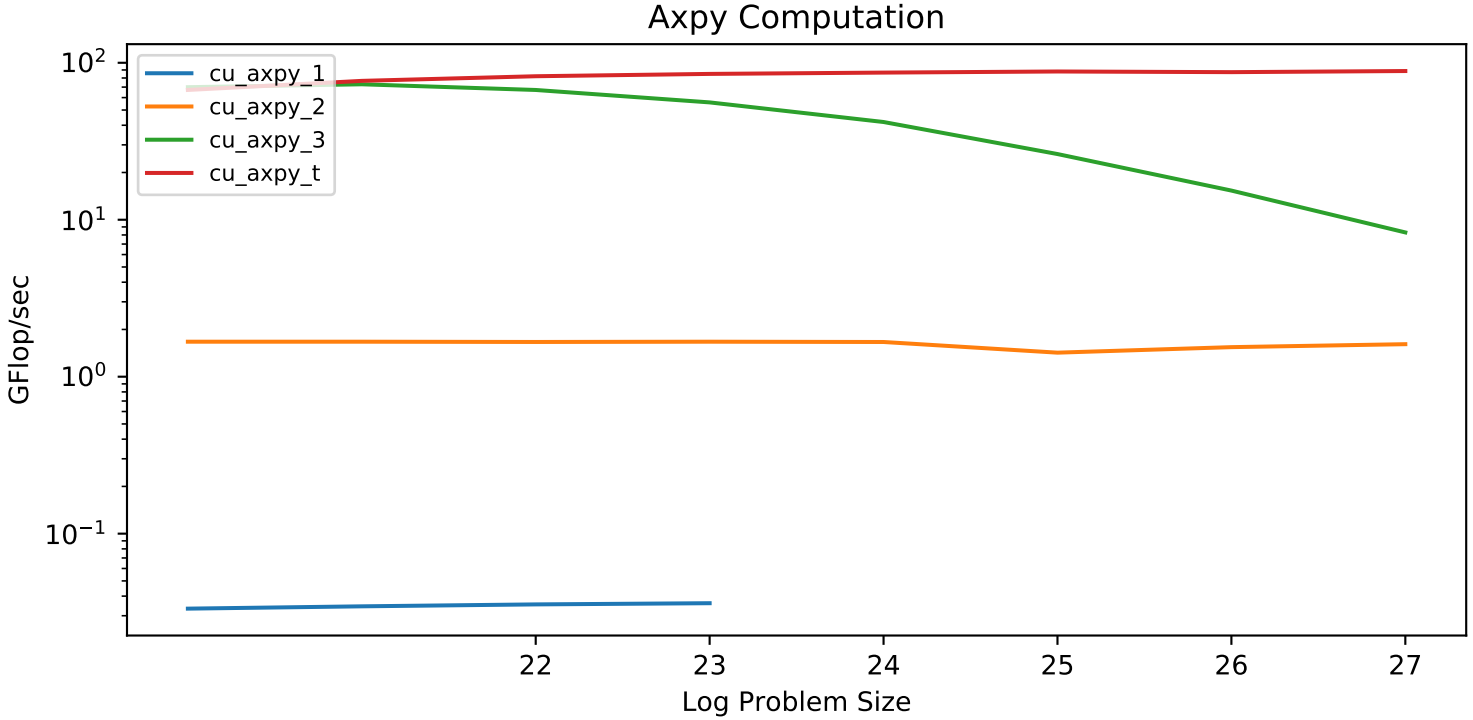
The reason we are using striding *here* is because of the data structure that threads are stored in (see pg 48 of lecture notes). Thread IDs are assigned within a block, so this striding is just a way of decoding its "global" address. One benefit of this approach is that since we're going through all of the blocks, we're "keeping all the blocks busy"/using the blocks, which is helpful for performance.

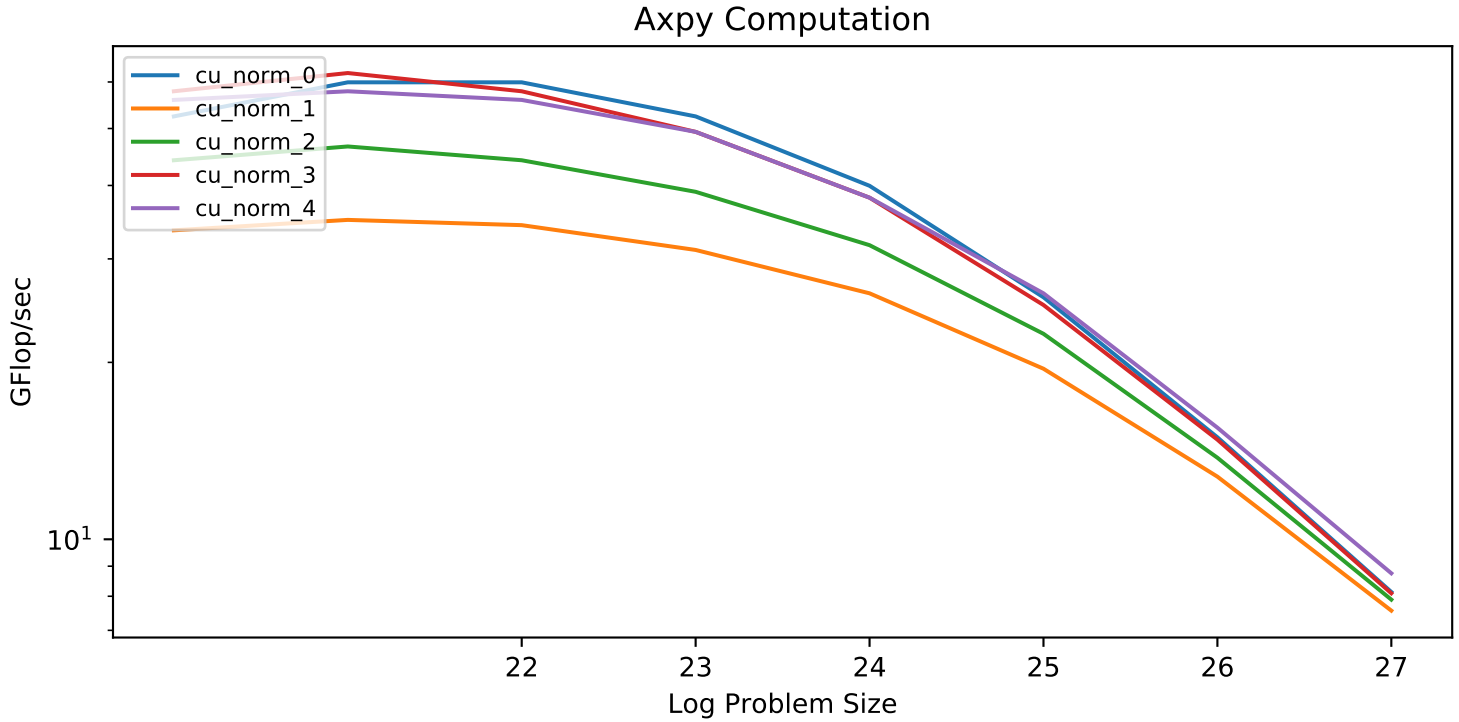
Put most succinctly, striding is here used to answer the critical question "Who am I?" (from threads)

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)?

62 GF/s. I tried this with CPU cores, but nothing was able to beat this.


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


Axpy Computation
cu_norm_0
cu_norm_1
cu_norm_2
cu_norm_3
cu_norm_4
GFlop/sec
10^1
22
23
24
25
26
27
Log Problem Size