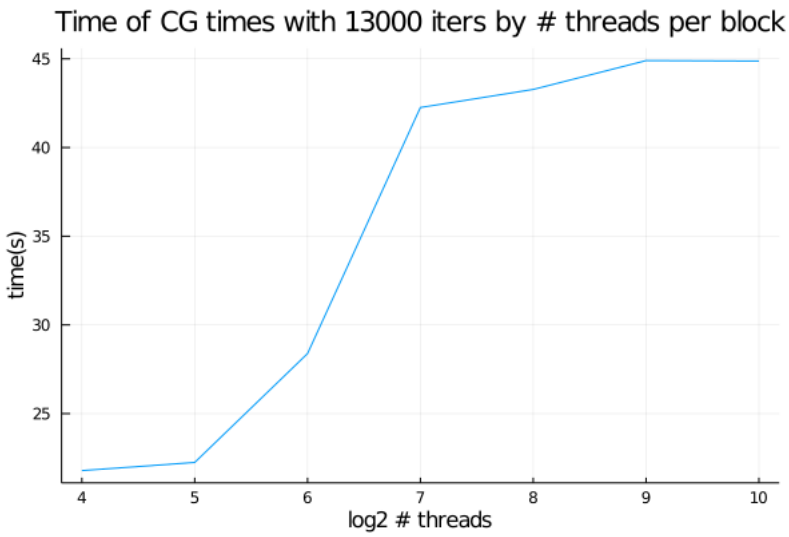
Parallel Processing Homework 6

Toby Harvey

December 8, 2020

In this homework I implemented a reduction , add vecs , element-wise multiplication kernel, to speed up CG on the GPU. Both the add and multiple kernel, were basic and embarrassingly parallel, in which each thread handles a single element of each vector. The reduction kernel is a bit more complicated. When launched on a vector each thread block reduces itself and stores its reduction in a new array that is the length of the number of threads at the index of that block. This means that on every iteration the original array of length N will be reduces to $\frac{N}{\#\text{threads per block}}$. This new resulting array is then reduced on itself, until we get to one final reduction on a single block in which the entire reduced array is stored at the first index. With this analysis is seems as though the larger blocks (more threads per block) that are launched the better, because there are $\lceil \log_{tpd}(N) \rceil$ where *tpd* is threads per block iterations. So there would be very few iterations. This doesn't actually have any affect on the total number of computations that need to be done though, as with fewer iterations there needs to be more reductions per iteration. In the below figure I plotting run times of CG vs number of threads per block, and it appears that actually few threads per block is better. To me this is unclear why? Is it possible that the optimal reduction would have similar number of iterations to number of threads per block? The arrays for the cant matrix are not big enough to test this theory, because there is never more than three iterations if we start within thread blocks of 16. i.e. the dimensions of cant are 62451 and $log_{16}(62451) \approx 4$. A few other things about the reduction kernel are worth nothing. The length of vectors worked on the reduction are actually increase up to the nearest power of 2 and padded by zeros past there actual values. This seems like a uniquely GPU way of doing things. Usually computing more will lead to lower performance, but saturating the GPU is considered "good", and avoids a lot of complicated code involving reducing on a non-power-of-two matrix. Lastly I was able to do the entire reduction with only two arrays in memory by switching back and forth between the two for each reduction read and write, this seems pretty since because memory won't really become an issue. For performance results I ran all of my tests on Tesla k80 on Talapas.


Time of CG times with 13000 iters by # threads per block
time(s)
45
40
35
30
25
4
5
6
7
8
9
10
log2 # threads