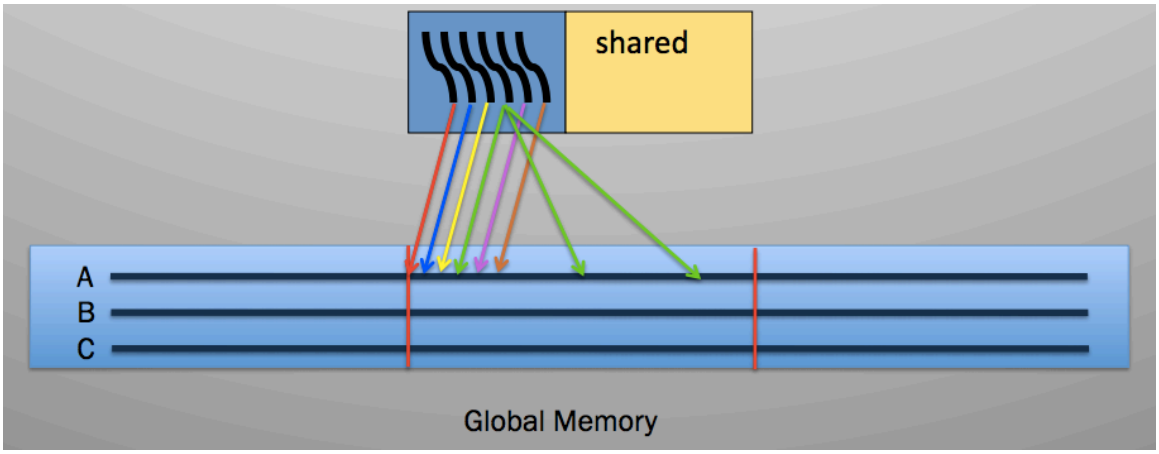
Michael Hoppal

Vector add: coalescing memory accesses

For this assignment the first program we looked at is vector add with Cuda programming. We were given a program that computes the addition of two vectors. In that program it is using non coalescing memory accesses to grab the data from the two vectors meaning that each thread is grabbing its own independent set of data to run on (threads access contiguous chunks in a partition).

Now although this works and you will get the correct answer is it the most efficient way to solve the problem? The answer is no. This can be derived by analyzing how non coalescing memory works. Each thread is grabbing its own independent set of data to work with meaning that each thread is not sharing any data. If you are doing this in parallel each thread is grabbing that chunk of data to work with and if large enough it will not all fit into the highest-level memory. This means that you will get inefficiencies with each thread fighting over trying to switch their data in.

How we can address this problem is by using coalescing memory accesses instead. In this model threads access memory in interleaved pattern, shown below.



Because of this you don't run into the problem of having all the threads trying to load their independent data into memory at the same time since they are working on the same data at the same time now. In result with this you should see a huge speed up to what the non coalescing had.

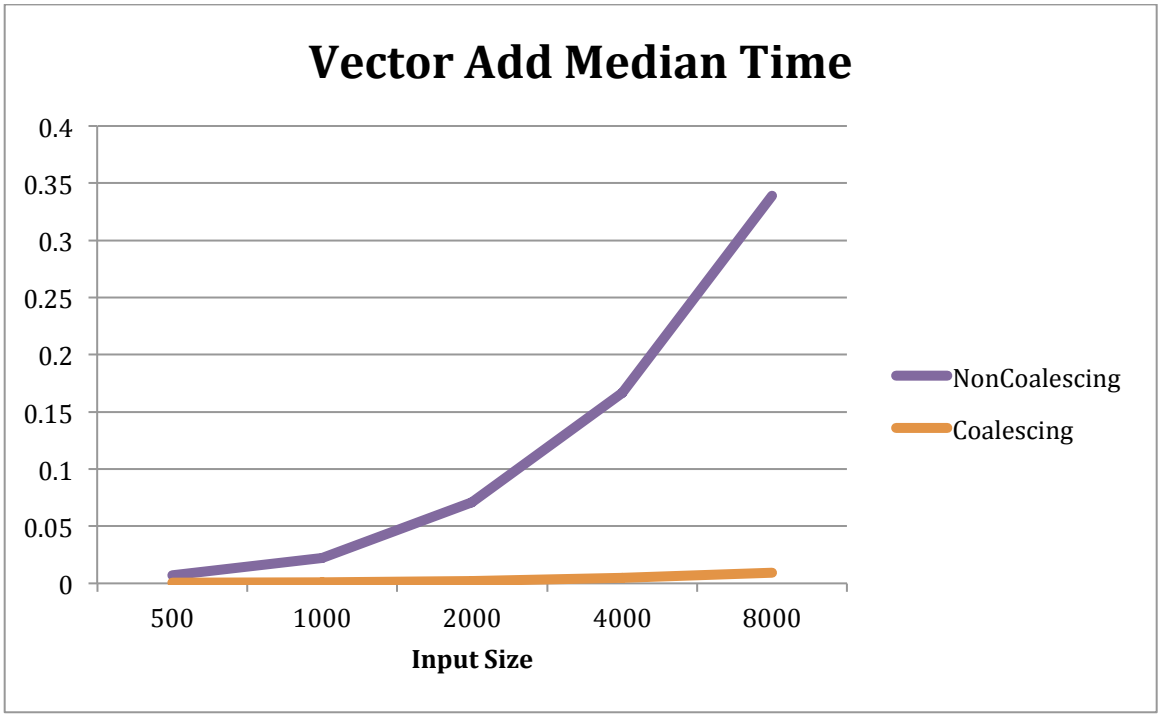
To experiment with this I rewrote the vector add to use coalesced memory accesses and then tested the same two programs on the same data set. After getting the results I graphed the median times against each other and the speed up the coalescing version saw. All of this is shown below.

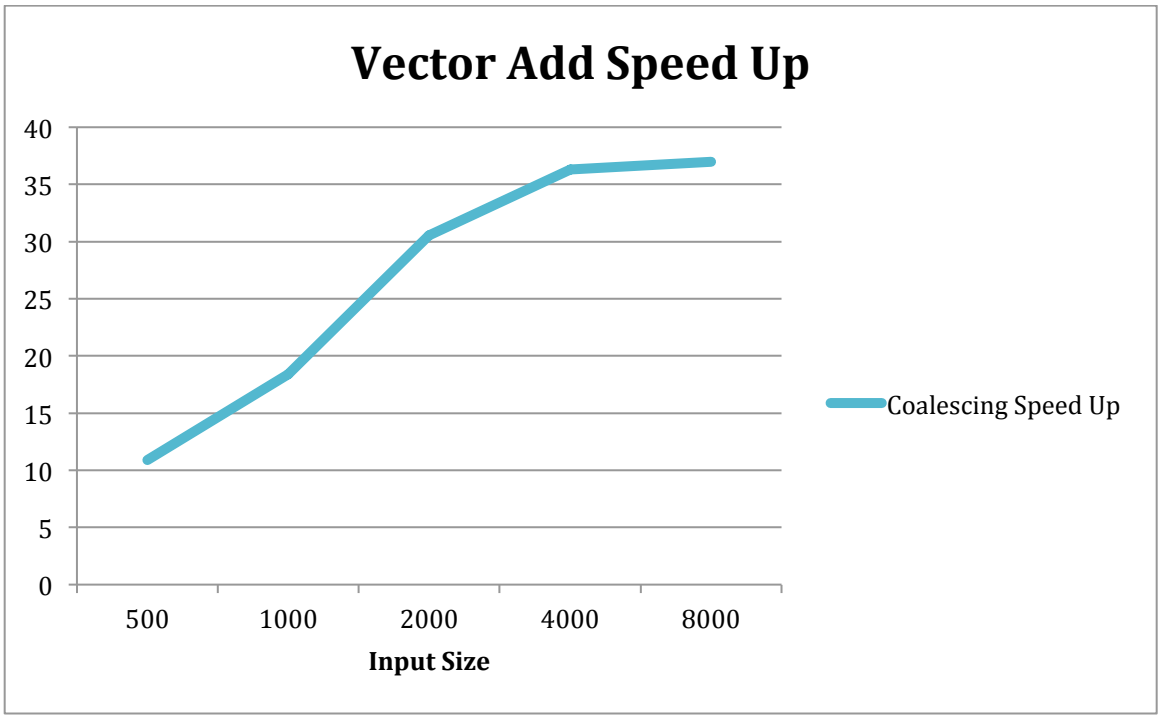
No coalescing

Input Value:	Median
500	0.006837
1000	0.021975
2000	0.070879
4000	0.166806
8000	0.338863

Coalescing

Input Value:	Median	Speed Up
500	0.000628	10.88694268
1000	0.001194	18.40452261
2000	0.002319	30.56446744
4000	0.004596	36.29373368
8000	0.009158	37.0018563





As you can see the coalescing version time increased at a very smaller rate than the non coalescing version and in result had an increasing speed up as the input grew. This result is what we expected as explained above due to the way the threads are accessing memory.

Shared / shared CUDA Matrix Multiply

For the next assignment we were given a matric multiplier Cuda program. It worked by taking in an input size for the matrix and then creating two square matrices of that size and multiplying those together.

How it did the multiplication with Cuda is there is a 16 by 16 block of threads which each had a 16 by 16 block of data from each matrix to be multiplied. Each thread would bring in one row from the first block of data and one column from the second block of data into shared memory from global memory and then multiple it together to get one value for the result matrix. It then would write the value to global memory.

Although this Cuda program works we were given the task to make it more efficient. The thing that I focused on was the amount of data that it is being brought into shared memory and how many calculations are being done. I also expanded on that and decided to change the size of the blocks of data each thread was working with.

With these two changes we can accomplish more work at once and assign more work to each thread. In class we discussed that the total memory traffic for matrix

multiplier being 2n^3/k. So the larger the k the less memory traffic and should result in a more efficient program.

In our new program we increased the block size from 16x16 to 32x32 and did four calculations at once so we could leave the thread block at 16x16. After this was accomplished we did an experiment with different matrix sizes with the original cuda program and the new one.

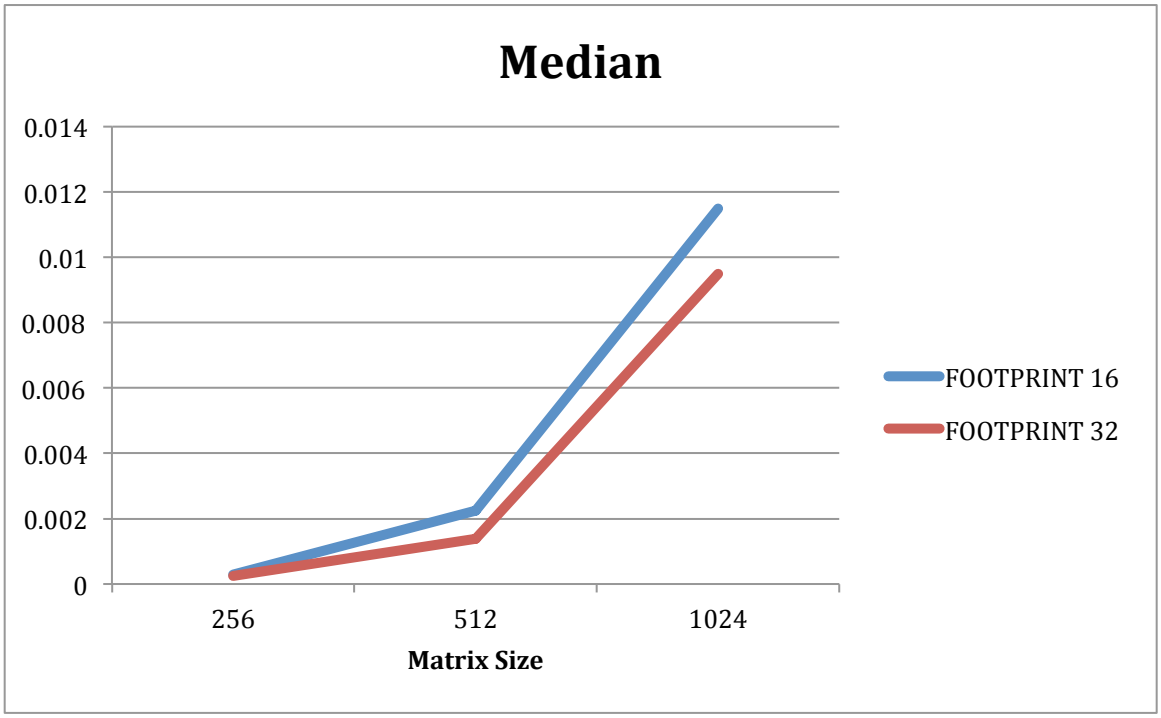
The results that we recorded are below.

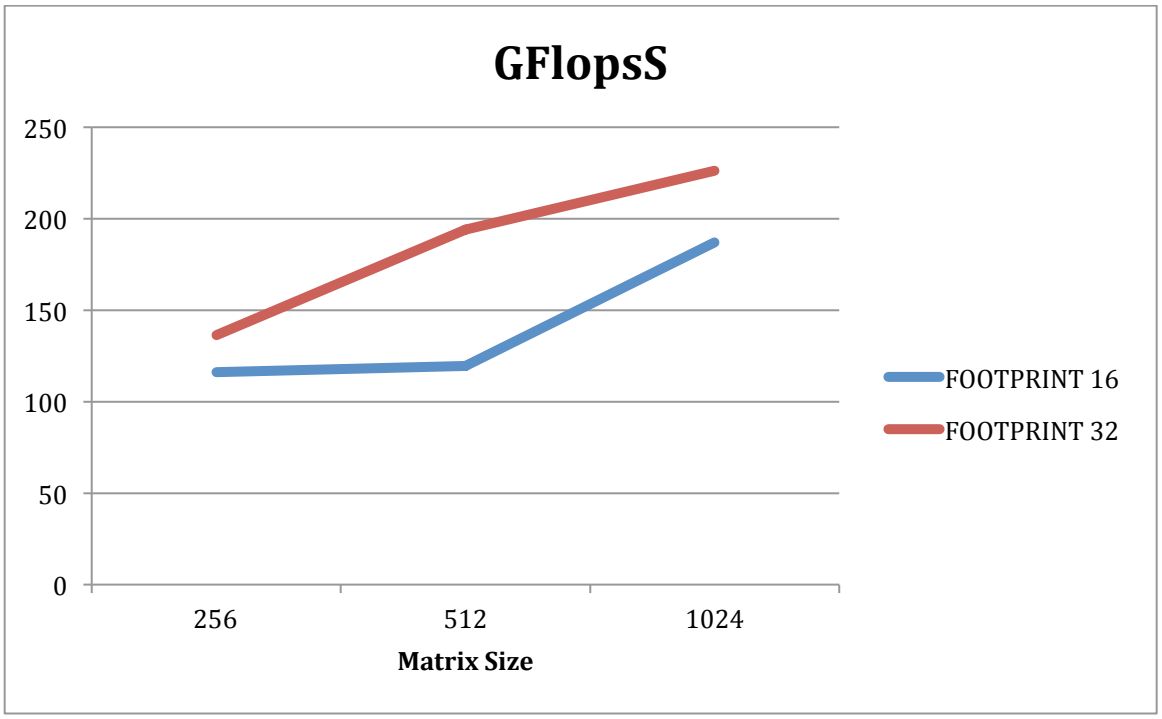
16 FOOTPRINT

Matrix Size	Median	GFlopsS
256	0.000289	116.12004
512	0.002245	119.573057
1024	0.011492	186.867477

32 FOOTPRINT

Matrix Size	Median	Speed Up	GFlopsS
256	0.000246	1.174796748	136.373535
512	0.001384	1.622109827	193.953472
1024	0.009499	1.209811559	226.07297





From the results you can see with the larger blocks of data each thread has to compute the faster the program is and the more GFlopsS it experiences. Like explained above this is due to the decrease in the total memory traffic due to the larger data blocks.

Conclusion for Cuda

After these experiments I saw first hand how important handling memory is in Cuda programming. By just changing the amount of data being read into shared memory in the matrix multiplier and the order of access in memory in the vector add we saw huge performance jumps in our experiments. This just proved that when coming to Cuda programming it is very important to think out how you are dealing with your memory whether that is reading from it or writing from it.