LU decomposition using GPU

Akash Desai
akashpra@buffalo.edu
31th March, 2015

1. Run

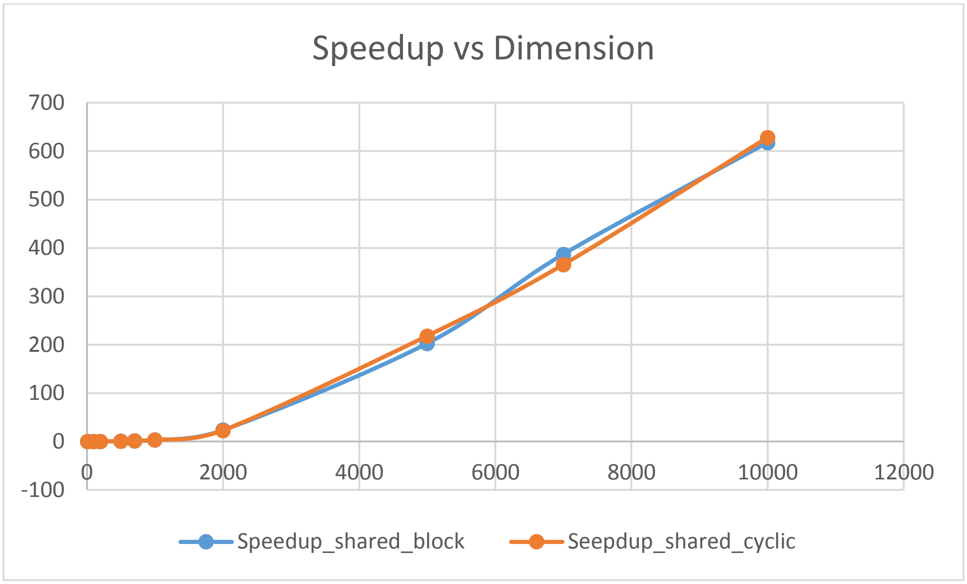
- In given zip folder, there is a folder called GPU.
- This time I have provided the shell script which we can run using sbatch command and passing 10 arguments as size of dimension.
- Basically this program will decompose the matrixes of different sizes provided by arguments.
- Verification method is under commented part: Part is highlighted as verification code, removing that comment will verify the matrix.
- Verifying matrix is very time consuming on server, so for that reason that has been commented for executing period. One can always remove the comments and verify the matrix.
- Example of shell file run: sbatch SLURM_MM_cuda_1.sh 10 100 200 500 700 1000 2000 5000 7000 10000
- All the result of given arguments will be stored in a '.csv' file, from which I have collected the data and plotted the graphs.

2. Code Implementation and Results

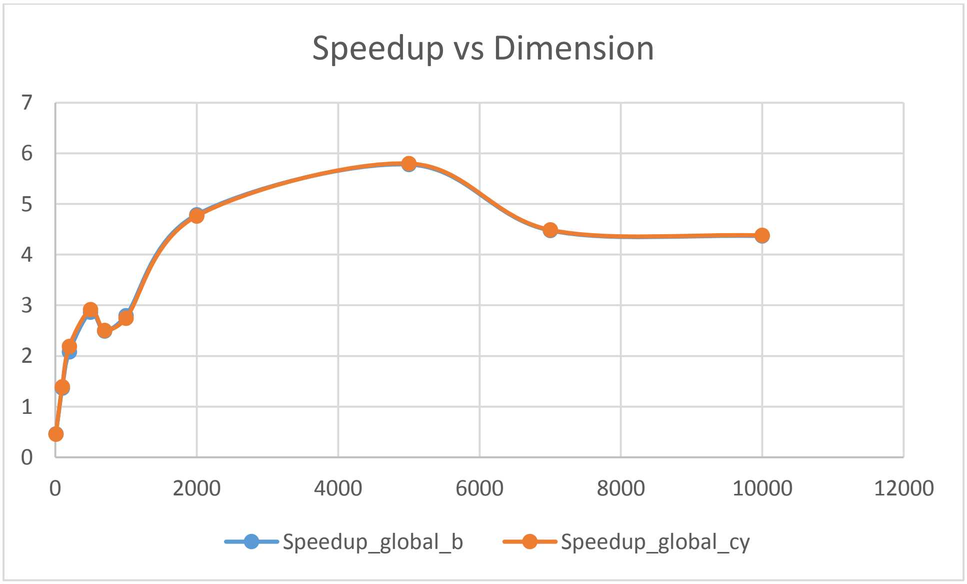
- My implementation of CUDA code for Gaussian decomposition is based on " Linear equation solver based on Gaussian elimination by Xinggao Xia and Jong Chul Lee.
- I have launched kernel only once, you can launch different kernels for each row and column operation separately but that is not an efficient way to do it. That will decrease your speedup by 10(divided by 10).
- In kernel I have used blockIdx and threadIdx to get temporary variable called lower part of decomposed array.
- While dimension is below 1000, you don't need to worry about the indexing, but once it crosses zero, you have to handle the case, since you can only have 1024

thread in a block so can't use pure threadIdx as column anymore. That modification has been done in kernel function called DUkernel.

- More over in shared memory implementation, I have assigned an array of dimension x dimension size in one dimensional array. U part of the original matrix is stored in the shared memory and after completing all the functions, we put back all the values back to original array for further verification.
- File containing all the results, is saved as, Results.xls in the folder given.



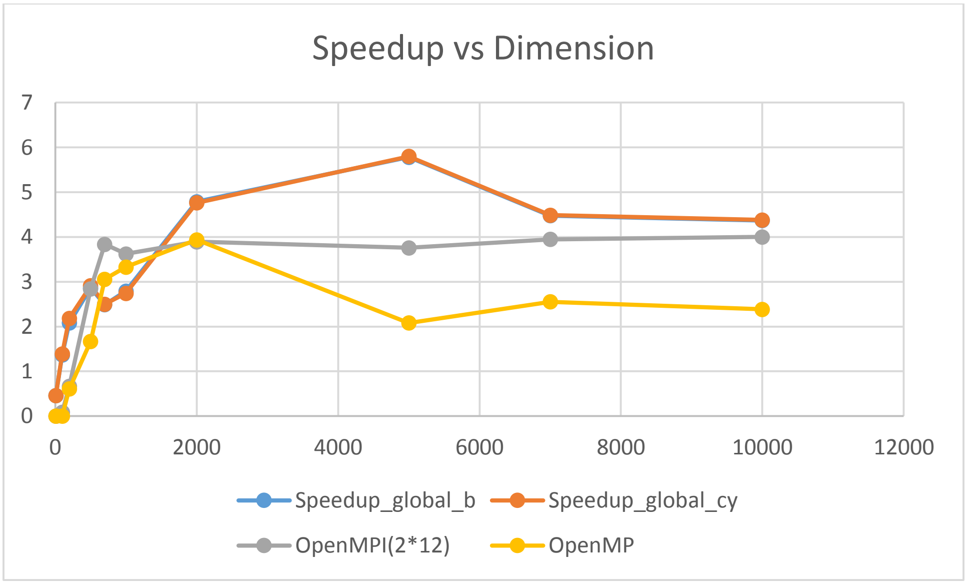
Shared Memory: SpeedUp vs Dimension



Global Memory: SpeedUp vs Dimension

- **a. 1. Increase in size of the problem:** As you see in the graph, with the increasing size, speed is increasing exponentially till 10000 dimension. Where are in global memory, you get a pick value at 5000 and after that it decreases a bit then, it will stabilize.
- Here in my implementation each thread is working on single data, so cyclic distribution and block distribution won't affect the result since you can't distribute more than one data to any thread. Distribution come in to action only when thread has to handle more than one data.
- Shared memory is limited, in CUDA 1.0, it is 16 kb , that suggests that this exponential rise will take a hit after the value of dimension that exhaust the memory.
- In gpu I am spawning 1000 threads per block, so essentially that many threads will work in parallel to get higher speedups.
- In global memory, it takes a lot of time to get access to global memory and make changes to that, so it basically won't give any high speedups compared to sequential or OpenMP and OpenMPI.

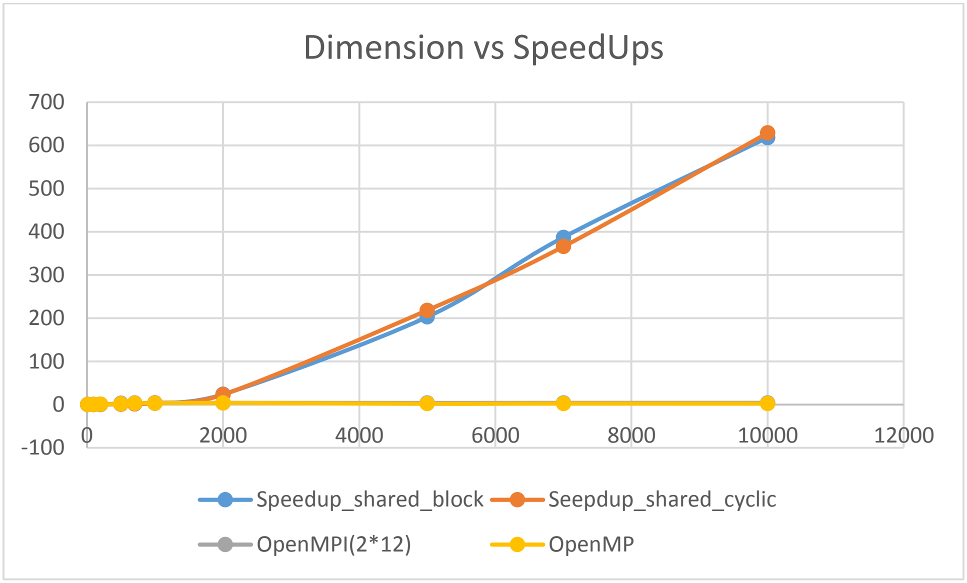
- **2.b. Comparison with OpenMP and OpenMPI:**



- This graph suggests the speedup difference between the results of OpenMP, OpenMPI and Global memory of gpu.
- You will see that though, global memory speedup is higher than OpenMP and OpenMPI, it is not a significant improvement on comparison of number of cores

used for the calculation. It is down to memory access time for the node to get the value of original matrix. That can be removed in shared memory that is why shared memory performs much better than Global memory.

- This is the graph comparing Shared memory gpu and OpenMP and OpenMPI



Comparison of OpenMP, OpenMPI and Shared Memory GPU

- Here you will notice that Shared memory's speedup results are incomparable with any other speed that you get.
- In my gpu implementation, I was able to achieve almost linear speedup, whereas in OpenMP and OpenMPI, my implementation couldn't get to anywhere near the linear speedup.
- From the graph shown above, you can compare the results. Moreover I have included all the OpenMP and OpenMPI files with the submission, including the codes and graph and the results for your reference.
- **2.c. Comparison between Cyclic and Block Distribution**

- Though I have included the folders for cyclic and block distribution, thr result would not change, since I have launched enough amount of thread to the kernel so each thread can work on the different data. In this manner no thread will have to deal with 2 pieces of data. So basically this will not encourage any kind of distribution since for distribution you need more than 1 data per thread so thread can choose whether it wants to execute the data in cyclic manner or in blocked fashion.

- For other implementation, which my peers have tried, might need a data distribution, so in their code, data distribution will matter, since their threads have more than 1 data to work on.

- My kernel launching method refrain me from doing so. Moreover from all the online resources available I have read, it suggested to launch threads in a manner that each thread has to deal with minimum amount of datas in order to get better speedups. Now we have enough blocks available to our disposal so it doesn’t make much sense to give more than 1 data to any thread.