Assignment - T-61.6030 Dynamical models for prediction and decision making

Rakshith Shetty (466945)
rakshith.shetty@aalto.fi

April 21, 2015

1 Platform

I used python programming language for all the tasks below. The python version used was 2.7.6. For efficient matrix manipulation I have bechmarked performance on two libraries, numpy and theano.

Numpy [1] is a scientific computation package for python which has efficient functions for matrix manipulations . It can also be linked to use openblas libraries and hence can many of core linear algebra operartions using these c libraries. The numpy version I used was '1.9.1.dev-90ae342' and was linked against openblas library.

Theano [2][3] is a python library which works on a slightly different principles. In theano you define your program as a series of mathematical expressions relating your input to output. Theano then builds a corresponding computational graph, optimizes it and then generates efficient C function which can be called using theano interface in python. All this happens during interpretation of python code hence there is a slight overhead. Another main strength of this library is that it can run the same python code on GPUs by generating code using cuda libraries [4]. For the user this involves just changing an option indicating theano to use the GPU. I have benchmarked my code on both GPU and CPU and we can see the performance gains we get when using the GPU.

I ran all my tests on gpu001 node on triton as I wanted to keep the cpu common when comparing performance against a gpu. It has a 64 bit Intel processor with 6 cores per cpu and is clocked at 2.66 Ghz. This node also has 23 GB of RAM. The gpu available on the system is a NVIDIA Tesla M2090 gpu with 5GB of memory.

All the time information reported is calculated within the python code and is only for the section doing the actual computation (i.e excluding the initializations). This corresponds to wall clock time but since I was the only user using the machine at the time it should be quite close to best possible running times for these implementations.

2 Dense Matrix multiplication

The basic python implementation was done using lists and is quite slow. I didn't run the native implementation for N=10000 as this was taking too long (I estimate around 40k seconds and have used this in the plot). All the running times for this task are shown in Table 1. This data is also plotted in figure 1. Figure 2, shows the speedups obtained using these libraries taking the native python implementation as the baseline. As we can see running on GPU using theano library is fastest for large matrix sizes, outperforming the Numpy and Theano cpu versions by a factor of 8. But for smaller matrix sizes the gpu implementation is a bit slower. This due to the quite significant overhead involved in moving the data from the cpu memory to the gpu memory and back.

The tests broke at N = 10^5 with memory error as it would require about 37GB of memory to store the transition matrix of size $10^5 \times 10^5$, assuming 32 bit float numbers.

N	Native Python	Numpy	Theano CPU	Theano GPU
1	0.001525	0.001925	0.00384	0.081004
10	0.024394	0.002326	0.003777	0.082799
100	1.591991	0.003915	0.005589	0.083527
1000	205.644113	0.171891	0.174472	0.11776
10000	-	35.254669	34.945581	4.077473

Table 1: Running times in seconds for different libraries for dense matrix multiplication

3 Sparse Matrix Multiplication

For sparse matrix multiplication, I assumed each row in the matrix has only few elements non-zeros, specified exactly by sparsity S. I store only these non-zero elements and their indices for each row and multiplication is also carried out only for these elements in the native implementaion. Both Numpy and Theano libraries have support for storing and manipulating sparse matrices. Because of sparsity we can now run systems of much larger sizes and I could go upto systems of size 10^6.

Figure 3 shows the running times of these implementaions for varying sparsity for a fixed system size of 10^4. We can see here that numpy version performs the best although theano on cpu is quite close. It is interesting to note that GPU version is slower that both numpy and theano cpu version but gets quite close at larger sparsity value S. This is again probably due to the overhead in gpu data transfers.

Figure 4 shows the running times for fixed sparsity of 8 with increasing system sizes. We can see that the running time scales linearly (after initial part) with N and again numpy version is the best by a small margin.

References

[1] http://www.numpy.org/

[2] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley and Y. Bengio. "Theano: A CPU and GPU Math Expression Compiler". Proceedings of the Python for Scientific Computing Conference (SciPy) 2010. June 30 - July 3, Austin, TX

[3] http://deeplearning.net/software/theano/

[4] http://deeplearning.net/software/theano/tutorial/using_gpu.html

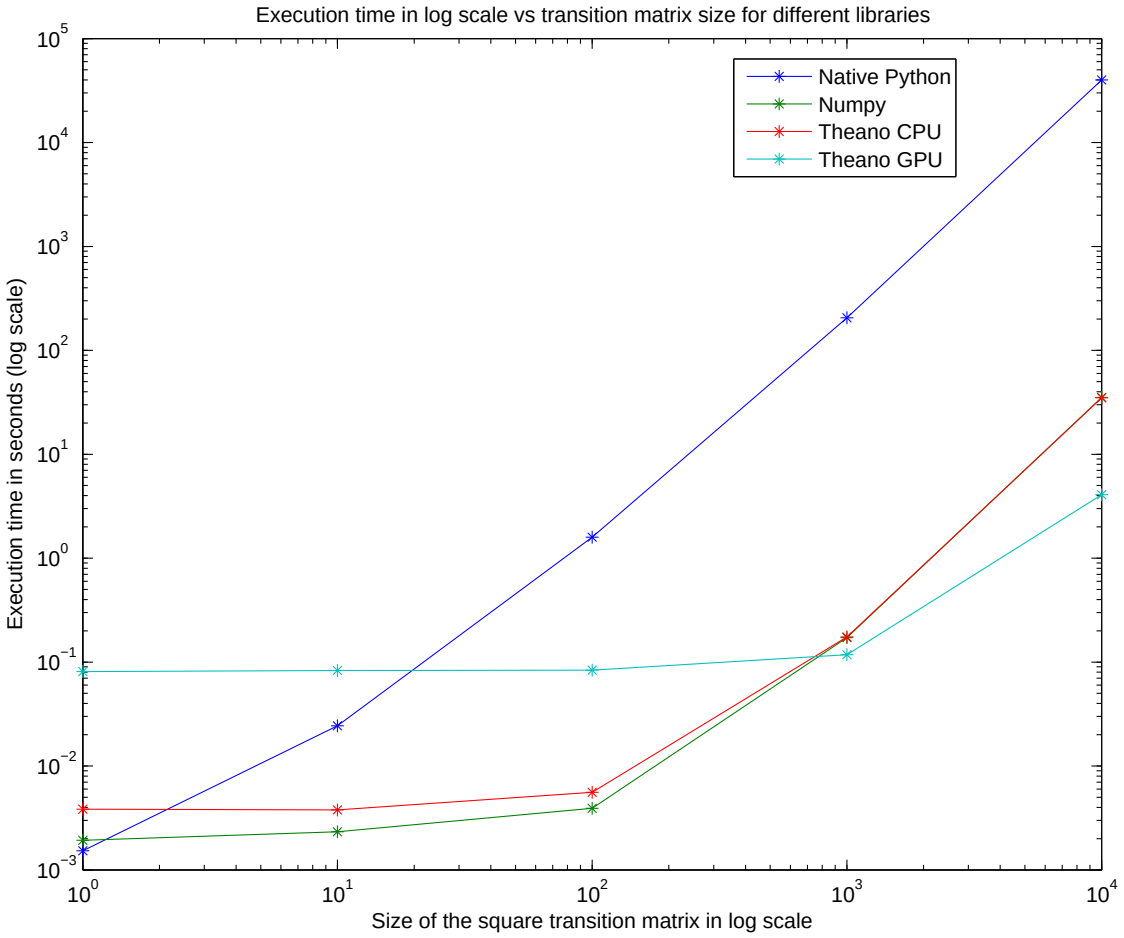

Figure 1: Running times against system size for various libraries

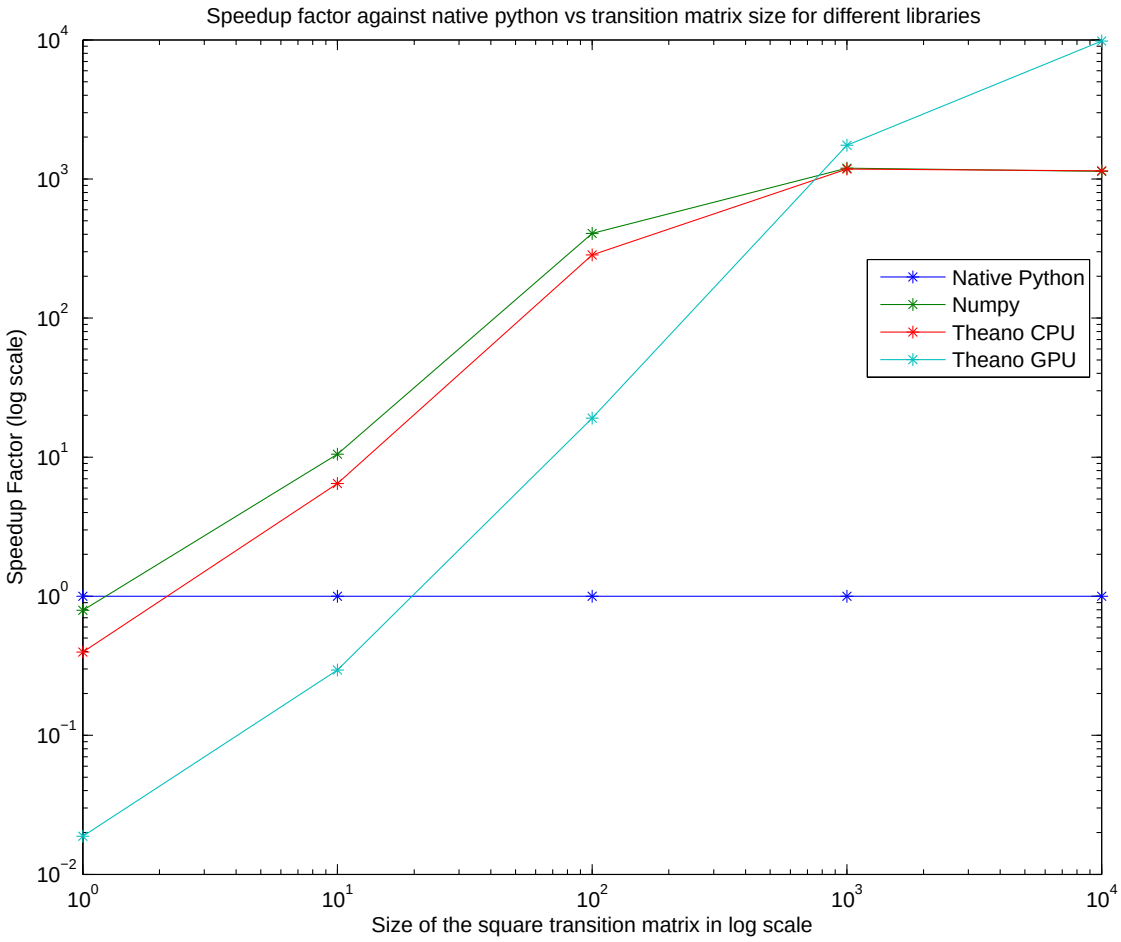

Figure 2: Speedup factor against the native python implementation

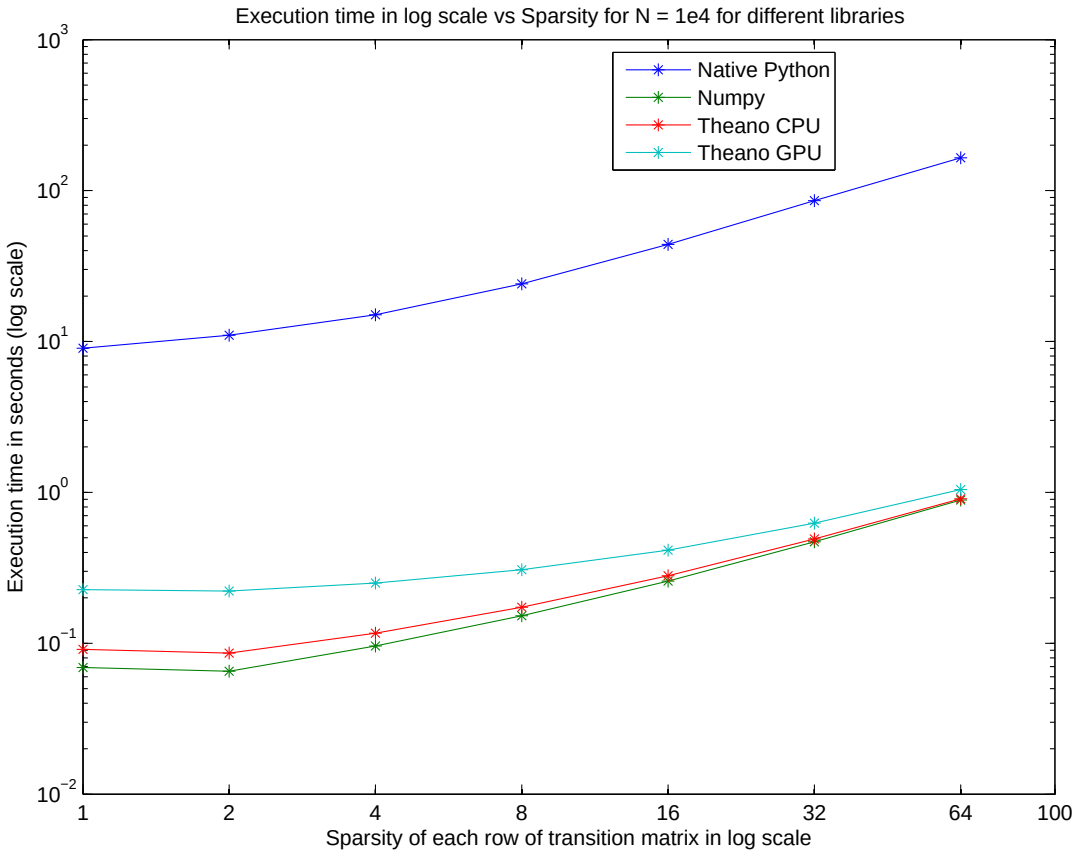

Figure 3: Running times against sparsity S, for system size N =10000 for various libraries

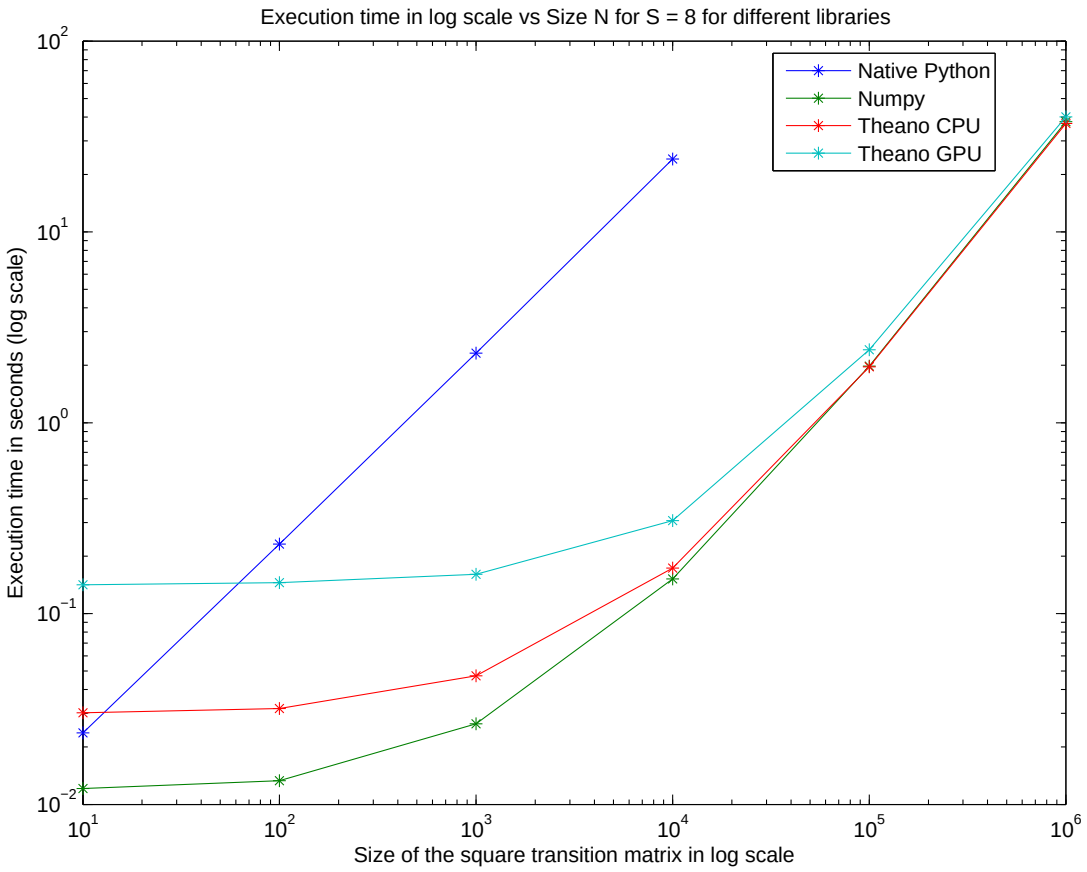

Figure 4: Running times against system size N, for sparsity S = 8 for various libraries