A Parallel Implementation of Smolyak Method

Iskander Karibzhanov

Summary

- Evaluating large polynomials on a large grid size is the main challenge when utilizing projection methods. However, this challenge can be easily overcome by parallizing the projection algorithm using Graphical Processing Unit (GPU).
- In this note, I summarize my results of parallelizing the "Smolyak" algorithm, a popular projection method. I illustrate the practical application of my method by solving the ten region international real business cycle model with twenty states (Malin, 2011).
- My algorithm improves performance in double precision up to 400 times compared with serial MATLAB implementation in Judd, K. L., Maliar, L., Maliar, S., & Valero, R. (2014). The third level of approximation finishes in 6 minutes on Nvidia Tesla V100 GPU rather than 41 hours on Intel CPU. This proves that the algorithm can be efficiently parallelized on the GPU and makes third-level approximation computationally feasible which was not possible before.

Introduction

The Smolyak method is one of the promising lines of research that tackles the curse of dimensionality, the biggest challenge of projection methods. It was Invented by Russian mathematician Sergey Smolyak in 1963. This method is a sparse tensor product construction of Chebyshev polynomials and their extrema. It solves high-dimensional problems avoiding curse of dimensionality (hence it is useful for smooth functions only). Krueger and Kubler (2004) and Malin (2011) were first to introduce the Smolyak method in economics which allowed them to study far larger problems than other methods. However, they showed that its computational burden is prohibitive in very high dimensional problems. Judd et al. (2014) (henceforth, JMMV) have proposed a more efficient implementation based on the combination of anisotropic grids and Lagrange interpolation. This method is of high arithmetic intensity (compute-bound) and is still costly for high-order approximation. In this note, I show how to parallelize Smolyak method using GPUs and significantly improve its computational efficiency compared to JMMV.[1]

[1] For a survey on parallel computing and its applications in economics using GPU architectures see Aldrich (2014), Brumm (2015), Brumm (2015), Hatcher (2015), Scheffel (2015), Aldrich (2011), Morozov (2011), Lee (2007), Bruno (2012), Cai (2014).

Projection methods try to approximate a function (policy function, value function, etc.) on *d* state variables using the polynomial f:$[-1,1]^d \rightarrow R$. Instead of taking the tensor product of basis polynomials and grid points, the Smolyak method allows us to select a subset that is most important for the quality of approximation. Due to judicious choice of points, the number of elements in this subset does not explode with number of dimensions *d*. By choosing a smaller level of approximation along less-important dimensions (e.g. exogenous shocks) it is possible to significantly reduce the number of grid points and basis functions without compromising the accuracy.

I implemented two versions of the Smolyak method: a the serial version as a MATLAB function and a parallel version as a Compute Unified Driver Architecture (CUDA) kernel function. The code can be found on my GitHub page: https://github.com/ikarib/smolyak

Sequential Implementation (smolyak.m)

My algorithm builds on Judd, Maliar, Maliar and Valero (2014). It combines their four MATLAB functions (Smolyak_Elem_Isotrop.m, Smolyak_Elem_Anisotrop.m, Smolyak_Grid.m and Smolyak_Polynomial.m) into one function (smolyak.m). It is fully vectorized and runs much faster than JMMV especially for smaller models.

The original JMMV multi country code spent 90% of the time just in one function called `Smolyak_Polynomial.m` which evaluates the following Smolyak polynomial at state grid vector $x_k \in R^D$:

$$y_{n,k} = \sum_{i=1}^{M} c_{n,i} \prod_{j=1}^{\mu} \phi_{S_{i,j}}(x_k) \quad \text{for all} \quad k = 1, \dots, L \quad \text{and} \quad n = 1, \dots, N,$$

where L is the number of grid points, *D* is the number of states, *N* is the number of countries, μ is the level of approximation and M is the number of polynomial terms, $c_{n,i}$ are Smolyak coefficients and $\phi_{S_{i,j}}$ are Smolyak basis functions indexed by Smolyak indicies $S_{i,j}$.

After optimization and vectorization of the original JMMV codes, the run time decreased several times with 99.5% of the runtime now spent at the interpolation step in constructing the Smolyak basis matrix and multipliying it by the Smolyak coefficient vector:

```
B=phi(S(:,1),:);
for i=2:mu
    B=B.*phi(S(:,i),:);
end
y=c*B;
```

For example in IRBC model with N=10 countries, D=20 states and level of approximation $\mu = 3$, the grid size becomes large (L=231,220 and M=11,561) so the parallelization can be done over columns of B matrix independently. This calculation problem is embarassinly parallel over L grid points/integration nodes.

Parallel Implementation (smolyak_kernel.cu)

The parallel implementation of the Smolyak algorithm works in MATLAB by invoking a precompiled CUDA kernel function as a Parallel Thread Execution (PTX) assembly for NVidia GPUs. This approach allows us to keep using the rest of our existing MATLAB codes without having to translate them into the C language.

To implement the algorithm in CUDA, I followed the following four steps:

1. Write the the GPU CUDA kernel code as CU code (based on C programming language),
2. Manually compile the source CU code into PTX code using nVidia's NVCC compiler to produce the assembly-level file smolyak_kernel.ptx (.ptx stands for "Parallel Thread eXecution language"),
3. Use MATLAB function from parallel computing toolbox called parallel.gpu.CUDAKernel to create the CUDA kernel object in MATLAB workspace,
4. Use the kernel object to call the CUDA kernel using feval function. The inputs can be constants or variables in MATLAB workspace, the output will be gpuArray. It is more efficient to use gpuArrays for large input matrices to avoid repetitive memory copy from host to device.

This approach offers several computational advantages compared to JMMV. First, this implementation constructs the anisotropic subset directly, which is computationally more efficient when many anisotropic dimensions (e.g. productivity levels) require lower levels of approximation ($\mu=2$) than more important dimensions (e.g. capital stocks) where more accuracy is needed ($\mu=3$).[2] As a result, three separate JMMV functions (`Smolyak_Elem_Isotrop.m`, `Smolyak_Elem_Anisotrop.m` and `Smolyak_Grid.m`) that constructed the anisotropic

[2] In JMMV, they first construct an isotropic Smolyak interpolant, and then remove the terms accordingly to arrive to the target anisotropic construction. This procedure requires additional running time which reduces the savings from anisotropy.

grid were merged conveniently into one function `smolyak.m`. The new function performs both the construction of the Smolyak indices and the evaluation of the Chebyshev polynomials. The sparse storage of the Smolyak indices not only avoids warp divergent branches inside the main loop but also allows these indices to be stored in a fast read-only constant memory.[3]

I tested the relative performance of the parallel implementation of the Smolyak method using the multi-country model of Malin (2011). Figure 1 and Table 1 show speedup of solving the model with ten countries and twenty state variables on GPU vs CPU. Two different GPUs were tested separately: single Nvidia Tesla K80 (available on the Bank of Canada Edith cluster) and single Nvidia Tesla V100 (now available on Microsoft Azure NCv3-series Virtual Machine in Canada Central region). The serial CPU version of the code was tested in multi-threaded MATLAB on Microsoft Azure Fsv2-series Virtual Machine based on the newest 36-core 2.7 GHz (3.7 GHz turbo) Intel Xeon Platinum 8168 processor.

Figure 1. Multi-country performance test: Nvidia Tesla K80 and Tesla V100 GPUs

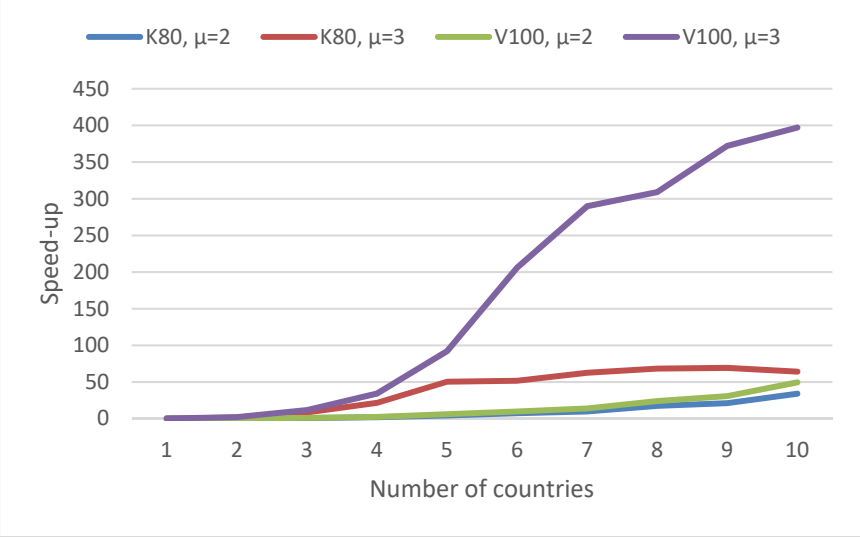

Table 1. GPU speed-up in 10-country model at 3rd level of approximation

	Intel Xeon CPU	Tesla K80 GPU	Tesla V100 GPU
Total runtime	41 hours	39 minutes	6 minutes
Speed-up vs CPU		x 64	x 397

When assessing the relative performance of different implementations, It is also important to choose the correct performance metric. Since the Smolyak algorithm has a high arithmetic

[3] The ability to store the indices was not previously possible because the full matrix of Smolak indices occupied more than the 64 KB of constant memory available on the GPU.

intensity (compute-bound), peak performance should be assessed in GFLOP/s. For every of M*J elements loaded, the kernel theoretically performs (peak) $N(2M-1)+M(\mu_{max}-1)$ flops (floating point operations per second). In case of $N=10$ and $\mu=3$, the kernel achieves peak calculation rates of 198 Gflops on Nvidia Tesla K80 and 1.8 Tflops on Nvidia Tesla V100.

Since the proposed implementation's Smolyak kernel uses local memory to store an array of basis functions ϕ and does not offload data to shared memory, the larger the number of countries and the higher the level of approximation, the more registers are allocated per thread. This register pressure lowers the occupancy. The number of registers varies from 36 to 78 and always exceeds 32 (the maximum number for full 100% occupancy). Therefore the maximum theoretical occupancy of my kernel varies from 75% down to 50%. In order to achieve the highest possible occupancy of 75%, a block size of 128 threads is chosen as it does not depend on the number of registers per thread.[4] As a result, this block size is optimal for any number of countries in the model and any level of Smolyak approximation

Further efficiencies are found by saving the precomputed inverse-of-basis-functions matrix on the GPU and solving the linear system on the GPU using simple matrix multiplication. Since the large basis functions matrix is no longer needed in the GPU memory, one can discard the previous term of the basis polynomial once it is multiplied by the interpolation coefficient matrix. Input matrices are read through texture memory (via the use of CUDA __restrict__ attributes).

Using Multiple GPUs in a Cluster

The Bank of Canada has a cluster with two Nvidia Tesla K80 GPUs available. Each of these GPUs has two more chips featuring 2496 CUDA cores with a double precision compute performance of 1.45 teraflops. To test Smolyak algorithm on four GPUs, I use spmd command to open one worker for each GPU. Unfortunately I found that there is a performance penalty since we have to collect and concatenate the results from each worker in every iteration. Since the program requires over 7000 iterations to converge, the scaling to multiple GPUs is poor.

[4] This optimal block size was found using CUDA occupancy calculator excel spreadsheet available at http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Conclusion

This paper demonstrates the efficiency gains of leveraging GPUs to solve high-dimensional nonlinear DSGE models in parallel using the Smolyak Algorithm. Recent contributions in solution methods for nonlinear DSGE models have highlighted the Smolyak method for its accuracy and speed. However, computational costs of existing algorithms are still too restrictive for practitioners interested in large models with multiple state-variables, such as multi-country international business cycle models. Applied to a large international business cycle model, The parallelized implementation proposed in this note reduces the running time of solving the multi-country model by up to 400 times on the newest Nvidia Tesla V100 GPU compared to the serial implementation of Judd, Malia, Maliar, and Valero (2014).

Bibliography

Aldrich, E. M. (2014). GPU Computing in Economics. *Handbook of Computational Economics*, 3, 557--598. doi 10.1016/b978-0-444-52980-0.00010-4

Aldrich, E. M., Fernández-Villaverde, J., Ronald Gallant, A., & Rubio-Ramírez, J. F. (2011). Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors. *Journal of Economic Dynamics and Control*, 35(3), 386--393. doi 10.1016/j.jedc.2010.10.001

Bruno, G. (2012). Metropolis-Hastings MCMC goes parallel on a GPU: first experiences and results. *Unpublished manuscript*. Retrieved from **giuseppe.bruno@bancaditalia.it**

Brumm, J., Mikushin, D., Scheidegger, S., & Schenk, O. (2015). Scalable high-dimensional dynamic stochastic economic modeling. *Journal of Computational Science*, 11, 12--25. 10.1016/j.jocs.2015.07.004

Brumm, J., & Scheidegger, S. (2015). Using Adaptive Sparse Grids to Solve High-Dimensional Dynamic Models. *SSRN Journal*. doi 10.2139/ssrn.2349281

Cai, Y., Judd, K. L., Thain, G., & Wright, S. J. (2014). Solving Dynamic Programming Problems on a Computational Grid. *Computational Economics*, 45(2), 261--284. doi 10.1007/s10614-014-9419-x

Fernández-Villaverde, J., & Levintal, O. (2016). Solution Methods for Models with Rare Disasters. doi 10.3386/w21997

Judd, K. L., Maliar, L., Maliar, S., & Valero, R. (2014). Smolyak method for solving dynamic economic models: Lagrange interpolation, anisotropic grid and adaptive domain. \emph{Journal of Economic Dynamics and Control}, 44, 92--23. doi 10.1016/j.jedc.2014.03.003

Hatcher, M. C., & Scheffel, E. M. (2015). Solving the Incomplete Markets Model in Parallel Using GPU Computing and the Krusell-Smith Algorithm. Computational Economics, 1--23. doi 10.1007/s10614-015-9537-0

Krueger, D., & Kubler, F. (2004). Computing equilibrium in OLG models with stochastic production. Journal of Economic Dynamics and Control, 28(7), 1411-–1436. doi 10.1016/s0165-1889(03)00111-8

Lee, D., & Wiswall, M. (2007). A Parallel Implementation of the Simplex Function Minimization Routine. *Computational Economics*, 30(2), 171--187. doi 10.1007/s10614-007-9094-2

Malin, B. A., Krueger, D., & Kubler, F. (2011). Solving the multi-country real business cycle model using a Smolyak-collocation method. *Journal of Economic Dynamics and Control*, 35(2), 229--239. doi 10.1016/j.jedc.2010.09.015

Morozov, S., & Mathur, S. (2011). Massively Parallel Computation Using Graphics Processors with Application to Optimal Experimentation in Dynamic Control. *Computational Economics*, 40(2), 151--182. doi 10.1007/s10614-011-9297-4

Scheffel, E. M. (2015). All Together Now! A survey of the GPGPU parallel paradigm in Economics. *Unpublished manuscript*. Retrieved from http://www.ericscheffel.com