A Parallel Implementation of Smolyak Method

Iskander Karibzhanov

October 21, 2016

Abstract

In this project, I show how to parallelize popular projection method called Smolyak algorithm involving sparse grids. The main hotspot in projection methods is the evaluation of a large polynomial on a large grid size. Fortunately, this problem turns out to be embarrassingly parallel. My program works in MATLAB by invoking a precompiled CUDA (Compute Unified Driver Architecture) kernel function as PTX (parallel thread execution) assembly for NVidia graphical processing units. This allows users to use their existing MATLAB codes without having to translate them into C language. I illustrate the practical application of my method by solving the same international real business cycle model with multiple countries. My algorithm improves performance in double precision by up to 66 times compared with serial implementation in Judd, Maliar, Maliar, and Valero's Smolyak toolbox also written in MATLAB. For example, ten country model with twenty states can be now solved with the third level of approximation in 1 hour and 3 minutes on Tesla K20c NVIDIA GPU rather than 70 hours on Intel CPU. The code is available at `https://github.com/ikarib/smolyak`

Introduction

The Smolyak method is one of the promising lines of research that tackles the biggest challenge of projection methods: curse of dimensionality. Krueger and Kubler (2004) and Malin et al. (2011) were first to introduce the Smolyak method in economics which allowed them to study far larger problems than other methods, but they showed that its computational burden is still rather expensive in very high dimensional problems. Judd et al. (2014) (henceforth, JMMV) have proposed a more efficient implementation based on anisotropic grids and Lagrange interpolation. In this paper, I will show how to parallelize JMMV method and improve its efficiency even further.

We consider the polynomial approximation of a function (policy function, value function, etc.) on d state variables $f : [-1, 1]^d \to \mathbb{R}$. 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 smaller approximation level μ 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 version of Smolyak method: the serial version as `smolyak.m` MATLAB function and parallel version as `smolyak_kernel.cu` file with CUDA kernel function.

1 Sequential Implementation (smolyak.m)

My implementation differs from JMMV in the construction of the anisotropic Smolyak grid. 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. Instead, my procedure avoids creation of the isotropic grid and constructs the anisotropic subset directly. This 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 ($\mu = 3$).

Three separate JMMV functions (`Smolyak_Elem_Isotrop.m`, `Smolyak_Elem_Anisotrop.m` and `Smolyak_Grid.m`) that together constructed anisotropic grid were merged conveniently into one function `smolyak.m` which implements both construction of Smolyak indices and evaluation of the Chebyshev polynomials. The original JMMV multi country code spent 90% of the time in function called `Smolyak_Polynomial.m` which made me think that this is an excellent candidate for parallelization. After vectorization the running time decreased several times less with 99.5% of the time spent in interpolation of Smolyak polynomial. The serial version achieves peak calculation rates of 1.7 Gflops which is far from the 60 Gflop benchmark score on Intel Xeon CPU E5-2690 8-core 2.9GHz for double precision multiplication of matrices of the same size. The crux of the runtime is spent at the following line of code:

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

Note that I was able to eliminate the costly "if" condition inside the above "for" loop by compactly storing Smolyak indices in `S` matrix with `mu_max` columns (compared with `D` columns in JMMV).

2 Parallel Implementation (smolyak_kernel.cu)

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

I parallelize the algorithm efficiently on GPU so that the third-level approximation is no longer computationally infeasible. I compare the performance of my parallel implementation of the Smolyak method using the multi-country model of Malin et al. (2011). As we can see from the table below, the second level approximation of the model with ten countries and twenty state variables using my CUDA kernel runs in just 56 seconds. For comparison, a serial JMMV code solves the same model in about 58 minutes. For the third level of polynomial approximation, the running time decreases from 70 hours on CPU to 1 hour on GPU.

It is important to choose correct performance metric. Smolyak algorithm has high arithmetic intensity (compute-bound), therefore we should strive for peak performance in GFLOP/s. For every element loaded, kernel performs theoretically (peak) $N(2M-1)+M(\mu_{max}-1)$ flops (floating point operations per second). The kernel operations have the same throughput for both CUDA compute capabilities 3.0 and 3.5. Effectively, the kernel achieves peak calculation rates of 156 Gflops for case $N=10$, $\mu=3$ which is not far from the 200 Gflops benchmark score on NVidia Tesla K20c for double precision multiplication of matrices of the same size. It is still far though from the theoretical limit of peak performance of 1.17 Tflops in double precision. Moreover, the profile kernel and host running times shows that kernel takes 88% of the total running time which means that matrix-matrix multiply is not optimized. My plan for the next step in code optimization is to replace use of textures by shared memory.

Since my Smolyak kernel uses local memory to store 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 is lower: from 75% down to 50%. In order to achieve such occupancy, I chose the block size to be 128 threads so that this number does not depend on the number of registers per thread and hence this block size is optimal for any number of countries in the model and any level of Smolyak approximation. 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`.

Due to sparse storage of the Smolyak indices, I was able to avoids warp divergence by eliminating divergent branches caused by "if" condition in the CUDA kernel code. Using sparse Smolyak indices also allowed me to store them in extremely fast, read only constant memory. This was not possible before because full matrix occupied more than 64 KB of available constant memory on GPU.

Instead of saving the large `B` matrix in global GPU memory, the kernel multiplies the its rows by coefficient matrix `c` which has very few columns. Depending on size, this input coefficient matrix can be either stored in constant or texture memory which provides fast and read only access.

Instead of solving the large linear system in each iteration, I multiply the precomputed inverse of basis functions matrix `B_inv` stored on the GPU.

The following table shows my runtime results. As you can see even my serial implementation of Smolyak method is much faster than the JMMV code.

N	JMMV		My serial		My parallel	
	$\mu=2$	$\mu=3$	$\mu=2$	$\mu=3$	$\mu=2$	$\mu=3$
2	23	71	3	5	4	5
4	143	1562	8	442	6	18
6	530	12278	74	6062	11	155
8	1670	70643	353	42958	27	820
10	3474	250420	964	186460	56	3785

References

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

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

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

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

Brumm, J., Mikushin, D., Scheidegger, S., & Schenk, O. (2015). Scalable high-dimensional dynamic stochastic economic modeling. *Journal of Computational Science*, 11, 12–25. doi: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

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

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

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

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

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

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

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