Solving the incomplete markets model in parallel using GPU computing and the Krusell-Smith algorithm

Michael C. Hatcher*
&
Eric M. Scheffel†

November 27, 2014

Abstract

This paper demonstrates the potential of graphics processing units (GPUs) in solving the incomplete markets model in parallel using the Krusell-Smith algorithm. We illustrate the power of this approach using the same exercise as in Den Haan et al (J Econ Dyn Control 34: 1-3, 2010). We document a speed gain which increases sharply with the number of agents. To reduce entry barriers, we explain our methodology and provide some example algorithms.

Keywords: GPU computing, heterogeneous agents, incomplete markets, interpolation, Krusell-Smith algorithm

JEL Codes: C6, C63, D52

1 Introduction

Models with incomplete markets and heterogeneous agents are used widely in macroeconomics. It is therefore important that researchers be able to solve these models quickly. This is a non-trivial problem since the set of state variables includes the cross-sectional distribution of wealth – an infinite-dimensional object. Krusell and Smith (1998) show that this problem can be circumvented by approximating the cross-sectional distribution of wealth with a small number of moments. This reduces the dimension of the state vector dramatically, making numerical simulations of incomplete market models tractable. Nevertheless, the Krusell-Smith algorithm is quite time-consuming, especially if the optimal

*Department of Economics, Faculty of Social and Human Sciences, University of Southampton, Southampton, SO17 1BJ. Email: m.c.hatcher@soton.ac.uk. Tel:

†Corresponding author. Nottingham University Business School China. 199 Taikang East Rd. Office AB478. 315100 Ningbo. People's Republic of China. Email: eric.scheffel@nottingham.edu.cn. Tel: +86(0)574 8818 2461

decisions of a large number of agents are computed sequentially or using only a few processing cores in parallel.

In this paper, we demonstrate the potential of graphics processing units (GPUs) in solving incomplete market models with heterogeneous agents and aggregate uncertainty using the Krusell-Smith algorithm. We rely on the compute unified device architecture (CUDA) of NVIDIA and show that using the GPU delivers a speed gain over the central processing unit (CPU) which rises sharply as the number of agents is increased. In particular, we document speed gains in the panel simulation stage of the Krusell-Smith algorithm of between 40 and 4000 times as the number of agents is increased from a relatively small number such as 10,000 to very large, but plausible, numbers such as 10 million agents.[1]

As discussed by Aldrich (2014), GPUs are relatively inexpensive pieces of hardware comprised of large numbers of individual processing cores capable of parallelization. This makes them ideal for computational work that has a high arithmetic intensity, that is, work which requires large numbers of computational operations which are almost identical and can be computed independently of one another. The Krusell-Smith algorithm fits this description because the optimal capital choice must be computed for each agent conditional on the mean of the wealth distribution, current capital holdings, the aggregate state of the economy, and idiosyncratic shocks. As this algorithm has been used widely in the heterogeneous-agent literature, our findings should be of use to other researchers. Our paper also contributes to a recent strand of literature that documents the potential of GPU computing in solving dynamic general equilibrium models in economics.

The seminal paper in that literature is Aldrich et al. (2011). They show that improvements in speed of up to 200 times are possible when solving a simple real business cycle model using value function iteration and CUDA architecture. Subsequently, Morozov and Mathur (2012) tackled a more complicated optimal control problem using CUDA. They consider imperfect information dynamic programming with a learning versus experimentation trade-off, so that the value function need not be convex and the policy function need not be continuous. For this problem, speed gains are 15-26 times are reported. Since GPU hardware has developed rapidly over the past few years, even larger gains should be well within reach.

A detailed survey of the current state of GPU computing in economics can be found in Aldrich (2014). He simulates an exchange economy with complete markets and agents with heterogeneous beliefs and documents speed gains of more than 1,000 times. Our paper goes beyond this because we consider an economy with incomplete markets and idiosyncratic shocks. In addition, our analysis differs from the early GPU literature on optimal control because we apply CUDA to the panel simulation stage of the Krusell-Smith algorithm.[2]

[1] Our results are compared against a central processing unit (CPU) where only a single core was utilized. Utilizing all four (or more if available) hardware cores would reduce the quantitative gains that we report, but our qualitative conclusions are unaffected.

[2] In other words, we do not apply CUDA to the Euler equation iteration stage. We choose

Consequently, we isolate the gains from CUDA in a context in which it has not previously been applied. We also investigate the computation time versus accuracy trade-off that arises with GPUs due to the choice between single- and double-precision arithmetic. Here, we find that single-precision arithmetic is roughly twice as fast as double-precision arithmetic but produces similar numerical results. Given that substantial speed gains are available in both cases, these results demonstrate the potential of GPU computing to make the trade-off between speed and accuracy somewhat less severe. Table **3** reports the performance of GPU and CPU used in this study.

The gains in computation time that we document are important for three reasons. First, solving even simple incomplete market models is quite time-consuming. Since this is likely to be a barrier to researchers entering the literature, it is important to lower solution times where possible. Second, non-trivial reductions in computation time would make it feasible to simulate richer economic models that include a larger state space, a task which might otherwise be considered prohibitively time-consuming by many researchers. Finally, improvements in computation speed would enable researchers to focus more effort on improving accuracy safe in the knowledge that total computation time could be kept relatively low. As pointed out by Den Haan (2010, p. 5), it is desirable to improve the accuracy of current algorithms for solving heterogeneous agent models by at least an order of magnitude.

Our paper is related to several computational papers in the heterogeneous-agent literature. Most directly, we solve the model in Krusell and Smith (1998) using their simulation-based methodology. They consider an incomplete markets version of the neo-classical growth model with a lower bound on capital holdings, aggregate productivity shocks, and idiosyncratic employment shocks. Their algorithm has the advantage that it is simple and relatively easy to program. In recent years, several alternative solution algorithms have been developed. These solution methods and their performance are documented in a 2010 special issue of the Journal of Economic Dynamics and Control. In that project entitled 'Computational Suite of Models with Heterogeneous Agents' a version of the Krusell-Smith model with unemployment benefits and 10,000 agents was solved using several different algorithms. As discussed by Den Haan (2010), the Krusell-Smith algorithm does relatively well in terms of accuracy, but it is not one of the fastest solution methods. For example, the backward induction algorithm of Reiter (2010) solves the model in less than 1 hour, while the explicit aggregation algorithm in Den Haan and Rendahl (2010) takes less than 10 minutes.[3] By comparison, the Krusell-Smith algorithm takes over 5 hours. It would therefore be desirable to speed up this method. We show how this can be done using CUDA, and we document very large speed gains as the cross-section of agents is increased to large but plausible numbers as 10 million.

not to because the value function iteration results reported in Aldrich et al. (2011) suggest that significant gains from GPU simulation would not be available for relatively sparse grids we.

[3]These alternative algorithms are, however, not as general and flexible as the Krusell-Smith algorithm, so it may not always be feasible for researchers to adopt these faster approaches.

Our work is also closely related to the Maliar et al. (2010) paper in the Computational Suite Project. They solve the model using the original Krusell-Smith algorithm by employing a consumption Euler equation method that iterates on a grid of pre-specified points. In a second stage they compute the aggregate law of motion as in Krusell and Smith (1998), that is, by simulating a panel for capital holdings and running regressions on the simulated data. Here, we follow the same approach, except that we use CUDA architecture to speed up the panel simulation stage.[4]

The paper proceeds as follows. Section 2 briefly sets out the model, introduces parallel computation on GPUs and describes our approach of exploiting this hardware in the context of the Krusell-Smith model. Section 3 reports a time comparison between GPU and CPU and discusses solution accuracy. Finally, Section 4 concludes.

2 Parallelizing the Krusell-Smith algorithm using GPU computing

This section presents an in-depth description of how we implemented our GPU parallelization of the Krusell-Smith algorithm using NVIDIA CUDA. It also includes a brief description of the specific CUDA language and GPU hardware features we exploited along the way, including any hardware and software limitations we had to code around in order to arrive at an efficient solution. Wherever possible, we followed the established best practices in GPU programming; see e.g. NVIDIA's supporting hardware documentation.

2.1 The Krusell-Smith model and algorithm

The basic model we consider was first solved by Krusell and Smith (1998), but we work with the version of the model set out in Den Haan et al. (2010). The model is a production economy with aggregate productivity shocks. Agents have different employment histories and can partially insure themselves by investing in capital. In addition, agents face a borrowing constraint which prevents them from taking short positions in capital. Since capital is the only asset available, markets are incomplete. In each period, agents face an idiosyncratic shock that can take on two different values: employed and unemployed. An employed agent receives an after-tax economy-wide wage rate; an unemployed agent receives unemployment benefits equal to a fraction of the economy-wide wage rate. Investment in capital yields a return equal to the marginal product of capital minus the depreciation rate. Both the economy-wide wage rate and the

[4] Our implementation of the individual policy function solution stage is almost identical to Maliar et al. (2010)'s variant coded in Matlab. It differs only in that it is written and compiled in C++, uses multi-threading on the CPU throughout, and employs a suite of industry-tested Fortran routines FITPACK for interpolating the policy function in each iteration. During the panel simulation, we instead rely on Catmull-Rom bi-cubic interpolation for both the GPU and CPU variants of our code.

return on capital depend upon the aggregate productivity shock. All agents in the economy have identical preferences and both aggregate and idiosyncratic shocks follow Markov processes.

The aim of the Krusell-Smith algorithm is to compute an aggregate law of motion (ALM) for economy-wide physical capital with which individual agents' optimal capital choices are consistent at each date. The algorithm works by using the individual-agent policy function, which we computed using an Euler equation iteration approach,[5]to simulate a panel of individual-agent capital holdings from which an updated belief about the ALM is computed via an ordinary least squares (OLS) regression of the aggregate capital stock on its past value. This process continues until the estimated OLS parameters settle down to values for which perceived beliefs about aggregate capital and the actual evolution of aggregate capital are consistent with each other. For each iteration of the KS-algorithm, these two objects - the simulated panel of individual capital holdings and the implied ALM, and the updated individual policy function - are computed in a sequential manner.[6]

The panel simulation step starts from a predetermined vector of each agent's capital holdings. Conditional on these values, the capital holdings of each agent are simulated for T periods using the current iterate of the policy function. The key insight of Krusell and Smith (1998) was that individual agent capital holdings could be computed accurately without full knowledge of the wealth distribution - an infinite dimensional object. Instead, it is sufficient to approximate the wealth distribution with its first moment. This makes the simulation stage of the Krusell-Smith algorithm an obvious candidate for parallelization because it means that, in any given time period, each agent's optimal capital holdings can be computed without any knoweldge of the capital holdings of the other agents, so that sequential computation is unnecessary. Given that we parallelize only the panel simulation stage of the Krusell-Smith algorithm, the upper bounds on the potential speed gains available are determined by Amdahl's law, which we now briefly discuss.

2.2 Amdahl's Law

As noted above, the first stage of our algorithm (i.e. Euler equation iteration) is not computed in parallel on the GPU and so should not solve any faster. Therefore, we should expect the speed gain for the algorithm as a whole to depend upon the fraction of total computation time that is spent in the panel simulation stage. This intuition is formalized in Amdahl's Law; see Aldrich (2014) for a useful discussion. Amdahl's law states that if a fraction P of an algorithm can be executed in parallel, then the theoretical maximum speedup

[5]In the descriptions that follow we assume that the current update of the individual-agent policy function has already been computed in a first step using Euler equation iteration (or some other approach). In our application, we relied on the Euler equation iteration algorithm available on Serguei Maliar's webpage (seeMaliar et al. (2010)), which we translated from Matlab to C++. We are grateful to Serguei Maliar for making his code available.

[6]We provide a full description of the two main stages of the Krusell-Smith algorithm and our numerical implementation in the Appendix.

when using N_s processing cores is $S = \frac{1}{(1-P)+P/N_s}$. The intuition is as follows. A fraction (1–P) of the algorithm will be run in the same time as in the serial case, while the remaining fraction P of the algorithm will take P/N_s units of time, because it can be run in parallel on N_s cores. Dividing one unit of time by the total compute time in the parallel case yields the above expression for S. The significance of the above equation is that large speed gains are likely to be realized for our algorithm if the panel simulation stage is a sufficiently large fraction of the overall algorithm and the number of CUDA cores on the GPU is sufficiently high. As documented in Table **1**, we use an NVIDIA Tesla K40 GPU with 2880 single-precision and 960 double-precision CUDA cores in this study.

Before turning to implementation details we provide a brief discussion of the current state of GPU hardware and best practices in GPU programming. We focus on NVIDIA, the manufacturer of the GPU hardware we used in this study, as this helps to explain several of our implementation choices discussed below.

2.3 GPU hardware and best-practice GPU programming

Despite recent efforts by both GPU hardware manufacturers and open-source organisations, using GPUs for scientific purposes still involves a relatively steep learning curve, with specially tailored solutions (such as solutions to non-trivial economic problems) requiring some low-level coding effort for programmers to tap peak performance levels.[7] One important GPU hardware consideration arises as a result of the translation process from double-precision accuracy parallel compute problems into a hardware domain (i.e. GPUs) that was originally designed for single-precision accuracy. Translating a specific numerical algorithm is a complicated task that can be done in several different ways, and it is not always obvious (even to experencied programmers) which route will yield the best results. More importantly, this translation process generates a significant trade-off for scientific researchers because accuracy is a crucial consideration, yet the slower computation speeds associated with using double-precision floating-point arithmetic on GPUs can be substantial.[8]And although double-precision arithmetic is now widely available on GPUs, many efficient and purpose-built functions are limited to single-precision accuracy. Given the non-trivial trade-off between using single-precision and double-precision arithmetic for computation on GPUs, we chose to experiment with several different approaches.

The key difference between CPUs and GPUs is that GPUs specialize in

[7]Creel and Goffe (2008) argued that the diffusion of GPU technology had been relatively low in economics and econometrics due to the relatively steep learning curve involved. However,Dziubinski and Grassi (2014) show that C++Accelerated Massive Parallelism (C++ AMP) lowers barriers to adoption of GPU due to its simplified programming style.

[8]For example, although the last three generations of CUDA-enabled GPUs (codenamed Fermi, Keppler, and Maxwell) added hardware support for double-precision arithmetic, most mainstream GPU models havce fewer CUDA cores capable of double-precision computation, leading to speed ratios of double to single precision arithmetic as low as 1/32. Morozov and Mathur (2012) show that double-precision arithmetic implies a substantial increase in computation time in the context of a complex optimal control problem.

handling comparatively lightweight transformations of multiple data elements massively in parallel. This approach is encapsulated by the SIMD (single-instruction-multiple-data) paradigm of computation, which leverages the notion of *data parallelism* as opposed to *task parallelism*. While CPUs typically possess a fairly small number of very fast, heavily cached and comparatively complex hardware cores which individually specialize in speeding up serial code, GPUs group together a grid composed of a very large number (typically somewhere between 100-3000) of far less complex compute cores (CUDA cores), which need to share resources such as memory and cache and are clocked at much lower speeds than CPU cores. These CUDA cores excel at speeding up calculations involving a large number of single-precision accuracy operations which are executed independently of one other, but they handle instructions with many conditional and divergent "if-else" branches very poorly. Nevertheless, the sheer number of hardware cores they can utilize in parallel is a major potential advantage. More generally, GPUs fill the specific role of dedicated co-processors to which easily paralellizable sub-tasks can be delegated from the CPU host system.

To solve the Krusell-Smith model on the GPU in an efficient manner, we abided (insofar as possible) by the following principles of best practice GPU programming. First, code for the GPU should exploit a language that overlaps with the C programming language but which also supports some popular constructs from C++, since three related activities are necessary in order to efficiently exploit the power of GPU hardware: (i) allocating and copying data between the host and the GPU's on-board memory; (ii) writing one or several so-called "kernel functions" which instruct GPUs how to transform multiple data elements stored in the GPU's memory in parallel; and (iii) grid launch expressions which launch kernel functions from the host side. Secondly, any code needs to be written in a way which minimizes data transfers between the host and the GPU device, because these transfers are relatively slow and often (but not always) may freeze parallel computation on the GPU, leaving it idle for the period of the data transfer. Therefore, programmers should aim to transfer all of the required data to the GPU's memory once-and-for-all upfront, so that hardware utilization rates of close to 100% can be attained over long periods of run-time. Thirdly, data accesses to the global memory of the GPU should be coalesced, the availability of fast on-chip cache memory should be exploited, and divergent "if-else" conditional branches in kernel functions should be avoided where possible. Since parallel tasks run on CUDA hardware are always executed in "warps" of 32 threads processed in lock-step, the existence of only one divergent "if-else" branch (i.e. where 31 threads in a warp evaluate to true, but only one evaluates to false) will result in that divergent thread being processed twice (once for true and once for false) in serial fashion. Thus, the higher the number of divergent threads per warp, the larger the performance penalty incurred.[9]

[9] GPUs are typically known to be limited in their speed not by their compute power (which is determined by the total number and speed of CUDA cores), but by the memory bandwidth available in swapping data inside their memory hierarchy. This explains why fine-tuning GPU code often involves giving priority to carrying out calculations in on-chip shared memory banks which posses a large bandwidth advantage over the larger but much slower *global* memory.

Table 1: Performance characteristics of a selection of GPU and CPU models

Type of GPU/CPU	Generation	HW (#Cores)	Freq. (GHz)	SP (GFlops)	DP (GFlops)	Mem (Gbs)
NVIDIA Tesla K40	Keppler	2880	0.75	4290	1430	12
INTEL I7-3770	Sandy Bridge	4	3.90	240	60	N/A

Note: Reported single-precision GFLOPS for the CPU are obtained by considering the simultaneous processing power available from using all 4 CPU cores jointly. Our main results are based on a comparison between the GPU and the CPU employing only one single core.

The GPU model we use in this paper is the Tesla K40. This combines a Keppler GK110B GPU chip comprised of a total of 2880 SP CUDA cores and 12GB of global on-board memory. At the time of writing, this particular model costs around $4,000 and represents one of NVIDIA's leading GPUs targeted at scientific users who require fast double-precision accuracy arithmetic.[10] In addition to the 2880 SP the Tesla K40 also houses another 960 DP CUDA cores on the same chip, resulting in a double to single precision arithmetic speed ratio of 960/2880=1/3 and a theoretical computational throughput of 4.29 TFlops for single precision and 1.43 TFlops for double-precision operations. The performance characteristics of the GPU and CPU models used in this study are reported in Table 1.

We installed the K40 GPU into a desktop PC equipped with an Intel i7 quad-core CPU and 16Gb of RAM running on a 64-bit Linux system and also installed all required CUDA drivers as well as the most recent version of the CUDA SDK 6.5 (software development kit) which is needed in order to write and compile code using GPU parallelization. Our code was written and compiled using a combination of C++, C, Fortran and CUDA-C with additional run-time dependencies on a small number of popular open-source libraries, including Boost, GSL (Gnu Scientific library), and FITPACK. These libraries were used, respectively, for general convenience, OLS estimation, and built-in CPU-based interpolation functionality. Given that the task of building executable files based on our code involves a series of complex compilation and linking steps, we streamlined this build process in Linux using the automated build system CMake.[11]

2.4 GPU implementation of the panel simulation

The computational problem we face in conducting the panel simulation step in parallel is that of having to interpolate on the policy function grid a total

[10] We are grateful to NVIDIA corporation for donating the Tesla K40 GPU used for this research.

[11] All of our source code for this project can be downloaded in the "Code" section at http://www.ericscheffel.com.

of $N \times T$ times in order to compute optimal next-period capital choices for each of the N agents over all T time periods, a necessary step for obtaining updated OLS estimates from the simulated aggregate physical capital series. However, these $N \times T$ interpolation calculations cannot be done out-of-order since, for each time period, the mean of the wealth distribution needs to be computed using agents' optimal capital holdings, in order to provide an input for computing the cross-section of agents' optimal capital holdings in the next period. This recursive property of the problem limits us to parallelizing code only in the cross-section dimension N for each of the T time periods.

Our implementation starts from the host side where we first simulated $N \times T$ idiosyncratic labour market shocks and T aggregate productivity shocks. Given that the idiosyncratic shocks are binary and a large N will quickly increase the memory requirement for the matrix of idiosyncratic shocks to prohibitive levels, we chose to encode the labour market shocks using an $(N/8) \times T$ memory block of 8-bit characters and employ bit-wise storage and operators inside the GPU's kernel functions (as well as on the host side). This data compression work-around was necessary since we consider cross-sections of agents as large as $N = 10$ million and set the simulation length at $N = 1{,}000$.[12] The $(N/8) \times T$ labour market shock matrix and the array of T aggregate productivity shocks were loaded into the GPU's memory, where they become accessible from within kernel function calls.

Instead of declaring the optimal policy function grids as conventional 'buffer memory' in the GPU's global memory pool - as we did with the shock matrices - we instead declared them as texture memory. Given that only two state dimensions of the policy function are near-continuous (individual and aggregate capital), we proceeded by uploading a total of four 2D grids into the GPU's texture memory: one for each possible productivity-employment combination. It is important to note that texture memory is special in that it is "read-only", limited to 32-bit single-precision storage, can be accessed in kernel functions using fast hardware-based texture functions, and makes repeated fetches from identical texture locations subject to very fast on-chip caching. However, some important limitations arise in this context, to which we now turn.

2.4.1 A fast hardware-based bi-linear texture interpolation approach

At first glance, CUDA's texture memory and the fast hardware-wired capabilities would appear to be the perfect tool for parallelizing repeated interpolation computations on a set of 2D grids. For instance, the various built-in 1D, 2D, and 3D texture routines available during kernel function programming via the CUDA-C language support various interpolation options directly in hardware.

[12] Had we not employed bit-wise storage in this particular case, storing the labour market shock matrix alone would have exceeded 10GB of memory, whereas data compression lowers the memory requirement to less than 1.5GB. We decided to keep memory requirements below 2GB because 32-bit operating systems cannot address memory larger than this threshold and many GPUs possess only 2GB on-board memory. This makes our code compatible with as many hardware platforms as possible, thus making it easier for other researchers to replicate our results.

Algorithm 1 Single-precision hardware-based interpolation kernel (Cuda-C)

```
1
2  __global__ void panelsim2d_catmull_rom_kernel(
3                   cudaTextureObject_t texObj_bad,
4                   cudaTextureObject_t texObj_good,
5                   float K, float *inarr, unsigned char *idios,
6                   int N, int ngrid2, int ngrid,
7                   float xmin, float xmax, float ymin, float ymax,
8                   float theta)
9  {
10    // Fetch the global index variable
11    unsigned int xi = blockIdx.x * blockDim.x + threadIdx.x;
12
13    if (xi < N) {
14
15       // Compute the normalized coordinates for given k(t-1) and K(t-1)
16       float xv = ((K - xmin)/(xmax - xmin)) *
17                  (float) (ngrid2 - 1) + 0.5f;
18       float yv = (powf((inarr[xi] - ymin)/(ymax - ymin),(1.0f/theta)) *
19                  (float) (ngrid - 1)) + 0.5f;
20       unsigned char empstatb = idios[(xi/8)] & (128>>(xi%8));
21
22
23       // Using coordinates read optimal k(t) from texture and store
24       float val = empstatb ? clamp(tex2D(texObj_good, xv, yv),
25                                    ymin, ymax) :
26                              clamp(tex2D(texObj_bad, xv, yv),
27                                    ymin, ymax);
28       inarr[xi] = val;
29    }
30 }
```

However, a careful study of CUDA documentation reveals two potentially important drawbacks. First, standard methods employed in declaring and uploading texture surfaces (i.e. 2D numerical grids) into the GPU's memory only support data represented with 32-bit (i.e. single-precision) accuracy. As a result of this limitation, all the built-in interpolation routines only work at this reduced level of numerical accuracy. Secondly, automated interpolation done entirely in GPU hardware leads to an additional and more serious loss in numerical accuracy because the computed weights used in interpolation are represented using as little as 8-bit decimal point precision.[13]

A major potential advantage of this approach is its simplicity. Algorithm 1 shows that an extremely simple kernel function can be used to employ hardware-based bi-linear interpolation on the 2D policy function grids in the texture memory. The kernel code consists of only one "global" function, which is callable from the host where the normalized texture coordinates are computed on lines 16-19 (these points are corrected for grid point bunching along the individual-agent capital dimension) before being passed to the built-in interpolation routine "tex2D", which computes and stores (subject to employment status) the interpolated values in parallel. However, given the potential accuracy drawbacks associated with this fast hardware-based interpolation approach, we choose to

[13]Consequently, this potentially fast approach could be too inaccurate for our purposes, *especially* for very sparse policy function grids.

also investigate two alternative approaches which deliver greater numerical accuracy.

2.4.2 Two software-based bi-cubic texture interpolation approaches

Algorithm 2 demonstrates our first alternative approach used in coding the panel simulation step, which is based on a kernel code that uses single-precision accuracy throughout.[14] Employing a total of 16 function calls to "tex2D", we then hand-coded the interpolation in software using a more complicated, purpose-built kernel function. Since the hand-coded interpolation method side-steps the hardware-wired alternative (which uses interpolation points with only 8-bit decimal point precision), it should lead to an improvement in accuracy and gives us the flexibility to choose from a large number of candidate interpolation approaches. This hand-coded approach remains very fast for two reasons. First, since we are still retrieving values from texture memory based on "tex2D" function calls, our code benefits from fast cache memory. Secondly, running the panel simulation using single-precision arithmetic makes use of a total of 2880 hardware CUDA cores in parallel - i.e. three times the availability of double-precision execution units on our GPU.

After some experimentation, we settled on a bi-cubic Catmull-Rom (Hermite) interpolation method. This method is 3rd-order accurate, preserves the shape of the underlying function and is obtained using a total of 16 nearest neighbours, as compared to 4 in the case of bi-linear interpolation. In this kernel code (see algorithm 2), the first four GPU-only kernel functions denoted "catrom_w0a", "catrom_w1a", "catrom_w2a", and "catrom_w3a" are used to compute and evaluate the four required Hermite basis functions, while the fifth GPU-only kernel function denoted "CatRomFiltera" combines these functions using a final convolution step.

The second implementation approach we consider is almost identical to the first but uses double-precision arithmetic throughout (the algorithm **A.2** is provided in the Appendix). Double-precision numerical accuracy was obtained for both operations carried out on memory locations and general local variable and texture storage using a 'work-around' method that allows textures which are otherwise 32-bit-only to hold elements with 64-bit double-precision accuracy. The trick employed by this method is to use *double-layered* 32-bit (i.e. single-precision) textures to distribute the storage of 64-bit double-precision elements across two 32-bit layers. In kernel functions the special procedure "hiloint2double(x,y)" can then be used to 'stitch together' at the bit-level the two layered 32-bit memory locations and convert them back to the original 64-bit value. Given that this interpolation approach proceeds using a hand-coded kernel function in which all intermittent operations are done using double-precision accuracy, only 960 hardware CUDA cores can be employed in this case, making execution performance somewhat slower.

[14] In this and subsequent kernel code the behaviour of the "tex2D" function is altered to a basic "point mode" under which any calls to it using 2D-indeces as arguments permit only the retrieval of points residing exactly on the nodes comprising the 2D texture surface.

Algorithm 2 Single-precision Catmull-Rom interpolation kernel (Cuda-C)

```
__host__ __device__ float catrom_w0a(float a) {
return a*(-0.5f + a*(1.0f - 0.5f*a)); }

__host__ __device__ float catrom_w1a(float a) {
return 1.0f + a*a*(-2.5f + 1.5f*a); }

__host__ __device__ float catrom_w2a(float a) {
return a*(0.5f + a*(2.0f - 1.5f*a)); }

__host__ __device__ float catrom_w3a(float a) {
return a*a*(-0.5f + 0.5f*a); }

__device__ float catRomFiltera(float x, float c0,
                               float c1, float c2, float c3) {
float r;
r = c0 * catrom_w0a(x);
r += c1 * catrom_w1a(x);
r += c2 * catrom_w2a(x);
r += c3 * catrom_w3a(x);
return r; }

__device__ float tex2DCatRoma(cudaTextureObject_t texObj,
                              float x, float y) {
x -= 0.5f;
y -= 0.5f;
float px = floor(x);
float py = floor(y);
float fx = x - px;
float fy = y - py;
px += 0.5f;
py += 0.5f;

return catRomFiltera(fy,
catRomFiltera(fx,tex2D(texObj,px-1,py-1),tex2D(texObj,px,py-1),
                 tex2D(texObj,px+1,py-1),tex2D(texObj,px+2,py-1)),
catRomFiltera(fx,tex2D(texObj,px-1,py),  tex2D(texObj,px,py),
                 tex2D(texObj,px+1,py),  tex2D(texObj,px+2,py)),
catRomFiltera(fx,tex2D(texObj,px-1,py+1),tex2D(texObj,px,py+1),
                 tex2D(texObj,px+1,py+1),tex2D(texObj,px+2,py+1)),
catRomFiltera(fx,tex2D(texObj,px-1,py+2),tex2D(texObj,px,py+2),
                 tex2D(texObj,px+1,py+2),tex2D(texObj,px+2,py+2)));
}

__global__ void panelsim2d_crk(cudaTextureObject_t texObj_bad,
                 cudaTextureObject_t texObj_good, float K,
                 float *inarr, unsigned char *idios, int N,
                 int ngrid2, int ngrid, float xmin, float xmax,
                 float ymin, float ymax, float theta)
{
   unsigned int xi = blockIdx.x * blockDim.x + threadIdx.x;

   if (xi < N) {
      float xv = ((K - xmin)/(xmax - xmin)) *
                 (float) (ngrid2 - 1) + 0.5f;
      float yv = (powf((inarr[xi] - ymin)/(ymax - ymin),(1.0f/theta)) *
                 (float) (ngrid - 1)) + 0.5f;
      unsigned char empstatb = idios[(xi/8)] & (128>>(xi%8));

      float val = empstatb ? clamp(tex2DCatRoma(texObj_good, xv, yv),
                                   ymin, ymax) :
                             clamp(tex2DCatRoma(texObj_bad, xv, yv),
                                   ymin, ymax);
      inarr[xi] = val;
   }
}
```

2.4.3 Description of host-side code

As illustrated in algorithm 3, we proceeded on the host side by executing our candidate CUDA kernels repeatedly for $T-1$ periods. In order to obtain the updated first moment of the distribution of capital holdings between periods, we instructed the GPU to execute a 'reduction' on the current N-sized vector of optimal individual-agent capital holdings, before launching a new kernel run for the next period. The 'reduction' (in this case a simple summation) over all agents' individual optimal capital holdings was done in parallel on the GPU using one of NVIDIA's abstraction libraries called "Thrust", which carries out basic tasks efficiently in parallel before copying the results back to the host memory pool.

In order to make our three interpolation approaches directly comparable, we employed the Catmull-Rom bi-cubic interpolation method on the CPU in a serial fashion, by traversing individual agent capital holdings agent by agent and period by period. This CPU-variant of the panel simulation was coded in order to allow for the possibility of multi-threaded execution based on the use of several CPU cores in parallel. However, as a benchmark, we compare the parallel panel simulation on the GPU against a CPU utilizing only one hardware core. One part of the Krusell-Smith algorithm - namely the individual-agent policy function computation - was done entirely on the CPU without recourse to the GPU hardware. Since the Euler equation iteration method requires repeated interpolation sweeps carried out over the entire policy function grid, we used the Fortran library FITPACK, which includes spline-based routines for bi-cubic interpolation on 2D grids.[15]

By designing the host code of the GPU-assisted panel simulation in this way we combined hundreds of synchronized GPU kernel calls with many intermittent data transfer operations between the GPU and host memory. In general this is inefficient, but since each intermittent data transfer involved copying only one double-precision floating point value (i.e. the sum over all individual-agent capital holdings), this had little impact upon performance. Indeed, diagnostic output from NVIDIA's profiler indicated that the executed code exhibited sustained GPU utilization rates of around 98% and texture cache hit rates close to 100%. Such peak-level cache hit-rates are to be expected in the context of the Krusell-Smith algorithm, given that a large number of agents possess similar levels of capital. This high degree of "locality" in successive texture fetches (see e.g. NVIDIA's online CUDA API manuals) implies that reading optimal choices off policy function grids stored in the GPU's cached texture memory represents an ideal implementation choice in the context of the Krusell-Smith algorithm, because it plays to the strength of the cached texture memory architecture.

We also violated the rule that divergent "if-else" branches should be avoided. We did so because populating the array of optimal next-period capital holdings for each of the N agents requires the employment status (which is determined by the idiosyncratic shock), which in turn determines which 2D policy grid

[15] In particular we employed a combination of the "regrid" and "bispeu" routines, which we also coded to allow for multi-threaded execution using several CPU cores.

should used for interpolation. While the "if-else" branch is not explicit in our kernel code, it creeps in implicitly through a C-style ternary operator, which for the single-precision kernel appears on lines 59-62 and for the double-precision kernel appears on lines 64-67. In practice, however, our profiling results exhibit near peak performance levels, suggesting that there were few incidences of warp divergence.[16]

3 Results

We report results for one converged simulation of the Krusell-Smith algorithm for each of the three GPU-based approaches discussed above and the CPU-only approach. Since these approaches differ only in the method and type of hardware used to interpolate off the optimal policy grid during the panel simulation stage of the Krusell-Smith algorithm, the equilibrium solution should be essentially identical across the four simulations. We started by simulating a panel of $N = 10,000$ agents with the same initial distribution of wealth as in the 'Computational Suite' project (see Den Haan, 2010).[17] When simulating larger panels, such as $N = 500,000$ or $N = 10,000,000$, we constructed an initial distribution of wealth with similar moments to the distribution when $N = 10,000$. We set the convergence tolerance at $\epsilon = 1E-7$. The results for the CPU-only approach are reported in Table **2** and those under the three GPU approaches in Table **3**. As indicated we employ policy grids of varying denseness. The coarsest grid specification we consider follows Krusell and Smith's original paper in which they employ 100 points in the individual agent and 4 points in the economy-wide capital dimension. Following Horvath (2012), we also consider policy grids a finely discretized as 500 points in the individual and 50 points in the economy-wide capital dimension.

The CPU results show that as the number of grid points for capital is increased, solution times increase at an exponential rate. This is simply a manifestation of the "curse of dimensionality". For our chosen convergence tolerance of $\epsilon = 1E-7$, the KS algorithm usually takes around 30 outer loop iterations to converge when it is solved on the CPU only. By comparison, the GPU approaches usually take a similar number of iterations (with the exception of the hardware-based single precision method) and solution times are usually much lower. The speed gains are largest when relatively coarse capital grids are combined with very large numbers of agents such as $N = 10,000,000$. For instance, with the double precision method there is a speed gain of around 1,200 times in

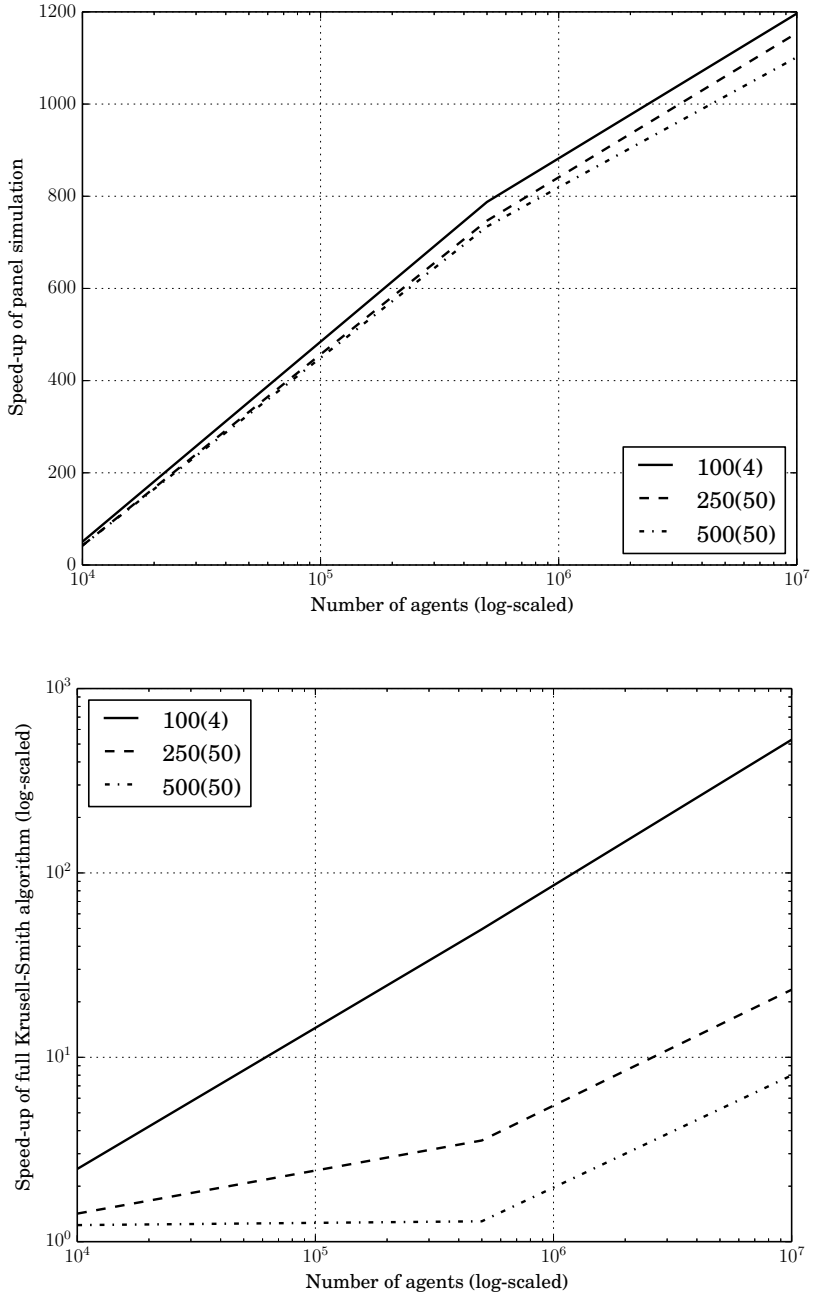
[16] A significant performance degradation due to warp divergence should only be expected in cases where, in a warp of 32 threads, one or several branches evaluate to, say, unemployed status, while the others evaluate to employed status. This implies that "if-else" branches in fact pose no performance problem, so long as parallelized threads executed within one warp always simultaneously cascade down one and same branch. Given that around 10% of agents are unemployed on average, the proportion of warps in which all 32 threads (i.e. agents) evaluate to employed should be relatively high. This provides one explanation for the finding that warp divergence did not pose a substantial performance penalty in our code.

[17] The initial distribution of wealth can be downloaded from Wouter Den Haan's webpage.

Algorithm 3 Host side code for panel simulation step [C,C++]

```
1
2  // Declare and initialize array of continuous state boundaries
3  double bounds[4] = {K_min, K_max, k_min, k_max};
4
5  // Declare and initialize array of GPU block size
6  int blockdim[3] = {512,1,1};
7
8
9  // Update the policy function texture objects on the GPU
10 cudaMemcpyToArray(texo_bu->cuArray,0,0,&fckprime[0],
11        ngrid2*ngrid*sizeof(float),cudaMemcpyHostToDevice);
12 cudaMemcpyToArray(texo_be->cuArray,0,0,&fckprime[ngrid2*ngrid],
13        ngrid2*ngrid*sizeof(float),cudaMemcpyHostToDevice);
14 cudaMemcpyToArray(texo_gu->cuArray,0,0,&fckprime[2*ngrid2*ngrid],
15        ngrid2*ngrid*sizeof(float),cudaMemcpyHostToDevice);
16 cudaMemcpyToArray(texo_ge->cuArray,0,0,&fckprime[3*ngrid2*ngrid],
17        ngrid2*ngrid*sizeof(float),cudaMemcpyHostToDevice);
18
19 // Call the N-sized parallelized kernel function T periods of time
20 for (int t=0; t<T-1; ++t) {
21
22          // For the good aggregate TFP state
23          if (h_aggs[t/8] & 128>>t%8) {
24
25          // Kernel call
26          panelsim2d_catmull_rom_kernel<<<dimGrid,dimBlock>>>(
27         texo_gu->texObj,texo_ge->texObj,
28              k_mean[t],thrust::raw_pointer_cast(&(*d_inarr)[0]),
29              thrust::raw_pointer_cast(&(*d_idios)[t*(N/8)]),N,ngrid2,ngrid,
30              bounds[0],bounds[1],bounds[2],bounds[3],theta
31      );
32
33          // Synchronize (wait) for device to finish parallel compute task
34          cudaDeviceSynchronize();
35
36          // For the bad aggregate TFP state
37          } else {
38
39          // Kernel call
40          panelsim2d_catmull_rom_kernel<<<dimGrid,dimBlock>>>(
41         texo_bu->texObj,texo_be->texObj,
42              k_mean[t],thrust::raw_pointer_cast(&(*d_inarr)[0]),
43              thrust::raw_pointer_cast(&(*d_idios)[t*(N/8)]),N,ngrid2,ngrid,
44              bounds[0],bounds[1],bounds[2],bounds[3],theta
45      );
46
47          // Synchronize (wait) for device to finish parallel compute task
48          cudaDeviceSynchronize();
49 }
50
51 // Update the aggregate K vector by saving current value
52 double sumor = thrust::reduce((*d_inarr).begin(), (*d_inarr).end());
53 double tval = sumor/(double)N;
54
55 // Make sure the computed first moment stays within permissible bounds
56 k_mean[t+1] = tval*(tval >= K_min && tval <= K_max) +
57               K_min*(tval < K_min) + K_max*(tval > K_max);
58 }
```

Figure 1: Speed-up of double-precision GPU method



Note: Figures show relative gain defined as $G = T_{CPU}/T_{GPU}$. Relative gains are substantially higher for single-precision methods and are not plotted here.

the panel simualtion stage and almost 600 times for the algorithm as a whole when there are 10 million agents and 100 grid points in the individual capital direction and 4 in the aggregate capital direction. By contrast, the gains from parallelization are relatively small for fine capital grids and small panel sizes, with gains in panel simulation stage of between 40 and 50 times and total simulation times which are only slightly faster, and in a couple of cases higher.[18]

The gains in the panel simulation stage are lower in these cases because a larger grid implies that the texture cache of the GPU is hit with a lower probability, while the overall solution times are fairly similar for the CPU and GPU due to Amdahl's law. In particular, both small panel sizes and fine capital grids decrease the relative importance of the panel simulation stage of the algorithm - the former because it means that fewer computations are run in parallel, and the latter because the Euler equation iteration stage (which is carried out on the CPU) is subject to the curse of dimensionality. It is important to note that in the cases where the speed gains from GPU computation are substantial, there are practically important reductions in computation times, since solving the model on the CPU takes several hours in these cases. In practice, these gains are likely to be magnified given that most researchers are interested in simulating a model a large number of times and not just once as in our experiments. Finally, it is worth noting that relative gains in computational speed tend to diminish as we consider increasingly more sophisticated (and thus more precise) interpolation schemes implemented on the GPU.

We provide a visual representation of the GPU results in Figure **1**, which plots the relative gain of the double-precision GPU approach in the panel simulation stage and for the algorithm as a whole. Even with this slower (and more accurate) methodology the results are very impressive, with gains of over 700 times for $N = 500,000$ and $1,100$ times for $N = 10,000,000$ in the panel simulation stage. For the hardware-based single-precision method, the implied speed-ups for the panel simulation can reach values as high as $3,700$, while the software-based single precision method gives gains of almost 2,500 times. The lower panel of **1** reports the relative gain for the algorithm as a whole as the number of agents is varied. The results here clearly demonstrate the operation of Amdahl's law: larger numbers of agents increase relative importance of the panel simulation stage of the Krusell-Smith algorithm, raising the potential gains from parallelizing using GPU computing.

In order to shed some light on the accuracy of our different methods, we stored the average economy-wide physical capital series for T periods obtained from the 5th outer loop iteration of each simulation. Since the transitional state of this series can be interpreted as the outcome from a learning process (Giusto, 2014), we computed the implied transitional "learning errors" using the

[18]Total solution times are higher for the hardware-based interpolation method because the lower degree of arithmetical precision means that the algorithm takes longer to reach convergence, as shown by the larger number of total iterations in Table **3**. In some cases, the outer loop did not converge within 50 iterations. In these cases we increased the tolerance criterion by one order of magnitude to $\epsilon = 1E{-}6$ and restarted the algorithm until convergence was attained.

Table 2: Computational times for panel simulation step (CPU-only)

#CPU Threads		1			2			4		
#Agents	Mem(Gbs)	100(4)	250(50)	500(50)	100(4)	250(50)	500(50)	100(4)	250(50)	500(50)
10.000	0.00116									
Panel Time		10.1	10.1	10.1	4.7	4.7	4.7	2.4	2.4	2.4
Polf. Time		7.6	380.5	1243.3	7.6	380.5	1243.3	7.6	380.5	1243.3
Iterations		30	32	27	30	32	27	30	32	27
Total Time		530.8	12499.2	33841.8	369.0	12326.4	33696.0	300.0	12252.8	33633.9
500.000	0.058									
Panel Time		433.2	433.2	433.2	215.1	215.1	215.1	107.2	107.2	107.2
Polf. Time		7.4	385.3	1279.2	7.4	385.3	1279.2	7.4	385.3	1279.2
Iterations		23	31	30	23	31	30	23	31	30
Total Time		10133.8	35373.5	51372.0	5117.5	18612.4	44829.0	2635.8	15267.5	41592.0
10.000.000	1.164									
Panel Time		8482.3	8482.3	8482.3	4241.4	4241.4	4241.4	2120.1	2120.1	2120.1
Polf. Time		7.37	381.1	1275.2	7.37	381.1	1275.2	7.37	381.1	1275.2
Iterations		27	32	33	27	32	33	27	32	33
Total Time		229221.1	283628.8	321997.5	114716.8	147920.0	182047.8	57441.7	80038.4	112044.9

Note: All figures indicate the total number of seconds elapsed until task completion in any iteration.
Also shown is the total amount of time it took for the KS algorithm to converge, including the total number of outer loop iterations.
Timings are reported for cases in which 1, 2 and 4 CPU hardware threads were employed. Memory storage requirements are also reported.
The convergence tolerance was set to $\epsilon = 1 \times E^{-7}$.

difference between the aggregate capital series obtained from GPU-based interpolation schemes and the one obtained via the CPU-only simulation approach. To this end we compute and report the maximum absolute percentage deviation of the GPU from the CPU series and report it in table **2** in the Appendix. Based on this we immediately observe that simulation runs in which convergence could not be attained after a total of 50 outer loop iterations usually also exhibited the largest absolute errors in learning about the evolution of the aggregate capital series. This finding confirms the intuition that the hardware-based single-precision approach is the least accurate of the three GPU approaches we considered and suggest that it may be unwise to use this approach in cases where accurate numerical results are imperative. [19]

Overall, these results suggest that 2 of the 3 approaches we investigated perform well in terms of both accuracy and speed, implying that GPU computing can help to make the trade-off between accuracy and speed somewhat less severe. For the other (hardware-based) approach, we are confident that this could be useful in practice if combined with very densely parametrized policy function grids, such as $1024 \times 64 = 65,536$ capital grid points. In fact, the results of Maliar et al. (2010) show that bi-linear interpolation is adequate in the context of the Krusell-Smith algorithm for sufficiently dense policy grids. However, given that the Euler equation approach employed in iterating on such dense grids would lead to prohibitively slow convergence if executed serially on the CPU, it is clear that only an overall computational strategy which parallelizes *both* the panel simulation and policy function iteration on the GPU would make this approach feasible. This is an interesting avenue which we consider worthwhile exploring in future research.

4 Conclusion

In this paper, we have demonstrated the potential of graphics processing units (GPUs) in solving models with heterogeneous agents and incomplete markets using the Krusell-Smith algorithm. In particular, utilizing the compute unified device architecture (CUDA) of Nvidia, we demonstrated that harnessing the power of GPUs can deliver a substantial speed gain over a CPU-only approach in the panel simulation stage of the algorithm. Notably, this gain increases sharply as the number of agents is increased. The GPU code we make available is optimized for speed by employing cached texture memory, an optimization choice ideal for the panel simulation step of the Krusell-Smith algorithm because of the highly concentrated mass of the wealth distribution.

To demonstrate the improvement in computation speed with GPU computing, we simulated a version of the Krusell-Smith model with a large number of

[19] It may be worth mentioning that in many cases convergence for the fast hardware-based linear interpolation approach could have been attained within less than 50 outer loop iterations had we relaxed the convergence tolerance only slightly to $\epsilon = 3.0E-7$ instead of lowering it by an entire order of magnitude to $\epsilon = 1.0E-6$. In this sense the fast hardware-based approach may still be a feasible option when a high degree of accuracy is not necessary.

Table 3: Computational times for panel simulation step (GPU/CPU comparison)

GPU Interp.		Hardware Single Precision			Software Single Precision			Software Double Precision		
#Agents	Mem(Gbs)	100(4)	250(50)	500(50)	100(4)	250(50)	500(50)	100(4)	250(50)	500(50)
10.000	0.00116									
Panel Time		0.19	0.20	0.23	0.19	0.24	0.25	0.20	0.24	0.24
Panel Speedup		53.20	50.51	43.90	53.20	42.11	40.41	50.51	42.11	42.11
Polf. Time		7.53	372.80	1284.80	7.70	388.20	1288.30	7.53	386.50	1286.70
Iterations		$84^{\dagger}$	$71^{\dagger}$	48	25	26	28	28	23	22
Total Speedup		1.78	0.45	0.56	1.88	1.24	1.00	2.48	1.42	1.23
$\bar{K}$ Error		0.875	0.100	0.029	0.013	0.014	0.014	0.012	0.014	0.014
500.000	0.058									
Panel Time		0.3	0.33	0.34	0.36	0.40	0.41	0.55	0.58	0.59
Panel Speedup		1444.00	1312.73	1274.12	1203.33	1083.00	1056.59	787.64	746.90	734.24
Polf. Time		7.80	385.20	1284.40	7.36	384.70	1288.80	7.36	383.90	1289.20
Iterations		$76^{\dagger}$	40	$68^{\dagger}$	29	25	26	26	26	31
Total Speedup		16.36	2.33	0.60	54.41	3.73	1.54	49.49	3.54	1.29
$\bar{K}$ Error		0.861	0.098	0.021	0.003	0.001	0.001	0.004	0.001	0.001
10.000.000	1.164									
Panel Time		2.28	2.29	2.31	3.42	3.61	3.78	7.09	7.35	7.70
Panel Speedup		3720.31	3704.06	3671.99	2480.20	2349.67	2243.99	1196.38	1154.05	1101.60
Polf. Time		7.75	384.3	1267.8	7.69	381.2	1282.3	7.57	386.5	1281.3
Iterations		$65^{\dagger}$	$73^{\dagger}$	$71^{\dagger}$	31	35	33	27	32	33
Total Speedup		195.04	10.05	3.62	666.36	21.25	8.02	582.00	23.26	7.98
$\bar{K}$ Error		0.867	0.098	0.026	0.027	0.027	0.026	0.003	0.000	0.000

Note: Times are in seconds. $\bar{K}$ error is defined as $K_{err} = max(abs(K_{t,GPU} - K_{t,CPU})/K_{t,CPU}) \times 100$
Also shown is the time it took for the KS algorithm to converge, including the total number of outer loop iterations.
$\bar{K}$ Error is the maximum absolute error between the CPU-based ALM and the GPU-variant in question, computed after the 5th outer-loop iteration.
Iterations superscripted with † indicate that convergence was not attained after 50 iterations, requiring a relaxation of ϵ by one order.

agents. We chose this model because it was recently simulated using several different algorithms as part of a project reported in a 2010 special issue of the Journal of Economic Dynamics and Control (see Den Haan, 2010) . Our main finding was that for very large, but realistic, numbers of agents such as 10 million, the panel simulation stage can be simulated more than 1,000 times faster than on the CPU, leading to very large reductions in computation time from a practical perspective.

It should be noted that since many GPUs are optimized for single-precision floating point operations, they tend to be fastest when operated at this level of precision. In the specific context of the Krusell-Smith algorithm, we found that employing single-precision floating point arithmetic during the GPU-based panel simulation was roughly twice as fast as double-precision but produced similar numerical results. This finding suggests that single-precision arithmetic may be adequate for some economic problems solved using GPUs. It should be emphasised, however, that very substantial time gains are realized even under double-precision arithmetic in our simulations. In this respect, GPUs have the potential to make the trade-off between speed and accuracy somewhat less severe. In similar fashion to Aldrich et al. (2011) we caution readers to interpret our reported speed gains as conservative lower bounds, given that additional speed gains are possible based on even more aggressively optimized code. To reduce entry barriers for other researchers, we provided details of the algorithm we used and explained its GPU implementation using key sections of source code. We hope that future research will exploit the computational benefits of GPUs that we have highlighted in this paper.

Bibliography

Aldrich, E. M., 2014. Gpu computing in economics. In: Judd, K. L., Schmedders, K. (Eds.), Handbook of Computational Economics. Vol. 3. Elsevier, Ch. 10.

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

Creel, M., Goffe, W., 2008. Multi-core cpus, clusters, and grid computing: A tutorial. Computational Economics 32 (4), 353–382.

Den Haan, W. J., January 2010. Comparison of solutions to the incomplete markets model with aggregate uncertainty. Journal of Economic Dynamics and Control 34 (1), 4–27.

Den Haan, W. J., Judd, K. L., Juillard, M., January 2010. Computational suite of models with heterogeneous agents: Incomplete markets and aggregate uncertainty. Journal of Economic Dynamics and Control 34 (1), 1–3.

Den Haan, W. J., Rendahl, P., January 2010. Solving the incomplete markets model with aggregate uncertainty using explicit aggregation. Journal of Economic Dynamics and Control 34 (1), 69–78.

Dziubinski, M., Grassi, S., 2014. Heterogeneous computing in economics: A simplified approach. Computational Economics 43 (4), 485–495.

Giusto, A., 2014. Adaptive learning and distributional dynamics in an incomplete markets model. Journal of Economic Dynamics and Control 40 (C), 317–333.

Horvath, M., 2012. Computational accuracy and distributional analysis in models with incomplete markets and aggregate uncertainty. Economics Letters 117 (1), 276–279.

Krusell, P., Smith, A. A., October 1998. Income and wealth heterogeneity in the macroeconomy. Journal of Political Economy 106 (5), 867–896.

Maliar, L., Maliar, S., Valli, F., January 2010. Solving the incomplete markets model with aggregate uncertainty using the krusell-smith algorithm. Journal of Economic Dynamics and Control 34 (1), 42–49.

Morozov, S., Mathur, S., 2012. Massively parallel computation using graphics processors with application to optimal experimentation in dynamic control. Computational Economics 40 (2), 151–182.

Reiter, M., 2010. Solving the incomplete markets model with aggregate uncertainty by backward induction. Journal of Economic Dynamics and Control 34 (1), 28–35.

Appendix

Algorithm A.1 Hybrid CPU/GPU implementation of KS algorithm

1. Specify initial values for the 4 ALM regression coefficients. We choose $B_0 = [0.0, 1.0, 0.0, 1.0]$, i.e. setting all OLS intercepts to zero and all slope coefficients to 1. Specify an initial guess of the individual-agent policy function $k' = \Phi_0(k, K, a, \epsilon)$; we simply set $k' = 0.9 \times k$ for all k,K,a and ϵ on the 4-dimensional discretized state space grid.

2. Conditional on current beliefs about K' implied by B_j for $j \geq 2$, solve the individual-agent policy function via Euler equation iteration on the CPU. Use as a starting value for the policy function either $k' = \Phi_0(k, K)$ (at the start of the simulation) or the last found policy function $k' = \Phi_{j-1}(k, K)$ for B_{j-1}**.** Convergence of this inner loop supplies the current update of the policy function in the shape of the 4-dimensional matrix $k' = \Phi_j(k, K, a, \epsilon)$.

3. Given that the aggregate (a) and idiosyncratic (ϵ) state variables are of binary nature, we decompose and re-write the current $k' = \Phi_j(k, K, a, \epsilon)$ in terms of a total of four 2-dimensional grids, $k' = \Phi_{00,j}(k, K)$, $k' = \Phi_{01,j}(k, K)$, $k' = \Phi_{10,j}(k, K)$, and $k' = \Phi_{11,j}(k, K)$, and upload them into the GPU's texture memory.

4. Initialize N-dimensional vectors v_t and v_{t+1} in the GPU's memory; v_t holds the first-period cross-section of agents' individual wealth, v_{t+1} is at first zero-valued. Copy the current economy-wide capital $K_t = \frac{1}{N}\sum^N v_t$, a matrix containing all agents' idiosyncratic shocks for all $t \in (1, T)$, and a vector of all aggregate shocks for all $t \in (1, T)$ into the GPU's memory. Given that each CUDA core can access these values, use fast interpolated texture fetches from the 4 uploaded policy functions in texture memory to populate v_{t+1} massively in parallel. Compute $K_{t+1} = \frac{1}{N}\sum^N v_{t+1}$ for the next iteration at $t + 1$ for which v_t will now be initialized with the values from v_{t+1}copied over from the previous iteration.

5. From step 4 obtain and copy back to the CPU host's memory the current simulated time path of economy-wide (average) physical capital $[K_0, K_1, \ldots, K_T]$. Employ this in two separate regressions, one for each aggregate TFP-state, in order to compute and update B_{j+1}. We compute this update as a weighted average of the old B_j and the new B_{j+1} using $B_{j+1} = \lambda B_{j+1} + (1 - \lambda)B_j$ with some relaxation parameter λ.

6. Given our new B_{j+1} we repeat steps 2-5 until convergence is achieved based on some $||B_{j+1} - B_j|| < \epsilon$. We set $\epsilon = 1.0E - 7$.

Algorithm A.2 Double precision Catmull-Rom interpolation kernel (Cuda-C)

```
__host__ __device__ double catrom_w0a(double a) {
return a*(-0.5 + a*(1.0 - 0.5*a)); }
__host__ __device__ double catrom_w1a(double a) {
return 1.0 + a*a*(-2.5 + 1.5*a); }
__host__ __device__ double catrom_w2a(double a) {
return a*(0.5 + a*(2.0 - 1.5*a)); }
__host__ __device__ double catrom_w3a(double a) {
return a*a*(-0.5 + 0.5*a); }

__device__ double catRomFiltera(double x, double c0, double c1,
                                double c2, double c3) {
double r;
r = c0 * catrom_w0a(x);
r += c1 * catrom_w1a(x);
r += c2 * catrom_w2a(x);
r += c3 * catrom_w3a(x);
return r; }

__device__ double tex2DCatRoma(cudaTextureObject_t texObj,
                               double x, double y) {
x -= 0.5;
y -= 0.5;
double px = floor(x);
double py = floor(y);
double fx = x - px;
double fy = y - py;
px += 0.5;
py += 0.5;

int2 vv1 =  tex2D(texObj,px-1,py-1); int2 vv2 = tex2D(texObj,px,py-1);
int2 vv3 =  tex2D(texObj,px+1,py-1); int2 vv4 = tex2D(texObj,px+2,py-1);
int2 vv5 =  tex2D(texObj,px-1,py);   int2 vv6 = tex2D(texObj,px,py);
int2 vv7 =  tex2D(texObj,px+1,py);   int2 vv8 = tex2D(texObj,px+2,py);
int2 vv9 =  tex2D(texObj,px-1,py+1); int2 vv10 = tex2D(texObj,px,py+1);
int2 vv11 = tex2D(texObj,px+1,py+1); int2 vv12 = tex2D(texObj,px+2,py+1);
int2 vv13 = tex2D(texObj,px-1,py+2); int2 vv14 = tex2D(texObj,px,py+2);
int2 vv15 = tex2D(texObj,px+1,py+2); int2 vv16 = tex2D(texObj,px+2,py+2);

return catRomFiltera(fy,
catRomFiltera(fx,__hiloint2double(vv1.y,vv1.x), __hiloint2double(vv2.y,vv2.x),
                 __hiloint2double(vv3.y,vv3.x), __hiloint2double(vv4.y,vv4.x)),
catRomFiltera(fx,__hiloint2double(vv5.y,vv5.x), __hiloint2double(vv6.y,vv6.x),
                 __hiloint2double(vv7.y,vv7.x), __hiloint2double(vv8.y,vv8.x)),
catRomFiltera(fx,__hiloint2double(vv9.y,vv9.x), __hiloint2double(vv10.y,vv10.x),
                 __hiloint2double(vv11.y,vv11.x), __hiloint2double(vv12.y,vv12.x)),
catRomFiltera(fx,__hiloint2double(vv13.y,vv13.x), __hiloint2double(vv14.y,vv14.x),
                 __hiloint2double(vv15.y,vv15.x), __hiloint2double(vv16.y,vv16.x)));}

__global__ void panelsim2d_catmull_rom_kernel(cudaTextureObject_t texObj_bad,
                  cudaTextureObject_t texObj_good, double K, double *inarr,
                  unsigned char *idios, int N, int ngrid2,
                  int ngrid, double xmin, double xmax,
                  double ymin, double ymax, double theta)
{
   unsigned int xi = blockIdx.x * blockDim.x + threadIdx.x;

   if (xi < N) {
      double xv = ((K - xmin)/(xmax - xmin)) *
                  (double) (ngrid2 - 1) + 0.5;
      double yv = (pow((inarr[xi] - ymin)/(ymax - ymin),(1.0/theta)) *
                  (double) (ngrid - 1)) + 0.5;
      unsigned char empstatb = idios[(xi/8)] & (128>>(xi%8));

      double val = empstatb ? clampd(tex2DCatRoma(texObj_good, xv, yv),
                                     ymin, ymax) :
                              clampd(tex2DCatRoma(texObj_bad, xv, yv),
                                     ymin, ymax);
      inarr[xi] = val;
   }
}
```

Supplementary Appendix (Online publication only)

B.1 Model description and calibration

The economy consists of a large number of households indexed by i. Agents face an idiosyncratic shock ϵ which can take on two different values: $\epsilon = 1$ (employed) and $\epsilon = 0$ (unemployed). An employed agent receives an after-tax wage $(1 - \tau_t)w_t$. An unemployed agent receives unemployment benefits μw_t. Investment in capital yields a return $r_t - \delta$, where r_t is the rental rate on capital and δ is the depreciation rate. The utility maximization problem of agent i is

$$\max_{c_t^i, k_t^i} U^i = E \sum_{t=0}^{\infty} \beta^t \frac{(c_t^i)^{1-\gamma} - 1}{1 - \gamma} \tag{1}$$

subject to:

$$c_t^i + k_t^i = r_t k_{t-1}^i + \left[(1 - \tau_t)\bar{l}\epsilon_t^i + \mu(1 - \epsilon_t^i)\right] w_t + (1 - \delta)k_{t-1}^i \tag{2}$$

$$k_t^i \geq 0 \tag{3}$$

where c_t^i is consumption, k_t^i is end-of-period capital, and $\bar{l}$ is the time endowment. The first-order condition for this problem is

$$U_c(c_t^i) = \beta E_t \left[U_c(c_{t+1}^i)(1 + r_{t+1} - \delta)\right] \tag{4}$$

A perfectly competitive firm uses capital and labour to produce output using a Cobb-Douglas production function. Let K_t and L_t denote capital per capita and the employment rate, respectively. Output per capita is given by

$$Y_t = a_t K_{t-1}^{\alpha} (\bar{l} L_t)^{1-\alpha} \tag{5}$$

where a_t is an aggregate productivity shock, which takes on two different values: $1 - \Delta_a$ (bad state) and $1 + \Delta_a$ (good state).

The firm maximizes profits, so the wage rate paid to labour and rental rate on capital are equal to their marginal products:

$$w_t = (1 - \alpha) a_t \left(\frac{K_{t-1}}{\bar{l} L_t}\right)^{\alpha} \qquad r_t = \alpha a_t \left(\frac{K_{t-1}}{\bar{l} L_t}\right)^{\alpha - 1} \tag{6}$$

Since these factor prices are functions of economy-wide capital and labour and the aggregate state, they are treated as given by individual agents.

The government taxes the wage income of employed agents and distributes the proceeds to the unemployed. Accordingly, the tax rate is

$$\tau_t = \frac{\mu u_t}{\bar{l} L_t} \tag{7}$$

where $u_t = 1 - L_t$ is the unemployment rate.

The aggregate productivity shock a and the idiosyncratic employment shock ϵ follow first-order Markov processes. As in Den Haan and Rendahl (2010), transition probabilities are calibrated so that the unemployment rate takes on only two values:: $u_b = u(1 - \Delta_a)$ (bad state) and $u_g = u(1 + \Delta_a) < u_b$ (good state).

If we define $y_t^i \equiv r_t k_{t-1}^i + \left[(1 - \tau_t)\bar{l}\epsilon_t^i + \mu(1 - \epsilon_t^i)\right] w_t + (1 - \delta)k_{t-1}^i$and use (2), the first-order condition for the individual-agent problem, (4), becomes

$$U_c(y_t^i - k_t^i) = \beta E_t \left[U_c(y_{t+1}^i - k_{t+1}^i)(1 + r_{t+1} - \delta)\right] \tag{8}$$

The individual-agent policy function will take the form $k_t^i = g(k_{t-1}^i, K_{t-1}, \epsilon_t^i, a_t)$. Individual income can be represented as $y_t^i(k_t^i, K_t, \epsilon_t^i, a_t)$ and the return on capital as $r_t(K_t, a_t)$. It follows that (8) can be written in the following form where dependence on shocks is suppressed for simplicity:

$$\begin{aligned} U_c\left(y_t^i(k_{t-1}^i, K_{t-1}) - g(k_{t-1}^i, K_{t-1})\right) = \\ \beta E_t \left[U_c\left(y_{t+1}^i(k_t^i, K_t) - g(k_t^i, K_t)\right)(1 + r_{t+1}(K_t) - \delta)\right] \end{aligned} \tag{9}$$

Equation (9) shows that the optimal capital choices of agents depend on two endogenous state variables: aggregate and individual capital. Consequently, standard 2-dimensional grid-based techniques can be used to solve this problem.

We follow Maliar et al. (2010) in using Euler equation iteration to solve for the individual agent policy function with a uniform grid for aggregate capital and an individual capital grid that follows a simple polynomial rule. We use the same calibration as in Den Haan and Rendahl (2010):

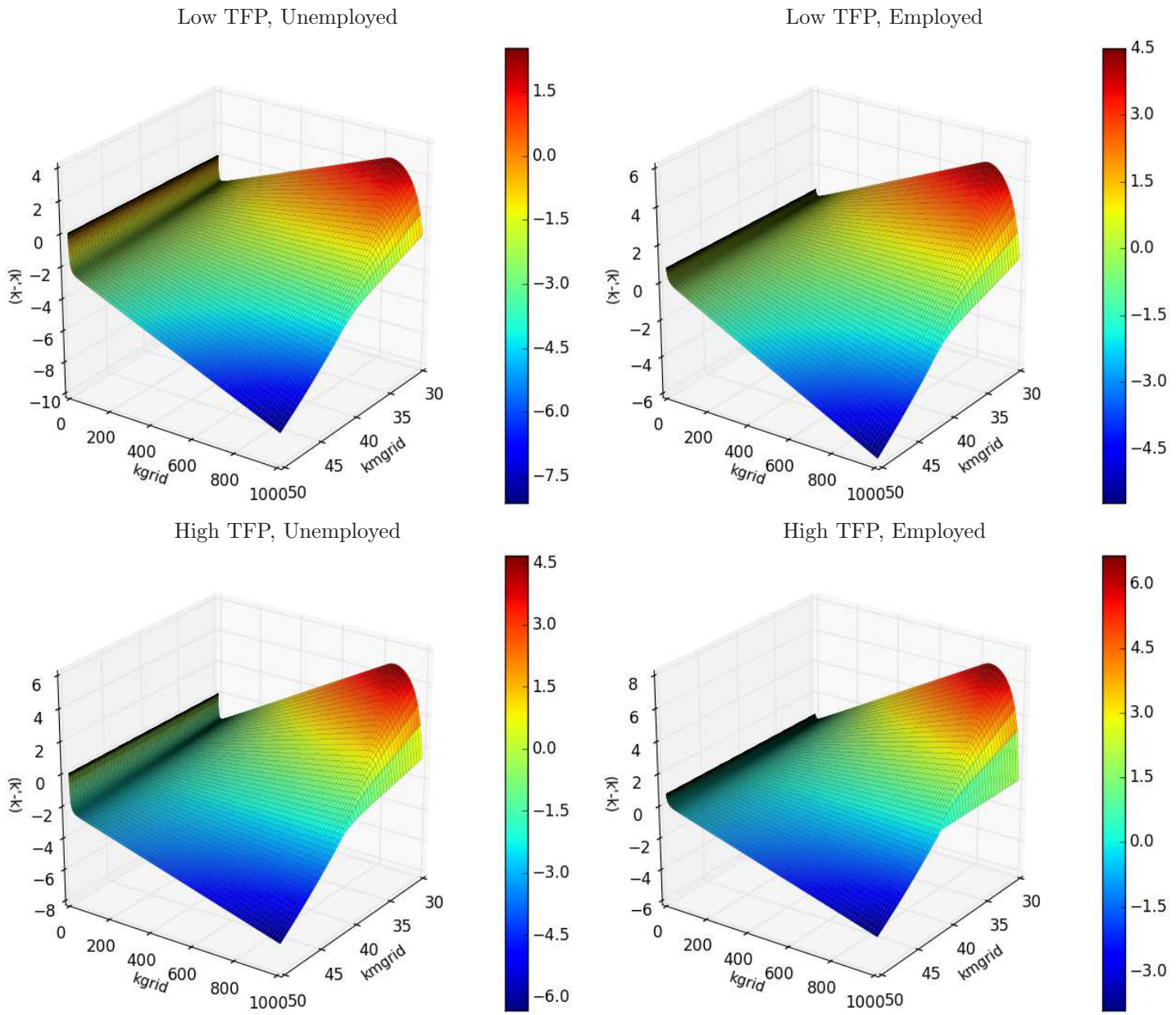
Table B.1: Calibration of KS economy

Parameters	β	γ	α	δ	$\bar{l}$	μ	Δ_a
Values	0.99	1.0	0.36	0.025	1.0/0.9	0.15	0.01

s,ϵ/s',ϵ'	$1 - \Delta_a, 0$	$1 - \Delta_a, 1$	$1 + \Delta_a, 0$	$1 + \Delta_a, 1$
$1 - \Delta_a, 0$	0.525	0.35	0.0.3125	0.09375
$1 - \Delta_a, 1$	0.03889	0.836111	0.002083	0.122917
$1 + \Delta_a, 0$	0.09375	0.03125	0.291667	0.583333
$1 + \Delta_a, 1$	0.009115	0.115885	0.024306	0.859604

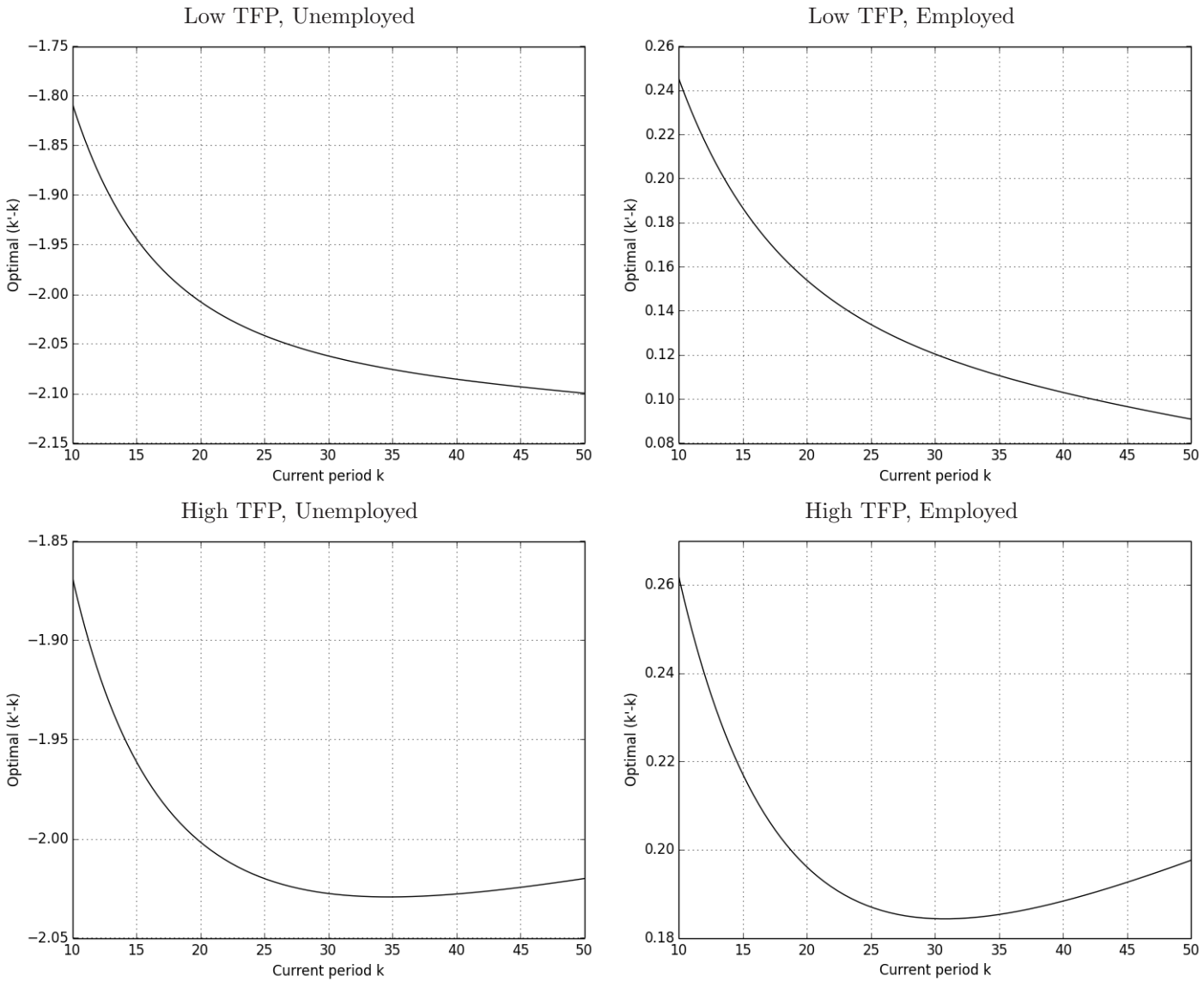
Note: Calibration matches that in Den Haan et al. (2010).

Figure B.1: Optimal individual agent policy function surfaces $(k' - k) = h(k, K)$



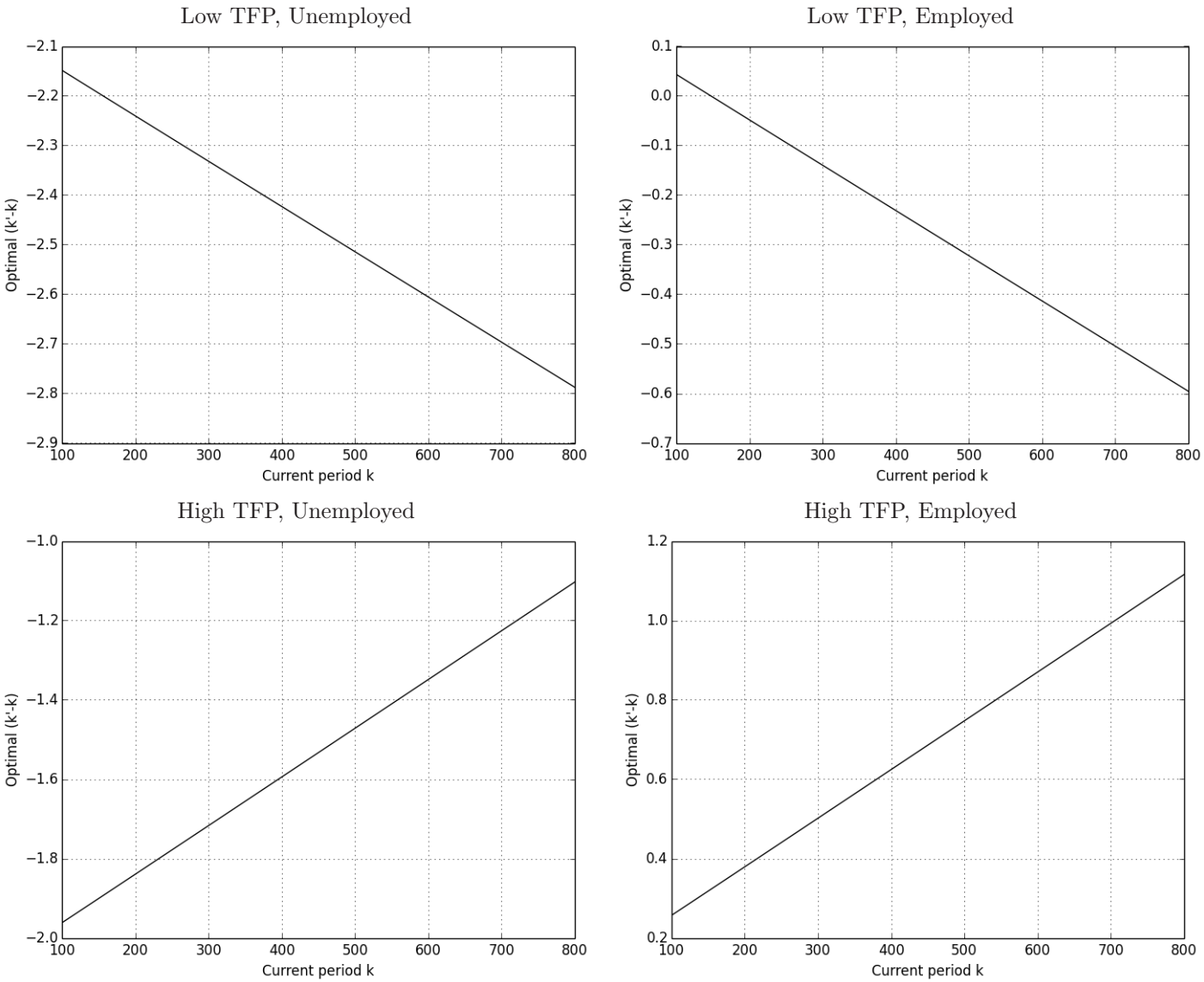
Note: The optimal policy function surfaces plotted here replicate the results presented in Horvath (2012). The optimal policy surfaces shown here are based on a discretization scheme employing 500 grid points in the individual agent and 50 grid points in the economy-wide capital dimension.

Figure B.2: $(k' - k) = h(k, K = 39)$ for $k \in (10.0, 50.0)$(poor)



Note: The optimal policy functions plotted here replicate the results presented in Horvath (2012).

Figure B.3: $(k' - k) = h(k, K = 39)$ for $k \in (100.0, 800.0)$(wealthy)



Note: The optimal policy functions plotted here replicate the results presented in Horvath (2012).