Agent-Based Economic Models in OpenCL

Dan Greenwald and Kevin Mullin *

December 28, 2012

1 Introduction

Our project is to simulate the macroeconomic behavior of an economy with incomplete asset markets using OpenCL on a GPU. A complete description of the environment and the algorithm that we use can be found in Sections 4 through 6. In general, the model is based on that of Krusell and Smith (1998) adapted so that a risk-free bond is traded, and there is no capital in the economy.

Overall we were able to obtain dramatic gains from parallelization in general, but also from moving from CPU to GPU computation. These gains were more dramatic for the solution algorithm than for the simulation algorithm, and were quite small for some scales, but showed speedup of nearly 1000 times for scales with large solution grids.

Our code is available on github under the repository

`git@github.com:dgreenwald/hpc12-fp-dlg340-kpm303.git`

and is intended to be free for anyone to use.

The write-up is organized as follows. The rest of Section 1 contains a summary of the project, notes on the relevant scale of the problem, instructions for running, and the division of labor. Section 2 contains the economic background for our problem. Section 3 describes the mathematical model we will be using. Sections 4 through 6 describe the computational algorithm. Section 7 contains the timing results. We conclude in Section 8.

*Dan Greenwald can be reached at daniel.greenwald@nyu.edu. We thank Andreas Kloeckner and Marsha Berger for doing a great job teaching such useful material, and especially Prof. Kloeckner for taking the time to debug the code when we were completely stuck.

1.1 Summary of Project

The problem contains two major components. The first is to solve for the agent's optimal consumption policy given his or her current state. This involves interating a functional operation on function approximations defined on a grid. Since the calculations for each gridpoint are independent, the algorithm is highly parallelizable. Each iteration involves solving an equation derived from the model's optimality conditions. This can be done without resorting to use of a nonlinear equation solver by use of the "endogenous grid method." The main computational challenge is to move back from the endogenous grid to the original grid, because it involves interpolation on an unknown grid, of which only a portion is held by each work-group.

The second component is to use the optimal policies to simulate the economy for a large number of agents over a large number of periods, in order to obtain a simulation of macroeconomic behavior. This type of simulation could be used to perform experiments, like studying the effects of a policy change. Alternatively, this type of simulation could be used for parameter estimation, which would seek to minimize the distance between simulated moments of macroeconomic variables and their counterparts in the actual data. The main computational challenge in this step is to set bond prices exactly so that markets clear (i.e. so that total saving equals total borrowing), which involves simulating the economy in each period for various guesses of the bond price until market clearing is attained.

Throughout the above steps, the agents use a forecasting rule to generate their expectations of the bond price in the future, based on the future macroeconomic state. In order that this expectation be unbiased, the above algorithm is iterated using guesses of the policy rule until the forecasting rule is within tolerance of the relevant sample means.

1.2 Scale of the Problem

The scale of the project that we are aiming at for the solution portion of the model is to use grids of between a few hundred and a few thousand gridpoints each for the continuous variables (x_i, q). This means approximating the optimal policy function on $4N_xN_q$ gridpoints.

At the high end, the scale of the solution step is limited by the available amount of memory for GPU computation, since two approximate functions must be stored on the device at all times, taking up $32N_xN_q$ bytes of space at double precision. At the low end, the ability to parallelize across gridpoints makes this procedure economical for nearly any scale.

The scale of the project that we are aiming at for the simulation portion of the model is to use

a few thousand agents over thousands of periods, for example, 5,000 agents over 12,000 periods (of which 2,000 is burn-in).

At the high end, the limitation is again the available amount of memory for GPU computation, as the program is currently written, since the simulation output arrays must hold $24N_{sim}N_t$ bytes on the device. At the low end, since parallelization can only occur across agents in the simulation, and not across time, this algorithm may not yield speedup for very small values of N_{sim}.

1.3 Distribution of Labor

The division of labor for the project was as follows: Dan Greenwald derived the simulation algorithm, and prepared the slides and writeup. Kevin Mullin derived the solution algorithm.

1.4 Instructions

A makefile is included in the code directory that should build all the relevant code. It is necessary to have OpenCL installed in order to run the code. The main file is `solve.c`, which in turn calls OpenCL kernels contained in the file `solve.cl`. The program `solve.c` can be run without any arguments, in which case default arguments are used. Alternatively, you can use the syntax

```
./solve [Nx] [Nq] [Nsim] [Nt] [use_gpu]
```

Where the first four variables are the desired values of Nx, Nq, N_{sim}, and N_t, respectively, and the last value is whether you want to run the algorithm on an NVIDIA GPU (if equal to 1) or on an AMD CPU platform (if equal to 0). A commented-out line of code in `solve.c` allows you to replace this with an Intel CPU platform if desired.

Inside of `solve.c`, the macro `OUTPUT`, determines whether the program saves the simulation results arrays to the directory in binary files. Set `OUTPUT` to 1 to save in this fashion, or to 0 to omit saving (e.g. for evaluating timings).

Two shell scripts were used to obtain the timing results, `timings.sh` and `timings_cpu.sh`, for the GPU and CPU results, respectively. To run the GPU shell script on the Bowery cluster, you can use the PBS file `job.pbs`, using the command `qsub job.pbs` which will set up a session on the appropriate `cuda` queue.

Finally, the Matlab script `results.m` was used to create the graphs and output parts of the tables used in the writeup.

2 Economic Background

This section describes the economic problem being solved in relatively theoretical terms. The reader interested only in the mathematical nuts and bolts of the model should skip to Section 3. The reader interested only in the computational algorithm should skip further, to Section 4.

2.1 Macroeconomic Modeling

Macroeconomics is the subfield of economics that deals with outcomes for an entire economy, as opposed to a single market. A standard macroeconomic model will seek to describe an economy in which prices adjust so that demand equals supply, and such that the interest rates on financial assets adjust so that the quantity of financial assets sold equals the quantity of financial assets purchased.

For several decades, the trend has been to develop "microfounded" models, in which macro-level behavior like total consumption is derived by considering the consumption decision of individual economic agents, and then aggregating over many agents' actions to obtain an overall result. These microfoundations are designed to keep macroeconomic models closer in line with reality, and provide testable checks on the theories macroeconomics propose (i.e., if your model is correct, then X, Y, and Z should be observed in micro-level data).

However, the task of aggregating over many individual decisions can impose serious mathematical difficulties. In particular, macroeconomic outcomes may depend on the entire distribution of individual states and characteristics across individuals. For example, the behavior of an economy with a large degree of wealth inequality may differ from one with less inequality, even if the two economies exhibit the same average level of wealth. Therefore, modeling the macroeconomy using a microfounded model may depend on keeping track of entire distributions for each variable — infinite-dimensional objects that are difficult to work with numerically.

These obstacles have often been overcome through the use of simplifying, but unrealistic, assumptions, to ensure that the distributions of states across agents do not matter, and that the overall state of the macroeconomy can be summarized in a few aggregate statistics. These assumptions often take the form of perfect insurance markets, in which agents can insure against any possible event that may occur. Since agents do not like risk, they generally insure away all their individual risk, so that their behavior only depends on aggregate conditions — allowing for easy aggregation.

2.2 The Heterogeneous-Agent Approach

While these assumptions have allowed for many years of productive macroeconomic research, they miss major features of the choices facing most individuals. In reality, people face many forms of uninsurable risk such as unexpected changes to wages or unemployment. In addition, most people only have access to a limited set of financial instruments with which to invest, and often face strict borrowing limits, especially on unsecured debt, which they cannot exceed.

Incorporating these features into a microfounded macroeconomic model leads to what is typically known as a "heterogeneous-agent" model, in which micro-level differences between agents are important, and the entire distribution of individual states must be accounted for. This leads to three kinds of problems, all of which are typically computationally intensive to overcome. A good example of this type of model, on which the model for this project is based, is that of Krusell and Smith (1998), although there are many other examples (see, e.g., Guvenen (2011) or Heathcote, Storesletten and Violante (2009) for surveys).

2.2.1 Solution

The first issue is that removing perfect insurance markets yields a much more complicated individual problem, in which agents must carefully consider the risks posed by uninsurable fluctuations at all times in the future. The optimal policies typically can only be calculated numerically, and standard grid-based approximations of policy functions are subject to the "curse of dimensionality" as the number of state variables increases, leading to large computational burden for all but the most simple models.

2.2.2 Simulation

The second problem posed by a heterogeneous-agent model is that macroeconomic behavior now depends on the entire distribution of individual states across agents. Therefore, analyzing the behavior implied by the model typically involves simulating the behavior of thousands of agents, and mechanically aggregating to obtain macroeconomic results over thousands of time periods. If prices must be set so that markets in goods or financial assets must clear, then each period may need to be simulated many times as an algorithm finds the correct price. Therefore, simulation can be a computationally intensive step even if the underlying model is very simple and easy to solve.

2.2.3 Forecasting

The final problem is that the agents in a realistic model are forward looking, meaning that agents have to have some way of forecasting what they expect to occur in the future based on current conditions. When current conditions are determined by an antire distribution across agents, then it is a challenging task to translate that object into a reasonable forecast. Instead, economists usually assume that agents use some forecasting rule based on aggregate variables. But to be realistic, these rules can lead to forecasting error, but should not be particularly biased (i.e. everyone should not always forecast incorrectly in the same way). But since behavior depends on the forecasting rule, and the bias of the forecasting rule depends on behavior, this leads to a "fixed-point" problem that may involve running the solution and simulation steps multiple times, adding to the computational burden.

2.2.4 Computational Performance

Despite these challenges, many of the computations involved in solving these models are highly parallelizable, and should therefore yield massive speedup relative to a serial computation. In particular, solving and simulating these models often requires the type of repeated simple calculation that is perfect for GPU computation. This should allow not only for convenience allowed by the reduction in wait times, but much more important, allow for more complex models or more accurate solutions to become tractable.

3 The Model

This next section will summarize the economic model that is being solved, which will be the foundation for the computational routine. The reader interested in the computational routine only should skip ahead to Section 4.

3.1 Environment

Time is modeled as a sequence of discrete periods, indexed by t (although I will typically suppress the dependence of variables on t in the notation). The economy is populated by a continuum of infinitely-lived agents indexed by i. Agents earn labor income based on their own individual employment state, and the overall state of the macroeconomy. The employment state for agent i is denoted e_i, with $e_i = 1$ when the agent is employed, and $e_i = 0$ when the agent is unemployed.

The macroeconomic state is denoted z, with $z = 0$ when the economy is in a low-productivity ("recession") state, and $z = 1$ when the economy is in a high-productivity ("expansion") state. Define $s_i = (z, e_i)$ to be the overall state for agent i, which can take on four possible values, and let $y(s_i)$ be the function (identical across individuals) that translates the states into an individual's labor income. We will assume that s_i follows a Markov chain for each agent with transition matrix P.

In practice, we parameterized the z values, the P matrix, and the y function following Krusell and Smith (1998), which in turn chooses these values to match various empirical features of the macroeconomy. Unlike Krusell and Smith, however, I chose the y function so that agents receive 1/3 of their employed income in the unemployed state, which roughly corresponds to U.S. unemployment insurance.

3.2 Preferences

In each period, agents consume resources. Agents choose state-contingent paths of lifetime consumption in each period t to maximize

$$V_{it} = \mathbb{E}_t \sum_{j=0}^{\infty} \beta^j u(c_{i,t+j})$$

where V is the present discounted value of lifetime consumption, $\mathbb{E}_t$ is the mathematical expectations operator conditional on time t information, β is the discount factor (which determines the agent's level of patience), u is a utility funciton representing the benefit an agent gets from some level of consumption in a given period, and c is consumption. Further, V and c should be interpreted as functions that may depend on the state at time t and $t + j$, respectively. We will restrict attention to problems where u is continuously differentiable and strictly concave.

3.3 Assets

Agents can save and borrow from each other by holding positive and negative positions in a one-period riskless bond. Denote agent i's holdings of the bond by b_i. Each agent can buy (or sell) one unit of the bond at price q, in which case they receive (or pay) one unit of consumption in the following period. This is equivalent to an interest rate of $r = 1/q$. Since there is no outside source of funds, in each period q must be set so that total saving equals total borrowing, which is known as "market clearing." Mathematically, this can be expressed by the condition

$$\int b_i \, di = 0$$

which must hold in each period.

3.4 Agent's Problem

Each agent enters a given period with wealth x, where x is the amount of the consumption good that the agent is due from his or her previous bond purchases ($x_{it} = b_{i,t-1}/q_{t-1}$). Each agent solves the problem

$$V(x_i, q, s_i) = \max_{c_i, b_i} u(c_i) + \beta \mathbb{E}\left[V(b_i, q', s_i') \middle| s_i\right]$$

subject to

$$c_i + q b_i = x_i + y(s_i)$$

$$c_i \geq 0$$

$$b_i \geq -B$$

where primes (i.e. x_i') represent values of variables in the following period.[1]

Given our earlier assumptions on the utility function, the agent's optimal policy $c(x, q, s_i)$ is uniquely defined by the first order condition

$$q u'(c(x_i, q, s_i)) \geq \beta \mathbb{E}\left[u'(c(x_i', q', s_i')) \middle| s_i\right] \tag{3.1}$$

which must hold with equality for $b_i > -B$, where $b_i = x_i + y(s_i) - c_i$. The goal of the solution algorithm will be to solve for this optimal policy function c up to a close approximation.

4 Optimal Policy Algorithm

4.1 Strategy and Approximation

The first computational task is to find the unique function $c(x_i, q, s_i)$ that solves the optimality condition (3.1). In general, the solution will proceed by starting with some guess of the policy function c^0, and then on each step, choosing c^{n+1} to satisfy

$$q u'(c^{n+1}(x_i, q, s_i)) \geq \beta \mathbb{E}\left[u'(c^n(x_i', q', s_i')) \middle| s_i\right] \tag{4.1}$$

for all (x_i, q, s_i), which again must hold with equality for $b_i > 0$.

[1] Hopefully this is not confusing when primes are also used for derivatives.

Since x_i and q are continuous variables, the function c is an infinite-dimensional object, we must approximate it to be able to solve numerically. For this project, we use a simple approximation by choosing grids of points $(\bar{x}_1, \ldots, \bar{x}_{N_x})$ and $(\bar{q}_1, \ldots, \bar{q}_{N_q})$, and approximate c by defining it at all combinations of these gridpoints and the state s_i. We can then use bilinear interpolation over the (x_i, q) dimension (holding s_i fixed) to evaluate the function.

For reasons that will become clear as the algorithm progresses, we implement this algorithm in OpenCL using work-groups of size $(K_x, 1, N_s)$, with $K_x \leq N_x$.

4.2 Endogenous Grid Method

A simple way to perform an iteration of (4.1) would be to use a nonlinear equation solver for each point $(\bar{x}_j, \bar{q}_k, \bar{s}_l)$ on the grids defined earlier. However, a nonlinear equation solver will typically require many function evaluations to find a solution, in addition to the added complexity of the inequality constraint, making it a relatively slow and complicated method to implement

A better method, assuming that the function u' has a known inverse, is to take advantage of the fact that the left side of (3.1) can be inverted in closed form (although the right side cannot). Along these lines, the improved method is to define a grid over bond holdings, $(\bar{b}_0, \ldots, \bar{b}_{N_b-1}$, and solve for the right hand side of (3.1) given $(\bar{b}_j, \bar{q}_k, \bar{s}_l)$. To perform this task efficiently in parallel, each work item with, say, global id $(p, 1, r)$ calculates $u'(\bar{b}_p, \tilde{q}(\bar{s}_r), \bar{s}_r)$ and stores the resulting value in local memory. Each work-item, again with arbitrary global id $(p, 1, r)$, now sums over the terms $P(r, r')u'(\bar{b}_p, \tilde{q}(\bar{s}_{r'}), \bar{s}_{r'})$ for $r' = 1, \ldots, N_s$, using the previous results. This is the logic for choosing the s dimension of each work-group to be equal to N_s.

With the expectation terms in hand on our $\bar{b}$ grid, we can then solve for c_i for each combination $(\bar{b}_j, \bar{q}_k, \bar{s}_l)$ using the relation

$$c_i = u'^{-1}\left\{\beta\bar{q}_k^{-1}\beta\mathbb{E}\left[u'(c^n(\bar{b}_j, \tilde{q}(s_i'), s_i'))\middle| s_i\right]\right\}$$

which allows us to solve (3.1) in one step. Note that we do not have to worry about (3.1) holding with equality because at most one point on our $\bar{b}$ grid is equal to $-B$. Define c_j^* to be the value obtained by solving (3.1) for $b_i = \bar{b}_j$. Then we can return to a mapping between x_i and c_i by using the definition $x_i = c_i + b_i - y(s_i)$ to solve for the implied starting wealth values x_j^* given c_j^* and $\bar{b}_j$.

Applying this algorithm on a given iteration involves many evaluations of c^n using bilinear interpolation. As long as we have chosen our $\bar{x}$ and $\bar{q}$ grids such that the mappings between some

value x_i and the nearest lower neighbor $\bar{x}_j$ can be easily evaluated, the bilinear interpolation procedure involves only a few floating point calculations, requires access to only four points of data, and is in general not a major computational burden. In particular, we used polynomially spaced grids

$$\bar{x}_j = x_{min} + (x_{max} - x_{min}) \left(\frac{j}{N_x - 1} \right)^{k_x} \bar{q}_j = q_{min} + (q_{maq} - q_{min}) \left(\frac{j}{N_q - 1} \right)^{k_q}$$

with $k_x = 0.4$ (more points for x_i small) and $k_q = 1$ (evenly spaced).

4.3 Recovering Original Grid

The method of Section 4.2 established an efficient way to solve (3.1) over a grid of values for bond holdings b_i, establishing a mapping between x_j^* and c_j^* for each $\bar{b}_j$. However, in order to be able to perform bilinear interpolation in the next iteration, we need to return to a uniform grid system for the x_i variable that does not depend on j. In particular, we will return to our original grid system via linear interpolation.

For fixed $\bar{q}_k$ and $\bar{s}_l$, we want to be able to evaluate $c^{n+1}(\bar{x}_j, \bar{q}_k, \bar{s}_l)$ for each point in our $\bar{x}$ grid. We can do this by finding the relevant values of x_j^* such that $x_m^* \leq \bar{x}_j \leq x_{m+1}^*$, and then interpolating accordingly between c_m^* and c_{m+1}^*.

There are two related challenges involved in this operation. The first is that the x^* grid is unknown (since it is a product of the algorithm) and has no known mapping to immediately obtain the correct bin given the value $\bar{x}_j$. Therefore, we will have to search for the correct bin before interpolating. The second challenge is that each work-group holds only a subset of the points of the x^* grid for a given $(\bar{q}_k, \bar{s}_l)$ if the x dimension is at all large. Since we cannot synchronize across work-groups, a second challenge is finding which work group should perform the interpolation for each point in the $\bar{x}_i$ grid.

We solved these problems by having each work-group load every element of the $\bar{x}$ grid, one $K_x \times 1$ sized block at a time. One element of each block is assigned to each work-item, and that work-item then checks whether that value of $\bar{x}_j$ falls in that work-group's subset of x^* points, which is easily done since the x_j^* points are monotonically increasing in j. If the value of $\bar{x}_j$ is not within that subset, the work-item does nothing and proceeds to load its entry from the next block. If the value of $\bar{x}_j$ is within that subset, the work-item searches for the relevant bin (in local memory) using a simply bisection algorithm, and then performs the linear interpolation. Once this is done, the work-item updates the relevant value of $c^{n+1}(\bar{x}_j, \bar{q}_k, \bar{s}_l)$ in global memory, and

moves on to the next block.

The final complication is that points on the $\bar{x}$ grid may fall below the bottom of the x^* grid. If we set $\bar{b}_0 = -B$, as we do in practice, then we know that these points must be in the constrained region with $b_i = -B$, and can correspondingly set $c_i = x_i + y(s_i) + B$.

4.4 Summary

This completes the algorithm to move from c^n to c^{n+1}. To summarize, the algorithm proceeds in the following steps.

1. Evaluate $\mathbb{E}_t u'(c^n(\bar{b}_j, \tilde{q}(\bar{s}_l), \bar{s}_l))$ forreach (j, l).
2. Invert (3.1) to construct a (x_j^*, c_j^*) mapping conditional on $\bar{b}_j$.
3. Load $\bar{x}$ block-by-block into each work-group, checking if points fall in the relevant x^* values, and interpolating if so to recover c^{n+1}.

The algorithm proceeds until the maximum distance between elements of c^n and c^{n+1} falls below some tolerance.

4.5 Implementation in OpenCL

Most of the specifics of how each work-group and each work-item is assigned has already been described in the preceding sections, so this section will focus on performance and parallelization issues.

It should be clear that all steps of this algorithm can be run in parallel, with only a few local synchronizations required. However, because of the need to consider the entire list of $\bar{x}$ points in each work-group, this algorithm will not scale linearly, and will lose efficiency the larger the $\bar{x}$ grid is compared to the local size K_x. In practice, we found that even for relatively large values of N_x, the cost of checking irrelevant values of the $\bar{x}$ grid appears tolerable. A further challenge is that depending on where points on the $\bar{x}$ grid are concentrated relative to the x^* grid, and so the work done by the various work groups in recovering the original grid may be unbalanced, and it is possible that the imbalance will increase with the number of $\bar{x}$ points. But overall the algorithm is highly parallelizable, and should exhibit massive speedup from running in parallel relative to in serial.

5 Simulation Algorithm

5.1 Overall Strategy

In order to determine macroeconomic behavior in a heterogeneous agent model, simulation is required. While other methods exist, for example keeping track of weights on a histogram, we use the method of simply simulating the paths of a large number of agents. The main computational challenge in this section is determining the bond price in each period of the simulation that makes markets clear.

A sample of some agents' simulated paths can be seen in Figure 1. Note that while the bond price makes large moves associated with the macroeconomic state (as in the agents' forecasting rules), it displays jitters associated with the specific market-clearing conditions, which correspond to forecasting error on the part of the agents. In general, it can be seen from the simulations that agents tend to build up a small buffer of assets (relative to their borrowing limit) while employed, which they then spend down when unemployed, so as not to have a drastic fall in consumption.

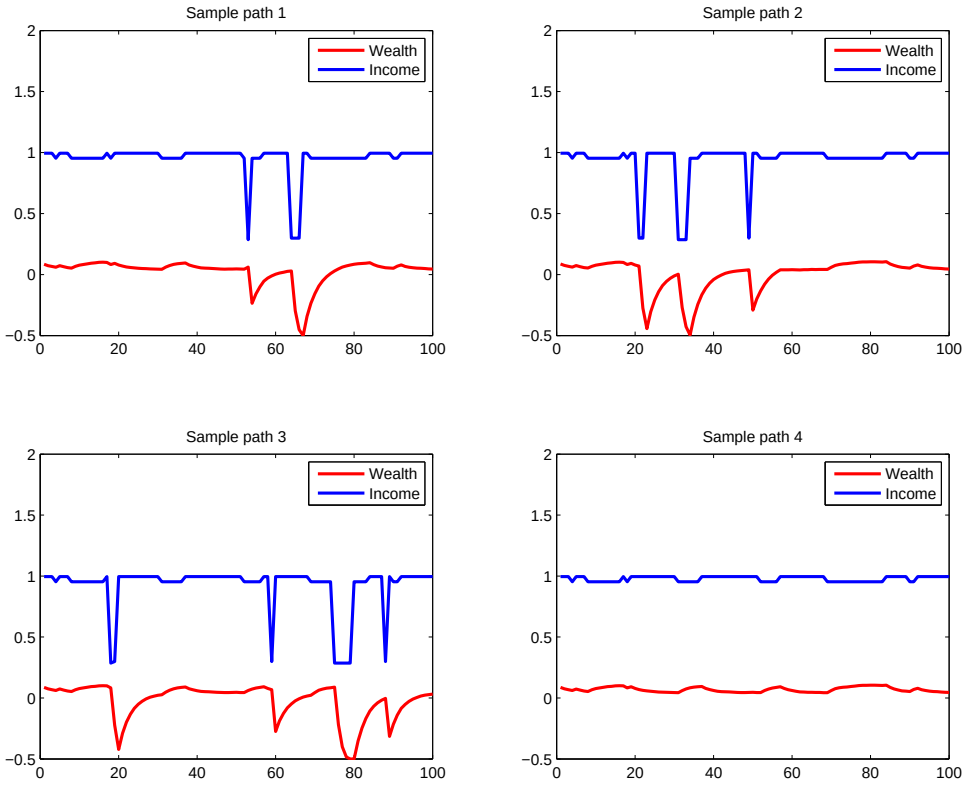
To initialize the simulation, we initialize agents with zero wealth, and assign some random previous states to each agent and to the overall economy. The initial condition can be relatively arbitrary, as we will use a long "burn in" period for which we discard the simulation results to ensure that the simulation has had time to lose its dependence on initial conditions.

Once the simulation is initialized, we update the simulation recursively, period-by-period. At the start of each period t, the previous period's simulation gives us the starting wealth x_{it} for each agent. We then draw the overall state z_t conditional on z_{t-1}, and then draw ε_{it} for each i conditional on $\varepsilon_{i,t-1}$, z_{t-1}, and z_t.

Given a guess for q_t, the bond price at time t, we evaluate agents' optimal consumption using the c function obtained from the methods of Section 4. Given the resulting values of c_{it}, we can calculate the implied bond holdings using $b_{it} = x_{it} + y_{it} - c_{it}$, and sum over agents to get aggregate bond holdings, $\sum_i b_{it}$.

If this sum is within some tolerance of zero, we are done for this period and move on. Otherwise, we adjust the guess of q_t and repeat the procedure until market clearing is obtained. In practice, we used a bisection scheme to solve for the equilibrium value of q_t, using the fact that $\sum_i b_{it} > 0$ when q_t is too low, and $\sum_i b_{it} < 0$ when q_t is too high.

Figure 1: Simulated Paths



5.2 Implementation in OpenCL

To implement this in OpenCL, we assigned each work-item to calculate the optimal policy for a single agent. This was performed using bilinear interpolation as usual. However, since the number of bilinear interpolation calculations over a large simulation of thousands of agents over thousands of periods is likely to be large relative to the number of gridpoints on which the c function is defined, we pre-calculated the interpolation coefficients over the entire grid on the device using an OpenCL kernel.

The only other complication with implementation in OpenCL is the need to evaluate the term $\sum_i b_{it}$, since the i indices are distributed across many work-groups. To address this, we used a reduction method from the book *OpenCL in Action* (2011), which takes a vector of entries in local memory and recursively adds the top half to the bottom half to efficiencly add b_{it} over each work-group. Because there is no global synchronization, we then wrote the sum for each work-group to global memory, and then called a second kernel to perform the same reduction

over the work-group sums, since reasonable numbers of simulations do not require more than two reductions.

Once the overall sum is calculated, it is transferred over to the host, which then determines whether the sum is within the desired tolerance, and if not, launches the entire routine again with a new value of q_t. In this way, the algorithm only requires that a single scalar be transferred between the host and device (once in each direction) on each iteration.

5.3 Summary

Once the market clearing value q_t is obtained, we then update time $t+1$ starting wealth using $x_{i,t+1} = b_{it}$, and continue on in the same fashion. The algorithm can therefore be summarized as follows.

1. Initialize agents starting wealth x_{i0}, previous employment states $\varepsilon_{i,-1}$, and previous macro state z_{-1}, and set $t = 0$.
2. Draw z_t given z_{t-1}.
3. Draw ε_{it} given z_t, z_{t-1} and $\varepsilon_{i,t-1}$.
4. Initialize a guess for q_t.
5. Calculate c_{it} and b_{it} for each agent using the optimal policy function.
6. If $\sum_i b_{it}$ is within tolerance of 0, proceed to Step 7, otherwise update the guess of q_t and return to Step 5.
7. Update $x_{i,t+1} = b_{it}$, increment $t = t + 1$, and return to Step 2.

The final step, once the simulated values have been calculated for each t, is to discard some portion of the initial observations as "burn-in" so that the resulting sample is not dependent on initial conditions.

Overall, this algorithm is completely parallel when calculating agents' policies, and highly parallel in the reduction step,although some work items are idle at some times during the reduction. Further, there is little overhead in terms of transferring data between host and device until the computation is complete, and there are relatively many floating point operations relative to memory operations, so overall we expected substantial speedup from parallelizing this step as well.

6 "Meta" Algorithm for Calculating $\tilde{q}$

Throughout Section 4 and Section 5, we assumed a forecasting rule for $\tilde{q}$, and the resulting optimal policy solution and simulation depend on our guess for this rule. However, in principle there is good reason to want this rule to be at least unbiased, so that agents' estimates are not perpetually too low or too high.

Since our forecasting rule was based on the aggregate state z, this is equivalent to demanding that $\tilde{q}(\bar{z}_j)$ be equal to the sample average of values q_t in periods in which $z_t = \bar{z}_j$. To solve this fixed point problem, we begin with a guess for $\tilde{q}$, and run the algorithms described in Section 4 and Section 5. We then evaluate the sample means for each possible z_t state, and update $\tilde{q}(\bar{z}_j)$ for the next iteration to be equal to this sample mean. With this new value of $\tilde{q}$ in hand, we proceed to the next iteration, and continue until the error between $\tilde{q}$ and the sample means falls within tolerance.

For this method, it is essential that the random draws of ε_{it} and z_t be kept constant across iterations, otherwise the algorithm may not converge. For computational efficiency, it is important that each calculation of the optimal policy in Section 4 begin with the solution from the previous iteration, which is likely to be very close to the new solution, and will substantially speed convergence time.

7 Timings

7.1 GPU Timings

The GPU timings are summarized in Tables 1 and 2, and in Figures 2 and 3.

There are four ways in which the scale of the problem can be changed, by adjusting the number of x gridpoints (N_x) or the number of q gridpoints (N_q) in the solution algorithm, or by adjusting the number of simulated agents (N_{sim}) or the number of periods (N_t) in the simulation algorithm. Since these produced different results, we varied each of these values while holding the others constant, to isolate the effects of changing each scale.

Increasing the N_x scale led to steadily increasing per-gridpoint timings, indicating a loss of efficiency with the increase of the scale. This is not surprising, and follows from the scheme described in Section 4 in which the entire $\bar{x}$ grid must be loaded into each work-group and checked, block by block. By increasing the size of the $\bar{x}$ grid, but leaving the size of each work-group

Table 1: GPU Timings

	N_x	N_q	N_{sim}	N_t
125	0.002	0.004	–	1.218
250	0.005	0.008	8.070	2.429
500	0.011	0.015	8.504	4.895
1000	0.030	0.030	8.770	9.702
2000	0.095	0.060	9.136	19.489
4000	0.335	0.118	9.613	38.860
6000	0.724	0.177	9.798	58.453
8000	1.251	0.236	10.130	77.714
12000	–	–	10.526	117.384
16000	–	–	10.881	155.566

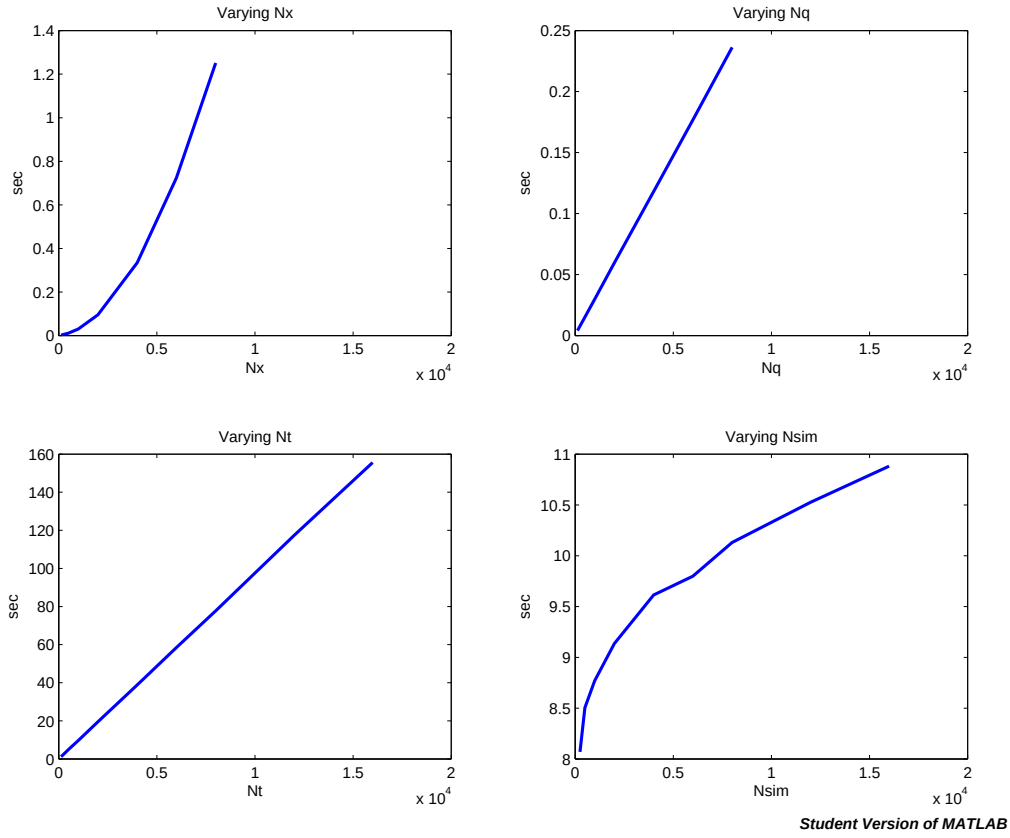
All timings are measured in seconds. The timings were obtained on a NVIDIA Tesla M2070 GPU on NYU's Bowery cluster. Timings are taken from the first overall iteration only in the "meta" algorithm. Timings for N_x and N_q values are per iteration of the solution algorithm. Timings for simulations are absolute timings for the entire simulation routine. The actual size used for the N_{sim} timings is the value in the first column rounded up to the nearest multiple of 256, the work-group size. The actual size used for the N_t timings is the value in the first column multiplied by 1.2, to allow for a 20% burn-in. While one size is being varied, the others are held constant at $N_x = 1000$, $N_q = 1000$, $N_{sim} = 5120$, and $N_t = 1200$. Because of rounding up, the 125 timing for the N_{sim} scaling is redundant and not included. The 12000 and 16000 timings for N_x and N_q would not run because of memory issues and are not included.

constant, this means more blocks loaded and checked per work-group, in addition to increasing the number of work-groups, explaining the more than linear increases in the timings.

Increasing the N_q and N_t scales led to linear increases in the timings, and relatively constant per-gridpoint timings, with the exception of slower per-gridpoint timings for very low values of N_q. This is largely in line with our expectations — increasing N_q changes the number of work-groups, but does not change the task of any given work-group. Similarly, each period is run in serial, and does not affect the parallelization of the simulation algorithm, and so we should again expect linear scaling.

Increasing the N_{sim} scale led to less than linear increases in the timings, and decreasing per-

Figure 2: GPU Timings



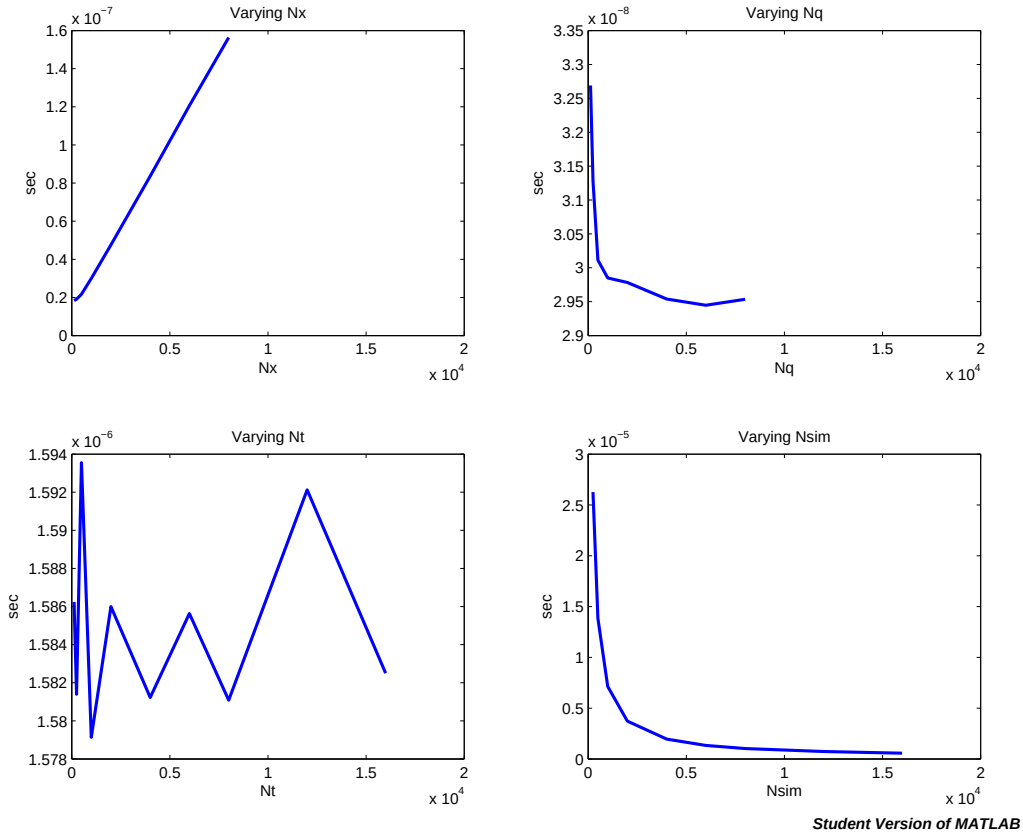
gridpoint timings. We are unsure exactly of why this is the case, but believe it to be because the number simulated agents is relatively small regardless of the numbers used, and the actual floating point computations can be handled nearly instantaneously by any powerful processor. Therefore, we suspect that the N_{sim} scale is not really the limiting factor in the timings.

7.2 CPU Timings

For comparison, we repeated the timings on the CPU, although we omitted some of the larger sizes for brevity, since overall the CPU timings were much slower than the CPU timings. The CPU timings are summarized in Tables 3 and 4, and in Figures 4 and 5.

For the N_x scaling, the CPU timings exhibited some strange properties, barely increasing linearly for small sizes of N_x, and then jumping by a factor of roughly 100 per gridpoint from $N_x = 2000$ to $N_x = 4000$, and then increasing roughly linearly again. I am unsure of the reason for this result (I suspect it is a memory issue), but it makes the GPU computation especially

Figure 3: GPU Timings Per Gridpoint



attractive for large values of N_x.

The N_q results on the CPU are similar to the N_x results, again with a strange jump from $N_q = 2000$ to $N_q = 4000$, and linear increases otherwise.

The N_{sim} timings no longer exhibit the same less than linear increases as in the GPU case, and in fact now show more than linear increases when N_{sim} is large. This may be because simultaneous computation of the agents' simulations is no longer so effortless for the CPU.

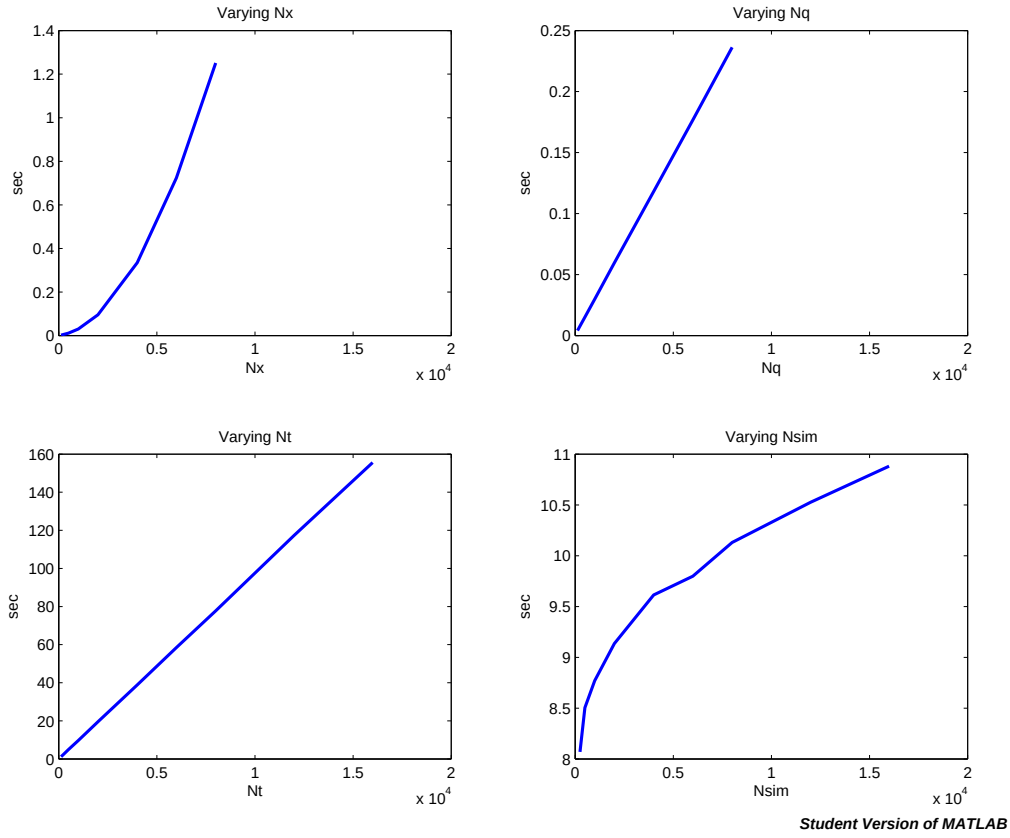
The N_t timings are very similar in their profile to the GPU case, once again increasing roughly linearly.

7.3 Speedup

The speedup from CPU to GPU appears to be quite large for some scalings, and speedup results are summarized in Table 5 and Figure 6.

In general, the speedup appears to be larger for the solution algorithm than for the simulation

Figure 4: CPU Timings

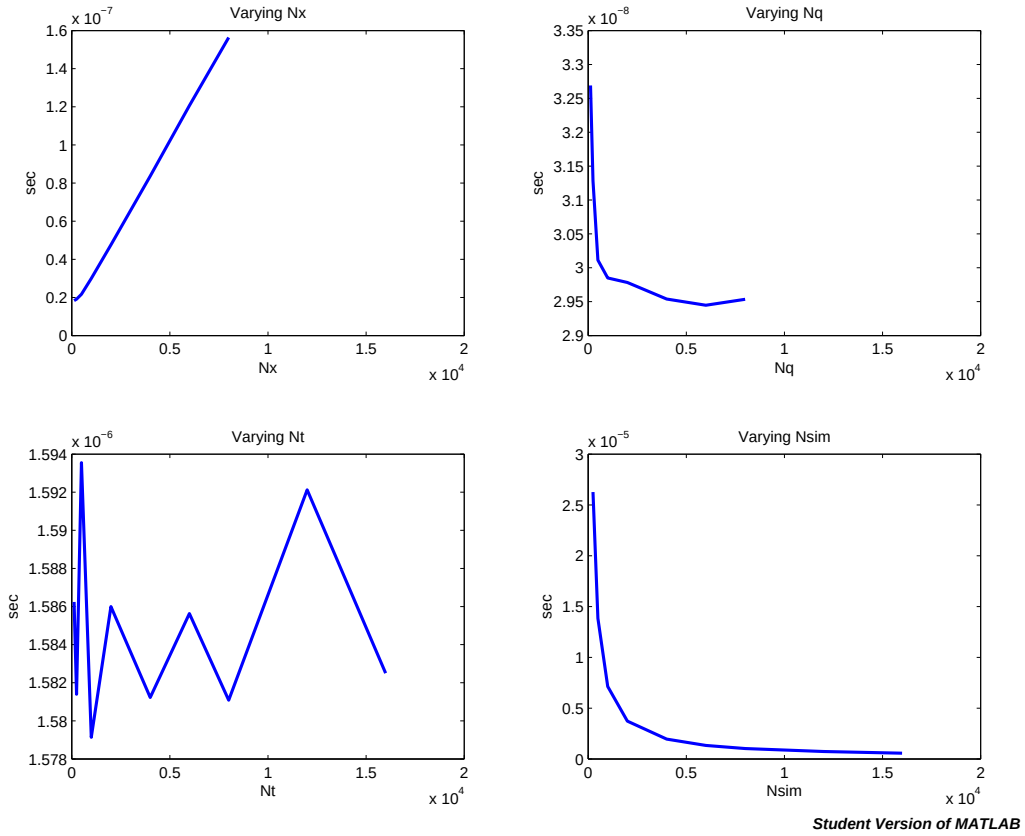


algorithm. This may be because the solution algorithm runs a larger set of computations in parallel (the entire grid), whereas the simulation algorithm only runs the simulation for each time t in parallel, and across times t in serial. Still, the GPU beats the CPU for both algorithms at all scales, and gains a large advantage over the CPU for large values of N_{sim}. It is also clear that the GPU holds a potentially huge advantage for the simulation step, making this type of parallelzation invaluable for solving optimal policy functions for complex models.

8 Conclusion

Overall this project was highly successful yielding tremendous returns to parallelization, and will be nearly directly applicable to Dan Greenwald's Ph.D. thesis work, as the model we use is a simplification of the types of model that he uses in his research. The speedup gained should allow for otherwise intractable models to be addressed or for existing computations to be made much

Figure 5: CPU Timings Per Gridpoint



more precise. GPU computation using OpenCL therefore appears to be a highly valuable tool for heterogeneous agents models, and one that we are very grateful to have learned.

Figure 6: Speedup: Ratio of CPU to GPU Timings

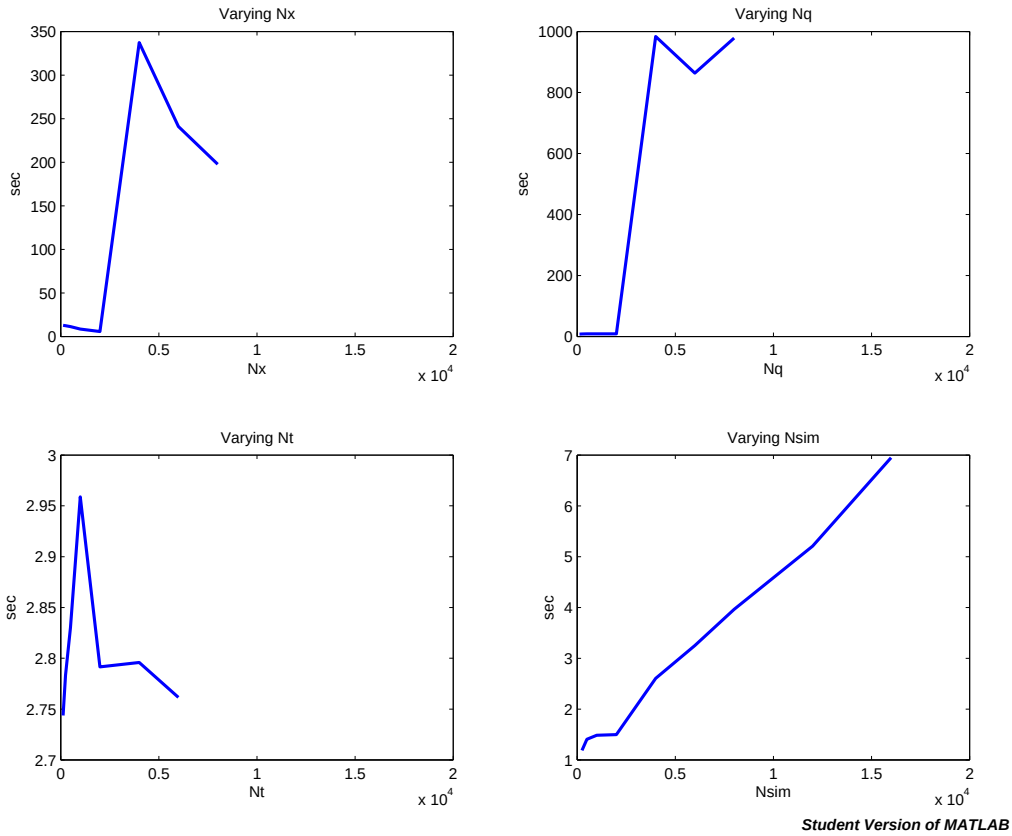


Table 2: GPU Timings per Gridpoint

	N_x	N_q	N_{sim}	N_t
125	1.837(-8)	3.269(-8)	–	1.586(-6)
250	1.907(-8)	3.128(-8)	2.627e-05	1.581(-6)
500	2.167(-8)	3.011(-8)	1.384e-05	1.594(-6)
1000	2.994(-8)	2.985(-8)	7.137(-6)	1.579(-6)
2000	4.755(-8)	2.978(-8)	3.717(-6)	1.586(-6)
4000	8.365(-8)	2.954(-8)	1.956(-6)	1.581(-6)
6000	1.206(-7)	2.945(-8)	1.329(-6)	1.586(-6)
8000	1.564(-7)	2.953(-8)	1.031(-6)	1.581(-6)
12000	–	–	7.290(-7)	1.592(-6)
16000	–	–	5.622(-7)	1.583(-6)

All timings are measured in seconds. Numbers in parentheses represent powers of ten for scientific notation. The timings were obtained on a NVIDIA Tesla M2070 GPU on NYU's Bowery cluster. Timings are taken from the first overall iteration only in the "meta" algorithm. The number of gridpoints is given by 4 N_xN_q for the N_x and N_q simulations, and $N_{sim}N_t$ foor the N_{sim} and N_t simulations. Timings for N_x and N_q values are per iteration of the solution algorithm. Timings for simulations are absolute timings for the entire simulation routine. The actual size used for the N_{sim} timings is the value in the first column rounded up to the nearest multiple of 256, the work-group size. The actual size used for the N_t timings is the value in the first column multiplied by 1.2, to allow for a 20% burn-in. While one size is being varied, the others are held constant at $N_x = 1000$, $N_q = 1000$, $N_{sim} = 5120$, and $N_t = 1200$. Because of rounding up, the 125 timing for the N_{sim} scaling is redundant and not included. The 12000 and 16000 timings for N_x and N_q would not run because of memory issues and are not included.

Table 3: CPU Timings

	N_x	N_q	N_{sim}	N_t
125	0.029	0.032	–	3.342
250	0.059	0.064	9.584	6.761
500	0.123	0.128	11.963	13.854
1000	0.256	0.256	13.026	28.707
2000	0.544	0.514	13.691	54.403
4000	112.886	116.237	25.045	108.647
6000	174.475	152.638	31.847	161.420
8000	247.291	231.218	40.149	–
12000	–	–	54.820	–
16000	–	–	75.595	–

All timings are measured in seconds. The timings were obtained on a 2.67 GHz Intel Xeon Processor using the AMD Platform on NYU's Bowery cluster. Timings are taken from the first overall iteration only in the "meta" algorithm. Timings for N_x and N_q values are per iteration of the solution algorithm. Timings for simulations are absolute timings for the entire simulation routine. The actual size used for the N_{sim} timings is the value in the first column rounded up to the nearest multiple of 256, the work-group size. The actual size used for the N_t timings is the value in the first column multiplied by 1.2, to allow for a 20% burn-in. While one size is being varied, the others are held constant at $N_x = 1000$, $N_q = 1000$, $N_{sim} = 5120$, and $N_t = 1200$. Because of rounding up, the 125 timing for the N_{sim} scaling is redundant and not included. Other missing entries are omissions simply due to the long length of these timings.

Table 4: CPU Timings per Gridpoint

	N_x	N_q	N_{sim}	N_t
125	2.349(-7)	2.545(-7)	–	4.352(-6)
250	2.372(-7)	2.555(-7)	3.120(-5)	4.402(-6)
500	2.468(-7)	2.567(-7)	1.947(-5)	4.510(-6)
1000	2.557(-7)	2.562(-7)	1.060(-5)	4.672(-6)
2000	2.722(-7)	2.572(-7)	5.571(-6)	4.427(-6)
4000	2.822(-5)	2.906(-5)	5.095(-6)	4.421(-6)
6000	2.908(-5)	2.544(-5)	4.320(-6)	4.379(-6)
8000	3.091(-5)	2.890(-5)	4.084(-6)	–
12000	–	–	3.797(-6)	–
16000	–	–	3.906(-6)	–

All timings are measured in seconds. Numbers in parentheses represent powers of ten for scientific notation. The timings were obtained on a 2.67 GHz Intel Xeon Processor using the AMD Platform on NYU's Bowery cluster. Timings are taken from the first overall iteration only in the "meta" algorithm. The number of gridpoints is given by 4 $N_x N_q$ for the N_x and N_q simulations, and $N_{sim} N_t$ foor the N_{sim} and N_t simulations. Timings for N_x and N_q values are per iteration of the solution algorithm. Timings for simulations are absolute timings for the entire simulation routine. The actual size used for the N_{sim} timings is the value in the first column rounded up to the nearest multiple of 256, the work-group size. The actual size used for the N_t timings is the value in the first column multiplied by 1.2, to allow for a 20% burn-in. While one size is being varied, the others are held constant at $N_x = 1000$, $N_q = 1000$, $N_{sim} = 5120$, and $N_t = 1200$. Because of rounding up, the 125 timing for the N_{sim} scaling is redundant and not included. The 12000 and 16000 timings for N_x and N_q would not run because of memory issues and are not included.

Table 5: Speedup: Ratio of CPU to GPU Timings

	N_x	N_q	N_{sim}	N_t
125	0.029	0.032	–	3.342
250	0.059	0.064	9.584	6.761
500	0.123	0.128	11.963	13.854
1000	0.256	0.256	13.026	28.707
2000	0.544	0.514	13.691	54.403
4000	112.886	116.237	25.045	108.647
6000	174.475	152.638	31.847	161.420
8000	247.291	231.218	40.149	–
12000	–	–	54.820	–
16000	–	–	75.595	–

All timings are measured in seconds. The CPU timings were obtained on a 2.67 GHz Intel Xeon Processor using the AMD Platform and the GPU timings were obtained on a NVIDIA Tesla M2070 GPU, both on NYU's Bowery cluster. Timings are taken from the first overall iteration only in the "meta" algorithm. Timings for N_x and N_q values are per iteration of the solution algorithm. Timings for simulations are absolute timings for the entire simulation routine. The actual size used for the N_{sim} timings is the value in the first column rounded up to the nearest multiple of 256, the work-group size. The actual size used for the N_t timings is the value in the first column multiplied by 1.2, to allow for a 20% burn-in. While one size is being varied, the others are held constant at $N_x = 1000$, $N_q = 1000$, $N_{sim} = 5120$, and $N_t = 1200$. Because of rounding up, the 125 timing for the N_{sim} scaling is redundant and not included. Other missing entries are omissions simply due to the long length of these timings.

References

[1] Guvenen, Fatih, "Macroeconomics With Heterogeneity: A Practical Guide." NBER Working Paper, No. 17622, 2011.

[2] Heathcote, Jonathan, Storesletten, Kjetil, and Violante, Giovanni L., "Quantitative Macroeconomics with Heterogeneous Households." NBER Working Paper, No. 14768, 2009.

[3] Krusell, Per and Smith, Anthony A., Jr., "Income and wealth heterogeneity in the macroeconomy." *Journal of Political Economy* Vol. 106, No. 5, 1998.

[4] Scarpino, Matthew, *OpenCL In Action: How to Accelerate Graphics and Computation.* Manning Publications Co., 2011.