Programming FPGAs for Economics: An Introduction to Electrical Engineering Economics

Bhagath Cheela* André DeHon†

Jesús Fernández-Villaverde‡ Alessandro Peri§

December 26, 2022

Abstract

We show how to use field-programmable gate arrays (FPGAs) and their associated high-level synthesis (HLS) compilers to solve heterogeneous agent models with incomplete markets and aggregate uncertainty (Krusell and Smith, 1998). We document that the acceleration delivered by one single FPGA is comparable to that provided by using 78 CPU cores in a conventional cluster. The time to solve 1200 versions of the model drops from 10 and half hours to 8 minutes, illustrating great potential for structural estimation. We describe how to achieve multiple acceleration opportunities -pipeline, data-level parallelism, and data precision- with minimal modification of the `C/C++` code written for a traditional sequential processor, which we then deploy on FPGAs easily available at Amazon Web Services. We quantify the speedup and cost of these accelerations. Our paper is the first step toward a new field, electrical engineering economics, focused on designing computational accelerators for economics to tackle challenging quantitative models.

Keywords: FPGA acceleration; Heterogeneous agents; Aggregate uncertainty; Electrical engineering economics; Cloud computing.

JEL Classifications:: C6; C63; C88; D52.

*Department of Electrical and Systems Engineering, University of Pennsylvania, cheelabhagath@gmail.com

†Department of Electrical and Systems Engineering, University of Pennsylvania, andre@acm.org

‡Department of Economics, University of Pennsylvania, jesusfv@econ.upenn.edu

§Department of Economics, University of Colorado, Boulder, alessandro.peri@colorado.edu

1 Introduction

Computations play a crucial role in economics, from models of aggregate fluctuations to game theory, passing through econometrics, labor economics, industrial organization, international trade, and finance, among others. Thus, economists have developed sophisticated algorithms to solve or estimate their models. The over 3,000 pages of the four volumes of the renowned *Handbook of Computational Economics* (Amman et al., 2018) demonstrate this point sufficiently. Interestingly, economists have paid much less attention to hardware. Except for a few papers (among others, Nagurney, 1996, Aldrich et al., 2011, Fernández-Villaverde and Valencia, 2018, Duarte et al., 2019, and Peri, 2020), researchers have generally taken hardware as a given and not as an essential margin to improve computations.

In a quickly evolving computational landscape, this lack of interest is unfortunate. As the speeds of single-core central processing units (CPU) stall, specialized accelerators (i.e., hardware designed to perform specific operations) continue to deliver greater performance at inexpensive price points. Since advanced computation is playing a growing role in all industries and research, the development and programming of customized accelerator chips have become cheaper and more accessible, clearing the way for a computational revolution.

Fields like biology, genetics, medicine, and physics have been at the vanguard of this process. Over the years, research in these fields has seen widespread adoption of a chip whose hardware can be programmed to solve application-specific algorithms: field-programmable gate arrays (FPGA). FPGAs have been successfully deployed to accelerate a variety of applications: DNA matching (Hoang, 1993), molecular dynamics (Azizi et al., 2004), Basic Local Alignment Search Tool (Herbordt et al., 2006), astrophysics particle simulator (Berczik et al., 2009), and cancer treatment (Young-Schultz et al., 2020), among others.

Unfortunately, the skills required to implement algorithms at the hardware level have represented an entry barrier for fields usually organized around small research teams, like economics, where there might be a comparative lack of domain-specific knowledge related to hardware. Our paper describes a first attempt to reduce these barriers for economists.

How do we get higher performance from a chip while retaining a reasonable easiness of programmability? To answer this question, we illustrate how to use the compilers designed to work with FPGAs, both easily available at Amazon Web Services (AWS), to accelerate the solution of a major workhorse in economics: the incomplete markets, heterogeneous agent model with aggregate uncertainty.

After the pioneering work of Krusell and Smith (1998), heterogeneous agent models have been used extensively to study business cycle fluctuations, monetary and fiscal policy, climate change, life-cycle decisions, industry dynamics, and international trade. Also, there has been tremendous interest in the development of solution methods well-suited for these models, like

Algan et al. (2008), Reiter (2009), Den Haan and Rendahl (2010), Maliar et al. (2010), Reiter (2010), Young (2010), Algan et al. (2014), Pröhl (2015), Achdou et al. (2021), Bhandari et al. (2017), Brumm and Scheidegger (2017), Judd et al. (2017), Bayer and Luetticke (2018), Childers (2018), Mertens and Judd (2018), Winberry (2018), Fernández-Villaverde et al. (2019), Auclert et al. (2020), Bilal (2021), and Kahou et al. (2021), among many others.

More concretely, we will work with the canonical incomplete markets economy in Den Haan et al. (2010). The authors proposed this economy as a computational suite to test the accuracy of solution methods for heterogeneous agent models precisely because of its canonicity. We solve the model employing the algorithm developed in Maliar et al. (2010). We pick this solution algorithm because of its simplicity and accuracy. A further advantage of this code is that it was not written to extract performance from custom accelerators, limiting the possible biases in our analysis. We code our solution in `C/C++`, the fastest programming environment for computation in economics (Aruoba and Fernández-Villaverde, 2015).[1] For the FPGA, we employ the industry gold standard Xilinx high-level synthesis (HLS) compilers using Vitis HLS (v2021.2, 64-bit) and the `OpenCL` interface (Xilinx, Inc., 2020b). For the CPU, we use the `G++ 9.4.0` and `mpic++ 4.1.1` (Open MPI) compilers, which implement state-of-the-art sequential and parallel execution run-time optimizations (and whose importance will be clear momentarily).

Our first exercise runs the code in one FPGA and one CPU core. The code is *exactly* the same in both cases except when we apply the `#PRAGMA` directives available to the FPGA designer. The `#PRAGMA`s instruct the HLS compiler on how to *design* the FPGA hardware, i.e., on how to connect *physical* logical resources in the FPGA to exploit the maximum parallelism, pipelining, data distribution, and data streaming required to efficiently solve our algorithm.[2] The CPU cannot offer this feature because its hardware is not programmable. The CPU code is compiled using the `-O3` flag, the most aggressive standard-compliant optimization and one that delivers an executable file that is hard to beat via hand-made optimizations, even for highly experienced programmers. Thus, our code compares the best performance offered by one FPGA with the best performance offered by one CPU core, making the comparison meaningful. We measure that the FPGA delivers a speedup of nearly 78 times against an Intel Xeon Scalable Processor (Cascade Lake, second generation) core. In other words, we solve the same heterogeneous agent model 78 times faster in one FPGA than in one CPU core, reducing the time to solve 1200 economies from 10 and half hours to 8 minutes.

Our second exercise scales up from one FPGA to eight and from one CPU core to 48 to

[1] We code our FPGA kernel functions in `C` to comply with the FPGA HLS `C`-to-gates compiler requirement. Similarly, we code our CPU kernel functions in `C` to avoid the abstraction speed penalties that may arise from `C++` object-oriented programming: we want to give the CPU core the best possible fighting chance against the FPGA acceleration.

[2] Our online tutorial at https://www.sas.upenn.edu/~jesusfv/KS_FPGA_tutorial.pdf supplies a detailed guide on how to utilize `#PRAGMA`s to accelerate the solution of our model.

compare their performance when a researcher has access to multi-core acceleration. In this case, we use Open-MPI (the de facto standard for parallelization in large clusters of CPUs) to parallelize our code as deployed in the CPUs. In particular, we ask each CPU core to solve the same model many times (we call each solution an "economy"). For example, if we have 48 cores, we ask each CPU core to solve 25 economies, for a total of 1,200 economies and we compare the time required against the time that the FPGAs require to solve 1,200 economies. This research design is the *best case scenario* for parallelization in CPU cores, as it minimizes the communication across CPU cores and its associated overheads. Other parallelization schemes, such as solving one economy simultaneously in several cores (perhaps more relevant in practice because the model to solve is complex), will deliver worse performance for multi-core CPU clusters because of time lost in data transfers. We find that one FPGA solves 1,200 economies 1.67 times faster than 48 CPU cores do and that eight FPGAs solve 1,200 economies 604 times faster than one CPU core and 13 times faster than 48 CPU cores.

The FPGAs speedups in these two exercises are accompanied by large cost savings (less than 18.35% of the CPU cost) and even more impressive energy savings (less than 5.46% of the CPU energy consumption), a growing concern due to environmental goals set up by universities and research institutions. An additional attractive feature of FPGAs is that they are easily available either for purchase at economical prices or at commercial cloud services, such as AWS (the service we use in this paper), at low costs per hour.

Next, we inspect the mechanisms that account for the FPGA speedups. First, we use pipelining of complex equations to start a new calculation composed of hundreds of primitive operations in each cycle. Second, we exploit loop-level data parallelism to simultaneously perform computations on multiple independent pipelines. Third, we employ coarse-grained, data-level parallelism to compute multiple economies in parallel. Fourth, we tune the precision and data representation to promote efficient pipelining and support additional parallelism.

While our application is focused on the canonical model in Den Haan et al. (2010), the acceleration techniques we describe for its solution can be easily generalized for solving more complicated heterogeneous agent models. We have in mind, for example, the classes of HANK (Kaplan et al., 2018; Bayer et al., 2019) and climate change models (Cai and Lontzek, 2019; Cruz Álvarez and Rossi-Hansberg, 2021; Krusell and Smith, 2022) that have become so influential in recent years. Both classes of models face a whole new range of computational challenges with respect to the basic framework in Krusell and Smith (1998) that have prevented their full deployment for policy advising despite their crucial importance. FPGAs are well-suited to deal with these larger models. Nonetheless, we prefer to illustrate our approach with the model in Den Haan et al. (2010) instead of jumping directly to more complex environments for transparency. Over the last 25 years, we have accumulated so much knowledge about what works (and what does not!) while solving models à la Krusell and Smith (1998) that the reader can

appreciate, more quickly, the novelty of our work, the speed gains, and its accuracy.

Similarly, the extra speed that FPGAs deliver can also be put to good use when structurally estimating the model using micro and macro data or, relatedly, while checking for the robustness of the results with respect to different parameter values. In both cases, we need to solve the model for many parameter values. Thus, speed is a first-order consideration.

In terms of the literature, the only other paper in economics using FPGAs we are aware of is Peri (2020). We replace the lower-level programming in the register-transfer level (`RTL`) language (`Verilog`) used in that paper with the more easily accessible FPGA HLS `C`-to-gates compilers. This switch follows a long history of increasingly sophisticated automation in computer science to map higher-level abstractions down to low-level implementations (see Appendix A). From the earliest demonstrations (Babb et al., 1999) to the launch of the first commercial compilers (`Streams-C`, Frigo et al. 2001, Snider 2002), FPGA compilers have steadily improved, making the programming of FPGAs cost-effective in terms of coding time.

In summary: there is much promise in a new field, electrical engineering economics, where specialized computational accelerators are designed for specific, yet vital, computational tasks in economics at attractive price points and reasonable programming complexity. Programming FPGAs to solve heterogeneous agent models is but a first step into a rich area of research.

The rest of the paper is organized as follows. Section 2 presents the model and its calibration. Section 3 details the solution algorithm. Section 4 describes the acceleration schemes. Section 5 reports quantitative results and performs robustness tests. Section 6 isolates the acceleration channels responsible for the speed gains. Section 7 concludes.

2 The Model

This section presents the incomplete markets model with aggregate uncertainty and borrowing constraints described in Den Haan et al. (2010) and its calibration. The model is an infinite-horizon production economy with aggregate uncertainty and a unit mass of infinitely lived ex-ante identical households that experience uninsurable idiosyncratic shocks to their employment status and are subject to an occasionally binding borrowing constraint.

The household's problem. Households choose sequences of consumption, $c_t \in \mathbb{R}_+$, and physical capital, $k_{t+1} \in \mathbf{K} \subseteq \mathbb{R}_+$, to solve:

$$\max_{\{c_t, k_{t+1}\}_{t=0}^{\infty}} \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\gamma} - 1}{1-\gamma} \tag{1}$$

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

$$k_{t+1} \geq 0 \tag{3}$$

subject to the budget constraint (2), the occasionally binding constraint (3), and the idiosyncratic shocks to their employment status $\epsilon_t \in \{0,1\}_\epsilon$, which equal 1 if employed and 0 if unemployed. We will specify the stochastic process for ϵ_t below. The prices w_t and r_t denote the equilibrium wage and interest rate, τ_t is the labor income tax rate, $\bar{l}$ is the time-endowment, μw_t is the (tax-free) unemployment benefit, and δ is the depreciation rate.[3]

The firm's problem. A representative, perfectly competitive firm uses per capita capital $K_t \in \mathbf{M} \subseteq \mathbb{R}_+$ and the employment rate $L_t \in \mathbb{R}_+$ to produce a per capita final good with a Cobb-Douglas production function $Y_t = A_t K_t^\alpha (\bar{l}L_t)^{1-\alpha}$, where $0 < \alpha < 1$. The aggregate productivity A_t follows a two-state first-order Markov process over the support $\mathbf{A} = \{a_b, a_g\}$, where $a_g = (1 + \Delta_A)$ is the good realization and $a_b = (1 - \Delta_A)$ is the bad realization.

Competition in the input and output markets implies that:

$$r_t = \alpha A_t \left(\frac{\bar{l}L_t}{K_t} \right)^{1-\alpha}, \tag{4}$$

and

$$w_t = (1-\alpha) A_t \left(\frac{K_t}{\bar{l}L_t} \right)^{\alpha}. \tag{5}$$

Government. In each period, the government uses labor income taxes τ_t to finance unemployment benefits (in terms of wages) μ,

$$\tau_t \bar{l} L_t = \mu(1 - L_t), \tag{6}$$

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

Aggregate law of motion. The cross-sectional distribution of households over capital holdings and employment status, $\mathbf{\Gamma}$, follows the law of motion:

$$\mathbf{\Gamma}_{t+1} = \mathcal{H}(\mathbf{\Gamma}_t, A_t, A_{t+1}). \tag{7}$$

Equilibrium. Given an exogenous transition law for $\{A, \epsilon\}$, a recursive competitive equilibrium is the set of prices $\{w, r\}$, policy function $k'(\cdot)$, tax rate τ, and law of motion $\mathcal{H}(\cdot)$ for the cross-sectional distribution $\mathbf{\Gamma}$ such that: i) given the individual household state $\{k, \epsilon; \mathbf{\Gamma}, A\}$, prices $\{w, r\}$ and the laws of motion of $\{A, \epsilon\}$ and Γ, the policy function $k'(\cdot)$ solves the Bellman equation representation of the household's sequential problem in (1)-(3); ii) given $\{\mathbf{\Gamma}, A\}$, input factor prices $\{w, r\}$ receive their marginal products (4)-(5); iii) given A, the labor income tax rate τ balances the government budget, (6); iv) the markets for labor and capital clear; v) given

[3] For the sake of comparability, the notation closely follows Den Haan et al. (2010), except that we drop the subindex for individual households. We are also more explicit about the choice sets than what is standard to facilitate readability. Notice that, when unemployment benefits are set to zero, the model coincides with the model in Krusell and Smith (1998).

$\{w, r, \mathbf{\Gamma}, k'\}$ and the transition laws for $\{A, \varepsilon\}$, the law of motion $\mathcal{H}(\cdot)$ satisfies (7).

Calibration. To ensure the maximum comparability of results, we replicate the calibration of Den Haan et al. (2010). The time unit is a quarter. We discretize the joint transition of aggregate productivity and idiosyncratic employment status using the transition matrix in Table 2 in Den Haan et al. (2010). The transition probabilities are designed such that the unemployment rate depends only on aggregate productivity and they take values $u_b = u(1-\Delta_A)$ in bad times and $u_g = u(1+\Delta_A)$ in good times, $u_b > u_g$. Table 1 summarizes the rest of the parameters as described in Den Haan et al. (2010).

Table 1: Calibrated Parameters

α	0.36	Output capital share
β	0.99	Quarterly discount factor
γ	1	Arrow-Pratt relative risk aversion coefficient
δ	0.025	Quarterly depreciation rate
μ	0.15	Unemployment benefits in terms of wages
$\bar{l}$	1/0.9	Time endowment
Δ_A	0.01	Aggregate productivity shock size

We will refer to the set of parameters $\underline{\theta} = \{\alpha, \beta, \gamma, \delta, \mu, \bar{l}, \Delta_A\}$ calibrated as in Table 1 as the *baseline economy*. Without loss of generality, we will refer to a generic $\underline{\theta}$ as an *economy*.

3 The Solution Algorithm

We solve the model using the stochastic simulation algorithm described in Maliar et al. (2010). Following Krusell and Smith (1998), we assume that households are boundedly rational and perceive that only a finite set of moments of $\mathbf{\Gamma}$ affect future prices. In the numerical exercise, we restrict our attention to the law of motion that describes the evolution of the first moment of the cross-sectional distribution of the per capita stock of capital, $m \in \mathbf{M}$ among households:

$$m' = H(m, A, A') \tag{8}$$

(notice the change in notation from $\mathcal{H}(\cdot)$ to $H(\cdot)$).

A. Grids. We define the intervals on the household's capital grid $\mathbf{K} \equiv [k_{\min}, k_{\max}] = [0, 1000]$ and first moment of the cross-sectional distribution $\mathbf{M} \equiv [m_{\min}, m_{\max}] = [30, 50]$. We discretize them with $N_k = 100$ and $N_M = 4$ grid points, respectively. Section 5.4 explores alternative grid sizes.

B. Individual households' problem (IHP). A stationary solution to the optimization

problem (1) is the saving policy function $k' : \mathbf{K} \times \{0,1\}_\epsilon \times \mathbf{M} \times \mathbf{A} \to \mathbb{R}_+$:

$$k' = \left[\mu(1-\epsilon) + (1-\tau)\bar{l}\epsilon\right] w + (1-\delta+r)k - \left\{\lambda + \beta\mathbb{E}\left[\frac{1-\delta+r'}{\left((\mu(1-\epsilon') + (1-\tau')\bar{l}\epsilon')\, w' + (1-\delta+r')k' - k'(k')\right)^\gamma}\right]\right\}^{-1/\gamma}, \quad (9)$$

where $k' \equiv k'(k,\epsilon,m,A)$, $\lambda \equiv \lambda(k,\epsilon,m,A)$, $k'(k') \equiv k'(k'(k,\epsilon,m,A),\epsilon',m',A')$ and that satisfies the occasionally binding constraint:

$$k' \geq 0 \quad (10)$$

and complementary slackness condition:

$$\lambda \cdot k' = 0 \qquad \lambda \geq 0. \quad (11)$$

We find a solution to the IHP using the following iterative Euler equation algorithm:

(i) Initial guess. Guess an initial policy function k'_0. In our case, we set $k'_0 \equiv k'_0(k,\epsilon,m,A) = 0.9 \cdot k$.

(ii) Iteration step. For each iteration step $i \geq 0$ and given the guess k'_i:

$$\widehat{k}'_{i+1} = \Phi k'_i \qquad \textit{(a)}\ \text{Solve (9)}$$
$$k'_{i+1} = \eta_k \widehat{k}'_{i+1} + (1-\eta_k)k'_i \qquad \textit{(b)}\ \text{Update Guess}$$

(a) For any state $(k,\epsilon,m,A) \in \mathbf{K} \times \{0,1\}_\epsilon \times \mathbf{M} \times \mathbf{A}$:

- Set the Lagrange multiplier $\lambda(k,\epsilon,m,A) = 0$.
- Substitute the guess k'_i and compute the right-hand side of equation (9).
- Update the left-hand side. If $\widehat{k}'_{i+1}$ falls outside the capital grid set, replace it with the closest boundary of the individual capital grid, $\{k_{\min}, k_{\max}\}$.

(b) After completing step *(a)*, let $\eta_k = 0.7$ and set the $(i+1)$-iteration policy function guess to:

$$k'_{i+1} = \eta_k \widehat{k}'_{i+1} + (1-\eta_k)k'_i. \quad (12)$$

(iii) Convergence criterion. Repeat the iteration step *(ii)* until convergence of the policy function in the sup norm:

$$\rho\left(k'_{i+1}, k'_i\right) = \max_{(k,\epsilon,m,A)\in\mathbf{K}\times\{0,1\}_\epsilon\times\mathbf{M}\times\mathbf{A}} |k'_{i+1} - k'_i| < \varepsilon_k = 1e(-8). \quad (13)$$

C. Aggregate law of motion (ALM). Following Maliar et al. (2010), we parameterize

the law of motion of the per capita stock of capital, $m \in \mathbf{M}$ as:

$$\ln m' = f(a, m; b) = b_1(a) + b_2(a) \ln m \qquad a \in \{a_b, a_g\}, \tag{14}$$

where the second equality assumes the conditional expectation of $\ln m'$ to be linear in $\ln m$ and aggregate-state dependent.

D. The fixed-point algorithm. We estimate the vector of aggregate state-dependent coefficients b (henceforth, ALM coefficients) using a nested fixed-point iterative algorithm:

(i) Initializations. Set initial values for $(b_1(a_g), b_2(a_g)) = (b_1(a_b), b_2(a_b)) = \{0, 1\}$. Set the size of the cross-sectional distribution to $J = 10,000$ households and the length of the stochastic simulation to $T = 1,100$. Use a pseudo-random number generator to draw the aggregate shocks $\{a_t\}_{t=1}^{1,100}$ and the idiosyncratic shocks to the employment status $\{\epsilon_{t,j}\}_{t=1,j=1}^{T=1,100,J=10,000}$. Set the initial cross-sectional distribution of the households' capital holdings.[4] Use Equations (4), (5), and (6) to compute wages, the interest rate, and labor income taxes.

(ii) Iteration step. For each iteration step $i \geq 0$:

(a) **IHP.** Estimate the household capital holdings policy functions $k'(k, \epsilon, m, A)$, by solving the IHP.

(b) **Simulation.** At each $t \in \{1, \ldots, 1100\}$, given the initial capital holdings distribution, the idiosyncratic and aggregate stochastic processes, and the policy functions:

(i) Accumulation step. Compute m_t, the cross-sectional average of household capital holdings

$$m_t = \frac{1}{J} \sum_{j=1}^{J} k_{j,t}. \tag{15}$$

(ii) Interpolation step. For each household $j \in \{1, \ldots, 10,000\}$, use linear interpolation to determine the next period household capital holdings, given the period t idiosyncratic $\{k_{t,j}, \epsilon_{t,j}\}$ and aggregate $\{m_t, A_t\}$ states.[5]

(c) **ALM.** Estimate $\hat{b}^i(a) = (\hat{b}_1^i(a), \hat{b}_2^i(a))$ with $a \in \{a_b, a_g\}$ by running the OLS regression associated with (14), after discarding the first 100 observations, $t = 1, \ldots, 100$.

[4] Following Maliar et al. (2010): *(i)* we set the initial capital distribution to the steady-state value of capital in a deterministic model with employment rate $L = 1/\bar{l} = 0.9$; *(ii)* we iteratively update the initial capital distribution (with the capital distribution associated with $T = 1,100$) if the metric measuring the convergence of the ALM coefficients in (16) is higher than $1e(-6)$.

[5] The `Matlab` code in Maliar et al. (2010) uses a spline interpolation scheme. Because moving the splines to an FPGA involves some extra work that would distract from the clarity of exposition, we leave the implementation of this feature for future research. In addition, we precompute aggregate and idiosyncratic shocks to ensure comparability across different software. These are the only two differences with their original code, available at https://lmaliar.ws.gc.cuny.edu/codes/.

(d) Let $\eta_b = 0.3$. Set the $(i+1)$-iteration ALM coefficients to:

$$b_l^{i+1}(a) = \eta_b \hat{b}_l^i(a) + (1-\eta_b) b_l^i(a), \qquad l \in \{1,2\}, \quad a \in \{a_b, a_g\}.$$

(iii) Convergence criterion. Repeat the iteration step *(ii)(a)-(ii)(d)* until convergence of the ALM coefficients in the Euclidean norm:

$$\sqrt{\sum_{l\in\{1,2\}, a\in\{a_b,a_g\}} (b_l^{i+1}(a) - b_l^i(a))^2} < \varepsilon_b = 1e(-8). \tag{16}$$

4 Acceleration Schemes and Hardware Architecture

This section describes the hardware architecture of FPGA acceleration and explains the hardware architecture of our CPU computations. Appendix B reports the hardware specifications of the different instances.

4.1 Hardware Architecture of FPGA Acceleration

The FPGA acceleration approach solves the algorithm in Section 3 by using Amazon F1 instances to deploy three configurations of FPGAs connected to a host: one FPGA (f1.2xlarge), two FPGAs (f1.4xlarge), and eight FPGAs (f1.16xlarge).

The computational flow is as follows. The host initializes variables (grids, shocks, guesses), allocates jobs across available FPGAs, and transfers the data to the FPGA(s). The FPGA(s) solve(s) the nested, fixed-point algorithm discussed in Section 3. The host reads back and saves the results. Unless differently specified, computations are performed using the IEEE754 double-precision floating-point format.

4.1.1 FPGA hardware description

F1 instances mount (up to eight) Xilinx VU9P FPGAs, each with 1.2 million 6-input gates (LUTs), 6,840 multiply-accumulate units (DSPs), and 346Mb of on-chip memory, given by 76Mb of Block RAM (BRAM) and 270Mb of Ultra RAM (URAM). These resources are organized in three dies, referred to as a Super Logic Region (SLR), connected by a silicon interposer (Figure 2). The AWS FPGA is further divided into two partitions: Shell and Custom Logic (CL). The shell is the FPGA platform implementing external peripherals like PCIe, DRAM with the FPGA. CL is the custom acceleration logic that users design. Users' CL designs can employ up to 895 thousand LUTs, 5,640 DSPs, and 284 of on-chip memory, given by 59Mb of BRAM and 225Mb of URAM per FPGA.

4.1.2 Custom logic hardware design

Our CL design is tailored to compute three economies in parallel, one per SLR, in the same FPGA.[6] `OpenCL` commands organize the computation flow in kernels. Each kernel is assigned a different economy $\underline{\theta}$ and is instantiated in a separate SLR. Kernels are then deployed in parallel.

A feature of our hardware design is that it is easily scalable with minimal modifications of the `C/C++` code. `OpenCL` commands collect the available SLRs and instantiate the kernels. In the case of one FPGA (f1.2xlarge), three kernels are instantiated across the available SLRs. In the case of eight FPGAs (f1.16xlarge), 24 kernels are launched in parallel –similar to having a 24-core CPU with separate memories running in parallel.

To load and deploy our custom logic hardware design on the EC2 F1 instances, we create Amazon FPGA Images (AFI) that combine the Shell and the CL design for multiple grid sizes. This step is accomplished with the creation of .AWSXCLBIN files, which can then be used to run inference on the Amazon cloud platform.

4.1.3 The single-kernel design

Our kernel implements in hardware the nested-fixed point algorithm presented in Section 3. The design organizes the three inter-dependent building blocks –IHP, simulation, and ALM– in a sequence, and performs the computations until convergence of the ALM coefficients (Equation (16)). Our final kernel design starts computations at every clock cycle at 250 MHz and provides flexibility to study multiple grid sizes.

Hardware Design: Common Challenges and Remedies. Our kernel defines customized hardware pipelines that compute hundreds of operations in the model equations every cycle. A well-designed application keeps these hardware pipelines constantly busy, starting new computations at every clock cycle. A pipeline that *initiates* computations every clock cycle is said to have an *initiation interval* of 1, or $\mathbf{II} = 1$. Applications face common challenges in the management of memory to reach this goal. Some of the common remedies include:

1. **Large memory access latency**. The FPGA device and host processor share a common memory referred to as global memory (Dynamic Random Access Memory, DRAM). Global memory is large (tens of gigabytes) and, thus, useful for storing input, intermediate, and output data. However, access to this memory is slow (tens of clock cycles), and it cannot provide enough data per cycle to perform the pipelined computations at $\mathbf{II} = 1$. This

[6] We could alternatively design the custom logic to span across the multiple SLRs. This design would require handling inter-SLR connections, complicating the coding and the management of tighter clock constraints. Since many of the benefits of accelerators are associated with the estimation of structural parameters that involve the computation of several thousand economies (each associated with a different set of parameter values), our hardware design is appropriate. Although we do not optimize the single economy acceleration, the single SLR solution is still 37.66 times faster than the sequential execution in the CPU (Section 5.4).

problem is addressed by copying chunks of data from the large global memories to on-chip local memories, which are small but numerous and can be accessed in a single clock cycle.[7] Our FPGA has enough local memories to store all of the input data for our application (Table A.2). Hence, we need to transfer the input data only once per kernel computation.

2. **Limited memory ports to access data in parallel**. Data in global and local memories can be accessed (for reading or writing) via a limited number of ports, two ports per local memory in our FPGA device. This creates a memory access bottleneck when we instantiate multiple instances of our pipeline to access the common memory in parallel. When more than two pipelines are instantiated in parallel, the memory reads are performed sequentially in multiple clock cycles, violating the targeted **II** of 1. The FPGA has many independent local memories. Hence, this problem is easy to fix by storing multiple copies of the same data in as many independent local memories as required by the number of pipelines performed in parallel. Even a single pipeline may require access to more than two data elements from the same data array, also forcing multiple clock cycle sequential reads. This issue is handled by distributing the single data array across different memory blocks, thereby scaling the number of ports.

A good memory design goes a long way toward helping the compiler automatically create efficient pipelines. Yet, it does not go all the way. Next, we discuss the application-specific interventions required to reach an **II** of 1.

Application-Specific Challenges and Remedies. After implementing the two steps above, there are still two pieces of computation that prevent the compiler from reaching an **II** = 1. Although specific to our application, these bottlenecks are common among many models in economics and are represented by the linear interpolation –in the **IHP** and **Simulation** steps– and the computation of the cross-sectional average of individual capital holdings in the **Simulation** step (Equation 15).

A well-known computational challenge in interpolation is to find the interval of interpolation. We accelerate this step by implementing an efficiently pipelined (**II** of 1) jump search algorithm with fixed-size loop bounds. To ensure that the algorithm does not introduce any unwanted bias, we verify that it outperforms alternative search algorithms in the CPU (Table 2).

The challenge in pipelining the cross-sectional average of individual capital holdings is the finite precision approximation used by the accumulation operation in Equation (15). Floating-point accumulations are complex arithmetic operations that take multiple clock cycles and prevent the pipeline from performing a new operation every cycle. To address this issue, we use knowledge of the range of our grids to find the best fixed-point approximation, which allows us

[7]The main on-chip local memories are represented by 2,160 BRAMs (of 36k bits each) and 960 URAM (of 288k bits each). Local memories also include DRAM (registers), whose availability changes by application.

to perform accumulation operations every clock cycle.

The Individual Household Problem (IHP) Design. The **IHP** design implements the previous steps to reach an **II** of 1 in Equation (9) by fully unrolling the operations involved in the computation of the expectation. We further accelerate the design by placing two pipelines to work in parallel on the (sequential) solution of Equation (9) at different states $\{k, \epsilon, m, A\}$ (more pipelines are hard given the hardware limits). Given the current guess of individual capital holdings, the IHP design solves Equation (9) for every state, and updates the guess according to Equation (12), as discussed in Section 3.B.*(ii)*. These steps are iterated until convergence, as per Equation (13).

The Simulation Design. The simulation design in Section 3.D.*(ii).(b)* is divided into two steps, which we accelerate with two different custom pipelines. The first step is represented by the accumulation operation required to compute the cross-sectional average of individual household capital holdings, m_t, in Equation (15). To achieve an $\mathbf{II} = 1$ at this step, we use a custom fixed-precision accumulation operator. After moving from floating-point to fixed precision, we accelerate the computation of the cross-sectional average by instantiating multiple pipelines to perform the accumulation in a reduce tree. The second step is represented by the interpolation step involved in the computation of next-period household capital holdings. We reach an $\mathbf{II} = 1$ by implementing the aforementioned jump search algorithm.[8]

The ALM Design. The ALM step involves several resource-expensive operations but has very small latency (0.755ms in our baseline economy).[9] Accordingly, we provide instructions to the compiler to not perform any pipeline or unrolling and further instruct it to limit the number of hardware resources used.

4.1.4 The Three-Kernel Design

Our CL design is tailored to compute three economies in parallel by exploiting the three hardware regions available for hardware design: the SLRs. To reach this goal, we need to slightly modify our single-kernel design to handle the resource limitations. Our single-kernel design barely fits in one of the SLRs, slightly leaking into one of the adjacent SLRs with non-time-critical operations (Figure 1). The Amazon Shell –which provides useful hardware resources to facilitate the HLS kernel design– exacerbates this problem by consuming part of the resources in two of the three adjacent SLRs. To fit the three kernels, we provide directives that limit the unrolling in the

[8]To achieve an $\mathbf{II} = 1$ in both steps, we create multiple copies of the input data to avoid memory access bottlenecks. The simulation requires a large number of individual shocks ($10,000 \times 1,100$). To minimize the memory usage, we encode the $\{0, 1\}$ shocks (from integers) into bits and pack them in groups of eight into a single byte of memory (8 bits) in both the FPGA and CPU. Further, we tell the compiler to store these shocks in the URAMs, which have wide arrays of 72 bits and allow us to store 64 shocks (of size 1 bit) in each of these arrays. By doing so, we need to access the memory only once every 64 iterations.

[9]The ALM step requires 16 BRAM, 133 DSP, 11,494 LUTs, and 76,736 Registers.

IHP. The resulting three-kernel design (Figure 2) trades off the single-kernel performance for the parallel execution speedup.

4.2 Hardware Architecture of CPU Multi-core Acceleration

The CPU multi-core acceleration approach solves our model using Amazon M5N instances by parallelizing data-independent tasks (economies) across multiple cores: one core (m5n.large), eight cores (m5n.4xlarge), and 48 cores (m5n.24xlarge).[10]

Our benchmark CPU kernel is an optimized `C++`-version of the original Maliar et al. (2010) `Matlab` code, whose sequential single-core solution of the baseline economy is four times as fast as the original `Matlab` code, an expected speedup given the results in Aruoba and Fernández-Villaverde (2015). The CPU kernel utilizes a jump search algorithm in order to determine the interpolation interval over the individual capital holdings grid. Table 2 shows that this algorithm outperforms standard alternatives in the CPU, ensuring the best CPU kernel performance.

Table 2: Benchmarking the CPU: Alternative Search Algorithms

	Linear Search	**Binary Search**	**Jump Search**
Execution Time	97348.3	49667.3	37854.5
Speedup	-	1.96	2.57

Note: Columns 2-4: Average execution time (in seconds) and speedups of alternative interpolation interval search algorithms. Speedups are computed relative to the linear search algorithm. Results are obtained by solving 1,200 baseline economies sequentially using a single core instance (m5n.large).

We operationalize the multi-core parallelization by using the open source Message Passing Interface (Open-MPI).[11] Open-MPI provides easily implementable, off-the-shelf routines for parallelizing data-independent tasks across multiple cores. In contrast with alternative multi-core parallelization (like OpenMP), Open-MPI does not require different cores to share the same memory. This feature makes Open-MPI particularly appealing for massive data parallelization, from small clusters up to medium/large-scale supercomputers (e.g., XSEDE, RMACC Summit Supercomputer, etc.).

Our computational flow is as follows. Open-MPI routines determine the number of cores available and uniformly spread the number of (data-independent) economies to be computed across the different cores. For example, the solution of 1,200 economies on a 48-core machine would have Open MPI allocating 25 economies per core. Each core will then independently complete the assigned tasks.

[10] In Appendix B, we justify the choice of these instances.

[11] See https://www.open-mpi.org. We use `Open MPI: Version 4.1.1.`

5 Quantitative Results

We assess the efficiency gains of FPGA acceleration against CPU cores by measuring efficiency gains in solution speedup, AWS costs, and energy savings. We choose as a benchmark the CPU rather than the graphic processing units (GPUs) for two reasons. First, CPUs are still more commonly used in economics than GPUs, making the interpretation of our results more transparent. Second, differently from the CPU, the effectiveness of GPU acceleration is heavily dependent on software optimizations (e.g., memory management), making the benchmark performance more susceptible to software engineers' choices. Nonetheless, and for completeness, we implement the Maliar et al. (2010) algorithm in `python/1.13` and accelerate it using the `Numba Cuda compiler` on an `NVIDIA A100` GPU. Consistent with the finding in Aldrich et al. (2011) for GPU acceleration on small grids, the GPU is "only" twice as fast as the original `Matlab` code. While the implementation of the algorithm with `C-cuda` and additional memory optimizations might yield further speedups, it is most unlikely that GPUs can deliver the same performance as the FPGA for this particular application.

To make our acceleration comparison as meaningful as possible, we proceed as follows. First, our FPGA and CPU kernel share the same `C/C++` code to solve the same algorithm in both platforms. Hence, the observed speedup is the result of a margin available to the electrical engineer economist (designing hardware) and not to the software engineer economist (writing better code for CPUs and GPUs). Using the `#PRAGMA`s, we have control of the flow of instruction over *time* and across *space*. For instance, we can program the logic of our FPGA to have additions, multiplications, and comparison operations to be performed in parallel, if consistent with our algorithm. Concretely, the FPGA HLS `C`-to-gates compiler would physically place these operators in the FPGA by specializing its custom logic.

In comparison, the CPU (and GPU) logic cannot be programmed. The CPU is pre-manufactured to perform as fast as possible the sequential instructions needed in daily computer activities, which not only include simulating our models, but also browsing the internet, checking emails, and so forth. As such, it lacks the gains from hardware specialization that are accessible to FPGAs. That said, the `C++`-compiler on a CPU (and GPU) goes a long way to execute the code as fast as possible by *automatically* performing low-level instruction parallelism. We can instruct the compiler to do this as much as possible by compiling the code using the aggressive `-O3` optimization flag in our `G++` compiler.

Second, our baseline comparison is that of a single FPGA chip against a single CPU core while computing N_E economies (which we will define momentarily). This benchmark provides an apple-to-apple comparison of the differential performance of the two accelerators.

Third, to get insights into the potential of FPGAs to structurally estimate our model by solving it for many different parameter values, we introduce a multi-economy parallelism. But

we do so in a controlled way. A well-known drawback of Open-MPI parallelization is that the total execution time is determined by the execution time of the slowest core. To temper this effect, we proceed as follows. First, we pick a number of economies ($N_E = 1,200$) that is uniformly divisible across the different CPU cores in our AWS instances (1, 8, 48 cores). Second, we determine the benchmark speedup (Table 3) by solving the same economy (i.e., the same set of parameter values) multiple times. In this way, we eliminate heterogeneity in the solution time attributed to differences in convergence of the iterative algorithm with different parameter values. Under this setup, we ensure that we use all CPU cores, and the speedup scales linearly with the number of cores. Thus, we eliminate any possible additional further gains from multi-core parallelization. Incidentally, this exercise illustrates how easy it is to implement increasingly aggressive acceleration by deploying multiple FPGAs in parallel.

In summary: our benchmarking strategy eliminates all differences in performance between FPGAs and CPUs that are not inherently linked to the differences in their architecture. If we could rewrite the solution algorithm more efficiently, that better code would improve the performance of both FPGAs and CPUs and leave the relative performance (roughly) unchanged.

Table 3: Efficiency Gains of FPGA Acceleration

CPU-cores	**Speedup**			**Relative Costs (%)**			**Energy (%)**		
	FPGAs			*FPGAs*			*FPGAs*		
	1	*2*	*8*	*1*	*2*	*8*	*1*	*2*	*8*
1	78.49	156.38	604.38	17.67	17.73	18.35	5.26	5.28	5.46
8	11.00	21.91	84.68	15.76	15.82	16.37	4.69	4.71	4.87
48	1.67	3.32	12.83	17.34	17.40	18.01	5.16	5.18	5.36

Note: Speedups provided by the FPGA and cost and energy usage of the FPGA relative to the CPU. These results are obtained by solving 1,200 baseline economies using AWS instances connected to 1, 2, and 8 FPGAs (columns) and using open-MPI parallelization on AWS instances with 1, 8, and 48 cores (rows). Speedup is obtained by dividing the total execution time in the CPU by that in the FPGA. Relative costs and energy are calculated using on-demand AWS prices and total energy consumption, and reported as FPGA usage as a percent of CPU usage. Table A.3 in Appendix C reports the details.

Table 3 reports how the relative performance of FPGAs vs. CPU cores varies as we vary the number of CPU cores and FPGA devices. We solve for 1,200 times the baseline economy on AWS instances connected to one (f1.2xlarge), two (f1.4xlarge), and eight (f1.16xlarge) FPGAs. We compare these results with the ones obtained by solving the model on AWS instances with one (m5n.large), eight (m5n.4xlarge), and forty-eight (m5n.24xlarge) cores. Then, we measure the relative performance across speedups, cost savings, and energy savings. Table A.3 in the Appendix complements this information by reporting execution time, costs, and energy consumption by instance.

5.1 Speedups of FPGA Acceleration

As mentioned before, our baseline comparison is a single FPGA vs. a single CPU core sequential solution. The FPGA acceleration delivers a 78 times speedup. The computation time drops from 10 and half hours in the CPU, to 8 minutes in the FPGA (Table A.3).

Next, we document the FPGA performance against that of multi-core CPUs. As many researchers have access to high-end laptops with at least eight cores, we first compare one FPGA with the 8-core acceleration. The computation time is reduced from approximately one and half hours in the CPUs to 8 minutes in the FPGA (Table A.3). FPGA acceleration also compares favorably with respect to the CPU multi-core parallelization on the AWS instance with the largest number of cores (48): a single FPGA device is 1.67 times faster than an Intel Xeon (Cascade Lake, second generation) platform running 48 cores in parallel.

Scaling these acceleration gains is easy, as it requires minimal modifications of the code to deploy multiple FPGAs in parallel. Table 3 shows that eight FPGAs in parallel reduce the execution time by 604x, 85x, and 13x when compared with the 1-core, 8-core, and 48-core acceleration, respectively.

Remark: The first three columns of Table 3 show how the differential performance of FPGAs vs. CPU cores scales linearly in the number of FPGA devices and CPU cores. This result corroborates the effectiveness of our benchmarking strategy in eliminating all contamination due to parallelization.

5.2 Cost Savings of FPGA Acceleration

We now turn to cost savings. The AWS cloud pricing schedules provide market-based measures of the cost of solving an application across different hardware architectures. We compute these costs by multiplying the total execution time for the AWS instance on-demand prices (Table A.1). Table 3 shows that our application solves in the FPGA instances at less than a quarter of the cost of the CPU instances. This result is relevant because the structural estimation of parameters may require computing up to one million different economies. Hence, moving from CPU to FPGA computing would reduce the estimation costs by hundreds of dollars (from $1043 to $184). Albeit the rental cost of FPGA instances is larger, the acceleration documented in our application more than justifies the expense.[12]

Remark: The linear speedup performance, combined with Amazon AWS linear price schedules, yields approximately constant costs and energy efficiency across different combinations of FPGA devices and CPU cores.

[12] Another cost comparison is with an in-house cluster. Even without entering into its purchase cost, clusters are expensive to maintain. In our application, a single FPGA chip can perform the same task as a cluster with 78 cores. This is a medium-to-high scale cluster, whose maintenance requires an HPC specialist, with a salary averaging around $85,000 per year in the US circa 2022. See Zip Recruiter.

5.3 Energy Savings of FPGA Acceleration

We determine the energy consumption (joules) by multiplying the total execution time for the power consumption (watts) of FPGA and the CPU chips. We use the AFI management tool to measure the FPGA peak power consumption, 33 watts per FPGA device. We use the procedure discussed in Appendix C.2.1 to measure the CPU power consumption, 8 watts per CPU core. The energy consumed by FPGA chips is less than 5.46% of the energy consumed by CPUs.[13] This statistic is relevant for organizations with in-house computational clusters (like research departments at central banks), whose computational needs are often constrained by the power limits on the cluster installation. Moving from CPU to FPGA computing enables more computations with the same energy.

Furthermore, the back-of-the-envelope calculations in Appendix D suggest that the associated energy savings may also be relevant to reducing the carbon footprint impact of research computing. As we describe in Appendix D, it has been estimated that the RMACC Summit and Blanca Supercomputers at the CU Boulder Research Computing Center emit 838.78 metric tons of CO_2 in the atmosphere every year to provide 150 million core hours to their research community. This is equivalent to the CO_2 emissions of 168 cars per year. If (a big if) we assume a type of acceleration similar to the one measured in our experiment and the FPGA power consumption recorded on Amazon Xilinx VU9P, transitioning all of these CPU-intensive computations to FPGA chips would reduce the CO_2 impact of these major supercomputing facilities to 27.12 metric tons of CO_2, or five cars per year.

5.4 Robustness

We perform a battery of robustness tests to study the performance of our hardware design. Different from the RTL approach proposed in Peri (2020), the compiler approach allows us to implement these experiments with minimal modifications of the `C/C++` code and hardware design configuration file.

In our first exercise, we solve the baseline economy using the single-kernel design in one FPGA and compare the performance with that of the one-core sequential solution. This result suggests that our single-kernel design can accelerate model experimentation, a valuable feature in the early stages of a research project when the ingredients of the final model are not yet determined.

The sequential computation of the 1,200 economies using the single-kernel design is relatively faster than that of the three-kernel design. Theoretically, instantiating three single-kernel designs

[13] As CPU power consumption is proportional to the number of CPU cores, Power(cores) $= P \cdot$ cores, and execution time is inversely proportional to the number of CPU cores, Execution Time (cores) $= T/$cores, first-order the energy is constant across CPU cores Energy(cores) $= P \cdot$ cores $\cdot\, T/$cores $= PT$.

Table 4: Speedup Comparison One-Kernel Single FPGA vs. Single CPU Core

FPGA-Time(sec)	*CPU*-Time(sec)	*Speedup*(x)	*Relative Costs*(%)	*Energy*(%)
0.84	31.54	37.66	36.81	7.30

Note: Columns 1-2: average execution time (in seconds) to compute the baseline economy in a single-kernel single-device FPGA (f1.2xlarge) and a one-core instance (m5n.large), respectively. Columns 3-5: efficiency gains of FPGA acceleration in terms of speedup, costs (in percent), and energy savings (in percent), computed as described in Table 3. Averages are computed over 1,200 economies. The FPGA peak power consumption on a single-kernel design is 22 Watts.

in the available SLRs should bestow a speedup of 37.66x $\cdot$ 3 SLR = 113x, yet the three-kernel design only reaches a speedup of 78x (Table 3). This loss in performance is due to the lower unrolling of the IHP design performed to save resources to fit the three kernels.

Table 5: Speedup Comparison across Grid Sizes

Individual Capital, N_k	*100*	*200*	*300*
1 FPGA vs. 8 Cores	11.00	14.16	14.59
2 FPGAs vs. 8 Cores	21.91	28.24	29.13
8 FPGAs vs. 8 Cores	84.68	109.68	114.50

Note: Speedups recorded by comparing the solution of 1,200 economies using AWS instances connected to 1, 2, and 8 FPGAs and using Open-MPI parallelization on AWS instances with 8 cores (rows) for different individual household capital N_k (columns).

In our second robustness exercise, we study how the speed gains depend on the size of the grids. Table 5 illustrates the speedup for increasingly finer grids on individual capital holdings $N_k = \{100, 200, 300\}$, obtained by comparing the performance of 1, 2, and 8 FPGA instances against that of an 8-core CPU. The performance improves with the size of the grids, that is, with the number of computations performed on the FPGAs.

6 Inspecting the Mechanism

What explains the observed speedup of FPGA vs. CPU multi-core acceleration? We address this question by discussing the performance benefits of gradual modifications of our code controlling the three critical acceleration channels: pipelining, data parallelism, and precision.[14]

[14]The bottlenecks faced in our design are common to most dynamic programming problems. So, our acceleration strategy provides easily transferable tools to accelerate a vast class of economic models.

We start by illustrating the performance of a baseline model, whose hardware image is created by automatic optimization of the HLS compiler (*Baseline Model*). Next, we build up the acceleration by resolving problems that prevent us from achieving an efficiently pipelined loop (*Pipelining Channel*). Once we achieve a pipelined kernel, we exploit available resources to instantiate multiple copies of the pipelined loops (*Within-Economy Data Parallelism Channel*). We conclude by instantiating the three-kernel design across available SLRs and FPGA devices to run multiple economies (*Across-Economies Data Parallelism Channel*).

Table 6: Speedup Gains: Acceleration Channels Accounting

	Baseline	*Pipelining*	*Data Parallelism*	
			Within Economy	*Across Economies*
Single-core Execution / FPGA Solution	0.40	0.57	37.66	78.49
CL Resources Utilization (%)				
BRAM	5.48	8.45	22.26	18.33
DSP	6.13	12.87	31.13	66.92
Registers	3.94	5.24	12.03	30.65
LUT	6.11	9.14	25.17	67.53
URAM	5.50	5.50	5.50	18.33

Note: Column 2 reports the speedup for a kernel design where all acceleration channels are switched off (baseline). Columns 3-5 report the speedup associated with implementing efficient pipelines (Column 3), introducing data parallelism in the kernel design (Column 4), and instantiating three kernels in the same FPGA (Column 5). The speedup (row 1) is computed by dividing the total execution time in the one-core CPU by the solution time (which does not include FPGA-host communications) in the FPGA. The acceleration in Columns 2-4 is performed using a single-kernel single-device FPGA (f1.2xlarge), where Column 4 coincides with the single-kernel design. The acceleration in Column 5 is performed by deploying the three-kernel design in parallel across the three SLRs in a single FPGA (f1.2xlarge). Averages are computed over 120 economies. Resources are expressed as a percentage of the Xilinx VU9P FPGA's resources utilized by AWS images associated with the different designs (columns). *Total Resources:* BRAM (1,680), DSP (5,640), Registers (1,790,400), LUTs (895 thousand), URAM (800). Total resources are computed using Xilinx Vivado.

Table 6 reports the speedup obtained by comparing the average time required to solve the baseline economy under each of these versions of our hardware design against the time to solve it in a single-core CPU. To isolate the acceleration channels, we replace the FPGA execution time with the solution time, which is the time required to solve the algorithm on the FPGA and abstract from the time absorbed by host-FPGA communications. The following paragraphs elaborate on the details.

6.1 Baseline

Column 2 in Table 6 reports the speedup of solving the model using the single-kernel FPGA baseline design and the one-core CPU sequential solution. The baseline design differs from the single-kernel design discussed in Section 4.1.3 because it does not explicitly call for any user-defined hardware optimizations (pipelining, unrolling, data precision). The only optimizations present in this design are the ones that are automatically performed by the HLS compiler. The compiler indeed tries to optimize the memory layout (to reduce the memory bottlenecks) and the loop pipelining (by trying to unroll inner loops). To reduce the latter effect, we use directives to limit the amount of automatic unrolling.

The FPGA solution is reached in 80 seconds, approximately half as fast as the CPU solution, which is reached in 32 seconds. This result is not surprising. FPGAs operate at a slower frequency than CPUs (in our case, 250MHz vs. 3.5GHz). Absent user-defined interventions –in pipelining, data parallelism, and data precision– the CPU should be faster. That said, the compiler goes a long way in optimizing the design, as we would have expected the FPGA to be 14x (3.5GHz/250MHz) slower than the CPU, if only due to differences in clock frequency.

This acceleration illustrates how FPGAs' gains are a by-product of hardware specialization and not the result of the chip being intrinsically faster. Slower but better-organized tasks in the FPGA deliver higher performance than faster but "poorly" organized execution of the same tasks in the CPU. The next subsections discuss how we specialize the custom logic of our FPGA to efficiently organize these tasks in assembly lines and how we improve performance by placing multiple assembly lines in parallel.

6.2 Pipelining

Next, we discuss the main changes we made in order to efficiently pipeline the **IHP** and **Simulation** steps in the single-kernel design.

Interpolation. We accelerate interpolation as follows. First, we declare the loop bounds of the individual and aggregate capital grids (namely, $\{0, N_k\}$ and $\{0, N_M\}$) as fixed constants, allowing the compiler to autonomously physically *place* the required CL resources (*space dimension*). Next, we implement a jump search algorithm to find the interpolation interval over the individual capital grid. The compiler instructs the hardware to pipeline a parallel reduce tree algorithm with three stages. Each stage determines the index of the smallest grid value larger than the interpolation point $k'(k, \epsilon, m, A)$ by performing comparisons in parallel. The number of comparisons varies by stage and grid size and ensures that the entire grid is examined, $i = \{0, \ldots, N_k\}$. The winner of each stage determines the search area of the successive stage. Since the result of this operation is part of a pipeline where the only dependence on subsequent loop iterations is through a final accumulation, we achieve an **II** of 1 in the **IHP** step.

Importantly for context, the CPU cannot physically place CL resources to make these comparisons in parallel, as its silicon is pre-manufactured and cannot be programmed. We could potentially implement the described parallel-search algorithm using multiple cores. But this design would be very inefficient, as the data transfer overhead costs would dominate the increase in performance. Conversely, our single FPGA vs. single CPU core and multi-core CPU benchmarking exercises are efficient, as they keep all CPU cores busy, minimizing data transfer overhead costs.[15]

Accumulation Data Precision. The efficient design of the interpolation function accelerates both the **IHP** and **Simulation** steps. While the **IHP** step attains **II** = 1, the **Simulation** step has an iteration-to-iteration limit in the computation of the cross-sectional average of individual capital holdings, m_t, and settles at an **II** of 5.

The way we perform accumulation in Equation (15) inhibits full pipeline parallelism. First, the floating-point addition is non-associative; hence, the compiler insists that the additions involved in the computation of m_t be performed in the original serial order.[16] Second, each floating-point addition takes 5 clock cycles to complete at 250 MHz on the Xilinx VU9P. Ultimately, m_t is required to compute the individual capital holdings policy functions $k'(k, \epsilon, m, A)$. This data dependency also prevents effective parallelism from unrolling, forcing a P-unrolled pipeline design to require $5P$ cycles per input (**II** = $5P$).

We circumvent this problem by performing the accumulation in Equation (15) in fixed-point precision.[17] Fixed-point additions are associative. Hence, the compiler can build a parallel reduce tree, avoiding the serialization for the addition across multiple unrolled pipelines. Besides, fixed-point additions using 72 bits are completed in a single cycle. These features allow fixed-point accumulations to run at least 5x faster than the floating-point computation with a single pipeline and to exploit unrolling parallelism to run even faster. After performing these changes the **Simulation** step reaches an **II=1**. Table 6 records a speedup of 0.57 with respect to a single-core CPU. Importantly, currently available CPUs do not provide access to user-defined fixed-precision arithmetic.

Our acceleration strategy trades off the accuracy of results for speed. The precision of the accumulations that occur in the algorithm can tolerate fixed-point calculations for the following reasons. First, the inputs to the accumulation are all positive values such that there will not be cancellation, which can often degrade the precision when moving from floating-point to fixed-point calculations. Second, since these are accumulations, we expect the accumulator

[15] The `C++` to CPU compiler can autonomously decide to perform these operations in parallel, but this step is not controlled by the coder.

[16] The associative property of addition is lost when we move from real numbers to their finite-precision approximation. Goldberg (1991) provides a textbook example, by showing how $(x + y) + z$ and $x + (y + z)$ give different results when $x = 1e30$, $y = -1e30$ and $z = 1$. The first equals 1 and the second equals 0.

[17] See Appendix E for a brief overview of finite-precision approximation.

Table 7: Precision Accuracy Analysis

Panel A: ALM Coefficients

	$\beta_1(a_b)$	$\beta_2(a_b)$	$\beta_1(a_g)$	$\beta_2(a_g)$
Floating-Point	0.1459	0.9599	0.1554	0.9587
Fixed Point	0.1459	0.9599	0.1554	0.9587

Panel B: Policy Function, k'

$\text{Mean}\left(\frac{\lvert\text{Fixed}-Float\rvert}{Float}\right)\%$	1.1e-09	$\text{Max}\left(\frac{\lvert\text{Fixed}-Float\rvert}{Float}\right)\%$	4e-08

Panel C: Individual Capital Holdings Distribution, $T = 1,100$

	Mean	Std	0.25	0.5	0.75
Floating-Point	40.49	133.44	12.23	16.00	19.78
Fixed Point	40.49	133.44	12.23	16.00	19.78
$\text{Mean}\left(\frac{\lvert\text{Fixed}-Float\rvert}{Float}\right)\%$	5.1e-08			$\text{Max}\left(\frac{\lvert\text{Fixed}-Float\rvert}{Float}\right)\%$	6.4e-07

Note: Panel A reports the equilibrium ALM coefficients $\hat{b}(a) = (\hat{b}_1(a), \hat{b}_2(a))$ with $a \in \{a_b, a_g\}$ under floating-point and fixed precision. Panel B reports the mean and max relative difference (in percent) between the policy functions computed under floating-point and fixed precision. Panel C reports moments of the distribution of individual capital holdings at $T = 1,100$ (mean, standard deviation, and quartiles) under floating-point and fixed precision. The last row reports their mean and max relative difference in percent.

not to hold a small value but rather converge to a value in a limited range and magnitude. In our baseline economy, the accumulator sums up to values between 15.371273672208304 and 404851.76387144416. Knowing the accumulator's range, we can determine the required precision. In particular, we need at least $D_1 = 6$ and $D_2 = 15$ decimal digits above and below the decimal point, to represent *404851*.76387144416 and 15.*371273672208304*, respectively. That is, we need $\lfloor \log_2 10^{D_1} \rfloor + 1 = 20$ binary digits to represent *404851*, and $\lfloor \log_2 10^{D_2} \rfloor + 1 = 50$ binary digits to represent *0.371273672208304*. Accordingly the minimum number of digits to represent our accumulator range is $50 + 20 = 70$. In order to accommodate a larger range of values (as may be required when we change economies θ), we set the number of digits to 72.

Not surprisingly, Table 7 shows that the results are very accurate. The estimated ALM coefficients are identical up to the 9th decimal place (R^2 are all 0.999 and therefore not reported). The approximations of policy functions and distribution of individual capital holdings at $T = 1,100$ are very good, as measured by the mean and max relative difference in percent of these objects under floating-point and fixed precision (less than 6.4e-7). Moments of the distribution of individual capital holdings are identical up to the seventh decimal place.

6.3 Within-Data Parallelism

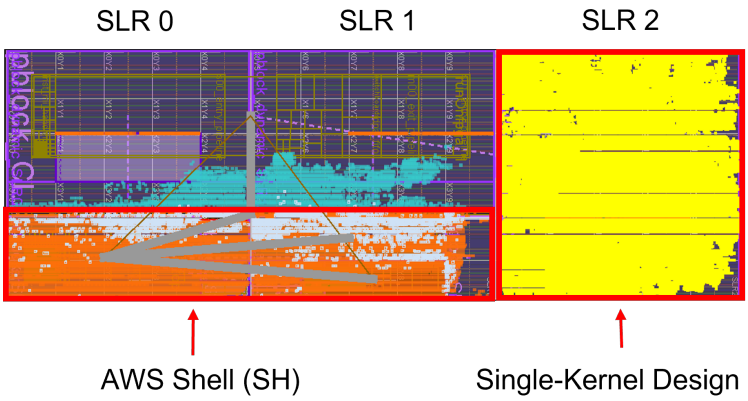
The interpolation and accumulation designs yield efficient custom pipelines but leave a considerable amount of CL resources unused in the single SLR (Table 6). Hence, our next step is to identify parts in the algorithm that can be parallelized. The computations involved in the interpolation step (Section 3.D.*(ii).(b)*) and custom fixed-precision accumulation operator of the **Simulation** provide suitable candidates.

The designer can perform a trade-off analysis between the resource utilization and the execution speedup to arrive at a conclusion. Eight copies of the pipeline work well for our design, as it brings the execution time of the **Simulation** step (371ms) closer to that of the **IHP** step (381ms), when we unroll the latter over the states by a factor of two. We call the design that follows from these changes the single-kernel design (Section 4.1.3).[18] Table 6 records a speedup of 37.66 with respect to the single-core CPU.

6.4 Across-Economies Data Parallelism

We tailored our CL design to compute three economies in parallel. However, the single-kernel design consumes more resources than the ones available in the largest SLR, leaking into the adjacent SLRs with non-time-critical operations (blue area in Figure 1). Also, the AWS shell covers a good share of resources in the adjacent SLRs (orange area in Figure 1).

Figure 1: Single-kernel Design: Resource Utilization

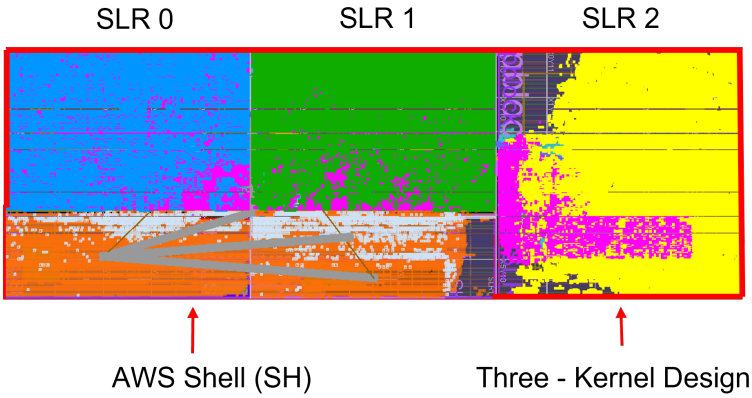


Note: Resources utilized by: *(i)* the single-kernel CL design (yellow area); *(ii)* by the AWS Shell (orange area); and *(iii)* available CL resources (other colors). The image is captured using Xilinx Vivado.

So, to fit three kernels, we slightly modify our design by reducing the usage of CL resources at the expense of performance. To do so, we provide directives to the compiler to halve the

[18]The recorded performance represents a lower bound to the performance that could be achieved by optimizing the individual grid size designs.

Figure 2: Three-kernel Design: Resource Utilization



Note: Resources utilized by: *(i)* the three-kernel CL design (yellow, green, blue areas each corresponding to one kernel); *(ii)* by the AWS Shell (orange area); and *(iii)* available CL resources (other colors, of which the pink area serves as a wrapper). The image is created using Xilinx Vivado.

amount of unrolling. In particular, we reduce the unrolling in the **IHP** design from 2 pipelines working in parallel to a single pipeline. Figure 2 shows the usage of resources associated with the three-kernel design.

Hence, we replicate the three-kernel design in each SLR and use `OpenCL` commands to launch one independent kernel in each of the three SLRs within a single FPGA device (f1.2xlarge). The FPGA design becomes 78.49 times faster than a single-core CPU. Table 3 shows how we exploit this parallelism further to deploy more than one FPGA device in parallel on the f1.4xlarge and f1.16xlarge instances.

7 Toward Electrical Engineering Economics

This paper proposes the design of FPGAs for the solution of economic models by using Xilinx HLS compilers. This approach requires minimal knowledge of hardware design principles, dramatically reducing the entry barriers to FPGA acceleration for economists. With small modifications of standard `C/C++` code, a single FPGA can deliver the same performance as 78 CPU cores when solving a canonical heterogeneous agent model. The associated energy savings make the compiler approach particularly appealing for organizations with in-house clusters (like research departments at central banks), whose computational needs are often constrained by the power limits on the cluster installation and the need to reduce the carbon footprint.

Our analysis leaves important venues for exploration. First, the recent popularization of machine learning techniques for the solution of economic models (Fernández-Villaverde et al., 2019, and Kahou et al., 2021, among many others) may benefit from decades of research in

the electrical engineering literature (Nurvitadhi et al., 2017) in terms of the design of efficient FPGAs. A similar argument applies to the acceleration of maximum likelihood estimators.

Second, notice that despite their acceleration potentials, FPGAs (like GPUs) still represent off-the-shelf application-specific integrated circuits (ASICs). Their routing network, logical units, and memory are pre-designed by the manufacturer to be configurable, but they are not customized to serve any particular algorithm. With a growing literature and the development of sophisticated heterogeneous agent models to assess the effect of monetary policy or the economic impact of climate change, we foresee a not-too-distant future where central banks and other policymaking institutions would invest in the design of ASICs specialized in the solution of these models. It is difficult to give a precise estimate, but customized silicons could likely improve the current speedup up to three orders of magnitude, with one order of magnitude only due to the faster clock cycle. FPGAs represent a first step in this direction, as they are actively used in the industry to test the functionality of the hardware design of customized chips. Of course, designing and manufacturing these pieces of silicon is not cheap (on the order of tens to hundreds of millions of $). Yet, more and more private companies are willing to incur these costs, and we can safely argue that the beneficial effects of a better-informed monetary or climate change policy dwarf these costs.

In conclusion: there is space for a new field, electrical engineering economics, focused on the design of computational accelerators for economics. Our analysis and successful experience in other areas suggest that such a field can provide computational breakthroughs in the years to come.

Acknowledgments

We thank Yicheng Li (UPenn, Engineering) for the initial implementation of our hardware design. We thank Lucas Ladenburger, Marina Leah McCann and Paro Suh (CU Boulder, Economics) for outstanding research assistance. We thank Andrew Monaghan and the CU Boulder Research Computing Center for providing valuable insights. We also thank Giuseppe Bruno and Riccardo Russo (Bank of Italy) for testing an early version of the tutorial associated with this paper and Victor Duarte, Mahdi E. Kahou, and Jesse Perla for comments. This project used: *(i)* the RMACC Summit supercomputer, supported by the National Science Foundation (awards ACI-1532235 and ACI-1532236), CU Boulder and Colorado State University; *(ii)* AWS Credits awarded under the NSF CC* Hybrid Cloud Award OAC-1925766, Research Computing, CU Boulder, 2022. This project was also supported by the Undergraduate Research Experiences for Diversity Grant, 2021, Institute of Behavioral Science, University of Colorado, USA.

References

Achdou, Y., J. Han, J.-M. Lasry, P.-L. Lions, and B. Moll (2021). Income and wealth distribution in macroeconomics: A continuous-time approach. *Review of Economic Studies 89*(1), 45–86.

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

Algan, Y., O. Allais, and W. J. Den Haan (2008). Solving heterogeneous-agent models with parameterized cross-sectional distributions. *Journal of Economic Dynamics and Control 32*(3), 875–908.

Algan, Y., O. Allais, W. J. Den Haan, and P. Rendahl (2014). Solving and simulating models with heterogeneous agents and aggregate uncertainty. In *Handbook of Computational Economics*, Volume 3, pp. 277–324. Elsevier.

Amman, H. M., D. A. Kendrick, J. Rust, L. Tesfatsion, K. Judd, K. S. C. Hommes, and B. LeBaron (Eds.) (2018). *Handbook of Computational Economics*. Elsevier.

Aruoba, S. B. and J. Fernández-Villaverde (2015). A comparison of programming languages in macroeconomics. *Journal of Economic Dynamics and Control 58*, 265–273.

Auclert, A., M. Rognlie, and L. Straub (2020). Micro jumps, macro humps: Monetary policy and business cycles in an estimated HANK model. Working Paper 26647, National Bureau of Economic Research.

Azizi, N., I. Kuon, A. Egier, A. Darabiha, and P. Chow (2004). Reconfigurable molecular dynamics simulator. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 197–206.

Babb, J., M. Rinard, C. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe (1999). Parallelizing applications into silicon. In *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00375)*, pp. 70–80.

Bayer, C. and R. Luetticke (2018). Solving heterogeneous agent models in discrete time with many idiosyncratic states by perturbation methods. Mimeo, University of Bonn.

Bayer, C., R. Luetticke, L. Pham-Dao, and V. Tjaden (2019). Precautionary savings, illiquid assets, and the aggregate consequences of shocks to household income risk. *Econometrica 87*(1), 255–290.

Berczik, P., R. Männer, G. Marcus, R. Banerje, A. Kugel, R. Klessen, and G. Lienhart (2009). Accelerating astrophysical particle simulations with programmable hardware (FPGA and GPU). *Computer Science Research and Development*, 231–239.

Bhandari, A., D. Evans, M. Golosov, and T. J. Sargent (2017). Fiscal policy and debt management with incomplete markets. *Quarterly Journal of Economics 132*(2), 617–663.

Bilal, A. (2021). Solving heterogeneous agent models with the master equation. Technical report, University of Chicago.

Brumm, J. and S. Scheidegger (2017). Using adaptive sparse grids to solve high-dimensional dynamic models. *Econometrica 85*(5), 1575–1612.

Cai, Y. and T. S. Lontzek (2019). The social cost of carbon with economic and climate risks. *Journal of Political Economy 127*(6), 2684–2734.

Childers, D. (2018). Solution of rational expectations models with function valued states. Manuscript, Carnegie Mellon.

Cruz Álvarez, J. L. and E. Rossi-Hansberg (2021). The economic geography of global warming. Working Paper 28466, National Bureau of Economic Research.

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

Duarte, V., D. Duarte, J. Fonseca, and A. Montecinos (2019). Benchmarking machine-learning software and hardware for quantitative economics. *Journal of Economic Dynamics and Control 111*, 103796.

Fernández-Villaverde, J., S. Hurtado, and G. Nuño (2019). Financial frictions and the wealth distribution. Working Paper 26302, National Bureau of Economic Research.

Fernández-Villaverde, J. and D. Z. Valencia (2018). A practical guide to parallelization in economics. Working Paper 24561, National Bureau of Economic Research.

Frigo, J., M. Gokhale, and D. Lavenier (2001). Evaluation of the Streams-C C-to-FPGA compiler: An applications perspective. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, pp. 134–140.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR) 23*(1), 5–48.

Herbordt, M. C., J. Model, Y. Gu, B. Sukhwani, and T. VanCourt (2006). Single pass, blast-like, approximate string matching on FPGAs. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 217–226.

Hoang, D. (1993). Searching genetic databases on splash 2. In *[1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 185–191.

IEEE Standards Committee (1985). *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE.

Intel Corporation (2021). *Intel® High Level Synthesis Compiler Pro Edition User Guide (UG20037)*. Intel Corporation.

Judd, K. L., L. Maliar, S. Maliar, and I. Tsener (2017). How to solve dynamic stochastic models computing expectations just once. *Quantitative Economics 8*(3), 851–893.

Kahou, M. E., J. Fernández-Villaverde, J. Perla, and A. Sood (2021). Exploiting symmetry in high-dimensional dynamic programming. Working Paper 28981, National Bureau of Economic Research.

Kaplan, G., B. Moll, and G. L. Violante (2018). Monetary policy according to HANK. *American Economic Review 108*(3), 697–743.

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

Krusell, P. and J. Smith, Anthony A (2022). Climate change around the world. Working Paper 30338, National Bureau of Economic Research.

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

Mertens, T. M. and K. L. Judd (2018). Solving an incomplete markets model with a large cross-section of agents. *Journal of Economic Dynamics and Control 91*, 349–368.

Nagurney, A. (1996). Parallel computation. *Handbook of Computational Economics 1*, 335–404.

Nurvitadhi, E., G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh (2017). Can FPGAs beat GPUs in

accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 5–14.

Peri, A. (2020). A hardware approach to value function iteration. *Journal of Economic Dynamics and Control 114*, 103894.

Pröhl, E. (2015). Approximating equilibria with ex-post heterogeneity and aggregate risk. Research Paper 17-63, Swiss Finance Institute.

Reiter, M. (2009). Solving heterogeneous-agent models by projection and perturbation. *Journal of Economic Dynamics and Control 33*(3), 649–665.

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

Snider, G. (2002). Performance-constrained pipelining of software loops onto reconfigurable hardware. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*, pp. 177–186.

Winberry, T. (2018). A method for solving and estimating heterogeneous agent macro models. *Quantitative Economics 9*(3), 1123–1151.

Xilinx, Inc. (2020a). *Performance and Resource Utilization for Floating Point*. Xilinx, Inc.

Xilinx, Inc. (2020b). *UG1145: Xilinx Vitis Unified Software Platform User Guide*. Xilinx, Inc.

Young, E. R. (2010). Solving the incomplete markets model with aggregate uncertainty using the Krusell–Smith algorithm and non-stochastic simulations. *Journal of Economic Dynamics and Control 34*(1), 36–41.

Young-Schultz, T., L. Lilge, S. Brown, and V. Betz (2020). Using OpenCL to enable software-like development of an FPGA-accelerated biophotonic cancer treatment simulator. *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 86–96.

Online Appendix for Programming FPGAs for Economics

This online appendix adds further details to the main paper.

A FPGA Compilers

Both software and hardware programming have a history of raising the level of abstraction, with increasingly sophisticated automation to map higher-level abstractions down to low-level implementations. FPGAs are no exception. After decades of effort, there are now commercially supported `C`, `C++`, and `OpenCL` compilers that allow programming FPGAs in these high-level programming languages (Xilinx, Inc. 2020b; Intel Corporation 2021).

The existence of these new compilers opens two routes. The first route is to compile already-existing code written for sequential processors without any change. This route often goes by the jargon "dusty deck," conjuring the image of pulling an old deck of punched cards, typically for `FORTRAN`, off the shelf and compiling it, untouched, to the new hardware. The second route is to write or tune the code specifically for the parallel hardware target. Currently, this is the best route. In particular, we can use `#PRAGMA`s to instruct the compiler on how to exploit parallelism, pipelining, data distribution, and data streaming. These directives disambiguate where you want serial or parallel implementations and guide the compiler to provide higher-performance implementation. We can expect the tuning of many of these directives to be automated in the future.

When we customize the hardware at the bit level, as we can in FPGAs, the hardware admits to richer data types and optimization than traditional fixed-architecture processors. Fixed-architecture processors and GPUs are just starting to catch up with support for some of the more economical data types. Similarly, languages originally designed for processors with fixed-width datapaths are lagging behind in their direct support for customized precision. Hence, the "dusty deck" programs will not exploit the richer and more efficient precision options automatically, but can often be upgraded to use them with minor modifications.

B AWS Instances Technical Specs

M5N Instances. *(a) CPU:* Intel Xeon Scalable Processors (Cascade Lake, 2nd generation), with sustained all-core Turbo CPU frequency of 3.1 GHz, maximum single core Turbo CPU frequency of 3.5 GHz; *(b) Network Bandwidth:* up to 25 Gbps; *(c) Storage* EBS.

Remark. We select M5N instances for three reasons. First, their architecture roughly belongs to the same vintage as our FPGAs –with the Xilinx VU9P being released a little bit earlier (2016) than the Intel Xeon Scalable Processor (Cascade Lake, second generation, 2019) featured

Table A.1: Technical Specifications

AWS Instance	**Cores**	**FPGAs**	**Pricing** ($/hour)	**Memory** (GiB)
m5n.large	1	-	0.119	8
m5n.4xlarge	8	-	0.952	64
m5n.24xlarge	48	-	5.712	384
f1.2xlarge	1	1	1.650	122
f1.4xlarge	4	2	3.300	244
f1.16xlarge	32	8	13.200	976

Note: Hardware architecture and AWS cloud pricing (Columns 2-5) for deployed AWS instances (Column 1). The column marked Cores reports the number of physical cores. The column marked FPGAs reports the number of connected FPGA chips (f1 instances only). The column marked Pricing denotes the AWS *On Demand* Pricing per instance per hour, as of September 2021. Memory is measured in Gigabytes. *Source:* Amazon AWS.

in M5N instances– thus, allowing us to control for technological improvements. Second, these CPUs compare favorably with respect to CPUs available in state-of-the-art supercomputers, for instance, the Intel Xeon E5-2680 v3 @2.50GHz (2 CPUs/node, 24 cores/node) provided by the CU Boulder RMACC Summit supercomputer. As a result, they provide a good benchmark of the expected performance. Third, they are the Amazon AWS general purpose instances with the largest number of cores (as of 2022); hence, they enable meaningful multi-core parallelism while preserving comparability.

F1 Instances. *(a) CPU:* Intel Xeon E5-2686 v4 Processor, with base CPU frequency of 2.3 GHz and Turbo CPU frequency of 2.7 GHz. *(b) Network Bandwidth:* up to 10 Gbps for f1.2xlarge and f1.4xlarge, and 25 Gbps for f1.16xlarge. *(c) Storage f1.2xlarge*: 470 GiB NVMe SSD *f1.4xlarge*: 940 GiB NVMe SSD *f1.16xlarge*: 3760 GiB (4 940 GiB NVMe SSD).

Source: For further information, visit https://aws.amazon.com/ec2/instance-types/.

C Hardware Designs: Resources and Performance

We report now resource utilization and performance measures associated with the hardware designs discussed in the main paper.

C.1 Resource Utilization

First, Table A.2 reports resource utilization by hardware design.

Table A.2: Resource Utilization by Grid Size

Individual Capital, N_k	*100*	*200*	*300*
BRAM(%)	18.33	20.97	24.72
DSP(%)	66.92	66.92	66.92
Registers(%)	30.65	30.51	30.76
LUT(%)	67.53	68.88	70.35
URAM(%)	18.33	18.33	18.33

Note: Percentage of Xilinx VU9P FPGA's resources (rows) utilized by AWS images associated with different grid sizes (columns). *Total Resources:* BRAM (2,160), DSP (6,840), Registers (2,363,536), LUTs (1,181,768), URAM (960).

C.2 Efficiency Gains of Baseline Economy

Next, Table A.3 reports the performance of different FPGA hardware designs and CPU-core platforms that yield the efficiency gains reported in the paper in terms of execution speedup, AWS costs, and energy savings.

Table A.3: Performance Comparison

	CPU cores			**FPGA devices**		
N.	1	8	48	1	2	8
Time (s)	37854.52	5303.73	803.63	482.30	242.06	62.63
Cost ($)	1.25	1.40	1.28	0.22	0.22	0.23
Energy (J)	302836.16	339438.72	308593.92	15915.90	15975.96	16534.32
AWS Instance	m5n.large	m5n.4xlarge	m5n.24xlarge	f1.2xlarge	f1.4xlarge	f1.16xlarge

Note: The table reports time (in seconds), cost (in USD) and energy (in joules) to solve 1,200 baseline economies using Open-MPI CPU multi-core acceleration on Amazon M5N multi-core instances (with 1, 8, 48 cores, Columns 1-3) and using FPGA acceleration on Amazon F1 instances (connected to 1, 2, 8 FPGA devices, Columns 4-6).

C.2.1 Energy Consumption

The FPGA power consumption is measured using the AFI management tool command sudo fpga−describe−local−image −S 0 −M. To make our energy performance comparison as meaningful as possible, we select the peak FPGA power consumption (across all our experiments, including different capital grids), which amounted to 33 watts per FPGA device.

The CPU power consumption can be determined using the Turbostat application.[19] However, Turbostat does not work on Amazon M5N instances. As a workaround:

- We use Turbostat to measure the power consumption of our application on the Amazon AWS metal instance.
- We then compare this number with the Thermal Design Power (TDP).[20] The comparison between the Turbostat application and the Thermal Design Power (TDP) establishes that our application requires approximately the maximum CPU power.

We map this estimate into our M5N instances with 1, 8, 48 cores using the formula:

$$\text{Power M5N(cores)} = \frac{\text{cores}}{\text{cores}_{\text{Metal}}} * \text{Power Turbostat}, \qquad \text{cores} \in \{1, 8, 48\}.$$

We estimate a power consumption of 8 watts per CPU core. To get the energy, we compute:

$$\text{Energy M5N(cores)} = \text{Power M5N(cores)} \cdot \text{Time(cores)}, \qquad \text{cores} \in \{1, 8, 48\}.$$

C.3 Performance Across Grid Sizes

Finally, Table A.4 reports the performance across different sizes of the grid.

Table A.4: Time Performance by Individual Capital Grid Size, N_k

	8-core CPU	**1 FPGA**	**2 FPGAs**	**8 FPGAs**
$N_k = 100$	5303.73	482.30	242.06	62.63
$N_k = 200$	9502.63	671.28	336.54	86.64
$N_k = 300$	15432.15	1057.53	529.75	134.78

Note: Seconds required to solve 1,200 baseline economies using Open-MPI CPU multi-cores (Column 1) and FPGA acceleration (Columns 2-4) for different grid sizes on the individual capital $N_k = \{100, 200, 300\}$ (rows). Open-MPI CPU multi-core results are obtained using the Amazon M5N instance with 8 cores (m5n.4xlarge). FPGA results are obtained using the Amazon F1 instance connected to 1 FPGA (f1.2xlarge), 2 FPGAs (f1.4xlarge), and 8 FPGAs (f1.16xlarge).

D Carbon Footprint of Scientific Computing

This appendix proposes a back-of-the-envelope calculation in order to estimate the carbon footprint of the Summit and Blanca Supercomputers. Calculations have been provided by indepen-

[19] Source: https://www.linux.org/docs/man8/turbostat.html
[20] https://www.intel.com/content/www/us/en/support/articles/000055611/processors.html.

dent research at the CU Boulder Research Computing Center and updated to 2020 data.[21]

The RC analysis assumes that each CURC HPC core consumes 13W, that is 0.013 kilowatts per CURC HPC core hour ($13\text{W/core} \cdot 1\text{hour}/1000 = 0.013kWh$). It then uses the Xcel Energy power generation breakdown in the state of Colorado in 2020[22] –37% Natural Gas, 26% Coal, 37% Renewables– and US EPA information on the emissions of CO_2 per kWh by source[23] –0.91 Natural Gas, 2.21 Coal, 0.1 Renewables[24]– to determine the pounds of CO_2 per Xcel Colorado kWh:

$$0.37 * 0.91 + 0.26 * 2.21 + 0.37 * .1 = 0.9483 \, \frac{\text{lbs } CO_2}{\text{kWh}}$$

Putting this information together, it estimates 0.0123 pounds CO_2 per CURC HPC core per hour:

$$0.9483 \, \frac{\text{lbs } CO_2}{\text{kWh}} * 0.013 \, \frac{\text{kWh}}{\text{core hour}} = 0.0123 \, \frac{\text{lbs } CO_2}{\text{core hour}},$$

On average the Summit and Blanca supercomputers (CU Boulder) serve 150 million core hours per year, and therefore produce on average

$$150 \cdot 10^6 \, \frac{\text{core hour}}{\text{year}} \cdot 0.0123 \, \frac{\text{lbs } CO_2}{\text{core hour}} = 1,849,185 \, \frac{\text{lbs } CO_2}{\text{year}},$$

which corresponds to 838.78 metric tons of CO_2 per year. To put this number in context, a typical US car emits about 5 metric tons per year. So, the annual Summit and Blanca carbon footprint is roughly the same as that of $838.78/5 \approx 168$ cars per year.

To explore the carbon footprint impact of moving all of these CPU-intensive computations to FPGA devices, let us assume an FPGA power consumption similar to the one measured on the Xilinx VU9P of 0.033 kWh per FPGA per hour. Accordingly,

$$0.9483 \, \frac{\text{lbs } CO_2}{\text{kWh}} * 0.033 \, \frac{\text{kWh}}{\text{FPGA hour}} = 0.031 \, \frac{\text{lbs } CO_2}{\text{FPGA hour}}.$$

If (a big if) we assume an acceleration similar to the one measured in our application (78.49x), the 150 million core hours per year would map into 1,911,071 FPGA hours per year. In this scenario, the carbon footprint would total:

$$1,911,071 \, \frac{\text{FPGA hour}}{\text{year}} \cdot 0.031 \, \frac{\text{lbs } CO_2}{\text{FPGA hour}} = 59,804 \, \frac{\text{lbs } CO_2}{\text{year}}$$

or approximately 27.12 metric tons of CO_2 per year. This is equivalent to a reduction in the carbon footprint from 168 cars to 5 cars per year.

[21] Andrew Monaghan, Andrew.Monaghan-1@Colorado.EDU.
[22] *Source:* Xcel Stats, https://co.my.xcelenergy.com/s/energy-portfolio/power-generation.
[23] *Source:* US EPA https://www.eia.gov/tools/faqs/faq.php?id=74t=11.
[24] This estimate is not given. The original analysis assumes it to be 0.1 for externalized carbon.

E Finite Precision Approximation: An Overview

Computers carry out computation on numbers with finite representations. This raises the question of how we adequately approximate the uncountable real numbers. The advent of the IEEE floating-point standard (IEEE Standards Committee, 1985) and readily available microprocessors that implemented it drove convergence to the modern floating-point representations. Most researchers get enough accuracy from the double-precision version of this standard, and they do not need to think carefully about the impact of finite-precision numeric representations for many uses.

Nonetheless, double-precision costs hardware and energy. Single-precision floating-point remained of interest for energy-conscious signal processing and the highest throughput computations, as did fixed-point representations where the significance of the bits does not change (i.e., the decimal point remains in a fixed position –it does not "float"). When custom hardware, both VLSI and FPGAs, is designed, precision optimization remains a point of leverage. For example, in modern Xilinx FPGAs, a double-precision floating-point add can take 700 LUTs, while a 32b-fixed-point add only takes 16. A double-precision floating-point multiply takes over 2400 LUTs, while a 32×32 fixed-point multiply is only 1100, and a 16×16 multiply is around 300 (Xilinx, Inc., 2020a).