Programming FPGAs for Economics: An Introduction to Electrical Engineering Economics

Bhagath Cheela* André DeHon†

Jesús Fernández-Villaverde‡ Alessandro Peri§

June 9, 2024

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 69 CPU cores in a conventional cluster. The time to solve 1,200 versions of the model drops from 8 hours to 7 minutes, illustrating a 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 nearly all fields of economics. The over 3,000 pages of the four volumes of the renowned *Handbook of Computational Economics* (Amman et al., 2018) demonstrate this point. 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 taken hardware as a given.

This lack of interest is unfortunate. While the speeds of single-core central processing units (CPU) are stalling, 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.

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.

So far, the skills required to implement algorithms in hardware have represented a significant entry barrier, particularly in disciplines characterized by small research teams, such as economics. These domain-specific knowledge requirements are apparent in Peri (2020), the only other attempt we are aware of employing FPGA technology in economics. Following the industry standard for chip design, Peri (2020) employs a hardware description language to describe a circuit at the register transfer level (`RTL`) that solves a Bellman equation via value function iteration on an FPGA. Much like working with `assembly` language, this `RTL` design approach uses an intricate low-level syntax to configure the hardware, rendering the code hardly accessible to economists. More importantly, it requires a working knowledge of digital design, a subject that most economists are unfamiliar with. Our paper describes how to reduce these barriers for economics.

How do we get higher performance from a chip while retaining a reasonable easiness of programmability? By replacing the lower-level `RTL` design in Peri (2020) (the `RTL` approach) with the more accessible FPGA `HLS` `C`-to-gates compilers (the `HLS` approach).[1] Here, we follow a long history of increasingly sophisticated automation in computer science to map higher-level abstractions down to low-level implementations. Our `HLS` approach allows economists to

[1] 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.

implement complex circuits with minimal to no knowledge of hardware design and delegate the technical details to the compiler. More significantly, our approach dramatically reduces the coding and debugging time compared to `RTL`, facilitating the implementation of far more complicated dynamic models and the exploration of different model features, both key aspects of practical research. We illustrate these ideas by accelerating a major workhorse model in economics: the incomplete markets, heterogeneous agent model with aggregate uncertainty of Krusell and Smith (1998).

We introduce the concepts of FPGAs' programming and optimizations in the context of a simple yet illustrative example: the accumulation of array elements. Here, we showcase *all* the acceleration techniques deployed to accelerate the Krusell and Smith (1998) algorithm intuitively, which illustrates how progressive code modifications translate into gradually higher-performance FPGA circuits. Along the way, we compile a list of hardware issues that economists are likely to encounter when accelerating their applications, including all the performance bottlenecks faced while accelerating the Krusell and Smith (1998) algorithm. This approach allows us to distill essential hardware design principles for accelerating dynamic models without necessitating economists to become hardware designers themselves. In the process, we highlight the remarkable development advantages over the `RTL` approach proposed in Peri (2020) in terms of accessibility, coding, and debugging time. Last, we discuss the portability of our code, discussing, among other aspects, how `Matlab 2022a` and later editions can directly implement the accumulator problem.

We deploy our `HLS` approach on Amazon Web Services (AWS) to accelerate the solution of an incomplete markets, heterogeneous agent model with aggregate uncertainty. After the pioneering work of Krusell and Smith (1998), heterogeneous agent models have been used 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, such as 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 work with the canonical heterogeneous agent model 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 Krusell and Smith (1998) algorithm as implemented by 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).[2] 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).[3]

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, and data distribution required to solve our algorithm efficiently.[4] 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. The FPGA delivers a speedup of nearly 69 times against an Intel Xeon Scalable Processor (Cascade Lake, second generation) core. That is, we solve the same heterogeneous agent model 69 times faster in one FPGA than in one CPU core, reducing the time to solve 1,200 economies from 8 hours to 7 minutes.

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

[2]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 penalty that may arise from `C++` object-oriented programming. Thus, we give the CPU the best fighting chance against FPGA acceleration.

[3]All replication codes are available on a `Github` repository: https://github.com/AleP83/FPGA-Econ.git.

[4]Our online tutorial Cheela et al. (2023) supplies a detailed guide on how to utilize `#pragma`s to accelerate the solution of our model.

1.48 times faster than 48 CPU cores do and that eight FPGAs solve 1,200 economies 549 times faster than one CPU core and 12 times faster than 48 CPU cores.

The FPGA speedups in these exercises are accompanied by large cost savings (less than 20 % of the CPU cost) and even more impressive energy savings (less than 6% 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. In the process, we tune the precision and data representation to achieve efficient pipelining and support additional parallelism. Second, we exploit loop-level data parallelism to perform computations on multiple independent pipelines simultaneously. Third, we employ coarse-grained, data-level parallelism to compute multiple economies in parallel.

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. 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 useful when structurally estimating the model using micro and macro data or 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 summary, the design of specialized computational accelerators for specific yet vital computational tasks in economics at attractive price points and reasonable programming complexity holds much promise. 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 introduces the building blocks of FPGAs'

programming and optimizations. Section 5 describes the acceleration schemes. Section 6 reports quantitative results and performs robustness tests. Section 7 isolates the acceleration channels responsible for the speed gains. Section 8 concludes. An online appendix and a tutorial provide further details (Cheela et al., 2023).

2 The Model

This section presents the incomplete markets model in Den Haan et al. (2010). This is a 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 a borrowing constraint.

The household's problem. Households consume $c_t \in \mathbb{R}_+$, supply labor $\bar{l}$ (paid at a wage w_t), and accumulate capital $k_{t+1} \in \mathbf{K} \subseteq \mathbb{R}_+$ (which receives a rental rate r_t). Households suffer idiosyncratic shocks to their employment status $\epsilon_t \in \{0,1\}_\epsilon$, which equals one if employed and 0 if unemployed. We will specify the stochastic process for ϵ_t below. Also, τ_t is the labor income tax rate, μw_t is the (tax-free) unemployment benefit, and δ is the depreciation rate.[5]

Thus, households choose sequences of consumption and physical capital to solve:

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

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

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

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 technology $Y_t = A_t K_t^{\alpha}(\bar{l}L_t)^{1-\alpha}$, where $0 < \alpha < 1$. The aggregate productivity A_t follows a two-state 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

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

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

Government. 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 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 $\{w, r\}$ equal the marginal products (4)-(5); iii) given A, τ balances the government budget, (6); iv) the markets for labor and capital clear; v) given $\{w, r, \mathbf{\Gamma}, k', A\}$ 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	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:

$$m' = H(m, A, A')$$

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

A. Grids. We define the intervals on the household's capital $\mathbf{K} \equiv [k_{\min}, k_{\max}] = [0, 1000]$ 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. Subsection 6.5 explores alternative grid sizes.

B. Individual households' problem (IHP). A stationary solution to the optimization problem (1) is the saving policy function $k' : \mathbf{K} \times \{0,1\}_\epsilon \times \mathbf{M} \times \mathbf{A} \to \mathbb{R}_+$:

$$\begin{aligned} k' = & \left[\mu(1-\epsilon) + (1-\tau)\bar{l}\epsilon\right] w + (1-\delta+r)k \\ - & \left\{\lambda + \beta\mathbb{E}\left[\frac{1-\delta+r'}{\left(\left(\mu(1-\epsilon') + (1-\tau')\bar{l}\epsilon'\right)w' + (1-\delta+r')k' - k'(k')\right)^\gamma}\right]\right\}^{-1/\gamma}, \end{aligned} \tag{8}$$

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 $k' \geq 0$ and complementary slackness condition $\lambda \cdot k' = 0$, with $\lambda \geq 0$.

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

(i) Initial guess. Guess an initial policy function k'_0. 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:

$\widehat{k}'_{i+1} = \Phi k'_i$	*(a)* Solve (8)
$k'_{i+1} = \eta_k \widehat{k}'_{i+1} + (1-\eta_k)k'_i$	*(b)* 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 (8).
- 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:

$$k'_{i+1} = \eta_k \widehat{k}'_{i+1} + (1 - \eta_k) k'_i. \tag{9}$$

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

$$\rho\left(k'_{i+1}, k'_i\right) = \max_{(k,\epsilon,m,A)\in \mathbf{K}\times\{0,1\}_\epsilon\times\mathbf{M}\times\mathbf{A}} |k'_{i+1} - k'_i| < \varepsilon_k = 1e(-8). \tag{10}$$

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:

$$\ln m' = f(a, m; b) = b_1(a) + b_2(a) \ln m \qquad a \in \{a_b, a_g\}, \tag{11}$$

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.[6] Use equations (4), (5), and (6) to compute w_t, r_t, and τ_t.

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

(a) **IHP.** Estimate the households' 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 households' capital holdings

$$m_t = \frac{1}{J} \sum_{j=1}^{J} k_{j,t}. \tag{12}$$

[6] 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 (13) is higher than $1e(-6)$.

(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.[7]

(c) **ALM.** Estimate $\hat{b}^i(a) = (\hat{b}^i_1(a), \hat{b}^i_2(a))$ with $a \in \{a_b, a_g\}$ by running the OLS regression associated with (11), after discarding the first 100 observations, $t = 1, \ldots, 100$.

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

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

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

$$\sqrt{\sum_{l \in \{1,2\}, a \in \{a_b, a_g\}} (b^{i+1}_l(a) - b^i_l(a))^2} < \varepsilon_b = 1e(-8). \tag{13}$$

4 Building Blocks of FPGAs' Optimizations

Next, we introduce the building blocks of FPGA optimizations: data precision, loop pipelining, loop unrolling, and memory management.

We do so by focusing on a simple example: the sum of eight numbers, $\sum_{i=0}^{8} X_i$ ("eight" is just for concreteness, nothing essential depends on it). This accumulation problem, which is equivalent to the problem of computing the cross-sectional average of individual capital holdings in equation (12) by summing J numbers, allows us to showcase all the acceleration techniques deployed in our main application. In the process, we illustrate how these circuit design tools can be exploited with minimal alterations of existing `C` code and we highlight the development advantages vis-á-vis the `RTL` approach proposed in Peri (2020), underscoring the potential of `HLS` programming for economics.

For expositional convenience, we focus our discussion on the *kernel*, that is, the algorithm the hardware designer seeks to accelerate by programming the custom logic of the FPGA (the accumulator in this section; the Krusell and Smith (1998) algorithm, later on). Our tutorial (Cheela et al., 2023) delves into the details of the non-kernel boilerplate code required to implement our application in hardware. Since the kernel is the only code changing across the exercises in this section, this choice comes without loss of generality.

[7] Maliar et al. (2010) use 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/.

4.1 Programming FPGAs Using HLS

Programming an FPGA is the *design* of a physical circuit by connecting the chip's custom logic (`CL`). Consequently, we will use the term "`CL` design" interchangeably to refer to circuit design moving forward. FPGA chips are built on the fundamental idea that we can implement Boolean combinational functions using memories, also referred to as lookup tables. By strategically organizing enhanced versions of these lookup tables, known as configurable logic blocks (CLBs), within a programmable grid, FPGAs enable the physical implementation of circuits that would otherwise require months to fabricate on traditional silicon chips.

This premise underscores a conceptual difference between *hardware* and *software* programming. While software programming yields an executable application designed to run on existing hardware, FPGA programming creates a circuit. Beyond this conceptual difference, software and hardware programming share a historical quest to reduce implementation costs by raising the level of abstraction. By leveraging the use of *compilers*, software programming has progressively replaced low-level (`assembly`) language with increasingly sophisticated higher-level languages (`C/C++`, `Fortran`, `Julia`, etc). The early adoption of compilers was driven by the necessity to take advantage of the rapid increase in the number of transistors in the late twentieth century (Moore's law). The stalling speed of single-core central processing units (CPU) has made the *compiler* approach increasingly more attractive in hardware programming, too, in order to make the design of circuits for customized hardware less time-consuming. After decades of effort, commercially supported compilers (Xilinx, Inc., 2020b; Intel Corporation, 2021) are now available to replace the widely used low-level register transfer level (`RTL`) design. Similar to `assembly`, `RTL` coding employs a low-level syntax to design the required hardware configurations, making the code hardly accessible to trained economists. Our goal is to explain how FPGA compilers break this barrier.

To make this point, we illustrate the simplicity of implementing the sum of $J = 8$ numbers in hardware using `HLS`. This operation requires just compiling existing `C` code written to be executed on the CPU with the `HLS v++` compiler in place of, say, the `g++` compiler,

Listing 1: Sequential Accumulator

```
void hw_loop(array_type st_k[J],array_type &reduced_sum){
    array_type sum = 0.;
    for(int i=0;i<J;i++) {
        sum+=st_k[i];
    }
    reduced_sum = sum;
    return;
}
```

where `array_type` is a double-precision floating-point

```
1 #typedef double array_type;
```

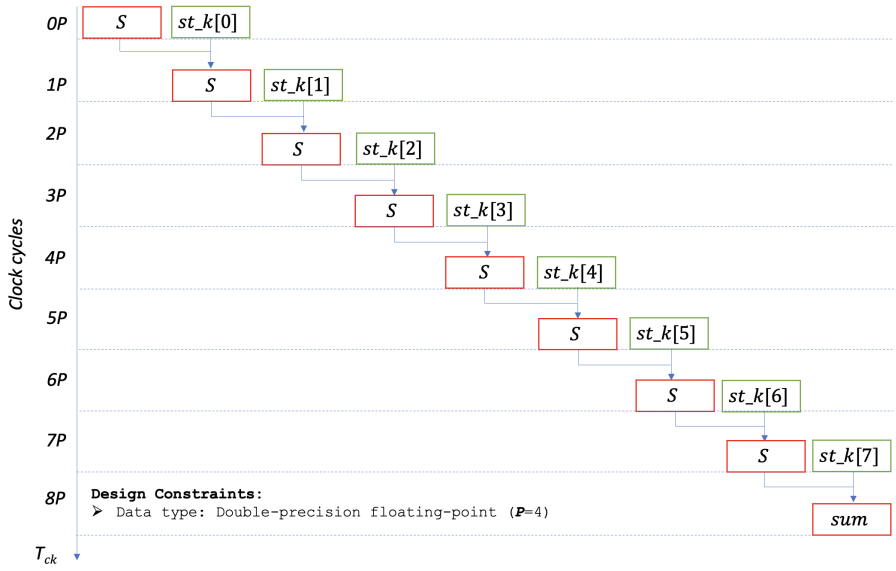
With this code, the `HLS` compiler *synthesizes* a circuit that executes our instructions. What is remarkable is that we can implement this circuit with little to no knowledge of hardware design. In contrast, attempting to represent this simple example using the `RTL` approach proposed in Peri (2020) cannot abstract from these technicalities. To demonstrate this point, Listing 5 in Appendix A.1 reports the `RTL` code required to implement the same circuit above.

Beyond the complexity of the syntax itself, coding in `RTL` is akin to describing a circuit. Accordingly, it demands proficiency in digital design, a subject rarely within the purview of economists. In contrast, the `HLS` approach delegates these details to the compiler. More significantly, the compiler autonomously recognizes and implements many of the performance optimizations discussed in the next sections without explicit directives from the user.

4.1.1 FPGA Optimizations

Our previous section implemented an accumulator without any optimization. This approach is often referred to as the "dusty deck" approach, evoking the image of pulling an old deck of punched cards off the shelf and compiling it, untouched, to the new hardware.

Figure 1: Data Flow of Sequential Accumulator: Dusty Deck Approach



Note: Data flow graph of a sequential accumulator circuit for adding double-precision floating-point elements of an array `st_k` of size $J = 8$. The vertical dimension illustrates in which clock cycles these operations are performed (*scheduling*). The circuit consists of one single double-precision floating-point adder, which performs an addition every $P > 1$ clock cycles.

While fully functional, the synthesized circuit is not efficient. Figure 1 illustrates the associated data flow graph to clarify this point. The vertical axis denotes the passage of time in clock cycles, while the horizontal axis represents the circuits' functional units, consisting of additions and memory accesses in our example. The graph shows that every P clock cycles, a new double-precision floating-point addition is performed. Accordingly, the synthesized circuit has a `for loop latency` of $J \times P$ clock cycles: this is the time (measured in clock cycles) required for a circuit with a single physical adder to sequentially execute $J = 8$ additions (`loop iterations`) when each one of them takes $P > 1$ clock cycles (`iteration latency`).

We can enhance the efficiency of this circuit by reducing its `for loop latency`. The next sections introduce the main tools that economists can leverage in order to *assist* the `HLS` compiler to design an effective `CL` design. We use the word assist deliberately. Unlike in the `RTL` approach, we do not design the circuit. Simply, we provide clues about acceleration opportunities that the compiler may not autonomously identify.

We introduce these optimization techniques in an order that aligns with industry standards for hardware design optimization. Our overarching strategy is to initially establish an efficient circuit (Subsections 4.2 and 4.3) and subsequently enhance its performance by harnessing parallel computing opportunities (Subsection 4.4).

Throughout this process, we address potential bottlenecks that might hinder efficient design. This enables us to compile a list of critical hardware issues that economists are likely to encounter when accelerating their applications. Most importantly, it provides the core hardware design principles for accelerating economic models.

4.2 Arbitrary-Precision Fixed-Point

We start our optimizations by targeting the `iteration latency`, P. This is the time required to execute the instructions in the body of the loop of Listing 1 at any given iteration. In our application, P consists of the time necessary to perform two operations: reading an array element and accumulating it (see Figure 1). Here, we illustrate how we can reduce the execution time of the accumulation from multiple clock cycles to a single one by transitioning from single or double-precision floating-point data type (hereafter referred to as floating-point) to fixed-precision fixed-point (hereafter referred to as fixed-point) to represent the array elements.

Floating-point operations provide the IEEE754 standard for the finite precision approximation of real numbers. However, they cost hardware and energy and take multiple clock cycles to complete. In contrast, integer and fixed-point operations execute within a single clock cycle and, when properly implemented, deliver high accuracy.[8] Among others, Yates (2009) provides guidelines for employing fixed-point arithmetic, and Subsection 7.2 operationalizes them in our

[8]See Appendix A.2 for a brief overview of finite-precision fixed-point approximation.

main application using HLS (Xilinx, Inc., 2021).

Here, we illustrate how a few lines of code can recast floating- into fixed-point arrays by defining a type `fixed_t` with the required precision in a header file, along with the necessary libraries.[9]

```
1 #define FIXED_ACC 1
2 typedef double real;
3 #if FIXED_ACC
4     #include <ap_int.h>
5     #include <ap_fixed.h>
6     typedef ap_fixed<72, 21> array_type;
7 #else
8     typedef real array_type;
9 #endif
```

Conversely, the implementation of fixed-point arrays in RTL can be tedious even for experienced hardware designers. For example, the RTL designer must explicitly manage the proper alignment and interpretation of values with different fixed-point precision representations.

Besides reducing the iteration latency, recasting our operation in fixed-point also addresses two well-known bottlenecks that hinder the efficient design of an accumulation with addition operations: *(i)* memory carry dependency; and *(ii)* the lack of associativity property of floating-point additions. Subsection 4.3 demonstrates how fixed-point addresses the memory carry dependency problem, facilitating the creation of efficient pipelines. Subsection 4.4 shows how properties of fixed-point arithmetic enable the implementation of a parallel reduce tree to solve our accumulation, further reducing the `for loop latency`.

4.3 Pipelining and the Quest for an Initiation Interval of 1

An ideal CL design keeps the circuit constantly busy (*goal 1*) and starts new computations at every clock cycle (*goal 2*). Here, we present an optimization tool that helps achieve both goals: hardware pipelining. In this context, we discuss how our solution of recasting our accumulator from floating- to fixed-point tackles a key obstacle to the efficient pipelining of our accumulation circuit: the memory carry dependency problem.

4.3.1 Pipelining

To understand hardware pipelining, we need to introduce the concept of *initiation interval*, denoted as **II**. The **II** measures the number of clock cycles elapsed between the launch of two

[9] This is one of the few optimizations that the HLS compiler cannot perform automatically since it requires users to specify the fixed precision parameters (72 and 21 in line 6). The macro `FIXED_ACC` allows flexible toggling between double-precision floating-point and fixed-point for debugging purposes.

consecutive iterations in a loop. An ideal circuit has an **II** = 1, and *initiates* computations at every clock cycle. The following code listing uses the `#pragma HLS pipeline II=1` directive to synthesize this design.

Listing 2: Pipelined Accumulator

```
1 void hw_loop(array_type st_k[J],array_type &reduced_sum){
2     array_type sum = 0.;
3     for(int i=0;i<J;i++) {
4         #pragma HLS pipeline II = 1
5         sum+=st_k[i];
6     }
7     reduced_sum = sum;
8     return;
9 }
```

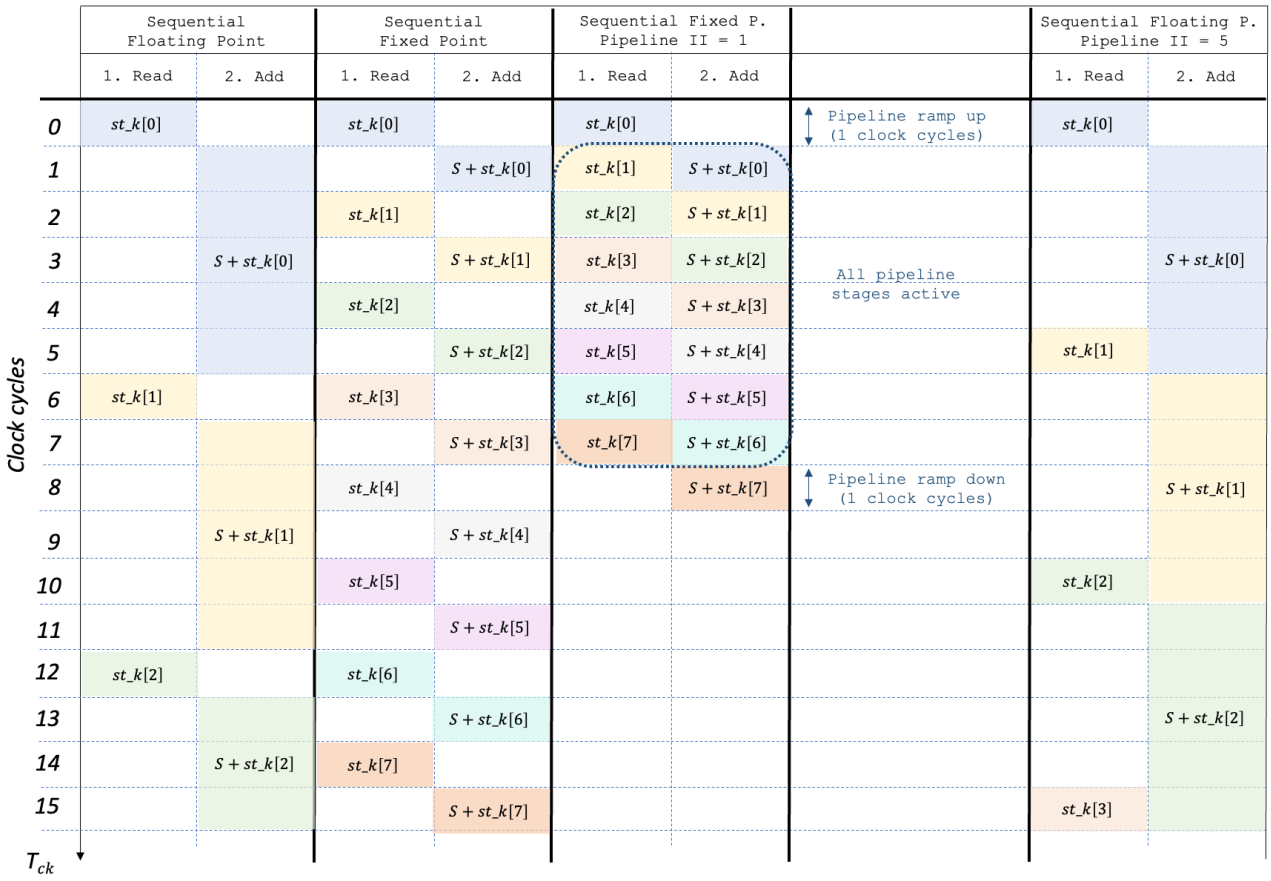
To understand the hardware implications of this optimization technique, Figure 2 reports the data flows associated with four distinct hardware designs of our sequential accumulator: floating-point (Subsection 4.1), fixed-point (Subsection 4.2), fixed-point with **II**=1, and floating-point with **II**=5. These graphs illustrate how the different `CL` designs engage their hardware components, often referred to as `pipeline stages`. They do so by reporting which data points each pipeline stage (horizontal axis) processes during each clock cycle (vertical axis).

In the case of our sequential accumulator, the pipeline stages associated with the hardware implementation of the body loop's instructions (line 5 of Listing 2) consists of two operations: a reading and an accumulate operation. Later in our main application, these pipeline stages will consist of the hardware steps required to solve either the **IHP** problem in equation (8) or the **Simulation** step.

The first two data flows on the left side of the figure illustrate how transitioning from floating- to fixed-point significantly reduces the execution time of the adder, from five to a single clock cycle, and, thus, the `iteration latency` from six to two clock cycles. Consequently, the total `for loop latency` drops from $P \cdot J = 6 \cdot 8 = 48$ clock cycles to $2 \cdot 8 = 16$ clock cycles.

Listing 2 improves over the sequential fixed-point design by instructing the `HLS` compiler to synthesize a fixed-point accumulator that initiates computations every clock cycle (**II**=1). This command generates a circuit that operates as follows. At clock cycle 0, the circuit starts reading `st_k[0]` from memory. At clock cycle 1, the circuit initiates the first fixed-point addition $S = S$+`st_k[0]`, while simultaneously loading `st_k[1]` for a new accumulation. From clock cycle 1 to 7, the circuit reaches its maximum degree of *instruction level parallelism*. At this point, all functional units of the circuit (`pipeline stages`) are working in parallel (attaining *goal 1*), yielding an addition at every subsequent clock cycle (attaining *goal 2*). The time required for all `pipeline stages` to become active is called pipeline `ramp up` and equals one clock cycle in our example. After clock cycle eight, the pipeline stages start to empty, as there are no more

Figure 2: Data Flow of Sequential Accumulator with Pipelining



Note: Data flow of four different hardware designs, from left to right: sequential floating-point (double precision), sequential fixed-point, sequential fixed-point pipeline with II=1, sequential floating-point (double precision) pipeline with **II** = 5.

array elements to read. By clock cycle nine, all pipeline stages are empty. The time required to flush out the pipeline stages is called pipeline `ramp down`, also equal to one clock cycle in our application.

As the state space increases, the pipeline ramp-up and ramp-down times become negligible with respect to the total for-loop latency. For example, when transitioning from an accumulation of an array of size 8 to an array of size 10,000 in equation (12), the portion of `for loop latency` spent on filling and flushing the pipeline shrinks from 25%, $\#\text{Stages}_{\text{ramp up+ramp down}}/J$=2/8, to a negligible 0.02% (2/10000).

Memory carry dependency problem. Add-reduce accumulations inherently suffer from memory-carry dependencies because adding a new element of the array to the accumulation requires the previous element of the array to have already been incorporated (Figure 1). In the context of floating-point accumulations, this data dependency may impose a constraint on the lag (in cycles) at which new computations can be initiated (**II**).

For example, attempting to design a circuit with an **II**=1 would result in a synthesis failure, flagging a memory carry dependency violation. The reason becomes evident when inspecting the behavior of this circuit on the right-hand side of Figure 2. The circuit must wait until clock cycle 5 for the value $S + st\_k[0]$ to become available before proceeding to the new addition $S + st\_k[1]$. Consequently, the best initiation interval for this design is every five clock cycles (**II**=5). Transitioning to fixed-point effectively addresses the memory dependency bottleneck by reducing the waiting time for the next element of the accumulation from five to just a single clock cycle, reaching **II**=1.

4.4 Loop Unrolling and Memory Management

So far, we have used `HLS` to design a circuit that efficiently initiates the necessary computations for a sequential accumulator at every clock cycle. Next, we further enhance its performance by introducing a powerful `HLS` tool that allows us to harness parallel computing opportunities: loop unrolling. Alongside this, we address limitations associated with memory management and provide effective solutions. Furthermore, we discuss a common performance hurdle faced by the hardware implementation of an accumulator, such as equation (12): the lack of associativity in floating-point additions. We conclude by explaining how our solution of recasting our accumulator from floating- to fixed-point effectively overcomes this challenge.

4.4.1 Loop Unrolling

Our prior circuitry executed additions sequentially, overlooking the opportunity to exploit the data independence of array elements for concurrent parallel addition operations. In our accumulation example, and later in the acceleration of equation (8) and Simulation step of the Krusell

and Smith (1998) algorithm, we harness this parallelism by deploying the `#pragma HLS unroll factor = X`. This directive creates multiple physical replicas (`X`) of the circuit described within the loop's body (line 6 of Listing 3) and *autonomously* organizes them to operate in parallel in a reduce-tree pipeline, which we will describe momentarily. Intuitively, the more physical replicas operating in parallel (`unroll factor`), the fewer iterations are necessary to perform the final accumulation, lowering the `for loop latency`. The following listing synthesizes a pipeline that processes in parallel all elements of the array, $J =$ `unroll factor` $= 8$.

Listing 3: Accumulator with Loop Unrolling of Factor 8

```
1  void hw_loop(array_type st_k[J],array_type &reduced_sum){
2      #pragma HLS array_partition variable = st_k factor = 4 type = cyclic
3      array_type sum = 0.;
4      for(int i=0;i<J;i++) {
5          #pragma HLS unroll factor=8
6          #pragma HLS pipeline II = 1
7          sum+=st_k[i];
8      }
9      reduced_sum = sum;
10     return;
11 }
```

Associative reduce tree and fixed-size loop bounds. Figure 3 shows that an unroll factor of 8 prompts the compiler to *autonomously* generate an associative reduce tree structure with a `pipeline depth` of three layers, resulting in an accumulation every three clock cycles.[10] This design is facilitated by the fact that the loop bounds are fixed (0 and J), and increments of the loop iterator i are constant (and equal to 1). These clues allow the compiler to aptly determine the required number of layers: $O(\ln_2$ `unroll factor`$)$.

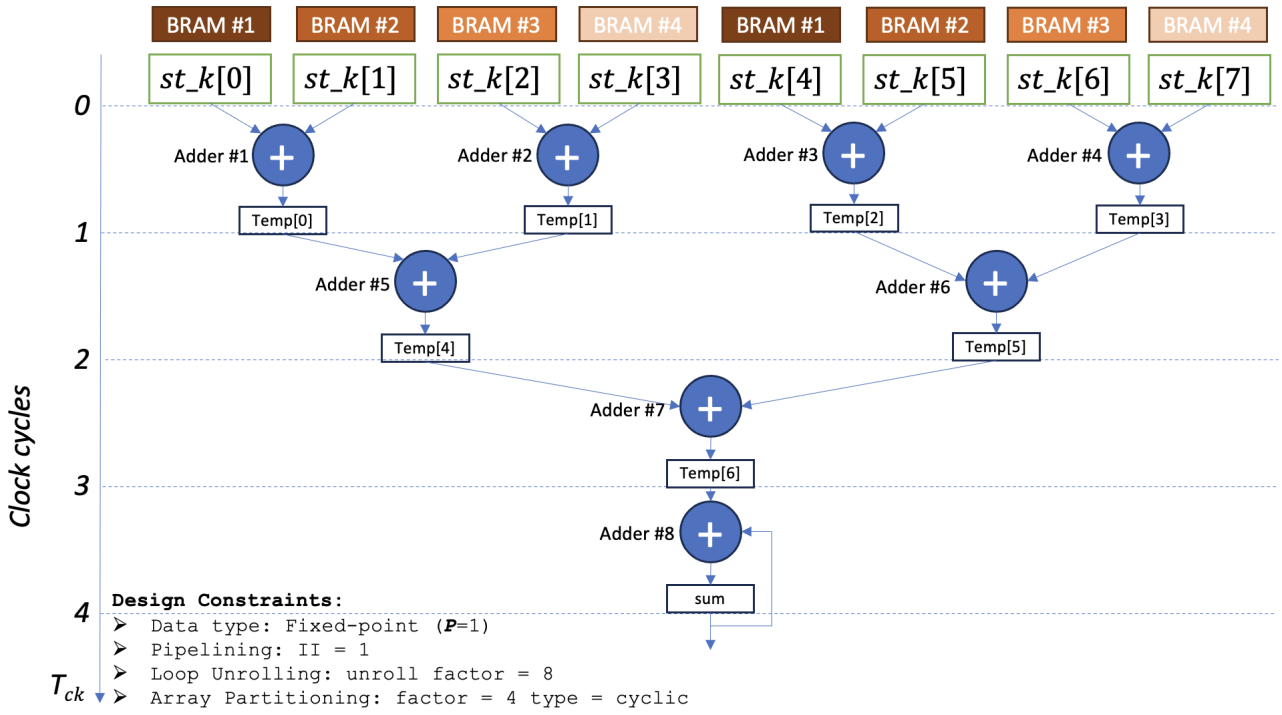
When the array of size J is larger than the `unroll factor`, this circuit produces an accumulation every $J/$`unroll factor` $+ \ln_2($`unroll_factor`$) + 1$ clock cycle. The first term illustrates how loop unrolling decreases the `for loop latency` from J to $J/$`unroll factor` by placing enough hardware, as determined by `unroll factor`. The second term captures the pipeline ramp-up time.[11] As discussed in Subsection 4.3, the pipelining performance of this circuit increases with the size of the state space. As J increases, the pipeline ramp-up time becomes negligible with respect to the total accumulation time, which converges to $J/$`unroll factor`.

While the acceleration achieved with loop unrolling resembles the single instruction, multiple

[10] In the first clock cycle, the compiler simultaneously reads all elements of the array. In the second clock cycle, it performs four additions in parallel. In the third clock cycle, it performs two additions and reduces the accumulation to a final digest. The fourth clock cycle appears when $J >$ `unroll_factor`, in which case the accumulation of 8 elements is added to the overall rolling sum. See Appendix A.3 for more details.

[11] The third term appears when $J >$ `unroll factor` and captures the time required to incorporate the result of the reduce tree into the rolling accumulation (clock cycle 4 in Figure 3). Otherwise, the term disappears.

Figure 3: Data Flow of Accumulator with Loop Unroll



Note: Data flow graph of a circuit that unrolls by a factor of 8 the addition of the fixed-point elements of an array `st_k` of size $J = 8$. The vertical dimension illustrates in which clock cycles these operations are performed (*scheduling*). The circuit consists of eight fixed-point adders, which perform an addition every clock cycle. `BRAM #1-#4` denote four local memories used to partition the `st_k` array.

data acceleration exploited by GPUs, it differs in important dimensions. First, the `HLS` compiler designs new hardware with the requested level of parallelism, whereas GPUs exploit existing one. Consequently, while GPUs may experience performance degradation when the required level of parallelism does not effectively engage all their cores (Aldrich et al., 2011), FPGAs are affected to a much lesser extent. Second, FPGAs allow the implementation of hardware designs of intricate algorithms, whereby various facets of the application can exploit different acceleration strategies. The Krusell and Smith (1998) algorithm exemplifies this complexity, transitioning from a policy function iteration algorithm that benefits from reduced evaluation latency through instruction-level parallelism and pipelining (Subsection 4.3) to a multi-parameter exploration or multiple-actor simulation step, which benefits from data-level parallelism. While GPUs thrive in the latter, they cannot reduce evaluation latency.

To conclude, note the `#pragma HLS array_partition` directive at line 2 of Listing 3. Next, we discuss how this command addresses a potential issue related to memory management that *may* prevent `HLS` from synthesizing the required circuit.

4.4.2 Memory Management

Now, we discuss two main limitations regarding memory management: large memory access latency and array partitioning.

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. In our accumulator design, we initialize our array in double-precision floating-point in the host processor, `st_in`, and make it available to the FPGAs by storing it in global memory. Access to this memory is slow (hundreds of clock cycles).

To solve this problem, we can copy 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: block RAMs (BRAMs), ultraRAMs (URAM), LUT RAMs, registers. The following listing illustrates how straightforward it is to copy the array from global (`st_in`) to local memories (`st_k`) through the implementation of a simple array assignment.

Listing 4: Store Data from Global to Local Memories

```
1 void hw_top_init(real *k_in, array_type st_k[J]){
2     init_1:
3     for (int i = 0; i < J; ++i)
4         st_k[i] = (array_type) k_in[i];
5     return;
6 }
```

Note the type cast operation that converts the floating-point values of the array initialized by the host, `st_in`, into the fixed-point array declared in the local memories, `st_k`. Without loss of generality, we assume that `st_k` has been stored in BRAMs in all our designs.

Array partitioning. When transferring data from global to local memory, compilers optimize on-chip resource use by storing contiguous elements of `st_k` in the same BRAM before moving on to the next. For instance, without further instructions, Listing 4 stores all the elements of `st_in` in a single BRAM, which, without loss of generality, we refer to as `BRAM #1`.

Each BRAM provides two read-write ports, allowing for a maximum of two concurrent read-write operations. Accordingly, while resource-efficient, this storage scheme can lead to a memory bottleneck when the total number of required reads from multiple pipelines (determined by the `unroll factor` multiplied by the simultaneous reads per pipeline) surpasses the available reading ports per shared BRAM.

The memory-access bottleneck becomes evident in our example when we instruct the compiler to synthesize four physical adders in parallel (Listing 3). In this scenario, during the first clock cycle, we can only read a maximum of two out of the eight `st_k` values stored in the single BRAM. Consequently, the synthesis process fails to produce the required `CL` design. In-

stead, it results in a circuit with an accumulation process that resembles the sequential addition described in Subsection 4.3 (see also Figure A.3 in Appendix A.3.1).

The memory bottleneck can be resolved by strategically distributing the elements of `st_k` across multiple BRAMs to accommodate the eight memory accesses required by the four parallel pipelines. The achieve this, we use the `#pragma HLS array_partition` directive at line 2 of Listing 3. This directive employs a `cyclic` array partitioning with a `factor` of 4 to interleave the elements of `st_k` across `BRAM #1-#4`, as depicted in Figure 3.[12]

Automatic optimizations. We explain here the emphasis on the word *may* at the end of Subsection 4.4. In our simple accumulator example, for a given `unroll factor`, the `HLS` compiler recognized the need to partition the array `st_k` without explicit clues from the array partitioning `#pragma`. As compiler technology evolves, we anticipate such automation to expand to even more intricate circuits. Our performance analysis of the Krusell and Smith (1998) algorithm in Section 7 demonstrates the advanced nature of these automations already in place.

4.4.3 Non-Associativity of Floating-Point Addition

Here, we discuss how using floating-point additions prevents the compiler from autonomously generating the associative reduce tree displayed in Figure 3. Computers carry out computation on numbers with finite representations. When we abandon the field of real numbers $(\mathbb{R}, +, \times, \geq)$ for their IEEE754 floating-point, finite-precision approximation, we lose the associative property of the addition. Goldberg (1991) provides an 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 (since $y + z$ rounds to $-1e30$ before performing the addition with x). Accordingly, compilers have historically been prevented from rearranging floating-point additions (like in Figure 3) in order to avoid producing a different result than the original code. This rule downplays the fact that the source code may have been written in a loop simply for convenience, without a specific order of additions in mind. As a result, the compiler still insists on performing the floating-point sum sequentially in $J \cdot P$ cycles, as shown in Figure 1, even if we allowed it to use more hardware.

In our accumulator example (later deployed in the Krusell and Smith, 1998, application), we tackle this issue by performing the accumulation in fixed-point, where the associative property still holds, and the compiler knows it is safe to re-associate the additions since the computed result will be the same. For alternative strategies to handle this issue while maintaining floating-point, we refer to Hrica (2012) and Kadric et al. (2016). In contrast to these strategies, our

[12] Cyclical array partitioning creates `factor = 4` smaller arrays from `st_k` by copying elements cyclically, and storing them in independent BRAMs. For instance, `st_k[0]` goes into `BRAM #1`, `st_k[1]` into `BRAM #2`, `st_k[2]` into `BRAM #3`, `st_k[3]` into `BRAM #4`, `st_k[4]` into `BRAM #1` and so on until completion. For a discussion of the different array partitioning types (cyclic, block, and complete), we refer to https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-array_partition.

fixed-point approach (Subsection 4.2) also addresses the long latency of floating-point additions with minimal compromises in accuracy, as demonstrated by our application.

4.5 Portability, Code Reusability, and Programming Languages

`C`-code designed for FPGA acceleration using `HLS` exhibits remarkable portability. As our accumulation example demonstrates, the code compiles and runs on CPUs (including vector and SIMD computations) without modifications. Non-FPGA compilers ignore the `HLS #pragma`s, eliminating the need to maintain a separate codebase for FPGA acceleration.

Furthermore, high-level computational components, such as accumulation (discussed here), interpolation, and simulation (discussed later), are common in economic models. Much like economists use function libraries to address recurring algorithms, the same `HLS`-enhanced codes and associated hardware components can be repurposed to accelerate recurring aspects of different models.

While `HLS`-enhanced `C`-code is highly portable, FPGA platform-specific optimizations and design may still require special considerations. First, fine-tuning an application for a specific FPGA involves adjusting parameters like loop unrolling or array partitioning, considering both application-specific and platform-specific factors, such as the execution frequency of particular portions of code and their resource utilization. These refinements only require changes to specific arguments in the `#pragma`s, leaving the base `C`-code unchanged. Recent advancements in automatic compilation and optimization hold great promise for automating these optimizations with solvers and autotuners (as discussed below), removing the need for manual changes by the developer.[13] Second, interfacing with platform-specific components, like memory interfaces and I/O ports, super logic regions (SLRs) may be necessary for advanced designs (see Subsection 5.1.2). To mitigate these platform-specific concerns, we showcase FPGA acceleration on `AWS`, a cloud-based service that offers widespread access to *identical* high-performance FPGA devices. In addition, our tutorial provides *all* the necessary boilerplate code to harness these platform-specific components.

To conclude, recent years have seen great advances in the incorporation of hardware design tools among popular programming languages, like `Python`, `Julia`, and `Matlab` (e.g., Quenon and da Silva, 2021, Biggs et al., 2022). Our accumulator best exemplifies these efforts. The `Matlab` suite has incorporated an `HDL` coder that generates the `RTL` code of a variety of functionalities, including the accumulator we just described in this section.[14] Overall, high-level synthesis

[13] We can determine these tuning parameters as the ones that maximize performance while staying within the constraints of available FPGA hardware resources. Solvers or autotuners can then automate the selection of these values, streamlining the optimization process (Whaley and Dongarra, 1998; Kapre and DeHon, 2009; Lo and Chow, 2016; Zhao et al., 2020).

[14] See https://www.mathworks.com/help/fixedpoint/ref/sum.html.

compilers are bound to become more popular among high-level software languages, further easing economists' access to custom accelerators.

5 Our Application

This section illustrates how we deploy the FPGAs' optimization tools introduced in our previous section to accelerate the Krusell and Smith (1998) algorithm in Section 3. We present the FPGA acceleration approach in Subsection 5.1 and the CPU counterpart in Section 5.2. Appendix B reports the hardware specifications of the different `AWS` instances.

5.1 Hardware Architecture of FPGA Acceleration

The FPGA acceleration approach solves the Krusell and Smith (1998) algorithm as implemented by Maliar et al. (2010) 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 FPGA(s), and transfers the data to the FPGA(s). The FPGA(s) then execute(s) the *kernel*: the nested, fixed-point algorithm detailed in Section 3. Subsequently, the host reads back and saves the results. Unless differently specified, computations employ the IEEE754 double-precision floating-point format.

5.1.1 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 SLR, connected by a silicon interposer (see Figure 5). The AWS FPGA is further divided into two partitions: shell and custom logic (`CL`). The shell is the FPGA platform implementing external peripherals, such as peripheral component interconnect express (PCIe) and dynamic random access memory (DRAM). 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.

5.1.2 Custom Logic Hardware Design

Now, we operationalize the FPGA optimization tools discussed in Section 4 to accelerate the Krusell and Smith (1998) algorithm. In addition, we show how our `CL` design exploits platform-specific characteristics of the F1 instances' FPGAs: the presence of three SLRs. As mentioned

above, the Xilinx VU9P FPGAs are organized in three dies, whose custom logic can be jointly or independently used for acceleration purposes. For our application, we tailor our `CL` design to compute three economies in parallel, one per SLR, in the same FPGA.[15]

Cheela et al. (2023) describe in detail how the `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 this 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.

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. We accomplish this step by creating.AWSXCLBIN files.

5.1.3 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 it performs the computations until convergence of the ALM coefficients in equation (13).

Following Amdahl's law, our `CL` design allocates hardware resources to balance the time spent on the two time-consuming steps of the algorithm: the **IHP** and Simulation design. Our final kernel design starts computations at every clock cycle at 250 MHz and provides flexibility to study multiple grid sizes.

To reach an **II** of 1, we address the challenges discussed in Section 4 as follows. First, we tackle the two memory management bottlenecks: *(i)* large memory access latency; and *(ii)* limited memory ports to access data in parallel. We fix the first issue by copying data in local memories that can be accessed in a single clock cycle. Our FPGA has enough local memories to store all of the input data for our application (Table A.3). Hence, we need to transfer the input data only once per kernel computation. We address the second issue by deploying the array partitioning `#pragma` to store multiple copies of the same data in as many independent local memories as required by the number of pipelines performed in parallel.

Next, we turn to application-specific interventions essential for achieving an **II** of 1, specif-

[15] We could alternatively design the `CL` 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. The single SLR solution is still 28.38 times faster than the sequential execution in the CPU (Subsection 6.5).

ically the acceleration of the linear interpolation within the **IHP** and **Simulation** steps. 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.[16] 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 individual household problem (IHP) design. The **IHP** design implements the previous steps to reach an **II** of 1 in equation (8) by pipelining the operations involved in the computation of the expectation as discussed in Subsection 4.3. We further accelerate the design by using the partial loop unrolling technique discussed before to place two pipelines to work in parallel on the (sequential) solution of equation (8) 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 (8) for every state and updates the guess according to equation (9), as discussed in Subsection 3.B.*(ii)*. These steps are iterated until convergence, as per equation (10).

The simulation design. The simulation design in Subsection 3.D.*(ii).(b)* is divided into two steps, each accelerated with a custom pipeline. The first step is represented by the accumulation operation required to compute the cross-sectional average of the individual household's capital holdings, m_t, in equation (12). To achieve an $\mathbf{II} = 1$ at this step, we deploy the custom pipelined fixed-point accumulation operator discussed in Section 4. Indeed, the best acceleration performance for our benchmark model is reached by letting the compiler perform automatic optimizations. 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.[17]

The ALM design. The ALM step involves several resource-expensive operations but has very small latency (1.25ms in our baseline economy).[18] 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.

[16] As discussed in Subsection 4.4.1, the presence of *fixed* interpolation grids' bound allows the compiler to autonomously determine the depth of the reduce tree required to find the interpolation interval.

[17] 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 the 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.

[18] The ALM step requires 16 BRAM, 133 DSP, 11,494 LUTs, and 15,265 Registers.

5.1.4 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 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 4). 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 degree of partial unrolling in the IHP. The resulting three-kernel design (Figure 5) trades off the single-kernel performance for the parallel execution speedup.

5.2 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 physical cores: one core (m5n.large), eight cores (m5n.4xlarge), and forty-eight cores (m5n.24xlarge). Appendix B justifies the choice of these instances.

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 ten 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**
Solution Time	73657.8	38392.0	28452.5
Speedup	-	1.92	2.59

Note: Solution 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`).[19] `Open MPI` provides easily implementable, off-the-shelf routines for parallelizing data-independent tasks across multiple cores. In contrast with alternative multi-core

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

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.

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.

6 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, 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" four times 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 optimization techniques discussed in Section 4, we program the logic of our FPGA to accelerate the Krusell and Smith (1998) algorithm. In comparison, the CPU and GPU logic cannot be programmed. The CPU is premanufactured to optimize the performance of sequential instructions needed in daily computer activities, which include not only 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, we introduce a multi-economy parallelism to get insights into the potential of FPGAs to structurally estimate our model by solving it for many different parameter values. 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, 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 that the speedup scales linearly with the number of cores. Incidentally, this exercise illustrates how easy it is to implement increasingly aggressive acceleration by deploying multiple FPGAs in parallel, a dimension not explored in Peri (2020).

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 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. To make the comparison as meaningful as possible, we compute the speedup using the solution time—which is the time required to solve the algorithm on the FPGA/CPU. This approach abstracts from the time absorbed by initializations and host-FPGA communications. Appendix C.2 details the reasons behind this choice by also reporting the execution time, that is, accounting for non-kernel operations. Table A.4 in the Appendix complements this information by reporting costs and energy consumption by instance.

6.1 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 speedup of 69 times. The computation time drops

Table 3: Efficiency Gains and Implementation Costs of FPGA Acceleration

Panel A: Efficiency Gains of FPGA Acceleration

CPU-cores	**Speedup**			**Relative Costs (%)**			**Energy (%)**		
	FPGAs			*FPGAs*			*FPGAs*		
	1	*2*	*8*	*1*	*2*	*8*	*1*	*2*	*8*
1	68.54	137.09	548.56	20.23	20.23	20.22	6.02	6.02	6.02
8	8.80	17.61	70.46	19.69	19.69	19.68	5.86	5.86	5.85
48	1.48	2.96	11.83	19.55	19.55	19.54	5.82	5.82	5.81

Panel B: Implementation Costs of FPGA Acceleration

	Kernel		*Non-kernel*	
	Number	Percent (%)	Number	Percent (%)
Extra Lines of Code	75	5.37	128	51

Note: Panel A reports speedups provided by the FPGA and cost and energy usage of the FPGA relative to the CPU. The results are obtained by solving 1,200 baseline economies using AWS instances connected to 1, 2, and 8 FPGAs and using open-MPI parallelization on AWS instances with 1, 8, and 48 cores (rows). Speedup is obtained by dividing the total solution 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.4 in Appendix C reports the details. Panel B estimates implementation costs for both kernel and non-kernel segments of our codebase by reporting the extra lines of code required by the `HLS`-enhanced `C` code when compared to standard `C` code designed to be executed on the CPU using `Open MPI`.

from 8 hours in the CPU to 7 minutes in the FPGA (Table A.4).

Next, we document the FPGAs' 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 1 hour in the CPUs to 7 minutes in the FPGA (Table A.4). 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 1.48 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. Panel A of Table 3 shows that eight FPGAs in parallel reduce the solution time by 549x, 70x, and 12x when compared with the 1-core, 8-core, and 48-core acceleration, respectively.

Panel A of Table 3 shows 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.

6.2 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 solution time for the AWS instance on-demand prices (Table A.2). Panel A of 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 \$784 to \$159). Albeit the rental cost of FPGA instances per hour is higher, the acceleration we document more than justifies the expense.[20]

Also, note that 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.

6.3 Energy Savings of FPGA Acceleration

We determine the energy consumption (joules) by multiplying the total solution time for the power consumption (watts) of FPGA and CPU chips. We use the AFI management tool to measure the FPGA average 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 6 % of the energy consumed by CPUs.[21] This statistic is relevant for organizations with in-house computational clusters (such as 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

[20] Another cost comparison is with an in-house cluster. Even without entering into their purchase cost, clusters are expensive to maintain. In our application, a single FPGA chip can perform the same task as a cluster with 69 cores. This is a medium-to-high scale cluster whose maintenance requires an HPC specialist, with a salary averaging around \$89,000 per year in the US circa 2024. See Zip Recruiter.

[21] Since CPU power consumption is proportional to the number of CPU cores, $\text{Power(cores)} = P \cdot \text{cores}$, and the solution time is inversely proportional to the number of CPU cores, $\text{Solution Time (cores)} = T/\text{cores}$, the energy, up to first-order, is constant across CPU cores, $\text{Energy(cores)} = P \cdot \text{cores} \cdot T/\text{cores} = PT$.

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 31.07 metric tons of CO_2, or six cars per year.

6.4 Implementation Costs of FPGA Acceleration

Panel B of Table 3 reports the number of lines required to implement the kernel and non-kernel portion of our application for FPGAs and CPUs. The kernel portion of the `HLS`-enhanced `C`-code requires 5.37% of additional lines with respect to `C`-code written to be executed on the CPU. With 75 extra lines of code (#`pragmas`), the user can achieve a 69x speedup with respect to the single-core CPU. The non-kernel portion of the code requires 128 more lines (51%) to implement the `OpenCL` communications between host and device. However, this code barely varies across applications and is available on our `Github` repository.

6.5 Robustness

Table 4 reports 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 (Panel A), we solve the baseline economy using the single-kernel design in one FPGA and compare its performance with that of the single-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 in the available SLRs should bestow a speedup of 28.38x $\cdot$ 3 SLR $\approx$ 85x, yet the three-kernel design "only" reaches a speedup of 69 x (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. The lower speedup is associated with higher costs than the costs of the three-kernel design, at still half of the cost. Energy savings remain significant.

In our second exercise (Panel B), we study how the speed gains depend on the size of the grids. The panel illustrates speedup, cost, and energy savings for increasingly finer grids on individual capital holdings $N_k = \{100, 200, 300\}$ and aggregate capital $N_M = \{4, 8\}$. To ensure a clear comparison, we keep contrasting the performance of the single-kernel FPGA design with the sequential single-core CPU execution. The speedup improves as the grid size of individual capital holdings gets larger, with stronger benefits when transitioning from 100 to 200 grid points, which then plateau at 300 grid points. The performance when moving from 4 to 8 grid

Table 4: Single-Kernel FPGA vs. Single CPU Core

Panel A: Benchmark Model, $\{N_k, N_M\} = \{100, 4\}$

FPGA-Time(sec)	*CPU*-Time(sec)	*Speedup*(x)	*Relative Costs*(%)	*Energy*(%)
0.84	23.71	28.38	48.86	7.49

Panel B: Speedup across Grid Sizes

Aggregate Capital, N_M	*4*			*8*		
Individual Capital, N_k	*100*	*200*	*300*	*100*	*200*	*300*
Speedup (x)	28.38	34.41	34.31	27.81	31.05	32.06
Relative Costs (%)	48.86	40.30	40.41	49.85	44.65	43.26
Energy (%)	7.49	6.18	6.19	7.64	6.84	6.63

Note: Figures are obtained by comparing the solution of 1,200 economies using AWS instances connected to 1 FPGA and sequential CPU execution on a single core. Panel A focuses on the benchmark economy, $\{N_k, N_M\} = \{100, 4\}$. Columns 1-2 detail the average solution time (in seconds) to compute the benchmark economy in a single-kernel, single-device FPGA (f1.2xlarge), and a single-core instance (m5n.large), respectively. Columns 3-5 display the efficiency gains of FPGA acceleration in terms of speedup, costs (in percent), and energy savings (in percent), computed as described in Table 3. The FPGA average power consumption on a single-kernel design is 17 watts. Panel B studies how speedup, relative costs, and energy consumption vary with the size (columns) of the individual household capital holdings grid (N_k) and aggregate capital grid (N_M).

points is slightly lower but comparable. Crucially, these results are obtained by keeping the *same* hardware design—pipelining, array partitioning, and loop unrolling—as the benchmark model with $N_k = 100$ and $N_M = 4$. This analysis suggests the potential for tailored optimizations for larger grid sizes, currently disabled in our design.

7 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 recasting data from floating- to fixed-point. Since the bottlenecks faced in our design are common to most dynamic programming problems, 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*).

7.1 Baseline

Column 1 in Table 5 reports the speedup of solving the model using the single-kernel FPGA baseline design and the single-core CPU sequential solution. The baseline design differs from the single-kernel design discussed above 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 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 113 seconds, approximately five times slower than the CPU solution, which is reached in 24 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, 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 deploy the optimization tools discussed in Section 4 to efficiently organize these tasks in assembly lines, and how we improve performance by placing multiple assembly lines in parallel.

7.2 Pipelining

Next, we discuss the main changes we made in order to pipeline the **IHP** efficiently 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. 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

Table 5: Speedup Gains: Acceleration Channels Accounting

	Baseline	*Pipelining*	*Data Parallelism*	
			Within Economy	*Across Economies*
Single-core Execution / FPGA Solution	0.21	6.94	28.38	68.54
CL Resources Utilization (%)				
BRAM	6.01	7.14	21.31	44.29
DSP	7.75	9.68	31.13	55.32
Registers	3.99	5.12	12.00	25.71
LUT	5.96	9.20	25.21	57.03
URAM	5.50	5.50	5.38	16.50

Note: Column 1 reports the speedup for a kernel design where all acceleration channels are switched off (baseline). Columns 2-4 report the speedup associated with implementing efficient pipelines (Column 2), introducing data parallelism in the kernel design (Column 3), and instantiating three kernels in the same FPGA (Column 4). The speedup (row 1) is computed by dividing the total solution time in the one-core CPU by the solution time in the FPGA. The acceleration in Columns 1-3 is performed using a single-kernel, single-device FPGA (f1.2xlarge), where Column 4 coincides with the single-kernel design. The acceleration in Column 4 is performed by deploying the three-kernel design in parallel across the three SLRs in a single FPGA (f1.2xlarge). Averages are computed over 1,200 economies, except for the Baseline and Pipeline designs, which for cost considerations are computed over 120 economies. Resources are measured (using Xilinx Vivado) as a percentage of the Xilinx VU9P FPGA's resources utilized by AWS images associated with the different designs (columns). *Available Resources: BRAM (1,680), DSP (5,640), Registers (1,790,400), LUTs (895 thousand), URAM (800).*

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.[22]

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 floating-point computation of the cross-sectional average of individual capital holdings, m_t, and settles at an **II** of 5. Section 4 demonstrates how recasting the pipelined accumulator in fixed-point addresses this issue, yielding an **II** of 1 in equation (12), and consequently in the **Simulation** step. Table 5 records a speedup of 6.94 with respect to a single-core CPU. Importantly, currently available CPUs do not provide access

[22]In the CPU, the `C++` compiler can autonomously decide to perform these operations in parallel, but the coder does not control this step.

to user-defined fixed-point 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 not to hold a small value but rather to 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 $\underline{404851}$.76387144416 and 15.$\underline{371273672208304}$, 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 A.6 in Appendix C.4 shows that the results are very accurate. The estimated ALM coefficients are identical up to the ninth decimal place (R^2 are all 0.999 and, thus, 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 the percent of these objects under floating- and fixed-point (less than 3.0e-08). Moments of the distribution of individual capital holdings are identical up to the seventh decimal place. The Euler equation errors reveal that transitioning from computations in floating-point on the CPU to fixed-point on the FPGA has no discernible impact on the accuracy of our policy function's discretization (with relative differences lower than 1e-6). The maximum Euler equation errors indicate that the accuracy of our solution increases with the individual capital holdings grid size, where the benefits of FPGA acceleration are even more pronounced.

7.3 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 5). Hence, our next step is to identify parts in the algorithm to parallelize using the loop unrolling technique described in Subsection 4.4.1. The computations involved in the interpolation step (Subsection 3.D.*(ii).(b)*) of the Simulation and the policy function iteration (equation (8)) provide suitable candidates.

Given Amdahl's law, we perform a trade-off analysis between resource utilization and the solution speedup. Eight copies of the pipeline work well for our design, as they bring the solution time of the **Simulation** step (371ms) closer to that of the **IHP** step (381ms) when we unroll the

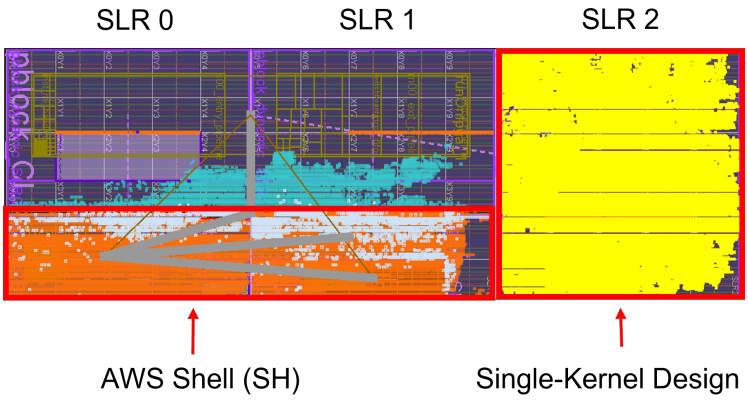
latter over the states by a factor of two. Lacking hardware programmability, the `CPU` execution is more imbalanced, with 72% of the solution time spent on the **Simulation** step and only 28% on the **IHP** step. Accordingly, our `CL` design yields a 46x speedup in the **Simulation** step and 17x speedup in the **IHP** step compared to a single-core CPU, totaling a speedup of 28.38x as shown in Table 5. We call the design that follows from these changes the single-kernel design.[23]

Also, it is clear from our acceleration strategy that the speedups at different stages of the algorithm are *endogenous* when programming FPGAs, where the hardware is not taken as given. Thus, FPGA acceleration depends less on the type of algorithms used compared to CPUs and GPUs. For example, replacing the stochastic simulation algorithm with a faster and more robust algorithm (e.g., Young, 2010) would allow us to free resources from this time-consuming step and reallocate them to accelerate the individual households' problem.

7.4 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 4). Also, the AWS shell covers a good share of resources in the adjacent SLRs (orange area in Figure 4).

Figure 4: Single-kernel Design: Resource Utilization

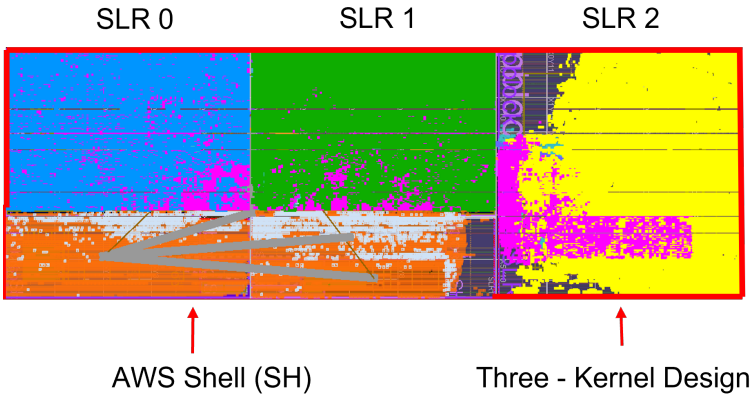


Note: Resources utilized by: *(i)* the single-kernel `CL` design (yellow area); *(ii)* by the AWS shell (orange area); and *(iii)* available `CL` resources (other colors). The image is captured using Xilinx Vivado.

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 amount of unrolling. In particular, we reduce the unrolling in the **IHP** design from 2 pipelines

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

Figure 5: 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.

working in parallel to a single pipeline. Figure 5 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 68.54 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.

8 Toward Electrical Engineering Economics

This paper proposes the design of FPGAs for the solution of economic models. This approach requires minimal knowledge of hardware design principles. With small modifications of standard `C/C++` code, a single FPGA can deliver the same performance as 69 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 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 repre-

sent 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 hard 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 dollars). Yet, 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 their 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. Schmedders, C. Hommes, and B. LeBaron (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 23*, 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.

Biggs, B., I. McInerney, E. C. Kerrigan, and G. A. Constantinides (2022). High-level synthesis using the Julia language. Technical report, Imperial College London.

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.

Cheela, B., A. DeHon, J. Fernández-Villaverde, and A. Peri (2023). *A Beginner's Guide to Programming FPGAs for Economics: An Introduction to Electrical Engineering Economics*. University of Pennsylvania.

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.

Hrica, J. (2012). Floating-point design with Vivado HLS. Xilinx Application Note.

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.

Kadric, E., P. Gurniak, and A. DeHon (2016). Accurate parallel floating-point accumulation. *IEEE Transactions on Computers 65*(11), 3224–3238.

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.

Kapre, N. and A. DeHon (2009). Performance comparison of single-precision SPICE model-evaluation on FPGA, GPU, cell, and multi-core processors. In *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 65–72.

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.

Lo, C. and P. Chow (2016). Model-based optimization of high level synthesis directives. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–10.

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*, 1–18.

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

Quenon, A. and V. R. G. da Silva (2021). Towards higher-level synthesis and co-design with python. In *Proceedings of the Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'21)*. ACM New York, NY, USA.

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.

Whaley, R. C. and J. J. Dongarra (1998). Automatically tuned linear algebra software. In *Proceedings ACM International Conference on Supercomputing*, Washington, DC, USA, pp. 1–27.

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.

Xilinx, Inc. (2021). *Overview of Arbitrary Precision Fixed-Point Data Types*. Xilinx. Accessed on 2023/11/02.

Yates, R. (2009). Fixed-point arithmetic: An introduction. *Digital Signal Labs 81*(83), 198.

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. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 86–96.

Zhao, J., L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He (2020). Performance modeling and directives optimization for high-level synthesis on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39*(7), 1428–1441.

Online Appendix for Programming FPGAs for Economics

This online appendix adds further details to the main paper. First, we include a table with all the abbreviations that we use for easy reference.

Table A.1: **List of Abbreviations**

ALM	Aggregate Law of Motion	Algorithm stage
AFI	Amazon FPGA Image	CL design implemented on AWS FPGAs
AWS	Amazon Web Services	Cloud service
.AWSXCLBIN	FPGA executable	Executable to be run on AWS FPGA
BRAM	Block RAM	Local memory
CL	Custom logic	FPGA logical units
CPU	Central processing unit	-
DRAM	Dynamic random access memory	Global memory
DSP	Digital signal processing unit	Accumulator unit
FPGA	Field-programmable gate array	Custom accelerator
GPU	Graphics processing unit	Graphics accelerator
HLS	High level synthesis	Compiler-based hardware design
IEEE754	Double-precision floating-point standard	Floating-point standard
IHP	Individual Household Problem	Algorithm stage
II	Initiation Interval	
LUT	Lookup table	Logical units available for CL design
OpenCL	Open Computing Language	https://www.khronos.org/opencl
Open MPI	Open message passing interface	https://www.open-mpi.org
PCIe	Peripheral Component Interconnect Express	Bus-connections with host
SLR	Super Logic Region	FPGA CL regions
URAM	Ultra RAM	Local memory
Xilinx VU9	FPGA on AWS	-

A More on Building Blocks of FPGAs' Optimizations

Now, we provide additional information on the building blocks of FPGA optimization presented in Section 4. Subsection A.1 presents the `RTL` implementation of the accumulator, Subsection A.2 overviews the arbitrary-precision fixed-point approximation, and Subsection A.3 delves into the details of implementing an associative reduce tree in hardware.

A.1 A comparison of RTL and HLS

The following listing reports the `RTL` description of the sequential accumulator in Section 7. For comparison purposes, we implement it using the `VHSIC` Hardware Description Language (`VHDL`), the same `RTL` language used in Peri (2020).

Listing 5: VHDL description of the Sequential Accumulator

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Adder module
entity single_acc is
    generic (
        din_WIDTH : integer := 64;       -- Width of input data
        dout_WIDTH : integer := 64        -- Width of output data
    );
    port (
        clk : in std_logic;                                -- Clock signal
        reset : in std_logic;                              -- Reset signal
        din0, din1 : in std_logic_vector(din_WIDTH-1 downto 0); -- Input data
        dout : out std_logic_vector(dout_WIDTH-1 downto 0) -- Accumulation
    result
    );
end entity single_acc;

architecture Behavioral of single_acc is
    -- Registers for storing input and output data
    signal din0_buf, din1_buf : std_logic_vector(din_WIDTH-1 downto 0);
    signal dout_buf : std_logic_vector(dout_WIDTH-1 downto 0);
begin
    -- Copy input data from wires to registers
    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                din0_buf <= (others => '0');
                din1_buf <= (others => '0');
            else
                din0_buf <= din0;
                din1_buf <= din1;
            end if;
        end if;
    end process;

    -- Perform accumulation
    dout_buf <= din0_buf + din1_buf;

    -- Output the result
    dout <= dout_buf;

```

```
43 end architecture Behavioral;
44
45 -- Copy the input stream to BRAM
46 entity runOnfpga_st_k_RAM_AUTO_1R1W is
47     generic (
48         DataWidth : integer := 64;    -- Width of data
49         AddressWidth : integer := 3; -- Width of address
50         AddressRange : integer := 8  -- Range of address
51     );
52     port (
53         address0 : in std_logic_vector(AddressWidth-1 downto 0); -- Address
54         ce0 : in std_logic;                                      -- Chip enable in
55         d0 : in std_logic_vector(DataWidth-1 downto 0);  -- Data in
56         we0 : in std_logic;                                      -- Write enable in
57         q0 : out std_logic_vector(DataWidth-1 downto 0); -- Data out
58         reset : in std_logic;                                    -- Reset in
59         clk : in std_logic                                       -- Clock in
60     );
61 end entity runOnfpga_st_k_RAM_AUTO_1R1W;
62
63 architecture Behavioral of runOnfpga_st_k_RAM_AUTO_1R1W is
64 begin
65     -- Internal RAM
66     (* ram_style = "auto" *)
67     reg [DataWidth-1:0] ram[0:AddressRange-1];
68
69     -- Read and write operations on RAM
70     process(clk)
71     begin
72         if rising_edge(clk) then
73             if reset = '1' then
74                 for i in ram'range loop
75                     ram(i) <= (others => '0');
76                 end loop;
77             else
78                 if ce0 = '1' then
79                     if we0 = '1' then
80                         ram(conv_integer(address0)) <= d0;
81                     end if;
82                     q0 <= ram(conv_integer(address0));
83                 end if;
84             end if;
85         end if;
86     end process;
87
```

```vhdl
end architecture Behavioral;

-- Top-level module
entity runOnfpga is
    generic (
        AddressRange : integer := 8    -- Number of elements in the array
    );
    port (
        ap_clk : in std_logic;          -- Clock input
        ap_rst : in std_logic;          -- Reset input
        ap_start : in std_logic;        -- Start input
        in_preinit : in std_logic_vector(63 downto 0); -- Initialization input
        ap_done : out std_logic;        -- Done output
        out_r : out std_logic_vector(63 downto 0);    -- Output data
        out_r_ap_vld : out std_logic    -- Output valid signal
    );
end entity runOnfpga;

architecture Behavioral of runOnfpga is
    -- Local signals
    signal accumulation_sum, loaded_data : std_logic_vector(63 downto 0); -- Accumulation and loaded data
    signal adder_result, temp_result : std_logic_vector(63 downto 0);     -- Adder and temporary result
    signal counter : std_logic_vector(3 downto 0) := AddressRange;        -- Counter to track elements
begin
    -- Add reset for the counter
    process(ap_clk, ap_rst)
    begin
        if ap_rst = '1' then
            counter <= "0000";  -- Reset counter
        elsif rising_edge(ap_clk) then
            if ap_start = '1' then
                if counter < AddressRange then
                    counter <= counter + 1;  -- Increment counter
                end if;
            end if;
        end if;
    end process;

    -- Instantiate an adder module
    adder_1 : entity work.single_acc
    generic map (
```

```
129         din_WIDTH => 64,
130         dout_WIDTH => 64
131     )
132     port map (
133         clk => ap_clk,
134         reset => ap_rst,
135         din0 => accumulation_sum,
136         din1 => loaded_data,
137         dout => adder_result
138     );
139 
140     -- Assign din0 from the previous result
141     process(ap_clk)
142     begin
143         if rising_edge(ap_clk) then
144             if ap_rst = '1' then
145                 accumulation_sum <= (others => '0');
146             else
147                 accumulation_sum <= temp_result;
148             end if;
149         end if;
150     end process;
151 
152     -- Copy din1 from local BRAM
153     process(ap_clk)
154     begin
155         if rising_edge(ap_clk) then
156             loaded_data <= q0;
157         end if;
158     end process;
159 
160     -- Copy the result to the next
161     process(ap_clk)
162     begin
163         if rising_edge(ap_clk) then
164             temp_result <= adder_result;
165         end if;
166     end process;
167 
168     -- Output the result
169     process
170     begin
171         if counter = "1000" then
172             out_r <= accumulation_sum;  -- Assign the accumulated value
173             ap_done <= '1';             -- Indicate accumulation done
```

```
174             out_r_ap_vld <= '1';
175         else
176             out_r <= (others => '0');    -- Default value when accumulation is
    not done
177             out_r_ap_vld <= '0';
178         end if;
179     end process;
180 
181 end architecture Behavioral;
```

A.2 Arbitrary-precision Fixed-point Approximation: An Overview

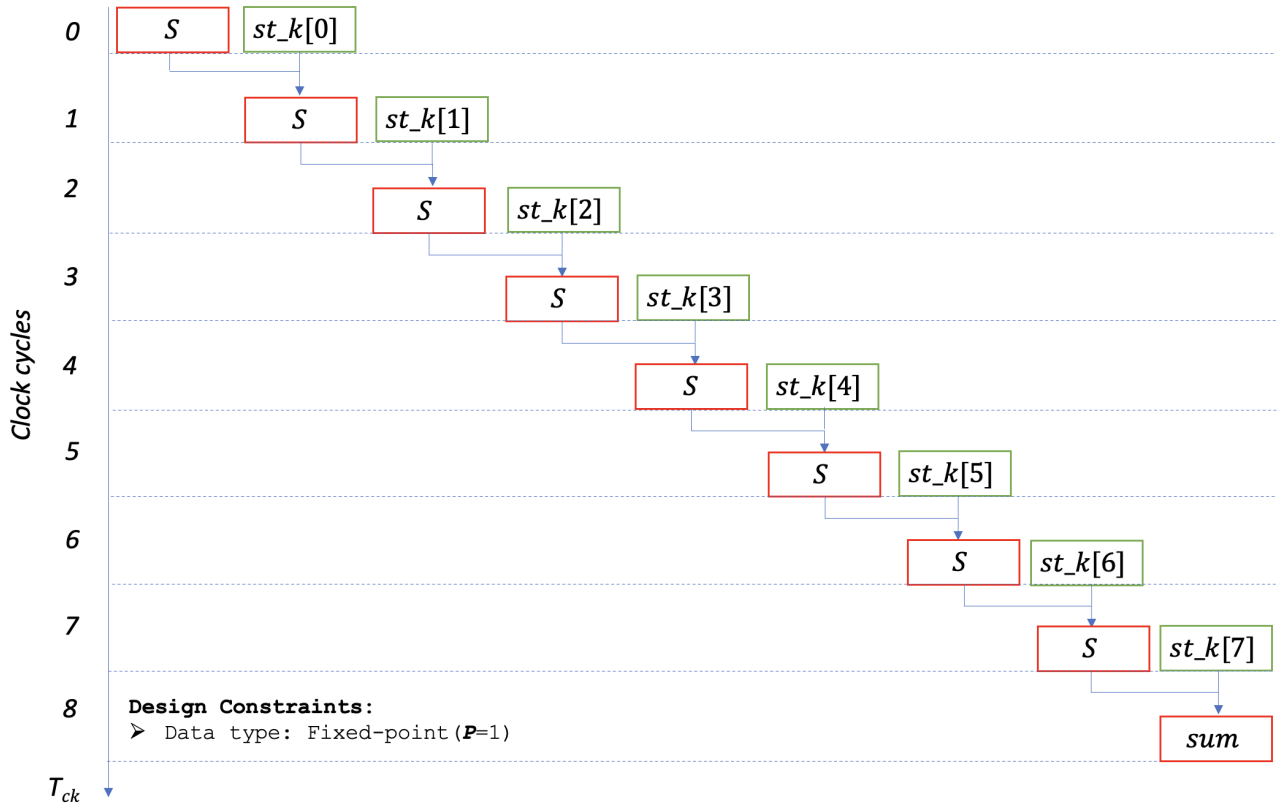
Computers carry out computation on numbers with finite representations. This raises the question of how we adequately approximate the uncountable real numbers. The advent of the IEEE floating-point standard (IEEE Standards Committee, 1985) and the readily available microprocessors that implemented it 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).

A.2.1 Implementation of fixed-point arithmetic in HLS

We refer to Xilinx, Inc. (2021) for a guide to the implementation of arbitrary precision in Vitis.

Figure A.1: Fixed-point Accumulation Operation



A.3 Associative Reduce Tree

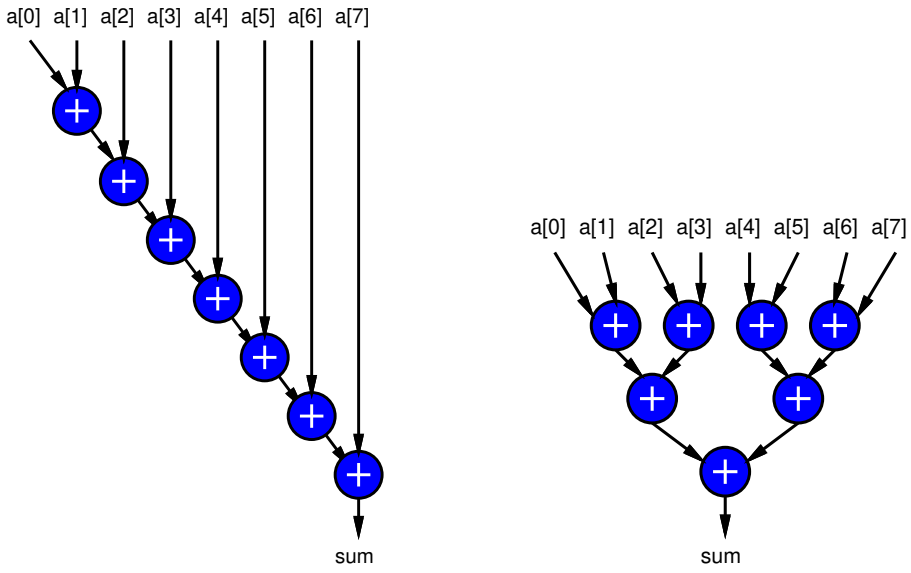
A *reduce* operation is a computational construct designed to reduce a large set of numbers into a single value. Common ways to reduce a set of numbers to a single digest include summing them up, multiplying them together, and identifying the maximum or minimum value of the set. For example, in Subsection 4.1 we consider an add-reduce tree of an array of $N = 8$ elements:

```
sum=0;
for (int i=0;i<8;i++)
   sum+=a[i];
```

This sequential summation performs $N - 1 = 7$ serial additions operations,

$$sum = (((((((a[0] + a[1]) + a[2]) + a[3]) + a[4]) + a[5]) + a[6]) + a[7])$$

exactly as illustrated in Figure A.2(a).



(a) Sequential (b) Associative Reduce Tree

Figure A.2: Associative Reduce Tree Transformation for Sequential Accumulation

When the reduce operation is associative—such as in the case of fixed-precision fixed-point (but not in the case of the IEEE754 double-precision floating-point format, as discussed in Subsection 4.2)—we can leverage parallelism to execute the $N-1$ operations in $\log_2 N$ steps.[24] This is achieved by employing a tree structure in which each step (or level in the tree) successively reduces the number of values by half through pair-wise combinations,

$$sum = (((a[0] + a[1]) + (a[2] + a[3])) + ((a[4] + a[5]) + (a[6] + a[7])))$$

as illustrated in Figure A.2(b). Given adequate hardware, an associative reduce tree (Figure A.2(b)) can perform $N - 1 = 7$ operations in $\log_2 N = 3$ sequential steps (also referred to as the *depth* of the tree).

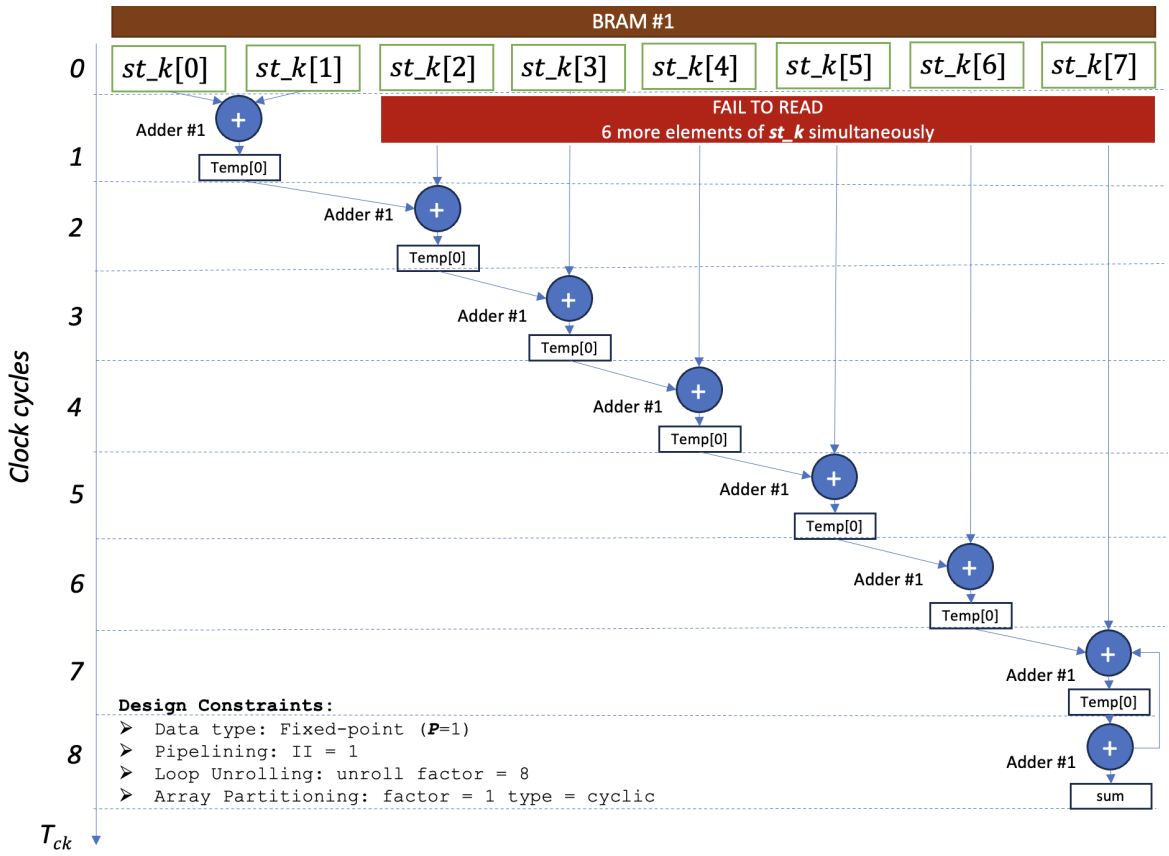
A.3.1 Memory-access bottleneck in absence of array partitioning

Figure A.3 shows the data flow of an accumulator with loop unroll and no array partitioning. This is a circuit that tries to unroll by a factor of 8 the addition of the fixed-point elements of an array `st_k` of size $J = 8$. The vertical dimension illustrates in which clock cycles these operations are performed (*scheduling*). The circuit fails to execute the prescribed unrolling because of a memory reading conflict that prevents reading more than two elements from the

[24] The logarithmic base 2 comes from the fact that we use a binary addition operation to digest pairs of numbers at a time. At each stage, we divide the number of partial sums in half, such that it takes us $\log_2 N$ steps to reduce to a single, final sum. Had we used a k-input operator to reduce k numbers to one at each stage, we would need $\log_k N$ steps.

same BRAM.

Figure A.3: Data Flow of Accumulator with Loop Unroll (No Array Partitioning)



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 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

Table A.2: 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	4	1	1.650	122
f1.4xlarge	8	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:* AWS instances, AWS specs.

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

C.1 FPGA Designs Performance and Resource Utilization

First, Table A.3 reports time performance and resource utilization by hardware design.

C.2 Efficiency Gains of Benchmark Economy

Next, Table A.4 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.

Differences in the execution time of initialization and printing operations between FPGA and CPU experiments are attributed to their parallel execution via `Open MPI` on the CPU

Table A.3: FPGA Designs Performance and Resource Utilization by Grid Size

	Three-Kernel	**Single-Kernel**					
Aggr. Capital	*4*	*4*			*8*		
Indiv. Capital	*100*	*100*	*200*	*300*	*100*	*200*	*300*
Time (s)	415.14	1002.62	1482.11	2245.56	2579.66	4627.80	7147.36
Cost ($)	0.19	0.46	0.68	1.03	1.18	2.12	3.28
Energy (J)	13699.54	17044.46	25195.90	38174.60	43854.19	78672.63	121505.20
BRAM(%)	44.29	21.31	27.32	33.10	27.32	37.92	47.26
DSP(%)	55.32	31.13	31.13	31.13	31.31	31.31	31.31
Registers(%)	25.71	12.00	12.00	12.12	12.06	12.17	12.26
LUT(%)	57.03	25.21	25.97	26.56	25.43	26.18	26.74
URAM(%)	16.50	5.38	5.38	5.38	5.38	5.38	5.38

Note: Solution time (in seconds), cost (in USD), energy (in joules) and FPGA resources (rows) across hardware designs (three- and single-kernel) and grid sizes on individual capital $N_k = \{100, 200, 300\}$ and aggregate capital $N_M = \{4, 8\}$ (columns). Time performance is measured in seconds required to solve 1,200 baseline economies on a single FPGA (f1.2xlarge) across the different hardware designs and grid sizes (columns). Resources are measured (using Xilinx Vivado) as a percentage of Xilinx VU9P FPGA's resources utilized by AWS images associated with the different hardware designs and grid sizes (columns). *Available Resources:* BRAM (1,680), DSP (5,640), Registers (1,790,400), LUTs (895 thousand), URAM (800). Available resources are lower than total resources because they exclude resources utilized by the AWS shell that are not available for `CL` design.

Table A.4: Performance Comparison

	CPU cores			**FPGA devices**		
N.	1	8	48	1	2	8
Exec Time (s)	28464.55	3656.52	613.81	431.60	223.40	69.51
Init Time (s)	0.36	0.04	0.01	0.81	0.67	0.84
Print Time (s)	11.70	1.58	0.28	15.10	14.50	14.81
Sol. Time (s)	28452.5	3654.74	613.37	415.14	207.55	51.87
Cost ($)	0.94	0.97	0.97	0.19	0.19	0.19
Energy (J)	227619.90	233903.34	235535.59	13699.54	13698.26	13693.02
AWS Instance	m5n.large	m5n.4xlarge	m5n.24xlarge	f1.2xlarge	f1.4xlarge	f1.16xlarge

Note: Execution, initialization, printing and solution 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 physical cores, Columns 1-3) and using FPGA acceleration on Amazon F1 instances (connected to 1, 2, 8 FPGA devices, Columns 4-6).

experiments and sequential execution on the CPU (host side) of the FPGA accelerated experiments. These differences can be eliminated by using `Open MPI` on the host side of the FPGA experiments. The FPGA has extra time allocation costs due to the OpenCL initialization of

host/device communications. Crucially, the relative magnitude of the non-kernel operations time washes out as the number of economies increases. Not surprisingly, the relative time spent on non-kernel operations disproportionately affects the experiment with 8 FPGAs, where non-kernel tasks account for roughly 25% of the total execution time. These results suggest that the use of 8 FPGAs may be more cost-effective when executing a large amount of economies in parallel.

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 FPGA average 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.[25] 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).[26] The comparison between the Turbostat application and the TDP establishes that our application requires approximately the maximum CPU power.

We map this estimate into our M5N instances with 1, 8, and 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 CPU Performance Across Grid Sizes

Finally, Table A.5 reports the CPU performance across different sizes of the grid.

C.4 Precision Accuracy Analysis

This section reports the accuracy analysis associated with FPGA and CPU implementation of the Krusell and Smith (1998) algorithm.

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

Table A.5: CPU Performance by Grid Size

Aggregate Capital, N_M	*4*			*8*		
Individual Capital, N_k	*100*	*200*	*300*	*100*	*200*	*300*
Exec. Time (s)	28464.55	51007.22	77061.15	71762.40	143718.80	229127.68
Init. Time (s)	0.36	0.38	0.39	0.37	0.40	0.41
Print Time (s)	11.70	12.72	14.94	14.38	15.94	18.38
Sol. Time (s)	28452.5	50994.12	77045.81	71747.64	143702.46	229108.89
Cost ($)	0.94	1.69	2.55	2.37	4.75	7.57
Energy (J)	227619.90	407952.96	616366.51	573981.11	1149619.67	1832871.10

Note: Execution, initialization, printing and solution time (in seconds), cost (in USD) and energy (in joules) to solve 1,200 baseline economies on a single core CPU (m5n.large) for different grid sizes (columns) on individual capital $N_k = \{100, 200, 300\}$ and aggregate capital $N_M = \{4, 8\}$.

Table A.6: 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	0.1460	0.9599	0.1554	0.9587
Fixed Point	0.1460	0.9599	0.1554	0.9587

Panel B: Policy Function, k'

$\text{Mean}\left(\frac{\lvert\text{Fixed}-Float\rvert}{Float}\right)\%$	4.0e-10	$\text{Max}\left(\frac{\lvert\text{Fixed}-Float\rvert}{Float}\right)\%$	2.6e-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

$\text{Mean}\left(\frac{\lvert\text{Fixed}-Float\rvert}{Float}\right)\%$	2.4e-09	$\text{Max}\left(\frac{\lvert\text{Fixed}-Float\rvert}{Float}\right)\%$	3.0e-08

Panel D: Euler Equation Errors (EEE)

	EEE	FPGA	CPU	$\lvert\Delta_{\text{FPGA}-\text{CPU}}/\text{CPU}\rvert\%$
$N_k = 100$	Mean (%)	0.12	0.12	1.35e-07
	Max (%)	1.03	1.03	4.85e-07
$N_k = 300$	Mean (%)	0.14	0.14	3.29e-07
	Max (%)	0.21	0.21	1.83e-07

Panel A of Table A.6 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- and fixed-point in the FPGA and CPU, respectively. Panel B reports the mean and max relative difference (in percent) between the policy functions computed under floating- and fixed-point. Panel C reports moments of the distribution of individual capital holdings at $T = 1,100$ (mean, standard deviation, and quartiles) under floating- and fixed-point. The last row reports their mean and max relative difference in percent. Panel D reports the mean/max Euler equation errors expressed in percent, associated with policy functions estimated in fixed-point using the FPGA (column 2), in floating-point on the CPU (column 3), and relative absolute difference, all in percent, for different individual capital holdings grid sizes, $N_k \in \{100, 300\}$ (rows), with $N_M = 4$.

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 independent research at the CU Boulder Research Computing Center and updated to 2020 data.[27]

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[28] –37% Natural Gas, 26% Coal, 37% Renewables– and US EPA information on the emissions of CO_2 per kWh by source[29] –0.91 Natural Gas, 2.21 Coal, 0.1 Renewables[30]– to determine the average 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 five 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.

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

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 (68.54x), the 150 million core hours per year would map into $2,188,583$ FPGA hours per year. In this scenario, the carbon footprint would total:

$$2,188,583 \frac{\text{FPGA hour}}{\text{year}} \cdot 0.031 \frac{\text{lbs } CO_2}{\text{FPGA hour}} = 68,489 \frac{\text{lbs } CO_2}{\text{year}}$$

or approximately 31.07 metric tons of CO_2 per year. This is equivalent to a reduction in the carbon footprint from 168 cars to 6 cars per year.