

Cheela, Bhagath; DeHon, André; Fernández-Villaverde, Jesús; Peri, Alessandro

Article

Programming FPGAs for economics: An introduction to electrical engineering economics

Quantitative Economics

Provided in Cooperation with:

The Econometric Society

Suggested Citation: Cheela, Bhagath; DeHon, André; Fernández-Villaverde, Jesús; Peri, Alessandro (2025) : Programming FPGAs for economics: An introduction to electrical engineering economics, Quantitative Economics, ISSN 1759-7331, The Econometric Society, New Haven, CT, Vol. 16, Iss. 1, pp. 49-87,
<https://doi.org/10.3982/QE2344>

This Version is available at:

<https://hdl.handle.net/10419/320332>

Standard-Nutzungsbedingungen:

Die Dokumente auf EconStor dürfen zu eigenen wissenschaftlichen Zwecken und zum Privatgebrauch gespeichert und kopiert werden.

Sie dürfen die Dokumente nicht für öffentliche oder kommerzielle Zwecke vervielfältigen, öffentlich ausstellen, öffentlich zugänglich machen, vertreiben oder anderweitig nutzen.

Sofern die Verfasser die Dokumente unter Open-Content-Lizenzen (insbesondere CC-Lizenzen) zur Verfügung gestellt haben sollten, gelten abweichend von diesen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Terms of use:

Documents in EconStor may be saved and copied for your personal and scholarly purposes.

You are not to copy documents for public or commercial purposes, to exhibit the documents publicly, to make them publicly available on the internet, or to distribute or otherwise use the documents in public.

If the documents have been made available under an Open Content Licence (especially Creative Commons Licences), you may exercise further usage rights as specified in the indicated licence.



<https://creativecommons.org/licenses/by-nc/4.0/>

Online Appendix for

Programming FPGAs for Economics:

An Introduction to Electrical Engineering Economics

Bhagath Cheela*

André DeHon†

Jesús Fernández-Villaverde‡

Alessandro Peri§

October 7, 2024

*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

Introduction

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 1: VHDL description of the Sequential Accumulator

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 -- Adder module
5 entity single_acc is
6     generic (
7         din_WIDTH : integer := 64;      -- Width of input data
8         dout_WIDTH : integer := 64      -- Width of output data
9     );
10    port (
11        clk : in std_logic;              -- Clock signal
12        reset : in std_logic;            -- Reset signal
13        din0, din1 : in std_logic_vector(din_WIDTH-1 downto 0); -- Input data
14        dout : out std_logic_vector(dout_WIDTH-1 downto 0) -- Accumulation
15        result
16    );
17 end entity single_acc;
18
19 architecture Behavioral of single_acc is
20     -- Registers for storing input and output data
21     signal din0_buf, din1_buf : std_logic_vector(din_WIDTH-1 downto 0);
22     signal dout_buf : std_logic_vector(dout_WIDTH-1 downto 0);
23 begin
24     -- Copy input data from wires to registers
25     process(clk)
26     begin
27         if rising_edge(clk) then
28             if reset = '1' then
29                 din0_buf <= (others => '0');
30                 din1_buf <= (others => '0');
31             else
32                 din0_buf <= din0;
33                 din1_buf <= din1;
34             end if;
35         end if;
36     end process;
37
38     -- Perform accumulation
39     dout_buf <= din0_buf + din1_buf;
40
41     -- Output the result
42     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

```

```

88 end architecture Behavioral;
89
90 -- Top-level module
91 entity run0nfpga is
92     generic (
93         AddressRange : integer := 8      -- Number of elements in the array
94     );
95     port (
96         ap_clk : in std_logic;           -- Clock input
97         ap_rst : in std_logic;           -- Reset input
98         ap_start : in std_logic;         -- Start input
99         in_preinit : in std_logic_vector(63 downto 0); -- Initialization
100     input
101         ap_done : out std_logic;          -- Done output
102         out_r : out std_logic_vector(63 downto 0); -- Output data
103         out_r_ap_vld : out std_logic      -- Output valid signal
104     );
105 end entity run0nfpga;
106
107 architecture Behavioral of run0nfpga is
108     -- Local signals
109     signal accumulation_sum, loaded_data : std_logic_vector(63 downto 0); --
110     Accumulation and loaded data
111     signal adder_result, temp_result : std_logic_vector(63 downto 0); --
112     Adder and temporary result
113     signal counter : std_logic_vector(3 downto 0) := AddressRange; --
114     Counter to track elements
115 begin
116     -- Add reset for the counter
117     process(ap_clk, ap_rst)
118     begin
119         if ap_rst = '1' then
120             counter <= "0000"; -- Reset counter
121         elsif rising_edge(ap_clk) then
122             if ap_start = '1' then
123                 if counter < AddressRange then
124                     counter <= counter + 1; -- Increment counter
125                 end if;
126             end if;
127         end if;
128     end process;
129
130     -- Instantiate an adder module
131     adder_1 : entity work.single_acc
132     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

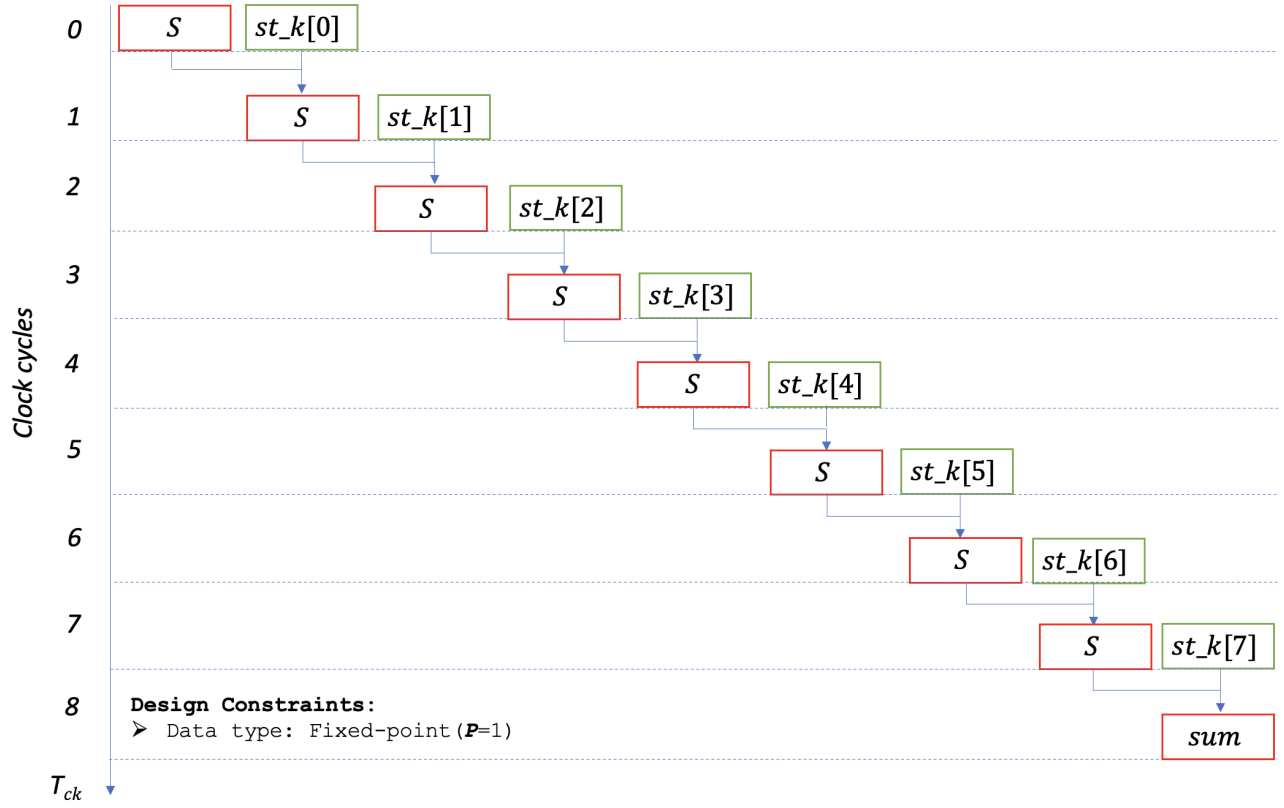
Computers carry out computation on numbers with finite representations. This raises the question of how we adequately approximate the uncountable real numbers. The advent of the IEEE floating-point standard ([IEEE Standards Committee, 1985](#)) and the readily available microprocessors that implemented 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., 2020](#)).

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).

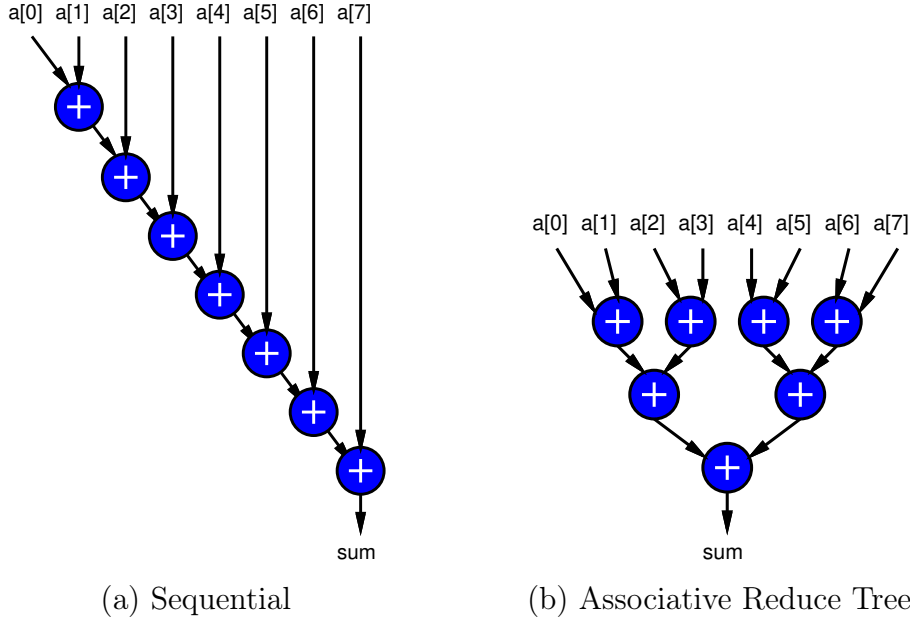


Figure A.2: Associative Reduce Tree Transformation for Sequential Accumulation

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.¹ 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).

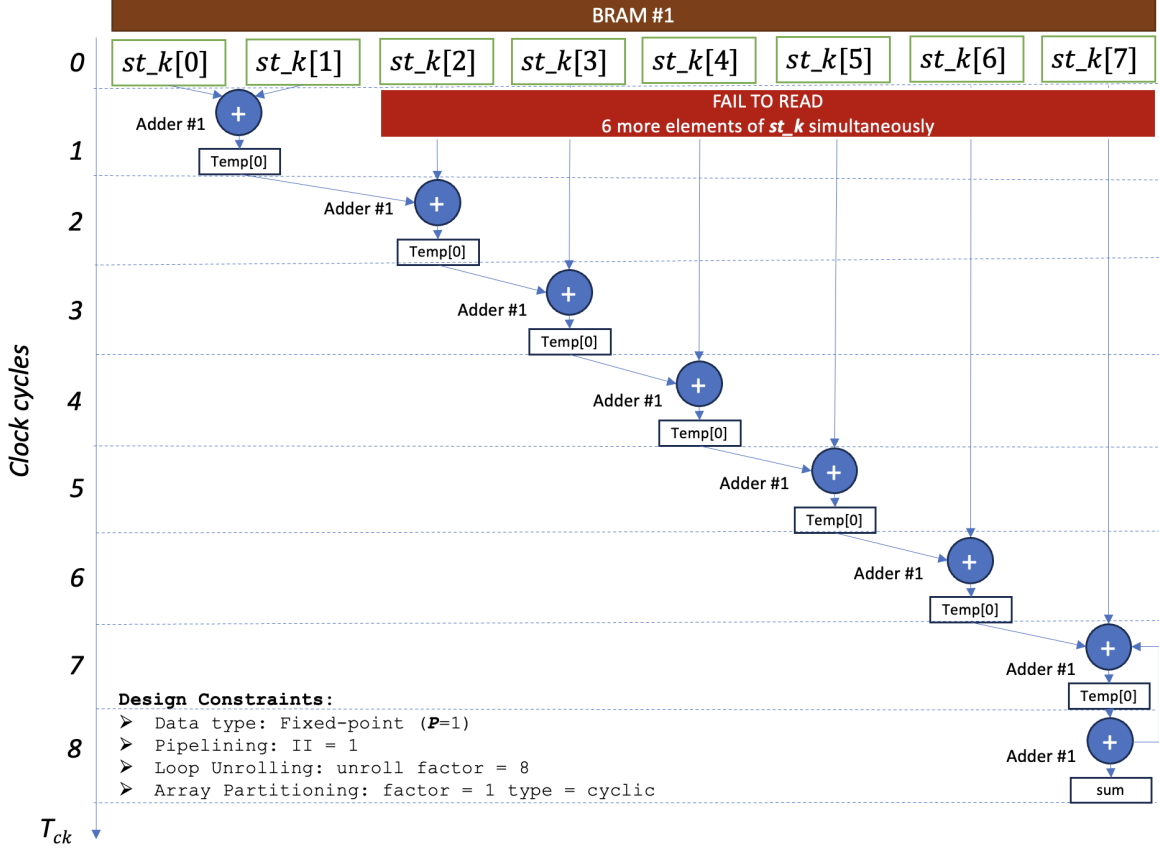
A.3.1 Memory-access bottleneck in absence of array partitioning

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

¹The 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.² 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).³ 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:

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

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

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

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.

²Source: <https://www.linux.org/docs/man8/turbostat.html>.

³Source: <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'

Mean $\left(\frac{ \text{Fixed}-\text{Float} }{\text{Float}}\right) \%$	4.0e-10	Max $\left(\frac{ \text{Fixed}-\text{Float} }{\text{Float}}\right) \%$	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
Mean $\left(\frac{ \text{Fixed}-\text{Float} }{\text{Float}}\right) \%$	2.4e-09			Max $\left(\frac{ \text{Fixed}-\text{Float} }{\text{Float}}\right) \%$	3.0e-08

Panel D: Euler Equation Errors (EEE)

	EEE	FPGA	CPU	$ \Delta_{\text{FPGA-CPU}}/\text{CPU} \%$
$N_k = 100$	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.⁴

The RC analysis assumes that each CURC HPC core consumes 13W, that is, 0.013 kilowatts per CURC HPC core hour ($13\text{W}/\text{core} \cdot 1\text{hour}/1000 = 0.013\text{kWh}$). It then uses the Xcel Energy power generation breakdown in the state of Colorado in 2020⁵ –37% Natural Gas, 26% Coal, 37% Renewables– and US EPA information on the emissions of CO₂ per kWh by source⁶ –0.91 Natural Gas, 2.21 Coal, 0.1 Renewables⁷– to determine the average pounds of CO₂ 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 CO₂ 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 CO₂ 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.

⁴Andrew Monaghan, Andrew.Monaghan-1@Colorado.EDU.

⁵Source: Xcel Stats, <https://co.my.xcelenergy.com/s/energy-portfolio/power-generation>.

⁶Source: US EPA <https://www.eia.gov/tools/faqs/faq.php?id=74&t=11>.

⁷This 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 CO₂ per year. This is equivalent to a reduction in the carbon footprint from 168 cars to 6 cars per year.

References

- IEEE Standards Committee (1985). *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE.
- Krusell, P. and A. A. Smith (1998). Income and wealth heterogeneity in the macroeconomy. *Journal of Political Economy* 106(5), 867–896.
- Peri, A. (2020). A hardware approach to value function iteration. *Journal of Economic Dynamics and Control* 114, 1–18.
- Xilinx, Inc. (2020). *Performance and Resource Utilization for Floating Point*. Xilinx, Inc.
- Xilinx, Inc. (2021). *Overview of Arbitrary Precision Fixed-Point Data Types*. Xilinx. Accessed on 2023/11/02.