CovaSimplify: An Agent-Based Epidemic Model Accelerator

Carson Brantley, Ishan Vyas

***Abstract*—In this paper, we present CovaSimplify, an agent-based epidemic model accelerator designed for any-scale, any-duration simulation of disease progression via the CovaSim epidemic model. We present a) a simplification of CovaSim that maintains key parameters while reducing the computational complexity and b) the microarchitecture of CovaSimplify. The microarchitecture features a parallelized initialization phase, a stream processing architecture, and individual modules along the datapath with fine-grained parallelism. CovaSimplify has been synthesized, placed, and routed onto an Amazon F2 FPGA and achieves a 2x speedup over conventional CovaSim implementation.**

***Index Terms*—Accelerator; Stream Processor; Microarchitecture; Epidemiology; Agent-Based Modeling**

I. Introduction

THE emergence of the COVID-19 pandemic brought to light the need for high-speed, highly-accurate simulations of disease progression. As the disease progressed, policymakers and healthcare centers needed information about potential outbreaks to support rapid decision-making [1]. For that reason, highly parameterizable stochastic simulations were developed to model possible outcomes. Unfortunately, the state of a certain area can change on the order of days or even hours. These simulations, though accurate, can take minutes per run. A representative sample of possible outcomes can take days, which slows down policymaking and reduces the response effectiveness, risking the lives of thousands of people vulnerable to disease.

Traditionally, dynamic models have used differential equations to track progression over time. However, these differential-equation-based models are suboptimal for multiple reasons [2]. First, they are abstract and inflexible: incorporating many parameters is impractical and reduces the accuracy of the model. Second, they are difficult to develop: model development decisions often conflict with observed population data. Third, they are difficult to explain to policymakers, who have to evaluate the validity of the model without substantial understanding of differential equations. For that reason, epidemiologists are investigating the use of Agent-Based Models (ABMs); these models simulate dynamics at an individual level by representing everyone as unique agents. These models use common population statistics as inputs to the model and progress intuitively using weighted random infections.

The main agent-based epidemiological model used during and after the COVID-19 pandemic is CovaSim, developed by the Gates Foundation's Institute for Disease Modeling (IDM). The model is renowned for its ease of use and its parameterizable nature. Unlike existing epidemic simulators like COVID_SIM, developed at McGill University, IDM's CovaSim is built into a python package with extensive documentation and support. Furthermore, CovaSim includes dozens of parameters related to the nature of the disease, the type of population network, and various potential interventions. Nonetheless, a single CovaSim simulation can take minutes to run; for that reason, we aim to accelerate the CovaSim model.

Associated with Pratt School of Engineering, Duke University. 305 Teer Engineering Building, Durham, NC 27708. Emails: {carson.brantley; ishan.vyas}@duke.edu

II. Algorithm

A. Existing Solutions

Agent-based models (ABMs) are a key tool in epidemiology for capturing complex population dynamics. Major ABMs are reviewed in [3], including those by Hoertel et al. [4], O'Neil and Sattenspiel [5], and Willem et al. [6], and analyze trade-offs in model parameters. Hunter et al. [7] demonstrates the importance of adaptable, data-driven models with a detailed measles simulation in Ireland.

To improve scalability and runtime, several optimized tools have emerged. Al-Handawi's COVID_SIM [8] uses CUDA acceleration but supports only COVID-19 and limited policy levers. FLAME GPU [9] targets cellular-level modeling but poses usability issues for non-programmers [10].

Hardware acceleration has also been attempted. Fain and Dobrovolny [11] use GPUs in a hybrid ABM/PDE model for viral spread in cell cultures. However, their tool is closed-source, not fully agent-based, and unsuitable for modeling human transmission.

Overall, despite progress in performance, existing solutions often lack flexibility, ease of use, or general applicability, highlighting the need for accessible, disease-agnostic models like CovaSim and accelerators such as CovaSimplify.

B. CovaSim Structure

CovaSim is a time-stepped agent-based simulation where individual attributes of each agent such as age, health state, and contacts are stored in parallel arrays, enabling efficient and modular computation. At each timestep, the simulation sequentially executes four primary modules: Update States (US), Compute Transmission and Susceptibility (CTS), Compute Infections (CI), and Apply Infections (INF). The simulation uses a stochastic "susceptible, exposed, infectious, recovered" model (Figure 1), where agents probabilistically transition through various health states based on internal timers and

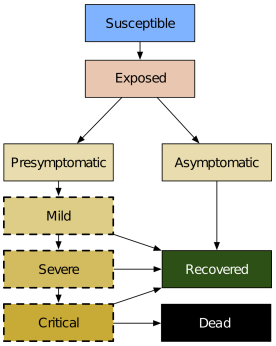

Fig. 1. Disease progression in CovaSim model.

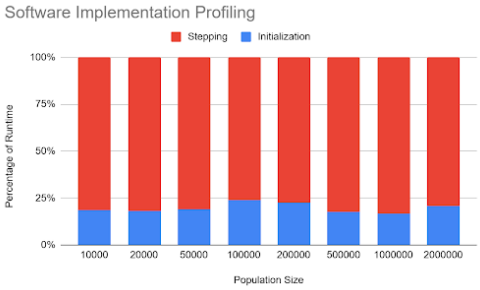

Fig. 2. CovaSim Runtime Profiling

infection pressures from contacts. These internal infection progression timers are calculated in the INF stage of the simulation, which probabilistically determines what final state an infected agent will progress to and the dates at which they progress to each state. The US stage checks to see if these dates have elapsed, and if necessary, updates the current state of the agent to reflect the new change. CTS computes the probability of an infection occurring between an infectious source agent and a susceptible target agent, while CI decides whether an infection actually occurs. The structured, per-agent and per-connection computational phases allow effective stream processing and facilitate hardware acceleration in CovaSimplify.

C. Profiling

Using Python's cProfile on representative CovaSim runs, we identified two primary runtime phases: the initialization phase (agent creation and network setup), accounting for approximately 20% of total runtime, and the compute phase (timestep simulation loop), accounting for the remaining 80%. A full analysis is shown in Figure 2. Per Amdahl's Law, achieving meaningful speedup requires accelerating both phases, since the serial portion limits overall performance gains.

This observation motivates our decision to offload both the timestep loop and the initialization phase. By targeting both major components, CovaSimplify avoids the diminishing returns typically associated with partial acceleration, and instead delivers meaningful improvements in end-to-end simulation performance.

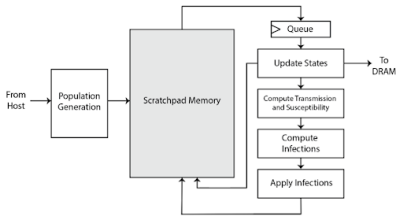

Fig. 3. Overview of CovaSimplify Microarchitecture.

III. CovaSimplify Microarchitecture

At a high level, the CovaSimplify accelerator employs a stream processor architecture shown in Figure 3. We address two key challenges: compute-intensive agent interactions and FPGA memory bandwidth limitations. CovaSim's random agent interactions result in irregular memory access patterns, hindering large batched reads and demanding fine-grained, low-latency memory operations. A streaming approach mitigates these issues by processing data as it arrives, minimizing memory round-trips, and maximizing compute unit utilization. The pipeline includes four modules, one for each stage of the algorithm. To overcome limited memory ports, tightly coupled pipelines ensure efficient bandwidth use and continuous processing despite serialized access patterns. Additionally, each module incorporates low-level parallelism for faster computations, and the scalable architecture supports multiple module instances for higher parallelism when resources permit.

A. Population Generation

The population generation phase initializes agents and their contact graph entirely on-chip, eliminating PCIe transfer bottlenecks and enabling rapid, highly parallel initialization. Given population size N and number of edges per person E, the module outputs arrays for agent ages, transmission factors, and social contacts, assigning attributes via pseudo-randomly indexed look-up tables drawn from CovaSim's statistical distributions, and marks the initial infected population. Edges are generated using parallel linear-feedback shift registers (LFSRs) and a single-cycle bit-masked modulo operation. Offloading these tasks to FPGA hardware results in faster setup and a structured, simulation-ready population.

B. Memory Architecture

The result of the population generation is sent into the scratchpad URAM; separate memories hold each individual statistic. This partitioned memory structure enables parallel access to different attributes of each agent, allowing CovaSimplify to stream in one agent's information per cycle. The contact graph is represented using two parallel edge arrays, and person-specific properties are stored in memories for state, relative transmission factor, and age. To support inter-agent interactions, each simulation step involves accessing both a source individual and their associated contacts, requiring randomly indexed memory reads. Due to the limited number of read/write ports available on FPGA BRAMs, we stagger

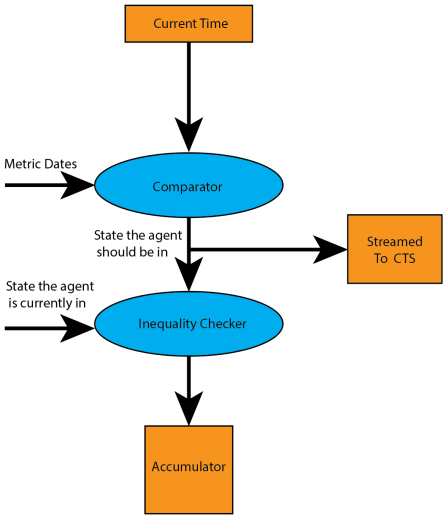

Fig. 4. Microarchitectural Implementation of the Update States Module.

memory accesses for source and target agents and coordinate reads via queues to ensure all data is valid before being sent to the Update States Module. To avoid memory data hazards, date and state memories are double-buffered, allowing simultaneous read and write access across alternate time steps. This allows the US and INF modules to write the next timestep's data while the current one is still being read.

C. Update States

This module reads in each person's current state and the dates they will transfer to another state. Each of these dates are compared against the current time in order to determine if a person should be in that state at that time. The US module then determines their next state based on both their current state and whether they should be in the next state. If a transition occurs, then counters for both the cumulative number of people entering that state and the daily total are incremented, and the person's information is sent to the CTS stage. Other relevant information, such as the target's susceptibility and age, is also passed through to the CTS stage. Figure 4 shows the microarchitectural implementation of the US module.

D. Compute Transmission and Susceptibility

The CTS module computes the source's relative transmission factor by multiplying it with their viral load, a scaling factor that represents how a person's infectiousness changes over the course of an infection. Both values are stored as 16-bit unsigned fixed-point numbers, with 6 integer bits and 10 fractional bits. If the source is infectious, as determined by the US module, then this value is passed to the CI module. Otherwise, the relative transmission factor is set to 0, which ensures that no infection will occur. Similar logic is used for the target's susceptibility, where the relative susceptibility is passed through if the target is unexposed to the disease and set to 0 if the target is currently infected. Figure 5 shows the microarchitectural implementation of the CTS module.

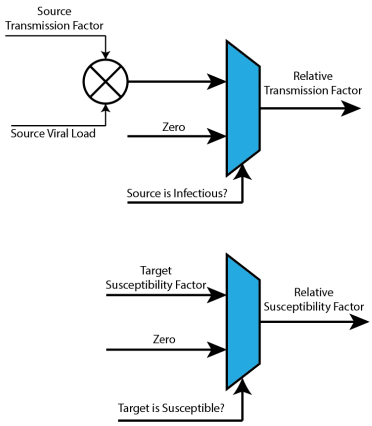

Fig. 5. Microarchitectural Implementation of the Compute Transmission and Susceptibility Module

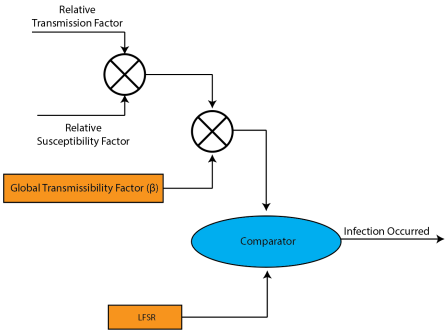

Fig. 6. Microarchitectural Implementation of the Compute Infections Module.

E. Compute Infections

This module calculates the total infection factor by multiplying the source's relative transmission factor with the target's relative susceptibility. This is then multiplied by the global infection probability β to determine the probability of an infection occurring. To determine whether an infection actually occurs, the probability of an infection is compared against a pseudo-randomly generated number, and if that number is less than the probability, the target's information is passed to the infect stage. Figure 6 shows the microarchitectural implementation of the CI module.

F. Apply Infections

Once it's been determined that a person will be infected, the INF module computes how far the disease will progress for them and the dates they will enter each state. Age-indexed LUTs determine the probability of a person entering the symptomatic, severe, critical, and dead states. Each LUT has an associated set of LFSRs that generates one pseudo-random number every cycle in order to determine if the infected person will advance to that stage. A separate set of LFSRs randomly selects stage durations from preloaded statistical LUTs. The datapath is designed to prevent an illogical outcome (such

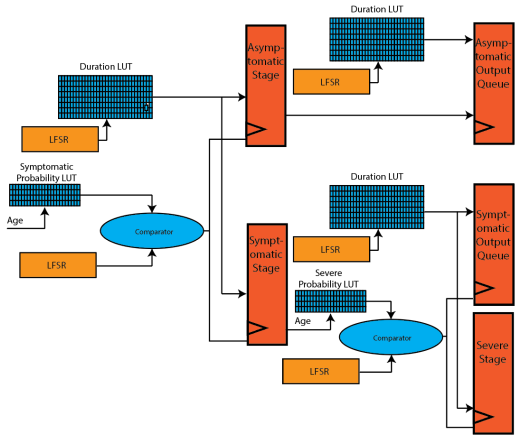

Fig. 7. Microarchitectural Implementation of the Apply Infections Module. Shown are the stages that determine whether an agent will develop symptoms and whether those symptoms will progress to the severe stage. Once an agent's final state is determined, their information is sent to an output queue. Not shown are the nearly identical stages that determine whether an agent will become critical, dead, or recovered.

as a person who will never develop symptoms dying from the disease) via each stage sending its output to one of two queues. If the person does not progress to a certain state, the person is sent directly to an output queue as a form of early exit, as it's impossible for them to progress to any further states. Otherwise, the person continues to the next stage, where asymptomatic cases exit and symptomatic ones are evaluated for severity. Figure 7 shows the microarchitectural implementation of the INF module.

IV. Experimental Methodology

To implement the CovaSimplify accelerator, all components were developed in Chisel HDL, enabling compile-time parameterization for agent count, edge density, and simulation time. This approach allowed for flexible experimentation across population sizes and disease parameters without modifying the core architecture.

We integrated the design into an FPGA system using the Beethoven framework, which provides a streamlined interface for memory-mapped communication and simulation-driven hardware development. The framework also facilitated interaction between our accelerator and host-based control logic.

Prior to synthesis, we performed cycle-accurate simulations using Beethoven's built-in VCS toolchain and DRAM simulator. These pre-synthesis tests were used to validate correctness and observe functional behavior under small-scale workloads.

For final evaluation, the design was synthesized, placed, and routed using Xilinx tools on an Intel Xeon m5zn instance, and then deployed onto an Amazon AWS F2 FPGA instance. This allowed us to determine realistic resource usage and clock timing on cloud-accessible reconfigurable hardware.

Our evaluations were based on runs that varied the population size and number of days the simulation ran. CovaSimplify was compared to the original Covasim software implementation, and the output from both were compared for accuracy. Python's cProfile library was used to measure the performance of both platforms.

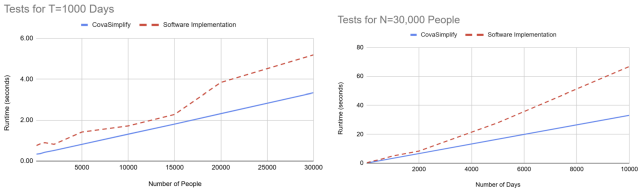

Fig. 8. Comparison of CovaSimplify's runtime against the software algorithm

V. Results

A. Validation

CovaSimplify's performance was first validated by analyzing waveforms generated by simulating the accelerator. These showed that one edge was being processed per cycle, and they also showed that per-day information was being written to DRAM after the edge list for that day was processed. Due to both the speed of the simulator and the size of the .vcd file generated, smaller simulations of under 100 agents for less than 10 days were performed. Once CovaSimplify was loaded on a physical FPGA, population size and the number of days to simulate were both increased substantially in tests, and performance was validated by observing that runtime increased linearly with growing population size, replicating results from the simulation. The accuracy of the model was validated by comparing the per-day output of CovaSimplify with the software CovaSim model for the same starting population size and duration; multiple trials were run to verify order-of-magnitude consistency.

B. Evaluation

CovaSimplify was compared against the original software implementation running on a personal computer. Across all tests, CovaSimplify returned final results in approximately half the time as the software implementation. This 2x speedup, while significantly smaller than what was expected, was fairly consistent across all tests. As the population size grows, the number of edges grows linearly with it. Since CovaSimplify's pipeline handles edges at an amortized throughput of one edge per cycle, runtime also grows linearly with population size. We believe the relatively small performance speedup is due to this single edge per cycle throughput. Processing more edges per cycle would enable a more significant speedup, but the current CovaSimplify scratchpad memory enables reading one person's information per cycle, which is unable to sustain higher throughput than approximately one edge per cycle. The results, both for varying population size and duration, are shown in Figure 8.

VI. Further Work

One avenue of future work is to better construct the memory hierarchy. Currently CovaSimplify's on-chip memory enables reading one agent's information per cycle. This is the limiting factor in preventing CovaSimplify from processing more than one edge per cycle, but this could be countered by increasing the number of read ports on each memory block and duplicating information into different memory blocks on chip. The

main challenge impeding this is the amount of data each agent requires: storing one copy of 32,000 agents on-chip utilized all of the available UltraRAM blocks and many of the BRAM blocks. More reads in parallel would require storing multiple copies of every agent, which could be accomplished by storing data for every agent in DRAM and only fetching a small subset of essential information to store in the on-chip scratchpads. This would increase the energy consumption of CovaSimplify, but it would also allow increased throughput compared to the current memory hierarchy.

Additional modifications to CovaSimplify include making the initialization stage more flexible. Currently, the lookup tables that determine how likely a person is to enter each state are preloaded with constant values at elaboration time, drawn from real-world statistical data. These LUTs are indexed by the target's age, and changing them from predetermined values to being dynamically loaded from DRAM would give the user more flexibility in how CovaSimplify models the infection. This reconfigurability is currently achievable by modifying the files that each LUT reads from in the hardware elaboration stage, but achieving it at runtime without re-synthesizing hardware would be ideal for end users.

VII. Conclusion

Working on this accelerator has yielded various insights, most of which reinforce insights that were introduced by prior accelerators. One strategy CovaSimplify employs is keeping almost all necessary data on-chip. This limited the number of DRAM accesses to only writing the output returned to the host, decreasing energy consumption. However, this also introduced some limitations to the number of agents that could be stored on chip and it introduced a bottleneck on how much data could be streamed per cycle.

In this paper, we presented CovaSimplify, an agent-based modeling accelerator that was able to efficiently compute interactions between agents and predict the spread of disease. With a compute pipeline and memory hierarchy designed for single edge per cycle throughput, CovaSimplify was able to accelerate both the initial population generation phase and the calculation of daily disease progression. Further modification of the memory hierarchy is needed to enable multiple edges per cycle throughput, but without this modification the accelerator was able to achieve a 2x speedup over a software algorithm.

Acknowledgments

Thank you to Professor Lisa Wu Wills and Chris Kjellqvist for their guidance.

References

[1] P. Dieckmann, K. Torgeirsen, S. A. Qvindesland, L. Thomas, V. Bushell, and H. L. Ersdal, "The use of simulation to prepare and improve responses to infectious disease outbreaks like COVID-19: Practical tips and resources from Norway, Denmark, and the UK," *Adv. Simul. (Lond.)*, vol. 5, no. 3, Apr. 2020, doi: 10.1186/s41077-020-00121-5.

[2] A. Afzal *et al.*, "Merits and limitations of mathematical modeling and computational simulations in mitigation of COVID-19 pandemic: A comprehensive review," *Arch. Comput. Methods Eng.*, vol. 29, no. 2, pp. 1311–1337, 2022, doi: 10.1007/s11831-021-09634-2.

[3] K. Al Handawi and M. Kokkolaras, "Optimization of infectious disease prevention and control policies using artificial life," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 6, no. 1, pp. 26–40, Feb. 2022, doi: 10.1109/TETCI.2021.3107496.

[4] N. Hoertel *et al.*, "A stochastic agent-based model of the SARS-CoV-2 epidemic in France," *Nat. Med.*, vol. 26, pp. 1417–1421, 2020, doi: 10.1038/s41591-020-1001-6.

[5] C. A. O'Neil and L. Sattenspiel, "Agent-based modeling of the spread of the 1918–1919 flu in three Canadian fur trading communities," *Am. J. Hum. Biol.*, vol. 22, no. 6, pp. 757–767, 2010.

[6] L. Willem, S. Stijven, E. Tijskens, *et al.*, "Optimizing agent-based transmission models for infectious diseases," *BMC Bioinformatics*, vol. 16, Art. no. 183, 2015, doi: 10.1186/s12859-015-0612-2.

[7] E. Hunter, B. Mac Namee, and J. Kelleher, "An open-data-driven agent-based model to simulate infectious disease outbreaks," *PLoS One*, vol. 13, no. 12, Art. no. e0208775, Dec. 2018, doi: 10.1371/journal.pone.0208775.

[8] K. Al Handawi and M. Kokkolaras, "Optimization of infectious disease prevention and control policies using artificial life," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 6, no. 1, pp. 26–40, Feb. 2022, doi: 10.1109/TETCI.2021.3107496.

[9] P. Richmond, D. Walker, S. Coakley, and D. Romano, "High performance cellular level agent-based simulation with FLAME for the GPU," *Brief. Bioinform.*, vol. 11, no. 3, pp. 334–347, May 2010, doi: 10.1093/bib/bbp073.

[10] R. A. Williams, "User experiences using FLAME: A case study modelling conflict in large enterprise system implementations," *Simul. Model. Pract. Theory*, vol. 106, Art. no. 102196, Jan. 2021, doi: 10.1016/j.simpat.2020.102196.

[11] B. G. Fain and H. M. Dobrovolny, "GPU acceleration and data fitting: Agent-based models of viral infections can now be parameterized in hours," *J. Comput. Sci.*, vol. 61, Art. no. 101662, 2022, doi: 10.1016/j.jocs.2022.101662.