Towards an FPGA Accelerator for Markov Brains

Arend Hintze[2,3] and Jory Schossau[1,2]

[1]Michigan State University, Department of Computer Science and Engineering, East Lansing, 48824, United States of America
[2]Michigan State University, BEACON Center for the Study of Evolution in Action, East Lansing, 48824, United States of America
[3]Dalarna University, Department of MicroData Analytics, Dalarna, 79188, Sweden
ahz@du.se

Abstract

The success of deep learning is to some degree based on our ability to train models quickly using GPU or TPU hardware accelerators. Markov Brains, which are also a form of neural networks, could benefit from such an acceleration as well. However, Markov Brains are optimized using genetic algorithms, which present an even higher demand on the acceleration hardware: Not only inputs to the network and its outputs need to be communicated but new network configurations have to be loaded and tested repeatedly in large numbers. FPGAs are a natural substrate to implement Markov Brains, who are already made from deterministic logic gates. Here a Markov Brain hardware accelerator is implemented and tested, showing that Markov Brains can be computed within a single clock cycle, the ultimate hardware acceleration. However, how current FPGA design and supporting development toolchains are limiting factors, and if there is a future size speed trade-off are explored here as well.

Introduction

Deep neural networks (DNN) together with deep learning are the most common technologies for the study and implementation of complex automation in the field of artificial intelligence. A DNN is usually optimized using backpropagation, but DNN have a predefined network structure. A more flexible option is neuroevolution that has been shown to be competitive to deep learning (Such et al., 2018; Real et al., 2020). A specific architecture amenable to neuroevolution is Markov Brains (MBs) (Marstaller et al., 2013) that is a form of recurrent neural network — similar to Cartesian Genetic Programming (Miller and Harding, 2009) — where computational units read from inputs and hidden states, and write their computational results into hidden states and outputs. Even the computational functions of MBs are not predefined, but optimized using a genetic algorithm (Hintze et al., 2017) . Markov Brains are particularly well-suited for embodied agent-based control and navigation tasks (Kvam et al., 2015; Edlund et al., 2011; Schossau et al., 2015; Marstaller et al., 2013) producing efficient digital logic circuitry tractable for information and representation analysis (Kirkpatrick and Hintze, 2020, 2019; Hintze et al., 2018; Albantakis et al., 2014) . Markov Brain technology remains yet underutilized compared to the widespread popularity of DNNs, despite a history of utility for research and having promising industrial application because of its ability to include arbitrary computational substrates (Hintze et al., 2019) (the Evolutionary Buffet Method), and for its natural resulting efficient circuit design. The use of Markov Brains is even well-supported by the Modular Agent Based Evolution Framework (MABE) (Bohm et al., 2017), which allows practitioners to easily implement high-performance agent-based tasks and other plugin modules. Even this convenient framework has only a modest worldwide user community. We assume that a key factor responsible for the lack of traction is the requirement of a genetic algorithm (GA), because any GA-based algorithm comes with numerous limitations that cannot be changed (Sivanandam and Deepa, 2008). Similar arguments were also made about training neural networks, but they are now one of the key technologies in the artificial intelligence domain (Soria-Frisch, 2017). So, we have to ask "what made deep learning so successful?" and "can this be applied to Markov Brain technology?"

Deep learning benefited from various technical improvements specific to neural networks and their training (LeCun et al., 2015; Wason, 2018). While those advancements are technology-specific, there were four other improvements that launched DNNs to popularity that could be applied to Markov Brains:

1. Cross-platform Support
2. Educational Resources
3. Large Free Datasets
4. Parallelization

TensorFlow (Abadi et al., 2016) provides the back-end for other end user-friendly Python libraries such as PyTorch (Paszke et al., 2019) and Keras (Chollet et al., 2015). Next, we detail the advancements of these enabling technologies for DNNs in each of the 4 areas above, and how and if MABE for Markov Brains can or has made these advancements.

Cross-Platform Support TensorFlow et al. are available for all major operating systems: Windows, Linux, and OSX. The MABE framework also supports these systems, and various build environments and integrated development environments each system might provide.

Educational Resources The DNN python tooling efforts provide quality educational resources through excellent documentation and numerous free tutorials. In the same venue, but of course on a much smaller scale, MABE provides documentation and user support. The extension of this effort is ongoing (ALIFE Conference 2018 Discussion on common research software, ALIFE Conference 2020 MABE Tutorial).

Large Free Datasets ImageNet (Deng et al., 2009) is only one of many examples how vast data sets for training deep learning have been made accessible. Similarly, websites like `Kaggle.com` even provide deep learning challenges further disseminating deep learning technology. MABE uses genetic algorithms, and in this domain of GAs we find similar endeavors. OpenAI Gym (Brockman et al., 2016) — while primarily a tool for reinforcement learning, provides tasks for neuroevolution. The annual GECCO Humies awards is another competitive event — Human-Competitive Results Produced by Genetic and Evolutionary Computation.

Parallelization GPU computing has been a necessary boon to DNN technology, enabling large-scale parallelization using Single Instruction, Multiple Data processing (Takizawa et al., 2009; Lahabar et al., 2008). The area of parallelization is ripe for Markov Brain advancement, to mitigate limitations of the GA and to capitalize on the digital logic gate intrinsics of the default substrate. The next section will discuss this in more detail.

For Markov Brains, the first three topics are already addressed by the community. This leaves parallelization as a fourth possible option for improvement. GPU computing plays a critical role in training deep neural networks as they can perform the required matrix operations in parallel. While a linear algorithm performing a matrix multiplication of a vector scales with $O(n^\alpha)$ (with $\alpha > 2$) doing the same in parallel reduces this to a linear complexity $O(n\alpha)$ (with $\alpha > 1.0$). It is this complexity reduction that leads to a much faster computation of deep learning algorithms when using GPU or TPU hardware. A Markov Brain uses a state vector, which is comprised of new sensor inputs, hidden states, and motor outputs. A set of computational operations now defines the next state of this vector (see Figure 1). While in most implementations (known to the authors) the operations of this set are computed sequentially, they are still applied in parallel. This is easiest imagined when considering the case where deterministic logic gates are used as computational operators. When two operators write into the same node their input is summed using a logical OR operation. An alternative is to have each computational operator to write into only one node, which allows also for more complex operators, such as they are found in genetic programming (Koza and Poli, 2005) to work in parallel. For that reason, a Markov Brain architecture where only deterministic logic operators each writing into separate nodes is considered here.

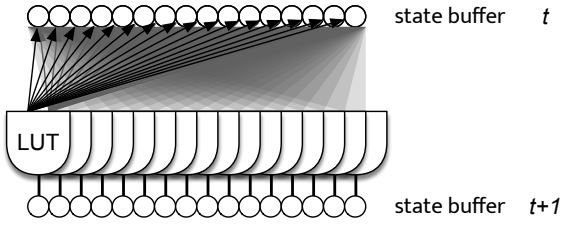

Figure 1: Illustration of a Markov Brain. Logic Gates, here depicted as and LookUp Table (LUT) gates with two inputs, can read from any node (bit) from the state buffer at time point t. Each gate then updates one state of the buffer at time point $t + 1$. For an implementation in an FPGA it is not only crucial to define the logic of each gate, but to also in principle allow each gate to connect to each state.

In this configuration, the computational complexity is constant regardless of the size of the Markov Brain $O = \alpha$. In other words, the time it needs to compute a single update of a Markov Brain — when using a parallel implementation — is theoretically independent of its size. While this might be obvious, there is currently no hardware that allows us to run a Markov Brain in such a way that its computation takes only a single clock cycle. Theoretically, because other computational constraints might limit the efficiency gain due to parallel execution, such as *Amdahl's Law* (Amdahl, 1967; Gustafson, 1988). Those issues can stem from computational overhead required to coordinate parallel execution of code. Regardless, it should thus be possible to take advantage of parallel processing to accelerate Markov Brain evaluation. Field Programmable Gate Arrays (FPGA) seem like the perfect solution. After all, a Markov Brain made from deterministic logic gates is nothing else than a network of logic gates, and so an FPGA should be the ideal solution (see Figure 1, the Markov Brain is already essentially an FPGA). Consequently, here we survey how current FPGA hardware supports building a Markov Brain accelerator, and how contemporary hardware solutions limit this endeavor.

GPU vs. FPGA One obvious alternative to an FPGA might be a GPU, where one could take advantage of its many parallel kernels allowing each one to compute a logic gate

of the Markov Brain. One argument against this approach is the better scalability and power-efficiency of the FPGA (Qasaimeh et al., 2019). Another problem arises from the strict scalar mapping of memory to kernels. Kernels can in principle perform random memory access, meaning for a Markov Brain implementation that the inputs for each gate can be accessed from any kernel. However, the design of a GPU prohibits true random access, and our experience using a GPU in this way shows that throughput is indeed significantly reduced due to cache localization and synchronization . Lastly, using a GPU would not provide the great ratio of transistors to Markov Brain logic gates that an FPGA would provide. For the FPGA a single LUT can be used to implement a single logic gate from the Markov Brain. In the case of the GPU, thousands of logic gates are needed for each of the kernels to compute a single logic function, which is inefficient. However, this allows each kernel to compute any of the other possible more complex functions Markov Brains can take advantage of (Hintze et al., 2019). Regardless, here we will focus on a possible solution using FPGAs.

In the following, the process of implementing an FPGA hardware accelerator parallelizing the execution for Markov Brains will be described and evaluated. However, we do not consider just any solution, but only those that follow the above introduced requirements for open science and accessibility to the three major compute platforms. Furthermore, it is not enough to attain speed of execution, but we must also be able to quickly interface with a GA to allow timely evaluation of candidate solutions. In the following Materials and Methods section, the technological choices will not only be reported, but the criteria that lead to them discussed.

Materials and Methods

General Structure

The typical domain of Markov Brains is agent-based control tasks in virtual environments, even though other applications can easily be imagined. MABE and other software that seeks to optimize Markov Brains using a genetic algorithm, thus needs to implement a perception-action cycle. This schema assumes a separation between world and brain. Input enters the brain via sensors, computation based on this sensor data and hidden states is performed, updating hidden states and creating output signals. Those output signals lead to actions in the environment, which in turn generate new sensor inputs, and so forth.

In a computational evolutionary model the performance of each individual of a population of agent controllers needs to be evaluated. Sometimes even in the presence of each other (Olson et al., 2013). There are numerous strategies to accelerate this process. First of all, if agents do not need to interact, their evaluation can be run in parallel using simple multi-threading. Physics, if needed, can be accelerated using modern physics libraries who take advantage of GPUs or other hardware accelerators. Lastly, in the case where the brain of the agent is, for example, a recurrent neural network again Tensorflow run on a GPU or TPU can be used. Even the selection and mutation of solutions can be accelerated using FPGAs Torquato and Fernandes (2019).

Assuming the above, any hardware accelerator regardless of computational neural substrate needs to communicate three types of data: the network structure (once), inputs, and outputs. While the network structure only needs to be conveyed at the very beginning of evaluation, every iteration the sensor inputs need to be relayed from the host (CPU) to the accelerator device, and the computational results of motor outputs need to be relayed back from the device to the host. There are a variety of proposals in ongoing research to solve the communications aspect of these issues for DNN systems with GPU devices (Espeholt et al., 2020; Lan et al., 2021; Dalton et al., 2020; Petrenko et al., 2021; Makoviychuk et al., 2021). However, these problems apply to all optimization problems of this nature with accelerator devices, so we will focus specifically on the acceleration of Markov Brains and not the communication aspects or how all of the previous technologies might interact (see Figure 2).

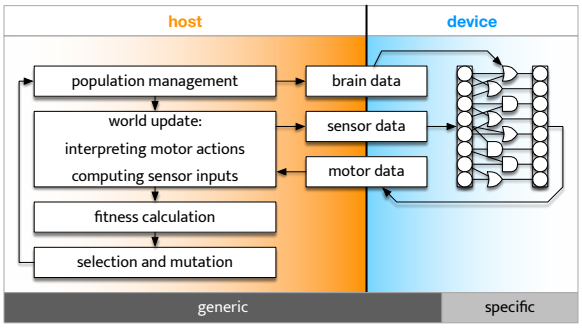

Figure 2: Illustration of the computational evolutionary modeling process. On the left side of the figure all steps that are generic to evolving any computational controller, including the communication part between host and device, and on the right side only the Markov Brain acceleration that is specifically studied here.
.

It is important to mention that all communication aspects can be further accelerated by merging host and device. For example, systems on a Chip (SoC) devices can provide such capabilities — assuming they solve the acceleration of Markov Brains.

Limitation of Existing FPGAs: Operating Systems & Encrypted Configuration

The two largest FPGA manufacturers are AMD (formerly Xilinx) and Intel (formerly Altera), and each provides a plethora of FPGAs together with proprietary design software (Vivado and Quartus) running only on Windows and Linux (Wikipedia, 2022). Neither development software nor

FPGA hardware seems to be produced for OSX on Apple hardware. Both companies fail to meet our requirements because they provide only proprietary development software, and enterprise solutions cost a licensing fee. Furthermore, the proprietary nature of the software makes the bit-stream packages that ultimately program the FPGA effectively encrypted — meaning it is not clear how they actually encode the FPGA configuration, only that the bit stream implements the previously described logic but not exactly how. While this is irrelevant for most industry applications, it matters when it comes to evaluating the actual configuration of the FPGA, as well as in the context of transparent AI (Castelvecchi, 2016).

The above considerations leave Yosys (Wolf et al., 2013) as the only development toolchain to run on all three operating systems. It is also open source, and the translation from Verilog to bitstream is known. This limits the number of FPGAs that can be programmed in this fashion to only a small few, mostly from Lattice Semiconductors. Here, an Icestick 40 HX1K (Evaluation Kit) was used. The Icestudio visual programming environment (icestudio.io) based on the Yosys development toolchain was used to design and program the FPGA.

One cannot avoid wondering "why are both major FPGA manufacturers using proprietary designs, especially if this increases the time it takes to compile the bitstream?" Aside from selling the FPGA hardware, there are recurring licensing fees for enterprise-only features of the development software. Revenue from licensing might be why FPGAs do not find a wider application. At the same time, this model allows companies to sell more than the relatively cheap hardware, especially if this hardware can be reconfigured into any other hardware.

Limitations of Existing FPGAs: Reprogramming Speed

The typical design flow for an FPGA starts with the user defining the desired logic layout. This is done using scripting languages such as Verilog or VHDL (Bailey, 2003), or using even higher level syntax such as C. Alternatively, graphical user interfaces (such as Vivado, Quartus, or Icestudio) allow an easier arrangement of modular components and how they connect to the FPGAs pins, clocks, or other devices such as memory controllers. Then a synthesis step is needed to create a bitstream file which carries the information necessary to configure the FPGA according to the previously defined layout. This bitstream is then transferred to the FPGA effectively reprogramming the device.

This process is a serious limitation when it comes to using FPGAs to accelerate a genetic algorithm. In a typical evolutionary run, a minimum of 100 solutions is tested over many generations. This would mean that the synthesis and reprogramming step, which takes seconds to minutes, is run 10 million times for a single experiment taking 100.000 generations. It is obvious that adding 100 CPU days (assuming around 1 second for synthesis and reprogramming) is preventative costly. The alternative would be to synthesize or optimize the bitstream directly, which for the most modern FPGAs is impossible because this synthesis step is proprietary. This problem is already known in other contexts (Leong et al., 2001; Skliarova and de Brito Ferrari, 2004; Chakraborty et al., 2013). Even if the bitstream is synthesized directly, it also does not remove the reprogramming step.

First generation FPGAs used fuses which were "burnt" during programming, effectively soldering them and allowing only for a single programming step. Then came EPROM and EEPROM-based configuration, where the bitstream is loaded into the EPROM, which in turn defines the FPGA's logic. This is a very tedious and time-consuming process, mostly replaced by SRAM-based configuration (Cofer and Harding, 2006). Here, an external none volatile memory storage like flash memory or an SD-card is used that configures on board SRAM, but technically allows fast reprogramming. While this two-step process is convenient because the bitstream is stored persistently on flash memory and could be used to configure the FPGA, it also prevents this configuration from being used for a GA. Non-volatile flash memory can only be reprogrammed up to a million times: an insufficient quantity for this application. Interfacing with the SRAM directly from the host may present an alternative, but the authors could not find such an FPGA, perhaps because such SRAM is distributed across the FPGA making access inconvenient.

A promising alternative could be partially reprogrammable FPGAs. An early version of this allowed the FPGA IO pins to connect back to its own JTAG programmer (Paulsson et al., 2007). While a creative solution, it seems that it did not find widespread application. A more modern version allowing the FPGA to be reprogrammed from within the FPGA is via an Internal Configuration Access Port (ICAP) (Maxfield, 2015), such as in a Xilinx virtex-4 (Ebrahim et al., 2014). This can be done reasonably quickly (800 Mhz) (Duhem et al., 2011), and also on a SoC (System on a Chip) (Al Kadi et al., 2013), allowing further speed improvements by also moving the GA and possible physics calculations all onto the same chip. In both cases the bitstreams were designed beforehand, and this disallows dynamically generated bitstreams during the application as would be required in a GA. Fortunately, a similar discussion is found in the context of fault tolerance of reconfigurable hardware (Garcia et al., 2006). Here, the idea is to reconfigure (reroute) a section of a chip that experienced a fault — a promising line of research, but such technology is not available yet.

With all the above, we find that a fast and quickly reprogrammable FPGA satisfying our design requirements does not exist, but that there have been small steps to solve some

of the intermediary problems. The best candidate at time of writing is a Zynq 7000 SoC chip, and one could configure it to support Markov Brains. However, this would not allow changing the configuration easily or in a transparent and open fashion. Furthermore, this would create a standalone system requiring a unique set of assembly instructions, instead of interacting with the three major operating systems (Win, *nix, OSX) and their well-developed software tooling.

Ironically, we must conclude that even though FPGAs are theoretically a great solution, practically, a different FPGA design optimized for open and fast reprogrammability would be needed. Consequently, here we use an FPGA to implement such a fast reprogrammable FPGA. In one sense this is incredibly wasteful, because we use universal logic gates to implement universal logic gates on a higher level[1]. However, this trade-off allows us to implement a fast reprogrammable FPGA using existing hardware options.

Reprogrammable Markov Brain implementation

The architecture of this implementation focuses on fast evaluation of Markov Brains together with the necessary code to load the data to configure the Markov Brain, to get input states from the host, and to return outputs to the host. This host-device communication can be implemented using a wide array of channels, such as a network adapter, SPI, PCI, or UART/USB. In a SoC one could even map the memory of the embedded host directly to the memory configuring the Markov Brain. Here, due to the hardware limitations of the IceStick-40 HX1K, the UART is the only choice. Observe, that the communication channel between host and device is not the focus of this investigation and could be separately optimized.

Since communication is in most cases serial, the Markov Brain can be configured while the data for that configuration arrives sequentially. Once the Markov Brain has been configured, only data for inputs needs to be received, and outputs returned. See Figure 3 for the general layout of this implementation, consisting of a Communication Controller (that also programs the Markov Brain), UART for IO, and the reprogrammable universal logic gates for the Markov Brain.

The part that implements the Markov Brain is a set of universal logic gates — equivalent to LookUp Tables in circuit design nomenclature (LUTs). Each of them is created as a block storing its own logic and which of the specific bits from the state buffer each gate needs to read. The outputs of all LUTs are pooled and returned to the communication controller. This implementation allows the configured Markov Brain to perform the computation updating the state buffer from $t \rightarrow t+1$ within a single clock tick. If hidden states should be made persistent, then only the new input states are updated by the Communication Controller. Similarly, only

[1]It also reminds us of Christopher Nolan's 2010 movie Inception

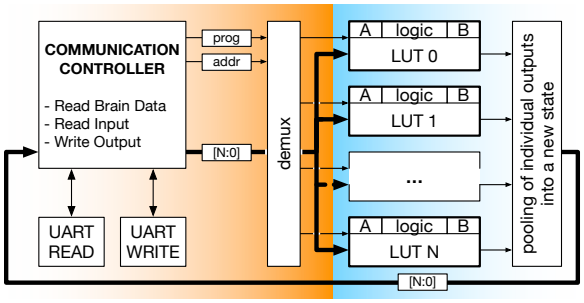

Figure 3: Schematic of the FPGA solution for a fast reprogrammable Markov Brain. In orange all necessary communication and configuration logic, in blue the actual Markov Brain.

.

output states can be communicated back to the host, again reducing the bandwidth for the communication. Here the entire buffer is communicated back and forth for verification purposes.

The current implementation allows Markov Brains with 16 LUTs and consequently a state buffer of 16 binary nodes to be implemented. This implementation also assumes that each node of the state buffer is updated by a single logic gate. If for writing, gates should arbitrarily be connected to states at $t+1$, then their addresses need to be stored as well. Outputs of gates will then not be single bits but a bus as wide as the state buffer. Performing an OR operation on all gate outputs would then be applied. While, without optimizing this process, requiring a serious amount of additional logic gates, it would not slow the computation because only an OR operation is needed. This kind of mapping has not been evaluated here.

Active Categorical Perception Task

An Active Categorical Perception Task (ACP) is one of the most-used tasks when evolving of artificial agents controlled by Markov Brains (Marstaller et al., 2013; Schossau et al., 2015; Hintze et al., 2018). In this task an agent with 4 sensors arranged as pairs of 2 separated by a wide gap (equivalent to 2 sensors) must catch or avoid blocks that are always moving roughly towards it. This "game" is played on a toroidal grid, 16 steps wide and 20 high. Blocks have a size of 2 or 4 and move to the left or right 1 step every time they advance lower. When blocks reach the bottom, the 6 blocks wide agent needs to intercept the 2 wide blocks to catch them, while blocks of size 4 need to be avoided. Agents are evaluated on all 64 combinations of blocks, directions, and possible start locations of the block while starting the agent always at column 0. The toroidal shape of the grid allows the agent leaving to either side to return from the other.

The fitness (see Equation 1 for the agent depends on the

number of properly caught and avoided blocks ('C' for correct outcomes, 'I' for incorrect ones):

$$W = 1.10^{(C-I)}. \quad (1)$$

A standard Genetic Algorithm was used, starting with a randomly generated population of 100 agents. Markov Brains were encoded as a set of 16 binary logic gates, with each logic gate having two input wires and four bits defining the output of all possible computations. When agents were selected by roulette wheel selection to propagate into the next generation each logic gate had a 1% chance to change its wiring and logic. Furthermore, each gate had a 1% chance to randomly copy the wiring and logic of another randomly selected gate, effectively implementing a gene duplication. This evolutionary optimization was carried out in MABE (Bohm et al., 2017).

Results

The accuracy of the implementation was evaluated by using a previously evolved agent with perfect performance on the active categorical perception task. The FPGA performed flawlessly. A python wrapper communicating with the FPGA also allows simple continual reprogramming during an entire evolutionary run. However, the FPGA used here runs at 12Mhz and is so slow that such a run would have taken too long. However, the proof of principle of evaluating a few generations on the FPGA by evaluating each Markov Brain on the ACP task worked flawlessly.

Next, we evaluate the speed of the FPGA. Both UART Rx and Tx (input and output communication between host and device) was run at 2 million baud, that is 6 clock cycles (`clk` in circuit design) per transferred bit. One whole round of communication starts with sending the character '`P`' (for programming) and sets the FPGA into a mode that the next 32 bytes are read and used to configure the 16 LUTs comprising the Markov Brain (see Figure 4 *command* and *bytes sent*). That is followed by sending the character '`T`' (for transfer) and sets the FPGA into a mode to receive 2 more bytes (16 bit) defining the state the Markov Brain should use to compute the next state. Once that is done, the letter '`R`' (for read) is sent setting the FPGA into a mode that sends back the 2 bytes defining the computed $t+1$ state. Sending a single byte, means sending 10 bits, one leading and one trailing 0. Each bit, at that baud rate requires 6 `clk` ticks. Consequently, we expect 32 times 60 `clk` ($1,920$ in total) ticks for that transmission (see Figure 4 *interval* and *measurements*). When measuring, we find this to take $1,921$ `clk` in over 90% of the $1,000$ measurements performed. The extra `clk` is used to clean up the state of the finite state machine of the communication controller. We measure more ticks (up to $4,500$) in the remaining 10% that we attribute to synchronization issues of the UART communication between the host and the device. This is followed by sending a character '`T`' that sets the FPGA into the state to receive two more bytes (16 bits) defining the current state of the Markov Brain. This should take 3 times 20 `clk` ticks (180). As above, this happens in 90% of $1,000$ measurements. Because the Markov Brain updates the $t+1$ buffer at every tick, once the t state buffer is loaded, the update is complete. This update is specified in Verilog as *wires* to not require additional `clk`. To receive the newly updated state, the character '`R`' is sent taking another 60 `clk`. As above, this can be confirmed in 90% of the $1,000$ measurements. Then, the FPGA sends back the updated state. Taking all `clk` together we expect this to take 32+3+1 times 60 ($2,160$) `clk`s. The extra `clk` for cleaning up the finite state machine in this case is executed in parallel to the next state and does not contribute. If the Markov Brain update would take additional `clk`s, we would measure more before the next state is returned. Alternatively, the signal might be returned, but the Markov Brain did not have a chance to complete the update. Since this does not happen, and we indeed find this process to actually take $2,160$ `clk` ticks in 90% of $1,000$ measurements, then we analytically confirm that this implementation allows the entire Markov Brain to be updated in a single `clk`. Any delays, measured in 10% of the cases must be attributed to the communication channel.

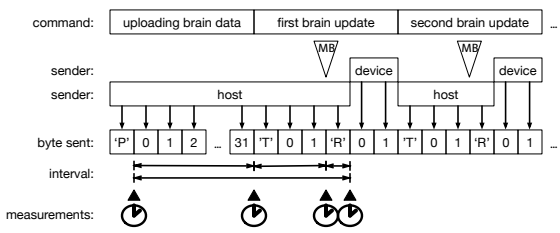

Figure 4: Illustration of the implementation timing. The host sends a '`P`' as the signal to encode the Markov Brain on the FPGA using the next 32 bytes. That is followed by a '`T`' and two more bytes to specify the first state buffer. At the end, the Markov Brain performs the proper update (inverted triangle with 'MB'). Technically, these updates happen every `clk`, but only now are they using the intended data. Typically the host then requests the $t+1$ state by sending an '`R`', which is answered with 2 bytes returning the $t+1$ buffer. After that more data is sent from the host. The clock symbols depict where `clk` counters were reset to time the FPGA. The arrows show the measured intervals.

.

Technically, it is possible that chaining very many logic gates — as it could happen within the LUTs encoding the Markov Brain — leads to signals that need an initial stabilization period. This, and similar phenomena, are summarized under the term "instability" in the FPGA engineering field. This could theoretically lead to faulty signals, and also cause a delay exceeding a single `clk`. FPGAs generally run

slower than modern CPUs thereby mitigating this problem. Additionally, our implementation does not relay the newly computed state back to the host, but waits at least 60 `clk` cycles for a signal from the host to do so. This is more then enough time to account for any instability that may occur. This 60 `clk` delay is of course a constant runtime cost and could be better optimized using different communication channels obviating the need for a delay.

Removing any of the LUTs up to the point where only 1 gate remains did not change the already maximal speed, as we are at the limit fitting on the FPGA used here (900 of the $1,280$ available logic cells). Obviously, removing gates is removing functionality.

Discussion

We showed that an FPGA can be configured such that it uses a single clock cycle (`clk`) to compute the t to $t+1$ update for a Markov Brain using deterministic logic gates. Furthermore, the implementation allows the proper computation of Markov Brains even when integrated into an evolutionary model. However, due to the extremely slow clock of the FPGA (12 Mhz) and the slow UART communication, this implementation does not provide a speed increase compared to the 3.5 Ghz Intel Core i5 host system the device was connected to and compared with.

The first question now is if there would be a benefit by economy of scale, that much larger Markov Brains implemented in this way would still update within a constant single `clk`? The largest currently (2022) available FPGAs have 10 million logic cells (LC), which can all update their outputs within a single `clk`. Speeds of these chips are often lower than that of contemporary CPUs, but speeds of 5Ghz haven been achieved in prototypes already 20 years ago (Clarke, 2001). However, these FPGAs require the conventional toolchain and are not designed to be quickly reprogrammable as defined within this context. Assuming the FPGAs were designed to be quickly reprogrammable — ideally following the design principles laid out here — then they would be able to perform the Markov Brain computation in a single `clk`.

Such reasoning about massively parallel computation in a single `clk` assumes that there is some scaling between the size of the Markov Brain and the required number of logic gates (or transistors needed). So, how does the use of logic cells scale with the size of Markov Brains?

The requirement that needs to be fulfilled here is to allow every logic gate to be able to read from arbitrary locations from the state buffer. Within modern FPGAs the wiring of logic cells to each other is not only proprietary, but also presumably optimized for processing speed, and likely not for reprogrammability. As discussed before, FPGAs are often only reprogrammed during testing and development. The method of implementing this feature, that each logic cell of the Markov Brain can connect to every possible bit in the

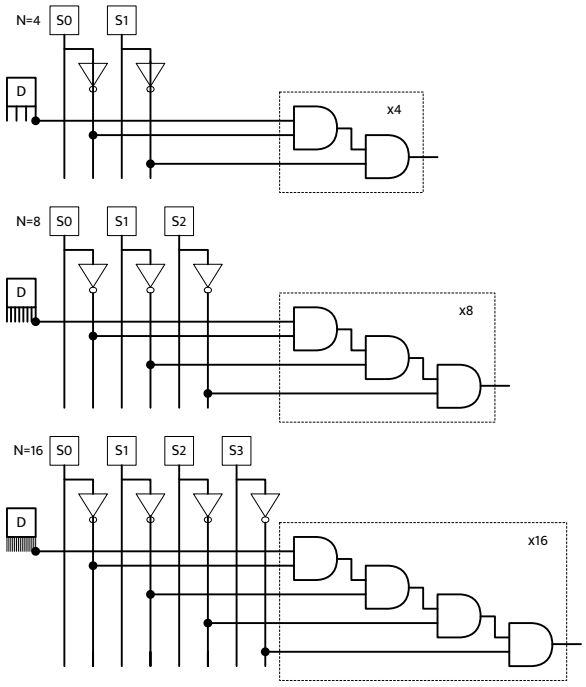

Figure 5: Examples of Demux circuits as they are used here to get the right bit from the current state buffer (D). Depending on the width N of the buffer, $\log_2 N$ signal wires labeled here S_0 to S_3 are needed. For each of the N wires, a set of AND gates is needed shown in the dashed box. The schema is shown for three different sizes of Markov Brains.

.

state buffer, was achieved using a structure similar to a de-multiplexer (see Figure 5). Depending on the size of the state buffer N a set of bits ($S_0 \rightarrow S_{\log_2 N}$)is needed for addressing. These bits specify which bit from the state buffer (D) needs to be selected. The equation to calculate the number of logic gates (L) scales exponentially with the number of bits necessary for addressing:

$$L = S2^S + S. \tag{2}$$

For Markov Brains with $N = 256$ gates, this would result in $500,000$ logic gates needed. While that sounds reasonable, and exceeds what has been used so far, the largest Markov Brain that could be currently supported would be $N = 65,536$, requiring about 70 million logic gates, as much as the largest available CPUs have these days. Observe that under perfect conditions a single CPU performing the same computation would need to retrieve the states from memory, evaluate the logic, and then update the $t+1$ state buffer, each requiring already many `clk`s. Which in serial computation would then be multiplied by N. However, this assumes the above described, and very wasteful use of such a de-multiplexer like arrangement. Fortunately,

there are alternatives for this design, such as transmission gates or combinations and stacking of de-multiplexers. Shift registers could be used for the same purpose, and while they reduce the gate footprint drastically, the require additional `clk`s.

An interesting question is, how FPGAs solve the same problem? After all, are FPGAs not able to connect logic cells arbitrarily? The answer is that FPGAs typically use switch boxes for interconnections, putting a limit on the possible configurations. Configuring these switch boxes is part of the synthesis step of the supporting software. It optimizes the layout of the FPGA. Often gates are not connected in a massive parallel fashion as done here, but in a linear sequence, which hides what is a shortcoming with respect to our requirements.

However, a combination of these methods could be imagined. What remains is that even extremely large Markov Brains can still be accelerated using FPGA technology up to a point. While a different chip design might allow us to to avoid speed/size trade-offs that stem from stereotypical FPGA designs. It seems that the current design, that makes FPGAs great at what they do right now, prevents them from being the perfect Markov Brain accelerator.

A common alternative idea is to use a GPU instead of an FPGA. This approach assumes that one uses each core of the GPU to perform the computation of each computational unit of the Markov Brain. From experience, GPUs work well, when data and processing is aligned. Whereas here, each core must be able to arbitrarily read from a shared state buffer, which drastically slows GPU performance. Secondly, if the Markov Brain is made from deterministic logic gates, a single universal logic gate would be needed to compute this function. Doing the same using a GPU kernel that uses hundreds or thousands of logic gates seems extremely wasteful. However, a GPU kernel could allow the implementation of much more complex computational units. Since GPUs are not optimized for arbitrary random memory access for each kernel in parallel, one could consider a hybrid: Using an FPGA to implement a GPU but with a different data access and pooling architecture, optimized for Markov Brain acceleration — an option we will explore in the future.

Conclusion

Exploring the implementation of a Markov Brain as an FPGA resulted in many interesting insights. First of all one can perform the computation of a Markov Brain made from deterministic logic gates in a single `clk` using an FPGA. However, the fast reprogrammability of such an FPGA was identified as the most important feature when it comes to an actual use within a genetic algorithm. This presents two major shortcomings of current technology: FPGAs are not optimized for the fast and repeated reprogramming that is needed here, and their usually proprietary for-profit development toolchains do not support the spirit of open source or open access. This severely limits the choice of FPGAs and toolchains, and limits a wider application of Markov Brain technology for the major operating systems.

This leaves us with three options: The use of an SoC FPGA, which would allow us to accelerate all computations including the GA and the physics calculation at the same time. However, instead of an accelerator, this would be a "standalone" solution, dependent on current toolchains, limited in the ways described above. It would also not solve the wiring problem discussed above and instead use valuable logic cells to create the inter-connectivity between all gates and the state buffer. Deploying the current solution on a larger FPGA and taking advantage of better host-device communication (for example, a Lattice ECP5 that has an open toolchain) is another viable option. This would also be the simplest way to explore larger Markov Brains, and could answer the question "how long can one maintain a single `clk` update for the entire Markov Brain before speed trade-offs need to be considered?" Lastly, one could explore new FPGA designs that are directly optimized for fast and repeated reprogrammability. They might even only support the Markov Brain paradigm. After all, updating a state buffer using arbitrary logic also allows a user to implement any finite state machine; only limited by the size of the buffer. Instead of running the Markov Brain algorithm on an FPGA, one could run an FPGA on a Markov Brain accelerator — a compelling thought worthy of further exploration.

Acknowledgements

The authors would like to thank David H. Ackley for pointing out that sometimes problems will only be solved by solving them yourself.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283.

Al Kadi, M., Rudolph, P., Gohringer, D., and Hubner, M. (2013). Dynamic and partial reconfiguration of zynq 7000 under linux. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–5. IEEE.

Albantakis, L., Hintze, A., Koch, C., Adami, C., and Tononi, G. (2014). Evolution of integrated causal structures in animats exposed to environments of increasing complexity. *PLoS Comput Biol*, 10(12):e1003966.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485.

Bailey, S. (2003). Comparison of vhdl, verilog and systemverilog. *Available for download from www. model. com*, page 29.

Bohm, C., CG, N., and Hintze, A. (2017). MABE (modular agent based evolver): A framework for digital evolution research. *Proceedings of the European Conference of Artificial Life*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Castelvecchi, D. (2016). Can we open the black box of ai? *Nature News*, 538(7623):20.

Chakraborty, R. S., Saha, I., Palchaudhuri, A., and Naik, G. K. (2013). Hardware trojan insertion by direct modification of fpga configuration bitstream. *IEEE Design & Test*, 30(2):45–54.

Chollet, F. et al. (2015). Keras. `https://github.com/fchollet/keras`.

Clarke, P. (2001). Sige process pushes reconfigurable fpga to 5 ghz.

Cofer, R. and Harding, B. F. (2006). *Rapid System Prototyping with FPGAs: Accelerating the Design Process*. Elsevier.

Dalton, S. et al. (2020). Accelerating reinforcement learning through gpu atari emulation. *Advances in Neural Information Processing Systems*, 33:19773–19782.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.

Duhem, F., Muller, F., and Lorenzini, P. (2011). Farm: Fast reconfiguration manager for reducing reconfiguration time overhead on fpga. In *International Symposium on Applied Reconfigurable Computing*, pages 253–260. Springer.

Ebrahim, A., Arslan, T., and Iturbe, X. (2014). On enhancing the reliability of internal configuration controllers in fpgas. In *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 83–88. IEEE.

Edlund, J. A., Chaumont, N., Hintze, A., Koch, C., Tononi, G., and Adami, C. (2011). Integrated information increases with fitness in the evolution of animats. *PLoS Comput Biol*, 7(10):e1002236.

Espeholt, L., Marinier, R., Stanczyk, P., Wang, K., and Michalski, M. (2020). Seed rl: Scalable and efficient deep-rl with accelerated central inference. In *International Conference on Learning Representations*.

Garcia, P., Compton, K., Schulte, M., Blem, E., and Fu, W. (2006). An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems*, 2006:1–19.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533.

Hintze, A., Edlund, J. A., Olson, R. S., Knoester, D. B., Schossau, J., Albantakis, L., Tehrani-Saleh, A., Kvam, P., Sheneman, L., Goldsby, H., et al. (2017). Markov brains: A technical introduction. *arXiv preprint arXiv:1709.05601*.

Hintze, A., Kirkpatrick, D., and Adami, C. (2018). The structure of evolved representations across different substrates for artificial intelligence. In *Artificial Life Conference Proceedings*, pages 388–395. MIT Press.

Hintze, A., Schossau, J., and Bohm, C. (2019). The evolutionary buffet method. In *Genetic Programming Theory and Practice XVI*, pages 17–36. Springer.

Kirkpatrick, D. and Hintze, A. (2019). The role of ambient noise in the evolution of robust mental representations in cognitive systems. In *to appear in the Proceedings of the Artificial Life Conference 2019*. Cambridge, MA: MIT Press.

Kirkpatrick, D. and Hintze, A. (2020). The evolution of representations in genetic programming trees. In *Genetic Programming Theory and Practice XVII*, pages 121–143. Springer.

Koza, J. R. and Poli, R. (2005). Genetic programming. In *Search methodologies*, pages 127–164. Springer.

Kvam, P., Cesario, J., Schossau, J., Eisthen, H., and Hintze, A. (2015). Computational evolution of decision-making strategies. *arXiv preprint arXiv:1509.05646*.

Lahabar, S., Agrawal, P., and Narayanan, P. (2008). High performance pattern recognition on gpu. *Proceedings of NCVPRIPG*, 2008:154–159.

Lan, T., Srinivasa, S., Wang, H., and Zheng, S. (2021). Warpdrive: Extremely fast end-to-end deep multi-agent reinforcement learning on a gpu. *arXiv preprint arXiv:2108.13976*.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

Leong, P. H. W., Sham, C.-W., Wong, W., Wong, H., Yuen, W. S., and Leong, M.-P. (2001). A bitstream reconfigurable fpga implementation of the wsat algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(1):197–201.

Makoviychuk, V., Wawrzyniak, L., Guo, Y., Lu, M., Storey, K., Macklin, M., Hoeller, D., Rudin, N., Allshire, A., Handa, A., et al. (2021). Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*.

Marstaller, L., Hintze, A., and Adami, C. (2013). The evolution of representation in simple cognitive networks. *Neural computation*, 25(8):2079–2107.

Maxfield, C. (2015). The mcu guy's introduction to fpgas: Configuration techniques & technologies.

Miller, J. F. and Harding, S. L. (2009). Cartesian genetic programming. In *Proceedings of the 11th annual conference companion on genetic and evolutionary computation conference: late breaking papers*, pages 3489–3512.

Olson, R. S., Hintze, A., Dyer, F. C., Knoester, D. B., and Adami, C. (2013). Predator confusion is sufficient to evolve swarming behaviour. *Journal of The Royal Society Interface*, 10(85):20130305.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai,

J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Paulsson, K., Hubner, M., Auer, G., Dreschmann, M., Chen, L., and Becker, J. (2007). Implementation of a virtual internal configuration access port (jcap) for enabling partial self-reconfiguration on xilinx spartan iii fpgas. In *2007 International Conference on Field Programmable Logic and Applications*, pages 351–356. IEEE.

Petrenko, A., Wijmans, E., Shacklett, B., and Koltun, V. (2021). Megaverse: Simulating embodied agents at one million experiences per second. In *International Conference on Machine Learning*, pages 8556–8566. PMLR.

Qasaimeh, M., Denolf, K., Lo, J., Vissers, K., Zambreno, J., and Jones, P. H. (2019). Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8.

Real, E., Liang, C., So, D., and Le, Q. (2020). Automl-zero: Evolving machine learning algorithms from scratch. In *International Conference on Machine Learning*, pages 8007–8019. PMLR.

Schossau, J., Adami, C., and Hintze, A. (2015). Information-theoretic neuro-correlates boost evolution of cognitive systems. *Entropy*, 18(1):6.

Sivanandam, S. and Deepa, S. (2008). Genetic algorithms. In *Introduction to genetic algorithms*, pages 15–37. Springer.

Skliarova, I. and de Brito Ferrari, A. (2004). Reconfigurable hardware sat solvers: A survey of systems. *IEEE Transactions on Computers*, 53(11):1449–1461.

Soria-Frisch, A. (2017). 3 practical thoughts on why deep learning performs so well.

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. (2018). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning.

Takizawa, H., Chida, T., and Kobayashi, H. (2009). Evaluating computational performance of backpropagation learning on graphics hardware. *Electronic Notes in Theoretical Computer Science*, 225:379–389.

Torquato, M. F. and Fernandes, M. A. (2019). High-performance parallel implementation of genetic algorithm on fpga. *Circuits, Systems, and Signal Processing*, 38(9):4014–4039.

Wason, R. (2018). Deep learning: Evolution and expansion. *Cognitive Systems Research*, 52:701–708.

Wikipedia (2022). Field-programmable gate array — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Field-programmable%20gate%20array&oldid=1063351721`. [Online; accessed 06-January-2022].

Wolf, C., Glaser, J., and Kepler, J. (2013). Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*.