A FULLY PIPELINED FPGA ARCHITECTURE FOR STOCHASTIC SIMULATION OF CHEMICAL SYSTEMS

David B. Thomas *

Imperial College London
email: dt10@ic.ac.uk

Hideharu Amano

Keio University
email: hunga@am.ics.keio.ac.jp

ABSTRACT

Simulation of chemical systems allows bio-chemists to understand how the interactions of individual molecules can lead to cellular and organism level behaviour. When the concentration of molecules is very small, it is necessary to model every single chemical interaction in a Monte-Carlo simulation, presenting a huge computational burden. This paper presents a new fully pipelined architecture for chemical simulation, which avoids the traditional approach of optimising for minimum operation count, and instead optimises for throughput and parallelism. We show that even though this leads to a higher asymptotic operation count per simulation step, it allows for a much greater degree of spatial and pipeline parallelism, and the increased area is offset by much greater throughput. The new architecture is implemented in a Virtex-6 SX475T and can sustain a rate of over 1 billion reactions per second for problems with less than 64 reactions. Compared against existing chemical simulators on small to medium size chemical models, the new architecture is 30-100 times faster than a commercial software simulator running on an 8-core 3.4GHz Core i7, and 12-30 times faster than the best existing FPGA simulators.

1. INTRODUCTION

Chemical systems can be modelled as the interaction and transformation of distinct chemical species within a container [1]. For example, we can model a biological cell in terms of basic proteins, and describe the way in which two proteins can combine to create a new protein. Many of these molecular models have been extracted from biological systems, but knowing the low-level protein interactions does not mean we understand what function they perform at a cellular level.

Stochastic simulation is a way of testing and exploring the functionality of these systems, by running a Monte-Carlo simulation which models a set of chemicals within a system, and tracking each reaction within the system according to probabilistic rules. It may require many millions of reactions before a particular regulatory reaction pathway takes effect, so this simulation process can be very slow in software. FPGA simulators have been suggested as a way of accelerating this process, as they are able to provide asymptotically fast simulations rates for large chemical models. However, this required sophisticated data-structures and simulation management algorithms, which limits overall throughput for moderately sized problems.

This paper proposes a new FPGA-optimised simulation architecture, which focusses on throughput for small to medium size models, at the expense of asymptotic performance for large models. Our contributions are:

- A new fully pipelined architecture for stochastic chemical simulation, which combines a simple structure with large amounts of spatial and pipeline parallelism.
- An evaluation of the architecture for multiple model sizes in a large Virtex 6 device, showing that it is a practical approach for models with up to 254 reactions.
- A comparison of the new architecture with existing simulators, showing that a Virtex-6 SX475T is up to 100 times faster than an 8-core software simulator, and is up to 30 times faster than previous best-in-class FPGA simulators in a Virtex-2 Pro.

2. BACKGROUND

A stochastic simulation contains two classes of object: *species*, and *reactions*. The **species** of a simulation are the types of molecules found within the cell, and are associated with some sort of countable instances, for example a type of molecule or protein. A key assumption is that individual instances of a given species are indistinguishable from each other, so we can just consider the total population of the species, and identify the m species using abstract labels $s_1..s_m$.

The behaviour of the simulation is determined by a set of **reactions**, which specify ways in which instances of the species can interact. Each reaction specifies a set of *reaction inputs*, identifying how many instances of each species will be consumed; a set of *reaction outputs*, describing how many instances of each species will be produced by the reaction; and a *reaction constant*, which is a scaling factor describing how "easy" it is for the reaction to occur. For example, the reaction:

$$r_2 : (s_2) \xrightarrow{c_2} (s_4, s_5) \quad (1)$$

describes a reaction labelled r_2 with associated reaction constant c_2, which consumes an instance from species s_2, and

*This research was supported by JSPS grant PE10014.

produces an instance of s_4 and one of s_5 – it essentially describes the fission of a large molecule into two smaller parts. Another example is the reaction:

$$r_4 : (s_3, s_4) \xrightarrow{c_4} (s_4, s_1) \tag{2}$$

which consumes one molecules of species s_3 to produce one molecule of s_1. A molecule of s_4 is both consumed and produced, so the total number of s_4 molecules is unchanged and s_4 is acting as a catalyst.

A central assumption is that the reactions occur within a "well-stirred" medium, so all the molecules from all the species are randomly jumbled up in one container. This allows a combinatorial view of the reactions: as the number of instances of molecule s_1 increases, the likelihood of any reactions which consume molecules of s_1 also increases.

For example, the above reaction r_4 requires one instance of s_3 and one of s_4, so the likelihood of the reaction increases as the product of the number of molecules in each species. If the current counts of molecules in each species are $a_1..a_m$, then the *reaction propensity* of r_4 is:

$$p_4 = c_4 a_3 a_4 \tag{3}$$

The reaction constant c_4 is used to scale the propensity; for example, if a reaction involves two tiny molecules we might use a lower reaction constant than a reaction involving two large molecules, as the large molecules are more likely to hit.

Given the count of molecules $a_1..a_m$, we can calculate the reaction propensities $p_1..p_n$. The relative magnitude of the propensities can be used to determine which reaction is most likely to happen next, and how long it is until that reaction occurs. This means that we can now analyse the system, by defining initial molecule counts at time zero, then moving the system forward in time according to the reaction propensities.

We can simulate the system exactly, reaction by reaction, tracking the changes in molecule counts caused by each reaction. However, because the reaction propensities only give us relative likelihood, we have to implement this as a Monte-Carlo simulation, with reactions chosen randomly according to likelihood. Given the same set of initial species counts, we will find a different set of reactions occur each time we simulate the system, but if we perform enough simulation runs then a picture of the average system can be formed.

The core simulation algorithm is:

1. Determine which reaction occurs next and when using the current species counters.
2. Update the species counters according to the reaction that occurred, and update the current simulation time.
3. If trigger conditions are met, gather statistics about the current simulation state.
4. If termination conditions are not met, go to step 1.

The bulk of the computational work is in steps 1 and step 2, working out what reactions happen, and when.

Given a stochastic system with n reactions and m species there are a number of ways of choosing the next reaction [2]. The Direct Method (DM) uses the reaction propensities directly, choosing the reaction in proportion to the relative propensities in the system, and then determining when the reaction occurs according to the total propensity of the system:

$$p_j^* = \sum_{i=1}^{j} p_i \tag{4}$$

$$\hat{r} = r_j : p_{j-1}^* \leq p_n^* \times U < p_j^*, \qquad \hat{\tau} = E/p_n^* \tag{5}$$

Here U is a uniform random number, and E is an exponential random number, and whenever either appears it represents a fresh random sample. [1] The outputs are $\hat{r}$, the identifier of the next reaction to happen; and $\hat{\tau}$, the time when the reaction occurs. DM works well, but has the disadvantage that it requires $O(n)$ work to calculate the sum of propensities, so it does not scale well.

An alternate mechanism is the First Reaction Method (FRM), which calculates a random waiting time for each reaction, then selects the reaction which occurs first (i.e. with the smallest waiting time):

$$\hat{\tau} = \min_{i=1}^{n} \tau_i, \quad \text{where } \tau_i = E/p_i \tag{6}$$

This method is also $O(n)$, and is computationally slightly worse as it requires n exponentials and divisions, but it does form the basis for a much more efficient method, called the Next Reaction Method (NRM).

The NRM relies on the observation that the propensity of a given reaction only changes when the species count of one of its inputs changes. For example r_4 only depends on two input species s_3 and s_4, so we only need to recalculate p_4 if a_3 or a_4 changes. So if we generate a waiting time τ_i, but end up choosing some earlier event $\hat{\tau}$, then $\tau_i - \hat{\tau}$ can be used for the waiting time until the event after that, *as long as p_i has not changed.*

The NRM method augments the simulator with a dependency function $dep(r)$, which gives the set of reactions whose propensity is affected by a given reaction r, and a vector $t_1..t_n$, containing the next time when each reaction will occur. The NRM simulation process is then:

1. For $t_1..t_n$ set $t_i = E/p_i$, and $\hat{t} = 0$.
2. Find the earliest (minimum) reaction time t_i.
3. Set $\hat{t} = t_i$, and use r_i to update the species counters.
4. For each $r_j \in dep(r_i)$, calculate $t_j = \hat{\tau} + E/p_j$.
5. If trigger conditions are met, gather statistics.
6. If termination conditions are not met, return to Step 2.

The advantage of this method is that, assuming $|dep(r)| << n$, the dominating cost is finding the minimum reaction time in Step 2.

The NRM method is very effective for software simulations of large stochastic systems, but simulation remains a

[1] So $x = E - E$ is the difference of two exponential random numbers, rather than zero.

computationally intensive task. Although the asymptotic time per step is $O(\log n)$, for many contemporary problems we find that n is still small enough that the dominating cost is the $O(|dep(r)|)$ part, as generating each new reaction time is comparatively expensive.

3. EXISTING APPROACHES

FPGAs are an attractive proposition for accelerating these systems, as cheap but accurate methods for generating exponential random numbers allow calculation of many billions of new random times per second, typically an order of magnitude more than a multi-core CPU using SIMD. However, the big problem in both hardware and software algorithms is the ordering and updating of propensities within the system, particularly as the number of reactions and species grows. Existing research into FPGA acceleration has tended to take two approaches: first is to encode the reaction structure of the chemical systems directly in HDL and compile it into the bitfile; second is to create a generic architecture containing reaction time calculation blocks and system update blocks, connected by an arbitrating interconnect.

Compiling the system directly into the FPGA fabric minimises the communication overhead when updating propensities [3, 4], but has the obvious drawback that a full place-and-route cycle will be needed whenever the model specification changes. Given biologists often uses simulators in order to refine models, incurring a multi-hour delay every time the model changes is not acceptable.

Generic architectures which contain a set of reaction time engines and a set of system update engines allow any model to be executed, as long as it does not exceed the number of reactions and species supported in a particular architecture [5]. There is also the potential to share resources, with many relatively cheap system update engines sharing a smaller number of expensive reaction time engines. However, this requires a sophisticated (and complicated) interconnect between the two, and some form of scheduling/arbitration engine. Because the systems are inherently stochastic, it can be difficult to efficiently interleave multiple simulations while ensuring the reaction engines have good utilisation.

4. FULLY PIPELINED SIMULATOR

Our goal is to develop an architecture for stochastic simulation with the following properties:

- **Generic**: a given instance of the architecture should support any chemical system meeting size constraints on the number of species and reactions.
- **Simple**: the architecture should be easy to describe and schedule.
- **Parallel**: the intrinsic parallelism of the FPGA should be exploited at all points in the simulation algorithm, both in terms of fine-grain pipelining and coarse-grain replication.
- **Fast**: simulation speed should be of the order of one simulated reaction per clock cycle, i.e. 100+ MReactions/sec.
- **Exact**: the architecture should follow the chemical master equation (Equation 4) exactly, and not use discrete-time approximations such as tau leaping.

The research hypothesis is that it is possible to redesign the stochastic simulation method into a streaming architecture which meets these goals.

4.1. Managing Reactions

While the FPGA can generate huge numbers of random reaction times, it is the management and updating of the system state (species counts and propensities) which is the most challenging problem. The NRM provides the best asymptotic cost for this update phase, as it is $O(\log n)$ in the number of reactions. In software this is achieved using a heap, which uses an iterative algorithm that may take between 1 and $\lceil \log_2(n) \rceil$ steps. Each step requires memory lookups and has a loop carried dependency, so cannot be performed in parallel. For the same reason, multiple insertions cannot be performed in parallel, as one insertion depends on the sorted list produced by the previous insertion.

The reason for using a heap-based priority queue in the NRM is to provide a data-structure using as few sequential steps as possible. So we can say that in terms of operations the heap insert requires $O(1)$ resources (there is only one processor), $O(\log n)$ operations, and $O(\log n)$ steps. To perform k sequential updates (assuming $k = |dep(r)|$), the costs become $O(1)$, $O(k \log n)$, and $O(k \log n)$. Extracting the head, i.e. the next reaction, is $O(1)$ for all metrics.

As $n \to \infty$ this ensures best-case performance for minimal resource usage, but previous work suggested that for $n < 1000$ (which is where biological researchers currently operate), the high proportional cost of the asymptotic algorithms makes them comparatively slow [5]. In particular, the priority queue implementation became a big sequential bottleneck. Rather than using sequential algorithms, we decided to explore a fully parallel pipelined solution, due to our goal of making each simulation step take roughly one clock cycle.

In hardware we can easily increase the spatial parallelism beyond one, allowing us to achieve different tradeoffs for the number of operations and the number of steps. If instead of trying to maintain an ordering between iterations, we simply store the reaction times as an unordered array, then the insertion time for one element is $O(1)$, as we simply store the time t_i at index i. Additionally, we can store k different reaction times in one step, as each species count goes to a different physical counter. However, to distribute k incoming times to the correct counter asymptotically requires $O(kn)$ operations spatially. We also have to apply the order every time we extract the minimum element. This ordering can be accomplished

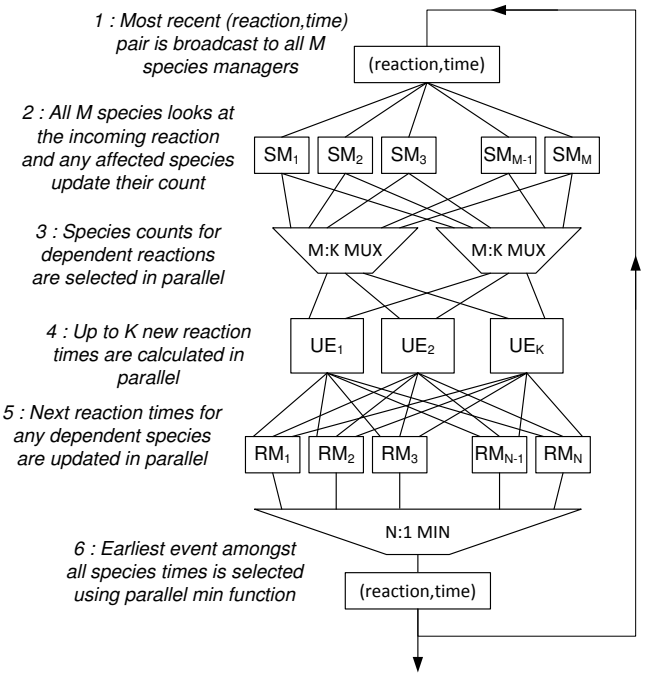


Fig. 1: Spatially parallel simulation method

with a binary tree of pairwise minimum operations, containing $n/2 + n/4 + n/8... = O(n)$ elements. So the spatial and operation costs are both $O(n)$, but the temporal cost, i.e. the cycles per reaction update, is still $O(1)$.

We propose that for a streaming architecture the unordered list approach is actually most appropriate, as it is able to support a throughput of one reaction per cycle. The drawback is that the resource cost may be very high, but for small and medium size problems we will show that this is more than compensated by the increase in speed.

4.2. Streaming Architecture

Having decided on a fully pipelined tree for the determining the next reaction, it is natural to extend the pipeline to encompass one entire simulation step. This leads to our new parallel/spatial algorithm for stochastic simulation shown in Figure 1. The architecture is fully pipelined from top to bottom, with spatial parallelism shown horizontally.

This pipeline structure is naturally very deep, and has a loop carried dependency, so to hide the latency of the pipeline we take a C-Slow approach. This means we have as many independent and concurrent simulations in the pipeline as there are pipeline stages, ensuring full utilisation. Such an approach is very effective for the stochastic simulation problem, as the stochastic nature of the systems means that users must run independent copies of the same model hundreds or thousands of times to discover the average behaviour. Each of the independent simulations progresses through the pipeline according to the following steps:

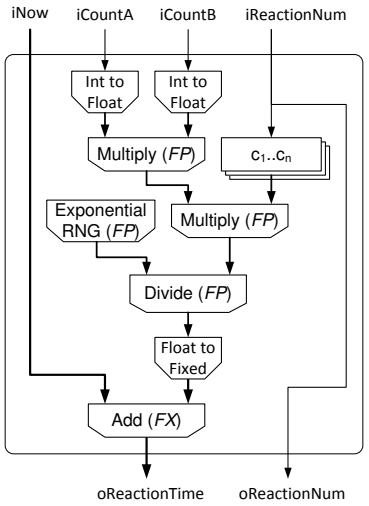


Fig. 2: Reaction time block

Step 1 : At the top of the pipeline a new event enters the system, describing which reaction just occurred, and when it occurred. The reaction id is then immediately fanned out to m Species Managers (SMs). All SMs receive the same pair, so this is essentially just wiring, but in practise it needs to be moderately pipelined due to the high fanout.

Step 2 : Each species manager is responsible for tracking the count of just one species, and for each type of reaction a different subset of species managers will modify their count. For example, if reaction r_4 has just occurred then SM_3 will decrement its count, and SM_1 will increment its count. The change in each species for a given reaction are held in an SM-local RAM, indexed by the incoming reaction number. The managers all update in parallel, and then immediately make the count for all m species available to the next stage.

Step 3 : The modified species counts will cause the propensities for some reactions to change, so their reaction times need to be re-calculated, requiring the selection of the input counts for those reactions. If there are k dependent reactions to be updated in this step, then we need to select k pairs of species counters in order to generate k new times. This selection is performed in parallel, using $2k$ multiplexors which can each select any one of the species counts. These multiplexors are large, but can be efficiently implemented using multilevel trees, and can also be aggressively pipelined. Modern 6-LUT architectures are also able to implement 16:1 multiplexors in just four LUTs, meaning the tree can be quite shallow.

Step 4 : Once we have k pairs of species counts, we can generate the new reaction times using the Update Engines (UEs), which are pipelined implementations of the τ_i part of Equation 6. A more detailed view of the update engines is shown in Figure 2, demonstrating the propensity calculation and random number generation. The new times are generated in k parallel UE instances, so overall the UEs consume a relatively large number of resources, including block-RAMs and

DSPs. The waiting time till the next reaction is calculated in floating-point, but is then converted to a fixed-point absolute time. Note that the fixed-point times are used for correctness, not because they are cheaper. Absolute times should be stored in absolute (fixed-point) precision; a floating-point representation would mean that simulation time had very fine granularity when t is closer to zero, but would become coarser (i.e. less accurate) as the simulation proceeds.

Step 5 : The update engines each generate a new time for one reaction, resulting in k pairs of reaction ids and times. The most recently calculated time for each reaction is held by n Reaction Managers (RMs), which are responsible for looking at all the reactions leaving the update engines, and storing the reaction time if it corresponds to their reaction id. As with the species managers, this happens in parallel, so the area required scales with the number of reactions, while the time required is constant.

Step 6 : Once all the reaction times are updated, the earliest reaction time is then found. Unlike software implementations and previous hardware implementations of the NRM, this is achieved through a parallel tree with a fixed throughput of one result per cycle, rather than an $O(\log n)$ cycle heap. However, the resource usage increases to $O(n)$, compared to $O(1)$ for a heap. Fortunately the comparators and multiplexors required are very efficiently supported in 6-LUT architectures, so this method is viable up to around $n = 500$ on a large FPGA, and is very efficient for $n < 100$.

This architecture allows for one simulation event to be processed on each pass through the pipeline, but the pipeline has very high latency - 60 or more cycles due to the floating-point operations in the update engines. This latency is hidden through the C-Slow approach of having multiple simulations in flight, with each simulation being identified by a unique context identifier flowing through the pipeline.

At points where per-simulation state is stored, such as the species managers and reaction managers, each storage register is expanded into a small LUT-RAM, indexed by the current context. This allows multiple simulations to safely flow though the system without interfering with each other. By managing the degree of pipelining, it is possible to keep the number of contexts to 64 or less, which means only one 64-bit LUT per state bit is needed in modern FPGA architectures.

4.3. Managing Dependencies

As described, the architecture needs to support the worst case k to operate correctly. For example, consider the (nonsensical) system:

$$r_i = \begin{cases} s_i \rightarrow s_{i+1}, & \text{if } i < n \\ s_2 + s_3 + ... + s_n \rightarrow s_1, & \text{if } i = n \end{cases} \quad (7)$$

Here r_n affects every single reaction, while all other reactions only affect three reactions. To achieve one step per cycle would require n parallel update engines, even though for most reactions $(n - 3)$ engines would be idle.

Our solution is to incorporate "virtual" reactions. These do not correspond to "real" reactions in the original system, and have no affect on the species count, but are simply present to allow reactions with larger numbers of dependencies to be handled. The idea is to choose some number k which represents the average number of dependent reactions, and then instantiate k update engines. This allows most reactions to be handled in one cycle.

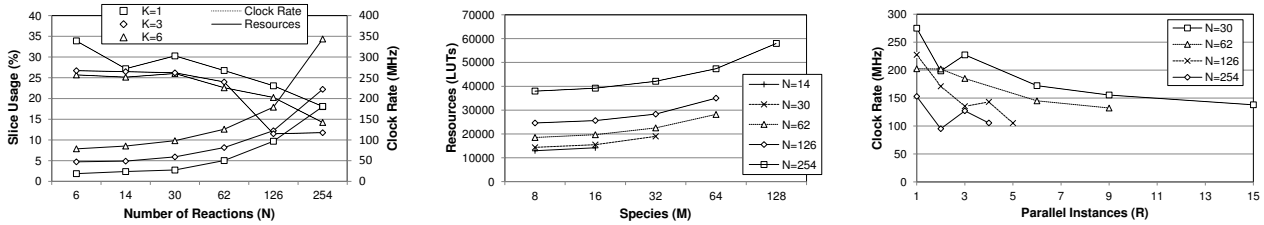
When a reaction has more than k dependents, the processing is split into two or more passes. On the first pass the normal k new times are generated, but rather than allowing processing to continue as normal, a virtual reaction is forced into the system. Virtual reactions have priority over all normal reactions, so in the next pass the virtual reaction will be processed, allowing the remaining dependencies to be handled. If necessary more passes can be added to process very long dependency lists.

5. EVALUATION

The streaming architecture has been implemented in platform independent VHDL, and tested both in simulation and in a Maxeler MaxStation containing a Virtex-6 SX475T FPGA. The implementation is parametrised to support any combination of: real reaction count (N); species count (M); number of update engines (K); absolute time width (WT); and virtual reaction count (VR). Pipeline depths are calculated automatically depending on the top-level parameters. Floating-point operators are all single-precision, derived from Xilinx CoreGen, and the exponential random number generator is from [6].

For smaller N and M each simulator instance will only occupy a small fraction of the FPGA. So as well as the spatial and pipeline parallelism within each instance, the system supports replication (R) of simulation instances within the FPGA. This allows even more simulations to be executed concurrently, and so effectively multiplies overall performance by R. The increased logic utilisation tends to decrease clock rate, but usually the overall gain in throughput is beneficial

Input for the system is managed by a simple register mapped interface, which allows simulation parameters such as reaction rates, reaction inputs, and reaction dependencies, to be loaded into all instances in parallel before simulation starts. The output from simulations is managed using an averaging snapshot: at a set of user-selectable time-points, the state of the species counters is captured and added into an accumulator, with one accumulator per SM. These snapshots can then either be streamed out during simulation, or added up over time to build up a statistical picture of the checkpoints over multiple simulations. Compared to the computational demands, very little IO is required, and is amortised over multiple simulations.



(a) Resource usage in slices and clock rate for increasing number of reactions.

(b) Resource usage in LUTs as number of reactions and species is varied.

(c) Clock rate as more parallel simulator instances are integrated in one FPGA.

Fig. 3: Post place-and-route metrics for Virtex-6 SX475T.

5.1. Area and Performance

The parametrisability of the architecture means that there are many variants to evaluate, and post-synthesis timing estimates are not reliable with large monolithic pipelines: Xilinx XST regularly estimated 400MHz+ performance, when in reality post place-and-route timing rarely exceeded 150MHz. In order to get post place-and-route timing for hundreds of large designs, we used Xilinx ISE 13.4 set to the fastest possible run-time, with no constraints on placement, and no clock constraints. This means the results reported here are accurate, but very conservative - informal manual tweaking of one large instance resulted in the clock rate doubling.

Figure 3a shows the resource usage in slices and post place-and-route clock rate as the number of reactions (N) is increased, with a species count of $M = N/2$. Due to the extensive use of multiplexors and LUT-RAMs in the SMs and RMs, and the large amount of pipelining, logic resources are the main resource bottlenecks, with DSP and block-RAM usage growing at a slower rate. However, the main size limit comes not from resource exhaustion, but from routing congestion, with larger N placing correctly, but failing at the routing stage. The steadily decreasing clock rate reflects this congestion, and in particular with K=3 and K=6 the congestion in the M:K multiplexors (Stage 3) becomes the limiting factor.

As well as the number of reactions, the number of species also significantly affects resource utilisation. Figure 3b varies both N and M, and shows the resource usage in LUTs, as this is the resource most needed to implemented the large multiplexors. The growth in resources with M is approximately linear, while the growth with N is close to linear. The large constant offset mainly represents the overhead due to the update-engines, which is fixed for a given K.

Parallelism is critical for fully exploiting the FPGA, which means as well as the pipeline and intra-simulator spatial parallelism, we must also use coarse-grain spatial parallelism to run multiple simulator instances. Figure 3c tracks the clock-rate as increasing numbers of instances are integrated into one FPGA. For N=30 we can instantiate 15 parallel instances, but as N increases each simulator gets larger, so for N=254 only four instances are possible – often the place-and-route tools refused to route the designs well before available device resources were exceeded.

5.2. Comparison with Existing Implementations

To assess the real-world performance of the system, we compare against the Matlab SimBiology toolbox, which translates each chemical system into C code before compiling it into a specialised executable. Multiple simulations are executed in parallel using the "parfor" command, with each simulation task chosen to be long enough to amortise both compiler overhead and parallelisation overhead. We consider two chemical systems: a linear cyclic colloidal model where N=M to represent very sparse systems; and a Heat Shock Reaction with M=16 and N=23. The linear model is inherently scalable, and to vary the size of HSR we replicate instances, and couple them into a connected ring via the NatP species.

Figure 4 shows the performance for SimBiology running 8 parallel simulations on a 3.4GHz Core i7, and for the pipelined simulator running on a Virtex 6 SX475T. For each value of N the software version is re-compiled, while for hardware we select a pre-compiled hardware configuration, choosing the fastest configuration with K=3 capable of handling problems of that size. For small N the FPGA simulator is extremely fast, achieving in excess of a billion simulated steps per second, as a large number of parallel simulators are supported. As N reaches the maximum of 254, the parallel instances decreases, but performance is always in excess of 100 MSteps/sec.

In comparison, the software produces a relatively constant simulation rate for both models, with only a slow decline. This is because the software simulator is compiled to reflect the model, meaning that the bottleneck at run-time is typically creating new reaction times. Looking at the relative speeds, the hardware is around 70 times faster than software for N$<$63, and at least 20 times faster for all other model sizes.

The most comparable FPGA implementation was developed in the ReSCiP project [5], which used the NRM to implement a system supporting 10-1000 species. Their approach split the calculation into two parts: Thread Private Units (TPUs), which manage the propensity table and an indexed priority queue for one simulation (i.e. one thread); and Thread Shared Units (TSUs), which are responsible for calculating new reaction times using deeply pipelined maths. The idea is to have a small number of TSUs shared between multiple TPUs, as it takes each TPU multiple cycles to update the various data-structures after each reaction.

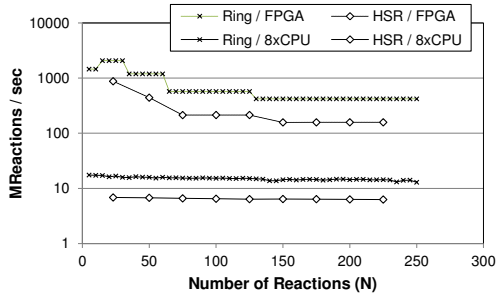


Fig. 4: Performance of linear ring (linear) and Heat Shock Reaction (HSR) models, executing on the pipelined architecture in a Virtex-6 SX475T and compiled using Matlab SimBiology and executing on 8 threads of a 3.4GHz Core i7.

N	Variant	Clock (MHz)	Slices (%)	MSteps per sec	Speedup
16	ReSCiP	77.3	63.2	6.29	1.0
	K=4;R=2	98.9	85.5	197.94	31.4
32	ReSCiP	77.3	63.2	6.22	1.0
	K=4;R=1	99.2	62.8	99.23	15.9
64	ReSCiP	77.3	63.2	6.15	1.0
	K=4;R=1	79.4	92.2	79.44	12.9

Table 1: Comparison of our architecture with the architecture from [5], both implemented in the Virtex-2 Pro xc2vp70.

The underlying platform used was the Virtex-2 Pro xc2vp70, which is of a vastly different generation and scale to the Virtex-6 devices targeted by the current architecture and implementation. For example, the current VHDL implementation assumes (but does not rely on) a 6-LUT architecture, so 4-input multiplexors and 64-bit RAMs are expected to map to one LUT, which is not true in older 4-LUT devices. However, to be fair, we synthesised exactly the same VHDL for the proposed implementation using ISE 9.2i, modifying only the bindings to use older V2Pro compatible floating-point cores.

Table 1 compares the two FPGA implementations in terms of clock rate, area, and throughput, using the D4S system [5]. In [5] it turned out that one architecture provides the best performance, but in our approach different variants are selected depending on the size of the simulated system. For small systems (N=16) we are able to fit two parallel instances (R=2), supporting two reactions per cycle, compared to just one reaction every ten cycles for ReCSiP, leading to an overall speedup of 31x. As N increases the resource overhead of the streaming architecture rapidly increases, and by N=64 we are only able to support a single instance with K=1. However, even at 1 reaction every four cycles, it is still 12x faster than the ReCSiP NRM implementation.

While the streaming approach is faster than ReCSiP, this only applies for small and medium systems - for N>64 the streaming approach simply does not fit, while ReCSiP can support up to N=1024 with little speed degradation. If ReCSiP were ported to a modern architecture, the same tradeoffs would be expected: for problem sizes which will fit, the streaming approach is much faster, but for big problems the ReCSiP TSU/TPU organisation is the only viable solution.

We have also informally compared against best-in-class GPU simulators, but do yet have a like-for-like comparison. For example, the ODM method presented in [7] achieves a throughput of 34 MSteps/sec for a sparse model with N=100 on a GPU, while our FPGA achieves over 300 MSteps/sec, but for other models of a similar size the GPU is ten times faster than the FPGA. We defer this analysis to a future paper, as it would over-simplify things to claim that one or the other was faster.

6. CONCLUSION

This paper has presented a new architecture for stochastic simulation of chemical models in an FPGA. The architecture is designed to be simple and deeply pipelined, and exploits parallelism at multiple levels. A key idea is that it makes sense to trade off sub-optimal asymptotic resource complexity in the algorithm in exchange for high throughput in small to medium size systems. For models containing up to 62 reactions the new architecture is at least 70 times faster than an 8-core software simulator, and is 12 times faster than previous FPGA approaches.

One focus for future work will be to improve the clock rate and resource consumption of the architecture, as the results presented here used low effort place-and-route settings, no floor-planning, and did not attempt to reduce the many congestion problems. A longer term goal is to develop hybrid methods which combine the asymptotic performance of other methods with the high throughput of this method.

7. REFERENCES

[1] D. T. Gillespie, "A general method for numerically simulating the stochastic time evolution of coupled chemical reactions," *Journal of Computational Physics*, vol. 22, pp. 403–434, 1976.

[2] M. A. Gibson and J. Bruck, "Efficient exact stochastic simulation of chemical systems with many species and many channels," *J. Phys. Chem*, vol. 104, pp. 1876–1889, 2000.

[3] J. F. Keane, C. Bradley, and C. Ebeling, "A compiled accelerator for biological cell signaling simulations," in *Proc. FPGA*, 2004.

[4] L. Macchiarulo, "A massively parallel implementation of gillespie algorithm on FPGAs," in *Proc. EMBS*, 2008, pp. 1343–1346.

[5] M. Yoshimi, Y. Iwaoka, Y. Nishikawa, T. Kojima, Y. Osana, A. Funahashi, N. Hiroi, Y. Shibata, N. Iwanaga, H. Yamada, H. Kitano, and H. Amano, "FPGA implementation of a data-driven stochastic biochemical simulator with the next reaction method," in *Proc. FPL*, 2007, pp. 254–259.

[6] D. B. Thomas and W. Luk, "Sampling from the exponential distribution using independent Bernoulli variates," in *Proc. FPL*, 2008, pp. 239–244.

[7] I. Komarov and R. M. D'Souza, "Accelerating the gillespie exact stochastic simulation algorithm using hybrid parallel execution on graphics processing units," *PLoS ONE*, vol. 7, 2012.