GPU-Accelerated Discrete Event Simulations: Towards Industry 4.0 Manufacturing

Moustafa Faheem, Adrian Murphy and Carlos Reaño
Queen's University Belfast, UK
{mfaheem01, a.murphy, c.reano}@qub.ac.uk

2021 IEEE Symposium on Computers and Communications (ISCC) | 978-1-6654-2744-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCC53001.2021.9631514

Abstract— **Discrete Event Simulations (DES) are the most commonplace tools for modelling today's manufacturing factories and their processes. DES are becoming steadfastly integrated into their corresponding physical counterparts to administer greater avenues for their analysis, control, forecasts and optimisations in real-time. However, this growth does not materialise without a penalty in the form of computational burden. The demand for flexible and alternate approaches to accelerate DES is made necessary. Hence, the utilisation of GPUs to comply with such acceleration presents a research topic of growing interest. This work investigates the use of the Machine Learning platform TensorFlow with GPUs to accelerate a variety of manufacturing-domain DES using the SimPy simulation framework. A range of results were gathered, of speed-ups spanning between x1.4 and x3.21, paving the way for further enhancements towards the vision of real-time communication between simulation and physical system in the form of a complete Digital Twin.**

I. INTRODUCTION

Of the ideologies that exist in the realm of High Performance Computing (HPC), the use of Graphics Processing Units (GPUs) to accelerate applications is becoming increasingly prominent in a spectrum of domains, such as biology, finance, modeling and simulation, etc. GPU acceleration can be defined as an application leveraging the high instruction throughput and memory bandwidth capabilities of GPU architecture so that it can execute faster than it does on Central Processing Unit (CPU) architecture [1]. Originally developed for the specific use of graphics rendering, GPUs have since evolved to become general purpose hardware - mainly thanks to the inception of the Compute Unified Device Architecture (CUDA) application programming interface (API) developed by the company NVIDIA in 2007. CUDA allows for high-level programming access to GPUs, greatly simplifying the compatibility of GPU threads to work with different codebases [2]. As a result, the GPU acceleration methodology has superseded the benefits of other acceleration techniques such as Field Programmable Gate Array (FPGA) acceleration, multi-core acceleration and distributed computing. GPUs provide both a higher level of parallelism compared to CPUs and FPGAs, as well as avoid issues related to the complex architecture and expensive hardware of distributed systems [3].

Simulation can be defined as the imitation of a real-world process or system over a period of time. Thus, owing to their general compute-intensive nature, it is of no surprise to find that the utilisation of GPUs to improve the performance of simulations is widely sought after by researchers and industrial institutions. Whether represented physically or by a computer, simulations allow one to draw important inferences concerning the system it represents. Simulations are the most prevalent modelling technique for all various types of processes ranging from medicine, education and environmental studies to manufacturing and military service [4]. This work is concerned with the use of GPUs and computer simulations of manufacturing processes, discussed in detail next, but findings are expected to have relevance beyond this single application domain.

The remainder of this article will be structured as follows. First, Section II establishes the necessary background. Second, Section III will delve into previous work related to the GPU acceleration of manufacturing simulations and note the prominent works of GPU-based DES. Next, Section IV will describe the approach proposed to accelerate SimPy-based DES using GPUs, Section V will explain the experimentation implemented alongside the presentation of tangible results. Section VI will end with a conclusion summarising the main findings, finishing with a note on the future work.

II. BACKGROUND

Manufacturing simulations can be split into three main categories: Discrete Event Simulation (DES), System Dynamics (SD) and Agent-Based Simulation (ABS). Of these three categories, rising firm above the two aforesaid simulations, DES is the most popularly adopted [5]. Hence, the field and category of simulation that forms the focus of this work are manufacturing and DES respectively (the reader here should understand that DES, ABS and SD are not solely utilised in the field of manufacturing and have a broader scope of use).

DESs model a system as an interacting set of entities that evolve through different states as internal or external events happen; when an event occurs, certain changes are triggered in the model and there are no changes in modelling variables between two events. In simpler terms, a system is modelled as a discrete sequence of events in time [6]. Hence, the variation of system complexity that DESs can model is considerably broader compared to SD and ABS (for a more comprehensive view, there are several detailed pieces of literature differentiating between the three simulation techniques [4], [5], [6], [7]. DES allows for a variety of metrics (e.g. system throughput, Work In Progress (WIP), resource utilisation and financial gain), what-if scenarios and future forecasts to be assessed and considered. There exists a large number of case-studies and literature concerning the applications and advantages of DES in industrial settings, relevant examples include: [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19].

Since DES is the most popular simulation technique in manufacturing, the demand of improving its performance in terms of execution time is high. The driving force behind this demand can be attributed to competitive markets that dominate the realm of manufacturing as well as the potential for DES to work in real-time alongside the system it represents, enabling physics-based predictions of future system behaviour (supporting future visions such as production facility *Digital Twins*). One of the methodologies to address speed-up in DES has been the ability to achieve its parallelisation across distributed multi-core computing systems, a feat in which the parallel-DES (PDES) community have constantly proven and optimised in several works. However, the noted issues that distributed computing inevitably brings to DES are that of expense, maintenance and specialised knowledge - such that the procurement of a DES speed-up is often neutralised by these factors [20], [21]. Hence, the prospect of using a commodity GPU to achieve higher DES parallelisation at low-cost and low-maintenance compared to distributed systems is one that has received attraction.

This is where difficulties have been identified between the compatibility of DES and GPU architecture. GPUs usually execute applications in the order they were submitted whereas DESs consist of an asynchronous/discrete time advance mechanism - creating an inharmonious mismatch between the two [1], [22]. These difficulties, however, have not prevented researchers from applying effort to rectify and create novel workarounds/alternatives for them to be minimised. Thus, there have been several frameworks and libraries developed to achieve GPU-applied DESs, resulting in experiments that have proven speed-up is possible. These will be detailed further in the next section. However, it is sufficient here to state that more research is required to maximise the full-potential that GPUs have to offer to DES.

To this end, this work has been set out to explore the use of GPUs for accelerating DESs. In particular, we will assess approaches to extend SimPy[1], a process-based discrete-event simulation framework based on standard Python.

III. RELATED WORK

Concerning current research trends towards the GPU acceleration of manufacturing simulations, it can be stated that ABSs have shown more success than DESs. The main attributable reason for this is that ABSs consist of an intrinsically parallel codebase with non-mutually-exclusive agents - perfectly complementing the architecture of a GPU and, hence, requiring less GPU-porting effort compared to an asynchronous DES codebase. Examples proving this include Li et al. work [23] of porting the Agent Management and Agent Interaction modules of a typical ABS codebase to a GPU for greater simulation efficacy. The results found that the performance of simulations running the GPU-enhanced versions of the modules outperformed the optimised CPU-versions substantially. Moreover, in a different work, Li et al. [24] developed a GPU parallelisation scheme for 3D ABS of In-Stent Restenosis (ISR). The scheme was successfully realised and conveyed significant results in the sense that a typical serial ISR model went from a run-time of greater than fifteen minutes to only thirty-five seconds when parallelised on a GPU. This paves the way for larger-scale ISR models to be developed without the disruptive worry of computing time and proves a reminder of the sheer power that a GPU can accommodate. In another notable work, Saprykin et al. [25] created a simulator by the name of GEMSim to accelerate large-scale agent-based mobility and traffic-forecasting simulations. Here, a large-scale mobility scenario simulating the entire country of Switzerland was executed on both GEMSim and an optimised MATSim (a non-GPU version of GEMSim). The results between the two simulators were compared and conveyed that GEMSim achieved a speed-up of sixty-eight times, requiring a total of less than five minutes to run on a standard workstation with a GPU.

Moving on to the literature related to GPU-based DES, there are several prominent works to note. Tang [26] presents a GPU DES kernel equipped with three algorithms to minimise the incompatibilities between DES and GPU architecture. The three algorithms consist of a breadth-expansion conservative time-window computation to minimise the cost of thread synchronization, a memory management algorithm to store events in a balanced manner and an event re-distribution algorithm to decrease the possibility of branch divergence. The kernel was named gDES and three CPU standard simulations were executed with and without the kernel. In all three cases, their performance with the kernel was higher. Chapuis et al. [22] develop a framework for the PDES engine, Simian, on GPUs. This framework allows modelers to quickly exploit the potential of GPUs for PDES via event handlers that fit the GPU paradigm. A hybrid simulation was tested on the framework and was able to achieve a speed-up compared to the non-GPU Simian engine. However, a limitation of this work is that the framework is not domain specific and the efficiency of its use on different PDESs will vary. Furthermore, Perumalla [27] discusses an alternative DES event-processing data structure, as opposed to the traditional queuing model, to benefit the maximum GPU utilisation. The essence of this data structure is to cast events into a stream-processing paradigm consisting of a stream of events and a stream of logical processes. However, this proposal appears to have not been tested in any tangible simulation thus far. Moreover, Park [2] was able to contribute rather significantly. Park's addition was the building of a CUDA-based library to support DES parallel event scheduling and queuing models on a GPU - a feat that proves it is possible to efficiently harness the power of a GPU on traditional queuing structures. This approach explicitly implements a parallel Future Event List (FEL) for selectively updating and scheduling events on the GPU

[1]https://simpy.readthedocs.io/

and was experimented against sequential heap-based FEL simulations. Although significant speed-ups were achieved using the developed library, the simulation results were slightly inaccurate by approximately 3% numerical error. In simulations with flexibility in output results, this could be an acceptable outcome for the advantage of speed-up.

To conclude the review, this work aims to add to the contributions of GPU-enhanced DES by proving the possibility of accelerating SimPy simulations with TensorFlow-GPU [28]. The forthcoming section will detail the approach proposed.

IV. Approach Proposed

The overall approach of this work was to employ TensorFlow-GPU across various DES SimPy models and assess the impact on simulation performance. For the evaluation of each model's performance, the computation size was varied between small, medium and large; with each computation's execution time being compared among CPU-only, TensorFlow-CPU (TF-CPU) and TensorFlow-GPU (TF-GPU) instances. The TensorFlow instances were applied and activated through the appropriation of the Anaconda distribution environment[2] via Kelvin2 (a HPC cluster at Queens University Belfast[3]). The remaining portion of this section will delve into greater detail regarding TF-GPU and its utilisation to run the SimPy models. Following this, each SimPy model under experimentation - all of which cater flexibility in terms of parameters and characteristics - will be introduced and described.

TensorFlow-GPU (TF-GPU)

TensorFlow (TF) is a software library that uses data-flow graphs to represent computation, shared-state and the operations that mutate shared-state. Although primarily exercised and exploited for Machine Learning operations and algorithms such as Deep Neural Networks, TF supports a wide variety of applications [28]. The basic principle behind TF is to adopt a unified data-flow graph to portray the computation in an algorithm and the state in which the algorithm operates. This includes individual mathematical operations, parameters and their update rules, and the input pre-processing. Edges in the graph carry tensors or multi-dimensional arrays between nodes and transparently inserts the appropriate communication between distributed sub-computations. The advantage of this paradigm, compared to traditional parameter-server designs where the management of shared state is built in to the system, is that the data-flow graph expresses the communication between sub-computations explicitly, thus making it easy to execute independent computations in parallel and to partition computations across multiple devices [28].

The TF platform supports running computations on both CPUs and GPUs, and is principally targeted for the Python programming language. Once a GPU is recognised on the system, TF code will transparently execute on a single GPU with no further changes required. This transparency proved to be an essential trade-off when deciding between TF-GPU and PyCUDA [29] for the experiments to follow. Whilst PyCUDA exposes the entire CUDA C/C++ API to Python and allows for fine-grained optimisations to be fashioned in Python code, a large proportion of the code-base would require to be re-written in C/C++ alongside other additions dealing with data transfer to and from GPU memory. As a result, the portability and transparency that is required to deal with the various SimPy models in question is lacking compared with TF-GPU.

There are three main channels to install TF. The first channel is the installation of TF directly in Python with the 'pip install' command. The second channel is through a virtual environment such as Anaconda and the final is through a container setting such as Docker[4]. Since the Kelvin2 cluster does not grant direct installation in Python and caters minimally for containers, the TF installation and activation avenue taken for this particular work was through Anaconda's virtual environment.

SimPy models

The purpose of this section is to describe the three SimPy models being investigated for their potential acceleration through the route of TF-GPU alongside the significance that this may deliver.

A. Basic BPMN model

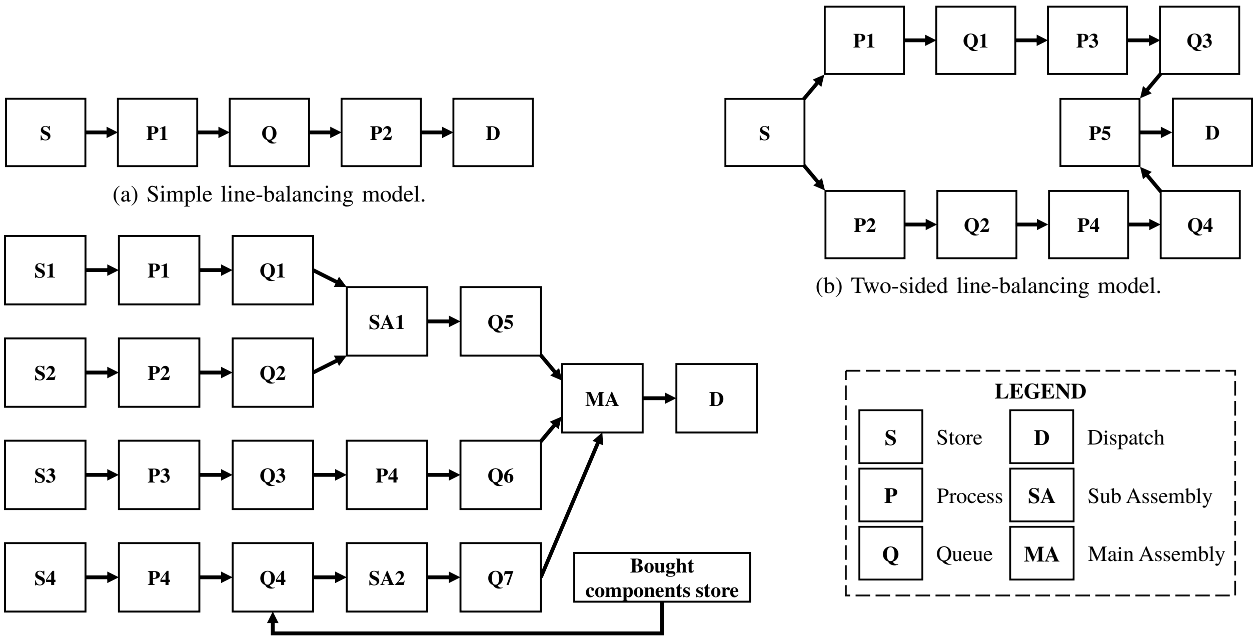
Of the three models under investigation, the basic Business Process Modeling and Notation (BPMN) model [30] is the most simplistic in terms of functionality. Regardless of its straightforward nature, the model represents and illustrates a core behaviour typical in DES SimPy applications. The model is a portrayal of a simple process whereby entities are created in a source that pass a process-step and leave the system via a sink. The main parameters at the source are inter-arrival time and number of entities with the computation size able to be varied by adjusting the latter.

Owing to the fact that this simple model acts as a building block to SimPy models of greater purpose, any significant speed-up gained has the capacity to enhance the speed-up towards its more complex counterparts.

B. Line balancing models of increasing complexity

Succeeding the aforementioned model comes three models of increasing intricacy (simple, two-sided and complex) that describe the same line-balancing problem. Akin to the production logistics model described below, the purpose of the simulation here is to exhibit a traditional line-balancing problem that involves balancing operator and machine time to equal the rate at which products are produced in order to adhere to customer demands. The simple model consists of two processes and one queue, with the two-sided and complex models increasing in number of queues, sub-processes and processes accordingly. The main parameters for the three models are store capacity, queue capacity and employees-per-process. Within each model, the computation size can be elevated by increasing the shop-floor working

[2]https://www.anaconda.com/
[3]https://www.ni-hpc.ac.uk/Kelvin2/
[4]https://www.docker.com/



(a) Simple line-balancing model.

(b) Two-sided line-balancing model.

(c) Complex line-balancing model.

Fig. 1: Structure of the three line-balancing models simulated in the experiments.

time. Figures 1a, 1b and 1c represent graphical abstractions of the model structure and data-flow as the model complexity increases. For emphasis, line-balancing problems are imperative and central to manufacturing factories. A speed-up that can be cultivated in this realm serves towards the goal of real-time interaction between a manufacturing simulation and the physical system that it represents.

C. *Reinforcement learning assembly line supply strategy and planning model*

The most complex model that this work investigates is that of Reinforcement Learning in an assembly line settings[5]. This model incorporates an ever-emerging Machine Learning strategy, Reinforcement Learning [31], with an important aspect of manufacturing factory production-logistics in assembly line supply strategy and planning. Moreover, this model forms the basis of a Digital Twin in the sense that it is able to produce accurate decisions/outputs when provided with real-time inputs. As an overview of the model's functionality, an abstracted version of an assembly line with a corresponding material supply system is developed in the form of an environment whereby a self-learning agent autonomously discovers successful strategies (in terms of maximising throughput and minimising bottleneck) through environment-interaction. In other words, without the provision of a specific solution strategy, an agent learns as it continues to interact with a specific setting. The agent here represents a Tugger[6] and deals with the following system parameters: processing time per station per product, demand per product of station type, Tugger movement speed, Tugger capacity, amount of material (un-)loaded per training step and time required per (un-)loading training step. To reiterate, production-logistics in assembly line frameworks is a fundamental element in manufacturing factory settings. Any speed-up that can be accomplished in this arena will contribute towards real-time integration with their mirrored physical counterparts.

[5]https://fladdimir.github.io/post/tugger-routing/#reinforcement-learning

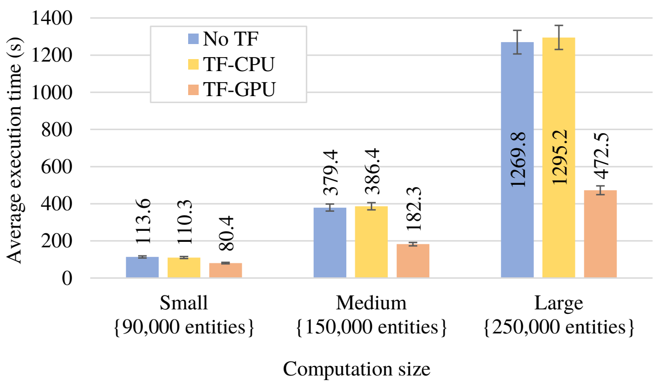
[6]In the context of manufacturing, a Tugger is a machine that pulls, drags and loads materials from one place to another.

V. EXPERIMENTS

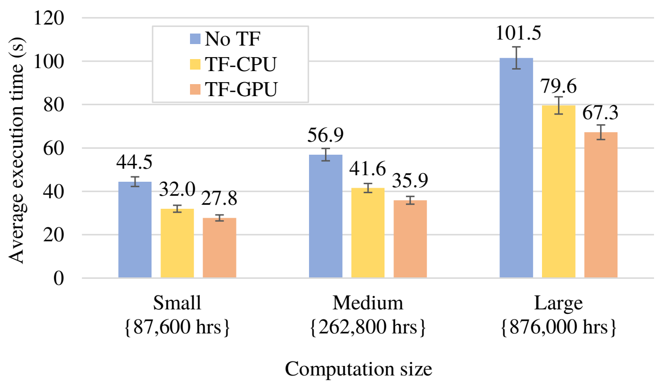
The purpose of this section is to present the outcome of applying TF-GPU to the three SimPy models explained in the previous section. For each SimPy model, TF-GPU was activated using the Anaconda virtual environment via the mechanisms of the Kelvin2 HPC cluster. The GPU version utilised throughout experimentation was an NVIDIA Tesla v100, coupled alongside two 24-core Intel Xeon Platinum 8168 CPUs at 2.70GHz, and 512GB of DDR4 SDRAM memory at 2.66GHz.

Commencing with the basic BPMN model, the target was to assess how the execution time varied amongst increasing computation size for CPU-only (No TF), TF-CPU and TF-GPU exponents. For each exponent, the computation size was raised by adjusting the 'number of entities' variable and keeping the 'inter-arrival time' constant. The smallest computation size consisted of 90,000 entities, with 150,000 and 250,000 entities attributed to medium and large respectively. Each computation size was executed three times and the value shown is the average.

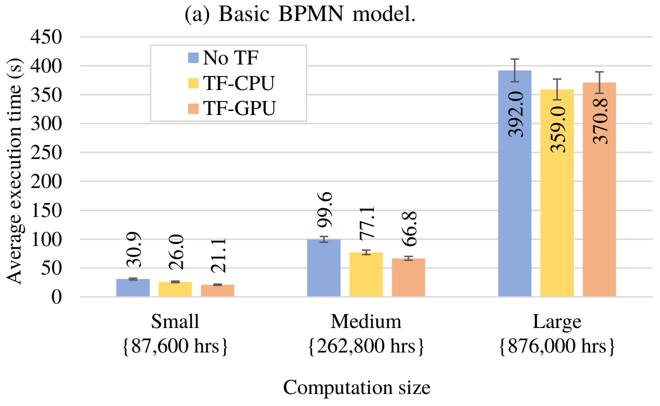
The CPU-only (no TF) instance proportionally ran the model at an average execution time of 1 minute 54 seconds, 6 minutes 19 seconds and 21 minutes 9 seconds for low, medium and high computation size, respectively. Following the activation of TF-CPU, the model performed almost



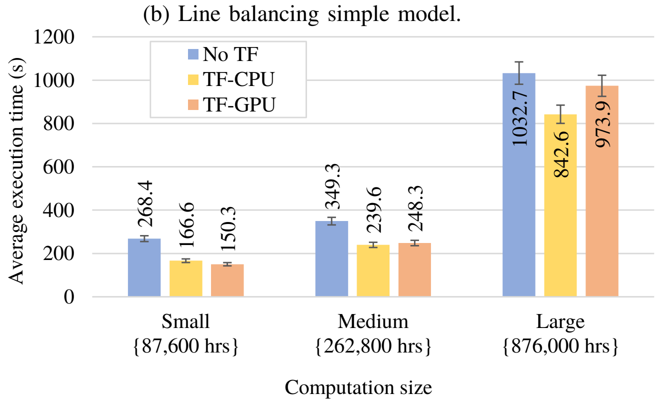
(a) Basic BPMN model.



(b) Line balancing simple model.



(c) Line balancing two-sided model.



(d) Line balancing complex model.

Fig. 2: Average execution time when simulating the different models by varying the computation size without TensorFlow (CPU-Only) and with TensorFlow (CPU and GPU).

equally at 1 minute 50 seconds, 6 minutes 26 seconds and 21 minutes 35 seconds as the computation size increased. However, once TF-GPU was activated and initiated to run the model, the distinctness in the average execution time became increasingly apparent as the computation size increased. The smallest computation size was only slightly more performant compared to TF-CPU, running 30 seconds faster. Whilst it was for the medium and large computation sizes that clear performance gains were achieved. Medium-size acquired a speed-up of x2.11, executing at 3 minutes 2 seconds, and large-size acquired a x2.74 speed-up, completing at 7 minutes 52 seconds. Figure 2a illustrates the model's performance between CPU-only, TF-CPU and TF-GPU.

The outcome of this model's behaviour after applying TF-GPU is that which is typically associated with GPU accelerated applications, in the sense that the performance grows greater as the computation size increases. This behaviour is connected with the fact that there are a larger number of operations being executed in parallel for large computation size as opposed to that of small, minimalising any overhead penalty caused by GPU data transfer.

The next SimPy model under experimentation pertained to a line-balancing model that could be varied in terms of complexity by altering the number of queues, sub-processes and processes. Hence, the model was partitioned into three categories of increasing complexity: simple, two-sided and complex (previously detailed in Figures 1a, 1b and 1c, respectively). For each category, the computation size was determined via the working-time parameter with 87,600, 262,800 and 876,000 hours adhering to small, medium and large computation size respectively. Moreover, each model complexity was executed against the CPU-only, TF-CPU and TF-GPU exponents to examine the developments in average execution time.

Mixed outcomes were gathered. Opening with the simple model, shown in Figure 2b, TF-GPU gained the superior performance ahead of its CPU-only exponent; approximately procuring a x2 speedup. TF-CPU maintained an average execution time lower than CPU-only and higher than TF-GPU. Next, the two-sided model, shown in Figure 2c, conveys the trend of both TF instances performing similarly with each other for all computation sizes whilst the CPU-only instance performs slower. The speed-up is minimal and does not reach greater than x1.4. Furthermore, the complex model, shown in Figure 2d, unexpectedly revealed that, of the three instances, TF-CPU was the greatest in performance; with CPU-only performing the lowest and TF-GPU in the middle. Again, any speed-up gained by TF-CPU was nominal and did not eclipse x1.4. Figures 2b, 2c and 2d depict the trends in average execution time for all three model categories across increasing computation size for the TF-GPU, TF-CPU and CPU-only instances. It is worth noting at this point that the variation in execution time generally increases with problem size owing to the rising non-linear program response to time

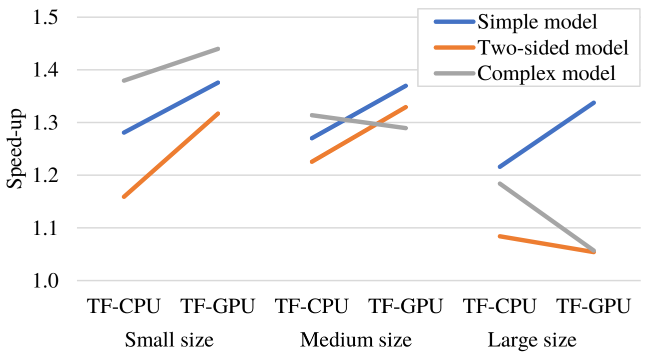


Fig. 3: Summary of TensorFlow (CPU and GPU) speed-up over no TensorFlow (CPU-Only) for all tests.

(illustrated in Figures 2b-2d with the max/min bars).

Figure 3 illustrates how the TF-CPU and TF-GPU acceleration speed-up evolves for all three model complexities as the computation size transforms from small to large. It is explicit to conclude with two main verdicts from the preceding results. Firstly, as the computation size increases with model complexity, the acceleration-ratio is less, and secondly, utilising a TF environment to run the three model categories, whether CPU or GPU, will earn a speed-up. The deviating outcome of TF-CPU outperforming TF-GPU for the complex model is one that requires additional examination and analysis. From the outset, the fact that the execution of the models is limited to a virtual environment articulates the likelihood of overhead playing a role, and thus opening opportunities for supplementary investigation. The conclusions section will seek to clarify as to what these investigations demand and assert the relevance of the speed-ups fulfilled in this work.

Proceeding to the Reinforcement Learning production logistics model, the aim was to evaluate the execution time for TF-CPU and TF-GPU instances only. The fact that the Reinforcement Learning algorithm present in the code is unable to run without TF alleviates the need for the evaluation of a CPU-only instance. Here, the computation size was elevated by growing the number of training steps. Small computation size made up of 10,000 steps, with 100,000 steps for medium and 1 million steps for large. Subsequently, the small-computation attained an average execution time of 2 minutes 24 seconds for TF-CPU compared to 45.38 seconds for TF-GPU, a speed-up of x3.17. Medium-computation surpassed this by procuring a x3.21 speed-up; TF-CPU executed for 25 minutes 18 seconds whilst TF-GPU did so for 7 minutes 53 seconds. However, this upward trend in speed-up was not sustained for large-computation. The average execution time for both TF-CPU and TF-GPU were almost resembling each other with 1 hour 15 minutes 1 second for TF-CPU and 1 hour 9 minutes 25 seconds for TF-GPU - a speed up of x1.08. Two figures represent these trends. Figure 4 shows a graph comparison of average execution time across all three computation sizes and Figure 5 equivalently expresses how the execution time of the most significant function in the code-base, specifically TF's internal function *pywrapTensorFlowInternal.TFSessionRunWrapper*, varied throughout.

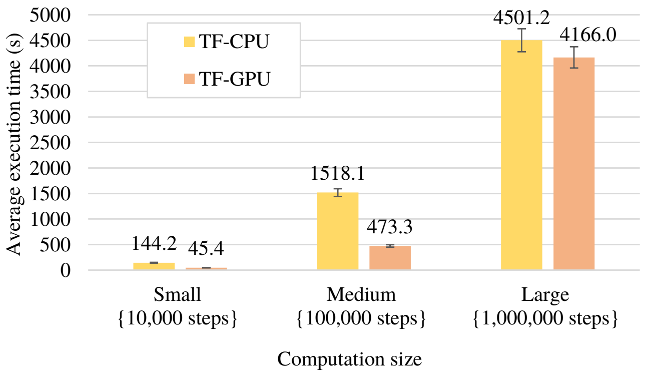


Fig. 4: Average execution time when simulating the Production Logistics model by varying the computation size with TensorFlow (CPU and GPU).

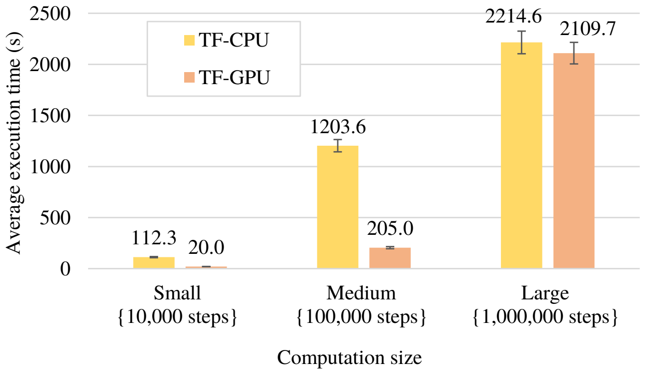


Fig. 5: Average execution time of most significant function when simulating the Production Logistics model by varying the computation size with TensorFlow (CPU and GPU).

Assessing the aforementioned results, it was anticipated for the acceleration to be the greatest with regards to the large-computation setting. In fact, the antithesis occurred. A reason for this could be that there is a higher number of data elements than there are threads in a grid, and hence a one-to-one mapping does not exist. Therefore, granular GPU optimisations such as grid-striding will be deemed necessary to exploit GPU architecture and memory. Another exploration could be the adoption of multi-GPU acceleration. Overall, the results in this context express that the faster the simulation runs, the quicker a high training accuracy can be realised. Hence, a high simulation throughput is attained more rapidly.

VI. Conclusions

To conclude, this work presented an inquisition as to how TensorFlow-GPU affects DES SimPy models in relation to performance. Three SimPy models within the sphere of manufacturing were utilised as case-studies, and the computation-size for each case was varied as TensorFlow-GPU was activated and applied. The purpose was to establish that a GPU-based acceleration is feasible in DES SimPy settings.

The results obtained confirm this, with an acceleration realised for each model and the largest acceleration induced being x3.21. However, perfect acceleration across the computation-sizes of the Reinforcement Learning and Line-balancing models, two of the cases being examined, was not procured. The Reinforcement Learning model conveyed a similar TensorFlow CPU to GPU performance across the largest computation-size whilst the most complex line-balancing model conveyed that TensorFlow-CPU performed better than TensorFlow-GPU. As a result of these anomalies, future work and research is necessitated. Whilst the use of the Kelvin2 cluster provided powerful compute-resource advantages, it also added the limitation of initiating TensorFlow inside Anaconda's virtual environment - hypothetically generating GPU overhead. Therefore, it is of primary interest that future work recognises the extent of this GPU overhead and investigates the potential shift in acceleration when TensorFlow is applied in a non-virtual native mode. Furthermore, accompanying research could also consider methodologies to accomplish greater acceleration. For example, the use of PyCUDA to probe how fine-grained GPU optimisations in Python affect the performance of SimPy models.

In short, it is expected that these contributions function as building-blocks in supporting the wider direction of manufacturing DESs becoming fully-integrated with their physical counterparts in the form of a Digital Twin.

REFERENCES

[1] D. W. Bauer, M. McMahon, and E. H. Page, "An approach for the effective utilization of GP-GPUS in parallel combined simulation," in *2008 Winter Simulation Conference*, 2008, pp. 695–702.

[2] H. Park and P. A. Fishwick, "A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation," *SIMULATION*, vol. 86, no. 10, pp. 613–628, 2010.

[3] S. J. Taylor, "Distributed simulation: state-of-the-art and potential for operational research," *European Journal of Operational Research*, vol. 273, no. 1, pp. 1–19, 2019.

[4] P. J. Sanchez, "Fundamentals of simulation modeling," *IEEE Engineering Management Review*, vol. 37, no. 2, pp. 23–23, 2009.

[5] S. M. Jeon and G. Kim, "A survey of simulation modeling techniques in production planning and control (PPC)," *Production Planning & Control*, vol. 27, no. 5, pp. 360–377, 2016.

[6] J. Stoldt and M. Putz, "Procedure Model for Efficient Simulation Studies which Consider the Flows of Materials and Energy Simultaneously," *Procedia CIRP*, vol. 61, pp. 122–127, 2017, the 24th CIRP Conference on Life Cycle Engineering.

[7] A. P. Galvão Scheidegger, T. Fernandes Pereira, M. L. Moura de Oliveira, A. Banerjee, and J. A. Barra Montevechi, "An introductory guide for hybrid simulation modelers on the primary simulation methods in industrial engineering identified through a systematic review of the literature," *Computers & Industrial Engineering*, vol. 124, pp. 474–492, 2018.

[8] F. Aqlan, S. S. Lam, and S. Ramakrishnan, "An integrated simulation–optimization study for consolidating production lines in a configure-to-order production environment," *International Journal of Production Economics*, vol. 148, pp. 51–61, 2014.

[9] I. Pergher and A. T. de Almeida, "A multi-attribute decision model for setting production planning parameters," *Journal of Manufacturing Systems*, vol. 42, pp. 224–232, 2017.

[10] Y. Zhang, J. Andrews, S. Reed, and M. Karlberg, "Maintenance processes modelling and optimisation," *Reliability Engineering & System Safety*, vol. 168, pp. 150–160, 2017, maintenance Modelling.

[11] S. Velumani and H. Tang, "Operations Status and Bottleneck Analysis and Improvement of a Batch Process Manufacturing Line Using Discrete Event Simulation," *Procedia Manufacturing*, vol. 10, pp. 100–111, 2017, 45th SME North American Manufacturing Research Conference, NAMRC 45, LA, USA.

[12] S. Kumar and D. A. Nottestad, "Capacity design: an application using discrete-event simulation and designed experiments," *IIE Transactions*, vol. 38, no. 9, pp. 729–736, 2006.

[13] D. Koulouriotis, A. Xanthopoulos, and V. Tourassis, "Simulation optimisation of pull control policies for serial manufacturing lines and assembly manufacturing systems using genetic algorithms," *International Journal of Production Research*, vol. 48, no. 10, pp. 2887–2912, 2010.

[14] J. L. Diaz C. and C. Ocampo-Martinez, "Energy efficiency in discrete-manufacturing systems: Insights, trends, and control strategies," *Journal of Manufacturing Systems*, vol. 52, pp. 131–145, 2019.

[15] A. Murphy, C. Taylor, C. Acheson, J. Butterfield, Y. Jin, P. Higgins, R. Collins, and C. Higgins, "Representing financial data streams in digital simulations to support data flow design for a future Digital Twin," *Robotics and Computer-Integrated Manufacturing*, vol. 61, p. 101853, 2020.

[16] L. Mönch, P. Lendermann, L. F. McGinnis, and A. Schirrmann, "A survey of challenges in modelling and decision-making for discrete event logistics systems," *Computers in Industry*, vol. 62, no. 6, pp. 557–567, 2011, special Issue: Grand Challenges for Discrete Event Logistics Systems.

[17] K.-P. Lin, M.-L. Wang, Y. Hong, Y. Yang, and J.-X. Zhou, "Discrete event simulation of long-duration space station operations for rapid evaluation," *Aerospace Science and Technology*, vol. 68, pp. 454–464, 2017.

[18] H. Golzarpoor, V. A. González, M. O'Sullivan, M. Shahbazpour, C. G. Walker, and M. Poshdar, "A non-queue-based paradigm in Discrete-Event-Simulation modelling for construction operations," *Simulation Modelling Practice and Theory*, vol. 77, pp. 49–67, 2017.

[19] P. Georgiadis, "An integrated System Dynamics model for strategic capacity planning in closed-loop recycling networks: A dynamic analysis for the paper industry," *Simulation Modelling Practice and Theory*, vol. 32, pp. 116–137, 2013.

[20] S. J. Taylor, "Distributed simulation: state-of-the-art and potential for operational research," *European Journal of Operational Research*, vol. 273, no. 1, pp. 1–19, 2019.

[21] Q. Liu and G. Wainer, "Multicore acceleration of Discrete Event System Specification systems," *SIMULATION*, vol. 88, no. 7, pp. 801–831, 2012.

[22] G. Chapuis, S. Eidenbenz, N. Santhi, and E. J. Park, "Simian integrated framework for parallel discrete event simulation on GPUS," in *2015 Winter Simulation Conference (WSC)*, 2015, pp. 1127–1138.

[23] X. Li, W. Cai, and S. J. Turner, "Supporting efficient execution of continuous space agent-based simulation on GPU," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 12, pp. 3313–3332, 2016.

[24] S. Li, L. Lei, Y. Hu, Y. He, Y. Sun, and Y. Zhou, "A GPU parallelization scheme for 3D agent-based simulation of in-stent restenosis," in *2019 IEEE International Conference on Cyborg and Bionic Systems (CBS)*, 2019, pp. 322–327.

[25] A. Saprykin, N. Chokani, and R. S. Abhari, "GEMSim: A GPU-accelerated multi-modal mobility simulator for large-scale scenarios," *Simulation Modelling Practice and Theory*, vol. 94, pp. 199–214, 2019.

[26] W. Tang and Y. Yao, "A GPU-based discrete event simulation kernel," *SIMULATION*, vol. 89, no. 11, pp. 1335–1354, 2013.

[27] K. S. Perumalla, "Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs)," *20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06)*, pp. 74–81, 2006.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[29] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[30] S. A. White, "Introduction to BPMN," *Ibm Cooperation*, vol. 2, 2004.

[31] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.