Empowering Data Centers with Computational Storage Drive-Based Deep Learning Inference Functionality to Combat Ransomware

Kurt Friday*, Elias Bou-Harb*, Ron Lee†, Pramod Peethambaran†, Mayank Saxena†
*Louisiana State University
†Samsung Memory Solutions Lab

***Abstract*—The exponential increase in data has significantly burdened data centers with heightened operational demands and challenges in managing, storing, and processing information. Computational Storage Drives (CSDs) mitigate these issues by accelerating processes, reducing energy consumption, and liberating CPU resources for other data center tasks. In light of the notable increase in harnessing the pattern recognition capabilities of Machine Learning (ML) for a variety of data center applications coupled with the aforementioned advantages of leveraging CSDs in data center environments, we build upon the state-of-the-art in CSD development via proposing a unique method for offloading ML inference to CSDs, such as Samsung's SmartSSD. Using various parallelization and optimization techniques, we demonstrate that the proposed CSD-based ML inference method surpasses the inference speed of a high-performance GPU by 344.6×. We further showcase the ability of the presented approach to promptly detect ransomware with high accuracy, precision, recall, and F1 scores, enabling effective and timely mitigation directly within the CSD. Indeed, this ML inference strategy offers the potential to enhance an assortment of other data center tasks.**

***Index Terms*—ransomware detection, deep learning classification, long short-term memory, computational storage, near-data processing, SmartSSD**

I. Introduction

In the evolving landscape of data center technologies, we are witnessing a pivotal shift from cloud-based to localized solutions, driven by increasing demands for data privacy, security, and immediate data processing. Localizing data handling not only enhances privacy and security by retaining sensitive information closer to its origin, thereby mitigating breach risks associated with centralized cloud storage, but also significantly reduces latency critical for real-time processing applications that depend on swift decision-making. This approach alleviates bandwidth limitations and connectivity issues by minimizing the distance data travels, thereby pre-empting the bottlenecks of transmitting data to distant cloud facilities. The rise of CSDs and Smart Network Interface Cards (SmartNICs) exemplifies this transition, marking a broader trend that has expanded from accelerating network-centric tasks such as load balancing [1] and security [2–4] to offloading such acceleration to CSDs and SmartNICs within data centers. This strategic move streamlines data throughput and enhances network management directly at the host level, facilitating a more seamless and powerful infrastructure capable of sophisticated, multi-faceted processing tasks that bolster the entire data center ecosystem.

ML algorithms have also been shown to benefit from the shift towards localized computing in data centers, particularly when offloading ML data preprocessing tasks to CSDs [5–7]. Data movement and I/O cost incurred are known to be substantial bottlenecks for ML applications, and CSDs can mitigate this issue by moving the data processing tasks near the storage itself. CSD-based processing tasks also profit from increased parallelization and an ability to bypass the CPU entirely, which allows for additional acceleration over traditional software rooted in the CPU. In addition, the lower-power processing capability of CSDs, compared to high-performance CPUs and GPUs, also decreases energy consumption under heavy workloads. As a result, CSDs not only have the capacity to reduce operational costs such as hardware procurement and cooling but also potentially extend the lifespan of more expensive components.

In light of the advantages that ML algorithms stand to gain from CSDs, we propose a unique method for offloading the entirety of a classifier to a CSD. In turn, data centers can execute the classifier continuously in the background for time-sensitive applications without exhausting the CPU or consuming inordinate amounts of energy. We also utilized novel parallelization techniques to overcome the parallel processing constraints of CPUs and the data movement bottlenecks of GPUs. Moreover, we employed a number of optimization strategies to further accelerate the inference time of the CSD-based model.

To verify the efficacy of the proposed approach, we performed a thorough evaluation in the Vitis Software Platform Development Environment [8] demonstrating that the approach not only realizes additional reductions in inference time from the aforementioned optimization strategies but that the resultant inference time is orders of magnitude faster than that of a high-performance CPU and GPU. As a use case to substantiate the utilization of the proposed approach in practice, we also showcase its capacity to promptly detect a preeminent threat prevalent in today's cyber attack landscape, namely, ransomware. To this end, a variety of prominent ransomware samples were aggregated and executed in a sandbox environment in order to extract associated Application Programming Interface (API) calls. The CSD-based classifier was then trained on these API calls in order to classify API call sequences associated with ransomware on the system housing the CSD. The classification results gave an accuracy, precision, recall, and F1 score of 0.9833, 0.9789, 0.9890, and 0.9840, respectively. This use case is intuitive, as the proposed approach resides next to the data that it is protecting and therefore can offer real-time mitigation upon detecting the presence of ransomware. We postulate that the proposed approach's appreciable results with ransomware detection can be extended to a variety of other time-sensitive data center applications.

In summary, we highlight the following core contributions of this work:

- Presenting a unique classification methodology entirely within CSDs to enable real-time inference for data center applications. The methodology leverages a sequential classifier capable of handling long-term decencies over time and thereby can generalize to any number of data center tasks.
- Implementing several enhancements to improve resource efficiency, parallel processing capabilities, and the speed of inference. Such enhancements were developed and tested using the Vitis Software Platform Development Environment. Evaluation results reveal that our CSD-centric inference method significantly outperforms a NVIDIA A100 GPU, achieving a speed increase of 344.6×.
- Showcasing the proposed approach's capability of detecting ransomware via evaluating it against a plethora of prominent ransomware samples. The evaluation demonstrates that the approach can identify ransomware with high accuracy, precision, recall, and F1 scores, thereby demonstrating its reliability in combating ransomware in practice.

II. CSD Primer

CSDs mark a significant evolution in data processing technologies by incorporating processing capabilities directly within the storage units, thereby situating computation proximal to the data source. A prominent example of this technology is Samsung's SmartSSD, which integrates conventional SSD components such as the SSD controller and NAND array with advanced elements including a Field Programmable Gate Array (FPGA) accelerator, FPGA DRAM, and a PCIe switch. This configuration is illustrated in Fig. 1, which highlights the architecture of the SmartSSD. The device pairs a 4 TB PM1733 SSD with a Xilinx KU15P Kintex UltraScale FPGA via a PCIe Gen3 x4 bus, optimizing the interaction between the host, storage, and processing elements.

The architecture of the SmartSSD, as portrayed in Fig. 1, enables the CPU to dispatch standard SSD read/write commands along with specialized FPGA computation and FPGA DRAM read/write requests. Additionally, a pivotal feature of the SmartSSD is its support for peer-to-peer (P2P) communication via the FPGA DRAM and the onboard PCIe switch, which facilitates direct data exchanges between its NVMe SSD and FPGA components. This capability drastically reduces PCIe traffic and CPU overhead, thereby lowering latency and enhancing overall system efficiency.

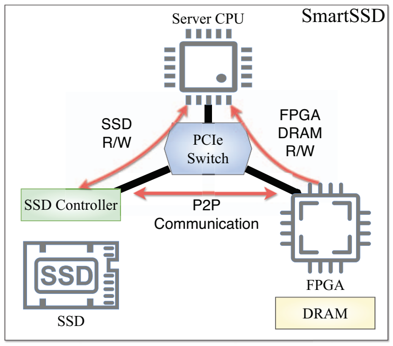

Fig. 1. SmartSSD CSD architecture.

Moreover, the SmartSSD is accompanied by a comprehensive development toolkit that includes a runtime library, an Application Programming Interface (API), a compiler, and necessary drivers for FPGA programming. This suite of tools enables developers to easily implement FPGA-based designs, further augmenting the computational potential of the SmartSSD. By offering these advanced features, the SmartSSD represents a scalable solution that overcomes traditional constraints related to space, power, and cost, allowing for the installation of multiple devices within a single node and significantly advancing near-storage computation capabilities.

The proposed ML classification model is fully integrated within the FPGA, allowing it to process input data directly from the SSD without necessitating CPU involvement. This arrangement optimally utilizes the unique capabilities of the SmartSSD, freeing the CPU to address other critical tasks within the data center, thereby streamlining operations and enhancing the overall performance and efficiency of the system.

III. Proposed Approach

In this section, we detail the DL model architecture harnessed by the proposed approach, and how it was optimized for deployment on the FPGAs within CSDs, such as Samsung's SmartSSD.

A. CSD-Based DL

Model selection. An LSTM model, a variant of Recurrent Neural Networks (RNNs) renowned for their capacity to capture long-term dependencies, is optimally suited for deployment on the FPGA within a CSD for tasks involving sequence prediction. In contrast to non-sequential models (i.e., those that do not process data in a time-dependent sequence) might only analyze static snapshots of data, LSTMs excel in handling time-series data, embodying long-term dependencies and temporal dynamics. Consequently, LSTMs are more adept at understanding patterns over time in order to provide more accurate predictions and insights for dynamic data center environments. To this extent, the proposed approach herein utilizes an LSTM model.

This selection is also grounded in the LSTM's robust track record across a spectrum of deep learning applications and its intrinsic compatibility with the reconfigurable architecture of FPGAs, which excel in managing both sequential and parallel processing demands efficiently. Unlike GPUs, which are predominantly oriented towards parallel processing tasks and may struggle with the sequential processing requirements of LSTMs, FPGAs provide a versatile platform that excels in handling the complex computational patterns of LSTMs by effectively balancing their sequential and parallel processing needs. This efficiency is further enhanced by the LSTM's architectural design, which employs a fixed set of cell parameters across all timesteps. This consistency allows for repeated use of the same computational logic, minimizing resource allocation on the FPGA and enabling more efficient data processing. Such a design aligns well with FPGAs, as they can be programmed to specifically optimize these recurrent operations, thereby significantly improving the model's execution efficiency and reducing the time complexity inherent in sequential tasks.

LSTM inner workings. An LSTM functions by processing data in a sequence and updating its state vector at each timestep, which encapsulates the historical information of all prior inputs. This aim is achieved through a sophisticated system of gates: the input gate i_t, forget gate f_t, and output gate o_t. Such gates are mathematically expressed as follows:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i),$$
$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f),$$
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o),$$

with σ denoting the sigmoid function, W the weight matrices, b the bias terms, h_{t-1} the previous hidden state, and x_t the current input. An LSTM also includes a process for cell modulation, represented as $C'_t = tanh(W_{C'}[h_{t-1}, x_t] + b_{C'})$, which generates a vector of new candidate values. These values are modulated by i_t and then combined with the old cell state C_{t-1} (modulated by f_t) to update the cell state. The cell state updates as $C_t = f_t * Ct - 1 + i_t * C'_t)$ in order to regulate the memory of the network by deciding which information to discard and which to retain. After the cell state is updated, the hidden state for the current timestep h_t is obtained by applying o_t via the equation $h_t = o_t * tanh(C_t)$. This hidden state h_t is then passed to the next timestep and ultimately used for predictions. This gated control of information flow also allows LSTMs to bridge long time intervals by mitigating the vanishing gradient problem inherent in traditional RNNs.

Porting the model to hardware. The LSTM model that will be deployed on the FPGA is first trained offline. It consumes a CSV dataset consisting n + 1 columns and N rows for sequences of n items plus a label and N samples, respectively. In general, it is also ideal to incorporate an embedding generation step for each item in a given sequence before passing it to the LSTM. Such embeddings enable the model to understand complex relationships and patterns within the data by representing it in a more informative, lower-dimensional space [9]. Once the embeddings and LSTM have been trained until convergence, the associated weights and biases are extracted and written to a text file. For example, TensorFlow allows one to extract parameters via the `get_weights()` function, which returns three Numpy arrays consisting of the weights W for x_t, the W for h_{t-1}, and the related b terms, respectively. For additional insight, we kindly refer interested readers to a notebook where we have implemented this procedure [10].

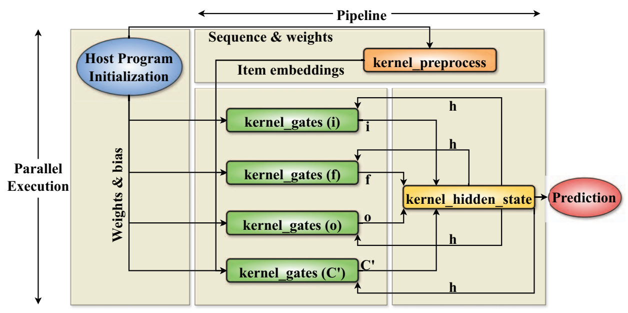

Fig. 2. Kernel implementation strategy for LSTM inference on CSDs.

Lastly, the host program that is responsible for general control flow, initiating data transfers, and managing the interaction with the FPGA ingests this text file amid initializing the FPGA. That being said, the structure of the proposed FPGA-based LSTM implementation described in subsequent subsections is constructed in such a manner that it remains fixed regardless of changes in the number of parameters or embeddings trained in the offline model and the length of the sequences that the offline model consumes. Consequently, the FPGA-based model is compiled once and can be updated at the operator's discretion. For instance, in the event that the proposed approach is leveraged for prompt ransomware detection and mitigation, it is advisable to update the FPGA-based model with a version that has been retrained on new ransomware strains once they are uncovered in Cyber Threat Intelligence (CTI) feeds.

B. Kernel implementations

In an FPGA, kernels are specialized blocks of code designed to operate on a stream of inputs to produce outputs. These kernels aim to accelerate computational tasks by efficiently executing multiple operations in parallel, which can substantially reduce the time required to process large volumes of data. This approach contrasts with traditional CPUs that, based on the classic Von Neumann architecture, typically execute instructions one after another in a sequential manner, often limiting their speed in data-intensive tasks.

To this end, we segregate the previously discussed LSTM inner workings into five well-defined kernels. These kernels and there overall role within the forward pass of the LSTM are portrayed in Fig. 2. The first kernel, `kernel_preprocess`, consumes a fully-formed data sequence (i.e., a sequence that has reached a pre-established number of items). For each item in the sequence, `kernel_preprocess` preprocesses the item by generating its corresponding embedding based off the weights from the offline training procedure. To achieve this aim, `kernel_preprocess` is initialized with a 1-dimensional buffer consisting of the flattened embedding vector consisting of parameters $p \in \mathbb{R}^{M \times O}$, where M is the magnitude of the set of all items in the N sequences and O is the embedding size. The embedding for the current item being processed in the given sequence by `kernel_preprocess` is obtained by taking the dot product of the one-hot vector of the item and the $M \times O$ matrix.

The next kernel is `kernel_gates`, which is tasked with generating the input gate i_t, forget gate f_t, and output gate o_t. `kernel_gates` takes the embeddings (i.e., x_t in the LSTM equations) generated by `kernel_preprocess` as input along with the previous iteration's h_{t-1} (i.e., the LSTM's output for the previous item in the sequence). `kernel_gates` also receives the weights for both the embeddings and h_{t-1}. Lastly, `kernel_gates` applies the sigmoid activation function to the result of the dot product and sum calculation.

The last kernel is `kernel_hidden_state`. As observed in the LSTM inner workings subsection, h_t is dependent upon C_t, and therefore `kernel_hidden_state` is used to generate both. Moreover, taking this approach allows us to maintain C_t entirely within `kernel_hidden_state`, in contrast to contending with the additional overhead associated with passing C_t to another kernel. Ultimately, `kernel_hidden_state` receives the inputs of i_t, f_t, o_t, and C'_t, and outputs h_t. Additionally, this kernel also receives the weights and bias for a fully connected layer in order to map h_t to a classification once all items in the sequence have been processed. `kernel_hidden_state` maintains a `static` counter in order to determine when the entirety of the sequence has been processed.

C. Parallelization

Note that the aforementioned kernel implementation was devised to support a balance between parallelization while reducing pressure on AXI Master interfaces used for high-performance, memory-mapped communications between the kernels and the FPGA's memory resources. These resources are not embedded within the FPGA fabric itself but are instead part of the global memory resources available to the FPGA. This often includes DDR SDRAM but can also refer to other types of accessible memory, depending on the architecture of the FPGA and its surrounding system. The proposed approach utilizes a conservative two DDR banks of global memory. For comparison, some Alveo cards (e.g., the u200 and u250) support four banks.

As shown in Fig. 2, our kernel implementation enforces parallelization between four `kernel_gates` Compute Units (CUs). To ensure that these four CUs truly run in parallel, `kernel_preprocess` creates four copies of the embedding of the given item in the sequence (i.e., x_t), and `kernel_hidden_state` performs the same copy operation with h_{t-1} such that each CU has its own copies. Note that streaming can be easily ported to the kernel implementation for additional acceleration if the FPGA supports it. The High-Level Synthesis (HLS) pragma `#pragma HLS DATAFLOW` was also employed in `kernel_gates` to promote added parallelization between independent operations within the CUs. Lastly, while an item in the the sequence is being processed by the `kernel_gates` CUs and `kernel_hidden_state`, `kernel_preprocess` preemptively processes the next item in the sequence to generate its embeddings in parallel so the embeddings can be consumed by the `kernel_gates` CUs when available.

D. Optimizations

Activation functions. As previously noted, LSTMs leverage sigmoid and tanh activations. While the sigmoid activation can be implemented with relative ease, the tanh function defined as follows, possesses complexities that can be taxing on FPGAs:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

To mitigate this issue, we replace all tanh functions with softsign activations, as subsequently defined:

$$softsign(x) = \frac{x}{abs(x) + 1}.$$

This is a sufficient replacement for the tanh function in LSTMs due to its similar S-shaped curve and asymptotic behavior, which offers a smooth and continuous gradient flow in order to mitigate vanishing gradient problems. Simultaneously, it provides computational efficiency by avoiding the exp() operation, thereby making it a practical choice for maintaining performance with lower computational costs.

Initiation Interval. The Initiation Interval (II) quantifies the number of clock cycles required between consecutive initiations of operations in a pipelined architecture. Consequently, it determines the throughput of the pipeline by specifying the minimum temporal separation at which the pipeline can accept new inputs or begin a new iteration of a loop. For instance, an II of 1 implies maximal pipeline utilization, where a new operation or loop iteration commences in every clock cycle, thereby achieving the highest possible data processing rate. Ultimately, II depends on various factors including the nature of the loop operations, data dependencies, and available hardware resources. As such, a sound technique to minimizing II is to employ `#pragma HLS PIPELINE II=1` prior the execution of loops in order to prompt the compiler to place added emphasis on II minimization. Additionally, we unroll loops to a degree via `#pragma HLS UNROLL` to help reduce II, as it can increase parallelism and reduce both data dependencies and loop overhead. In a similar vein, we also partition kernel buffers using `#pragma HLS ARRAY PARTITION complete` to promote the processing of indexed buffer items in parallel in order to further aid in II reduction.

Fixed-point arithmetic. Fixed-point arithmetic scales floating-point numbers by a factor, converting them to integers to perform operations more efficiently than floating-point calculations. This approach is advantageous for FPGA Digital Signal Processing (DSP) slices optimized for tasks like multiplication, accumulation, and filtering, crucial for deep learning inference involving extensive matrix multiplications.

Efficient DSP utilization also reduces FPGA Look-Up Table (LUT) consumption.

Since the vast majority of the floating points numbers used in the proposed approach are small numbers, we employ a scaling factor of 10^6, which places more emphasis on maintaining the mantissa of the floating point representation. We multiply the floating-point values of weights, biases, and embeddings by this factor before the host initialization shown in Fig. 2, converting them to integers while preserving significant digits. To minimize errors from finite precision, we round the results to closely match the original numbers. Moreover, after each multiplication, the product scales by 10^{12}, which requires a correction by dividing by the scaling factor squared (10^{12}) to maintain accurate final values with minimal errors for subsequent arithmetic operations in the FPGA.

IV. Evaluation

In this section, we conduct a comprehensive evaluation of our proposed methodology to assess both its efficiency and effectiveness. Specifically, we measure the execution times of our approach and compare them with that of high-performance CPUs and GPUs. Moreover, we illustrate the proficiency of our method in performing accurate CSD-based ransomware detection.

Testing environment. Our FPGA experiments were carried out using the Vitis Software Platform Development Environment [8], which enables complete FPGA hardware emulation and provides dependable performance metrics. These experiments were performed on a server running Ubuntu 20.04.6, equipped with an Intel Xeon Silver 4114 processor and 32 GB of RAM. All necessary code for the host and kernels was developed in C++ and made use of Xilinx Runtime (XRT) and HLS, respectively. We selected the Alveo u200 [11] as our primary experimental platform, which is part of the UltraScale family and similar to the SmartSSD's Kintex KU15P. Compilation of the host application was executed using `g++`, the standard GNU C compiler, while `v++` was utilized to compile the kernel objects into .xo files and to link these objects with the target platform when generating the FPGA binary (i.e., the .xclbin file). The experimental setup involved an LSTM model with an embedding dimension of 8, a hidden layer size of 32, and bias terms, which collectively accounted for 7,472 parameters (2,224 for the embeddings and 5,248 for the LSTM). The concluding fully-connected layer incorporated 32 weights and one bias term.

Execution time. We begin this evaluation by taking the execution time of the LSTM's forward pass for an individual item in a sequence using the Vitis Software Platform Development Environment's hardware emulation mode. Such mode is designed to provide an accurate estimate of how long the FPGA would take to execute the given program in real hardware. The execution times are given for a vanilla FPGA-based LSTM that leverages the parallel kernels and no other optimizations, and then incrementally adding the optimizations of II minimization and fixed-point arithmetic to better observe the acceleration enhancements realized by each.

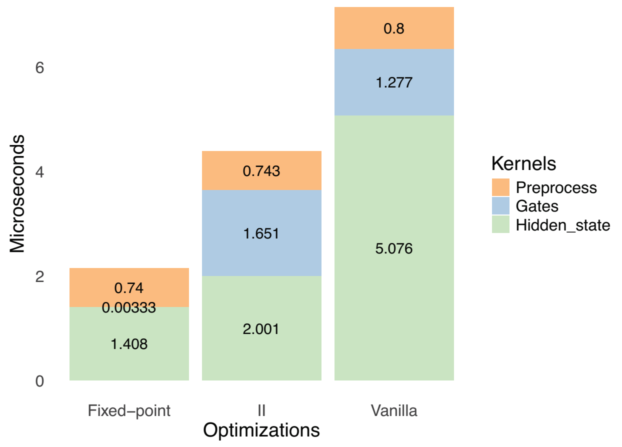

Fig. 3. FPGA-based LSTM inference time reductions realized through optimizations. The Vanilla implementation consists only of the kernel parallelization detailed in Subsection III-C. Subsequently, the Initiation Interval (II) optimizations are added to the Vanilla approach. Lastly, the mean is taken when utilizing fixed-point arithmetic and II (shown on the left). Incrementally integrating optimizations in this manner allows for visualizing both the reduction in inference time that each optimization offered and the final minimized inference time via all optimizations (i.e., the Fixed-point measurements).

The results are portrayed in Fig. 3 in microseconds. As can be observed, II minimization reduced the execution time of `kernel_hidden_state` by a relatively wide margin, and the leveraging of fixed-point arithmetic dramatically decreased the execution time of `kernel_gates`. Notably, the execution time of `kernel_preprocess` remained fairly fixed. Ultimately, with both optimizations, the total execution time of approximately 7.153 μs was decreased to roughly 2.15133 μs per forward pass of the FPGA-based LSTM. In addition, recall that the four CUs of the `kernel_gates` run in parallel, so the execution time of the gate operations is equivalent to the maximum execution time of each of the four CUs, which is given in Fig. 3.

Ransomware detection use case. To showcase the capability of the proposed approach to be harnessed in practice, we apply it to among the most concerning cybersecurity threats today, namely, ransomware. The advantages of pairing a ransomware defense mechanism with the storage itself are clear, as such a defense would allow near-instantaneous mitigation. Further, such a ransomware defense could be seamlessly paired with existing network-based detection mechanisms [12, 13]. To this extent, we feed API call sequences into the proposed FPGA-based LSTM, as API call sequences have proven to be a reliable method for identifying compromises. In order to derive an appreciable API call dataset comprising ransomware infections, we aggregated a number of prominent ransomware samples and utilized Cuckoo Sandbox [14] to generate associated API call sequences on Windows systems. We also collected an assortment of benign API call sequences to incorporate into the dataset as well. The dataset consisted of 29K sequences, of which 46% resulted from ransomware.

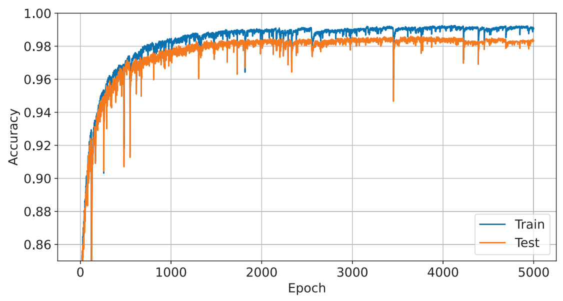

Fig. 4. Convergence of the LSTM training on ransomware API call sequences.

We make this dataset publicly available to promote future ransomware defenses and research [10]. Additional details pertaining to the dataset aggregation are covered in Appendix A. We then trained an LSTM model comprising a total of 7,472 parameters until convergence, as shown in Fig. 4. As can be observed, the model reached its peak detection accuracy of the testing dataset of 0.9833 at around 4K epochs. At this juncture, the model also gave a precision, recall, and F1 score of 0.9789, 0.9890, and 0.9840, respectively. Indeed, these results demonstrate that the proposed LSTM model can effectively pinpoint ransomware infections, and therefore could immediately thwart any subsequent encryption by the malware for extortion purposes.

GPU/CPU comparison. As a final experiment, we performed a comparison of the execution time of the LSTM's forward pass for an individual item in the sequence between an Intel Xeon CPU with 13 GB of RAM, an NVIDIA A100 GPU with 40 GB of video RAM, and the optimized proposed FPGA-based approach. The results are given in microseconds (μs) and shown in Table I, along with the corresponding 95% Confidence Interval (CI). Note that the CI for the proposed CSD-based FPGA inference strategy is listed as N/A since the measurement was taken via the Vitis Software Platform Development Environment's hardware emulation mode. As can be expected, the GPU outperformed the CPU. However, the proposed approach surpassed the GPU by 344.6×. Thus, not only is the proposed CSD-based inference approach more power-efficient than high-peformance CPUs and GPUs, but it also has edge over such hardware in terms of speed.

	Execution time	**95% CI**
FPGA	2.15133 μs	N/A
CPU	991.57750 μs	217.46576 μs - 1765.68923 μs
GPU	741.35336 μs	394.45317 μs - 1088.25355 μs

TABLE I
TRADITIONAL DL HARDWARE COMPARISON

V. Related Work

CSDs, which have proven to offer minimal power consumption in comparison to CPUs and GPUs, have also shown to offer appreciable acceleration for a variety of tasks including but not limited to hardware-based virtualization [15], encryption [16], database operations [17–22], big data analytics [23, 24], and several others [25]. In light of these advances, recent research endeavors have began leveraging CSDs for ML purposes. Largely, these approaches utilize the near-storage processing capabilities of CSDs to perform various data pre-processing procedures applicable to ML, such as tackling heavy data movement workloads [26], the management of graph-structure data [27, 28], data preprocessing to minimize related overhead associated with deep learning model training [5–7], and selecting important subsets of large datasets directly at the storage level [6].

Following the lead of the aforementioned noteworthy endeavors, efforts have been made to begin offloading ML inference tasks to CSDs. For instance, Tavakoli, Beygi, and Yao [29] integrated a dimensionality-reduced, k-Nearest Neighbors (kNN) classifier into a CSD. Further, An, Aliaj, and Jun [30] utilized a CSD to accelerate some convolution operations to facilitate Convolutional Neural Network (CNN) inference. It is our aim to extend these noble endeavors via offloading the entirety of the inference procedure of a sequential deep learning model to a CSD, demonstrating that CSDs have the capacity to not only assist such models but can also perform inference in an near-standalone manner comparable with CPUs and GPUs.

VI. Limitations and Future Directions

Despite the acceleration that accompanies offloading applications to FPGAs, along with the reduction of both energy usage and CPU overhead, the primary limitations in doing so are the domain-specific knowledge that it takes to program these devices and their resource constraints. While this work aims to facilitate addressing the former, future efforts exploring additional innovations for efficiently integrating classifiers into CSD-based FPGAs would be an interesting direction. Additionally, while the proposed approach was optimized using fixed-point arithmetic, mixed precision procedures are commonly utilized in deep learning models to enhance computational speed and efficiency by performing operations in lower precision where high precision is not necessary, and in higher precision where greater accuracy is required. As such, exploring mixed precision alternatives on CSDs would be a notable endeavor.

VII. Conclusion

In conclusion, this work introduces a novel approach for leveraging CSDs within data centers to offload deep learning classification, specifically LSTM models. This approach not only inherently reduces power consumption but we demonstrate that it dramatically reduces latency of high-performance CPUs and even GPUs, thereby addressing the dual challenges of operational efficiency and security in the face of escalating data center demands and sophisticated cybersecurity threats. Indeed, the showcased efficacy of our CSD-based classification technique in handling real-time ransomware detection underscores its potential to transform data center operations, paving the way for more resilient, efficient, and secure digital infrastructures.

REFERENCES

[1] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, “An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends,” *IEEE access*, vol. 9, pp. 87 094–87 155, 2021.

[2] K. Friday, E. Kfoury, E. Bou-Harb, and J. Crichigno, “Towards a unified in-network ddos detection and mitigation strategy,” in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2020, pp. 218–226.

[3] K. Friday, “On offloading network forensic analytics to programmable data plane switches,” *World ScientificNow Publishers series in business*, 2021.

[4] K. Friday, E. Kfoury, E. Bou-Harb, and J. Crichigno, “Inc: In-network classification of botnet propagation at line rate,” in *European Symposium on Research in Computer Security*. Springer, 2022, pp. 551–569.

[5] Y. Lee, J. Chung, and M. Rhu, “Smartsage: training large-scale graph neural networks using in-storage processing architectures,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 932–945.

[6] N. Prakriya, Y. Yang, B. Mirzasoleiman, C.-J. Hsieh, and J. Cong, “Nessa: Near-storage data selection for accelerated machine learning training,” in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, 2023, pp. 8–15.

[7] J.-H. Kim, S. Han, K. Park, S.-Y. Ji, and J.-Y. Kim, “Trinity: In-database near-data machine learning acceleration platform for advanced data analytics,” *IEEE Access*, 2023.

[8] Xilinx, “Vitis unified software platform.” [Online]. Available: https://www.xilinx.com/products/design-tools/vitis.html

[9] P. Rodríguez, M. A. Bautista, J. Gonzalez, and S. Escalera, “Beyond one-hot encoding: Lower dimensional target embedding,” *Image and Vision Computing*, vol. 75, pp. 21–31, 2018.

[10] Github. [Online]. Available: https://github.com/kfrida1/csdinf

[11] AMD, “Alveo u200 data center accelerator card.” [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u200.html

[12] K. Friday, E. Bou-Harb, and J. Crichigno, “A learning methodology for line-rate ransomware mitigation with p4 switches,” in *International Conference on Network and System Security*. Springer, 2022, pp. 120–139.

[13] M. Akbanov, V. G. Vassilakis, and M. D. Logothetis, “Ransomware detection and mitigation using software-defined networking: The case of wannacry,” *Computers & Electrical Engineering*, vol. 76, pp. 111–121, 2019.

[14] Cuckoo, “Cuckoo sandbox.” [Online]. Available: https://cuckoo.cert.ee/

[15] D. Kwon, W. Lee, D. Kim, J. Boo, and J. Kim, “Smartfvm: A fast, flexible, and scalable hardware-based virtualization for commodity storage devices,” *ACM Transactions on Storage (TOS)*, vol. 18, no. 2, pp. 1–27, 2022.

[16] Y. Yang, H. Lu, and X. Li, “Poseidon-ndp: Practical fully homomorphic encryption accelerator based on near data processing architecture,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[17] C.-G. Lee, H. Kang, D. Park, S. Park, Y. Kim, J. Noh, W. Chung, and K. Park, “ilsm-ssd: An intelligent lsm-tree based key-value ssd for data analytics,” in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019, pp. 384–395.

[18] J.-H. Park, S. Choi, G. Oh, and S.-W. Lee, “Sas: Ssd as sql database system,” *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1481–1488, 2021.

[19] M. Soltaniyeh, V. L. M. Dos Reis, M. Bryson, R. Martin, and S. Nagarakatte, “Near-storage acceleration of database query processing with smartssds,” in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 265–265.

[20] W. Qiao, J. Oh, L. Guo, M.-C. F. Chang, and J. Cong, “Fans: Fpga-accelerated near-storage sorting,” in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 106–114.

[21] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, “Nascent: Near-storage acceleration of database sort on smartssd,” in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 262–272.

[22] S. Salamat, H. Zhang, Y. S. Ki, and T. Rosing, “Nascent2: Generic near-storage sort accelerator for data analytics on smartssd,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 2, pp. 1–29, 2022.

[23] K. Chapman, M. Nik, B. Robatmili, S. Mirkhani, and M. Lavasani, “Computational storage for big data analytics,” in *Proceedings of 10th International Workshop on Accelerating Analytics and Data Management Systems (ADMS’19)*, 2019.

[24] P. Auradkar, T. Prashanth, S. Aralihalli, S. P. Kumar, and D. Sitaram, “Performance tuning analysis of spatial operations on spatial hadoop cluster with ssd,” *Procedia Computer Science*, vol. 167, pp. 2253–2266, 2020.

[25] D. Fakhry, M. Abdelsalam, M. W. El-Kharashi, and M. Safar, “A review on computational storage devices and near memory computing for high performance applications,” *Memories-Materials, Devices, Circuits and Systems*, p. 100051, 2023.

[26] S. Li, F. Tu, L. Liu, J. Lin, Z. Wang, Y. Kang, Y. Ding, and Y. Xie, “Ecssd: Hardware/data layout co-designed in-storage-computing architecture for extreme classification,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–14.

[27] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, “Graphssd: graph semantics aware ssd,” in *Proceedings of the 46th international symposium on computer architecture*, 2019, pp. 116–128.

[28] J.-H. Kim, Y.-R. Park, J. Do, S.-Y. Ji, and J.-Y. Kim, “Accelerating large-scale graph-based nearest neighbor search on a computational storage platform,” *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 278–290, 2022.

[29] E. B. Tavakoli, A. Beygi, and X. Yao, “Rpknn: An opencl-based fpga implementation of the dimensionality-reduced knn algorithm using random projection,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 4, pp. 549–552, 2022.

[30] J. An, E. Aliaj, and S.-W. Jun, “Precog: Near-storage accelerator for heterogeneous cnn inference,” in *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2023, pp. 45–52.

[31] “Best of...” [Online]. Available: https://www.portablefreeware.com/bestof.php

APPENDIX

Table II highlights the prominent ransomware samples that were aggregated to derive the dataset of ransomware API call sequences used in the proposed approach’s ransomware detection use case. In total, ten families were collected encompassing 78 variants. In order to ensure a plethora of varying malicious API call sequences, ransomware families that also exhibit self-propagation behaviors were incorporated into the dataset. Moreover, all aggregated variants encrypt files (as opposed to strictly locking behavior, which has largely become obsolete) and their network traffic.

Family	Instances	Encryption	Self-propagation
Ryuk	5 variants	✓	✓
Lockbit	6 variants	✓	✓
Teslacrypt	10 variants	✓	×
Virlock	11 variants	✓	×
Cryptowall	8 variants	✓	×
Cerber	9 variants	✓	×
Wannacry	7 variants	✓	✓
Locky	6 variants	✓	×
Chimera	9 variants	✓	×
BadRabbit	5 variants	✓	✓

TABLE II
RANSOMWARE DATASET OVERVIEW.

Each of the variants were then executed in a Cuckoo sandbox environment [14] using Windows 10 and 11 to extract all API calls that were made, in the order in which they would be observed on a system housing a CSD equipped with the proposed approach. An API call sequence for each variant of length 100 was taken, beginning with the first API call made to promote early detection. In order to facilitate generalizability to varying orders of malicious API calls, we also employed a sliding window of length 100 to extract sub-sequences at different stages in each variant's execution. As given by the appreciable ransomware detection accuracy achieved by the proposed approach, this sliding window procedure indeed did not hinder the model's performance (i.e., which could occur when such malicious sub-sequences are indistinguishable from those of benign nature), and therefore can only enhance its capacity for generalizing to malicious API call sequences not observed during training. Ultimately, 13,340 ransomware API call sequences of length 100 are encompassed in the dataset.

Lastly, an assortment of benign API call sequences were also extracted from Windows 10 and 11 environments. The benign API call sequences were derived from both manual interaction with such environments and via executing popular applications within them. Popular applications were selected from Top Ten lists on The Portable Freeware Collection [31] from years 2018 through 2021, as well as from the website's Popular Titles. In total, 30 popular applications were collected and executed. The aforementioned technique based on a sliding window of length 100 was subsequently utilized to extract benign API call sub-sequences from the popular applications and manual interaction, resulting in 15,660 sequences. The final benign and ransomware API call sequences were then merged and shuffled for binary classification tasks. The final dataset is publicly available [10].