Journal of
Instrumentation

OPEN ACCESS

A latency-constrained, Gated Recurrent Unit (GRU) implementation in the Versal AIE

To cite this article: M. Sapkas *et al* 2026 *JINST* **21** C03015

View the article online for updates and enhancements.

You may also like

- Cloud Server Oriented FPGA Accelerator for Long Short-Term Memory Recurrent Neural Networks
Jiasheng Wang, Yu Zhou, Yuyang Sun et al.
- rule4ml: an open-source tool for resource utilization and latency estimation for ML models on FPGA
Mohammad Mehdi Rahimifar, Hamza Ezzaoui Rahali and Audrey C Therrien
- Ferroelectric $Hf_{0.5}Zr_{0.5}O_2$ Memcapacitor Array for in-Memory Computing
Jiyoung Kim, Minjong Lee, Geon Park et al.

This content was downloaded from IP address 37.120.137.216 on 16/07/2026 at 07:11

Published by IOP Publishing for Sissa Medialab

Received: *November 6, 2025*
Accepted: *January 27, 2026*
Published: *March 10, 2026*

Topical Workshop on Electronics for Particle Physics
Rethymno, Crete, Greece
6–10 October 2025

A latency-constrained, Gated Recurrent Unit (GRU) implementation in the Versal AIE

M. Sapkas[a,*] **A. Triossi**[a] **and M. Zanetti**[a]

[a] *Università di Padova and INFN, Sezione di Padova, Via Marzolo 8, 35131 Padova, Italy*

E-mail: michail.sapkas@phd.unipd.it

Abstract: This work explores the use of the AMD Xilinx Versal Adaptable Intelligent Engine (AIE) to accelerate Gated Recurrent Unit (GRU) inference for latency constrained applications. We present a custom workload distribution framework across the AIE's vector processors and propose a hybrid AIE-Programmable Logic (PL) design to optimize computational efficiency. Our approach explores the parallelization over the rows of the matrices by utilizing as many of the AIE vectorized processors effectively computing all the elements of the resulting vector at the same time, an alternative to cascade stream pipelining.

Keywords: Acceleration cavities and superconducting magnets (high-temperature superconductor, radiation hardened magnets, normal-conducting, permanent magnet devices, wigglers and undulators); Data processing methods; On-board data handling; Trigger detectors

*Corresponding author.

© 2026 The Author(s). Published by IOP Publishing Ltd on behalf of Sissa Medialab. Original content from this work may be used under the terms of the Creative Commons Attribution 4.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.

https://doi.org/10.1088/1748-0221/21/03/C03015

2026 JINST 21 C03015

Contents

1 **Introduction** 1

2 **Recurrent Neural Networks-Gated Recurrent Unit** 1

3 **The Versal Adaptable Intelligent Engine** 2

4 **Model implementation** 3

5 **Latency measurements and conclusions** 4

1 Introduction

With the introduction of the *Versal* family, AMD presents the *Adaptive Compute Acceleration Platform* (ACAP) [1]. One of the key innovations within this platform is the *Adaptable Intelligent Engine* (AIE), a specialized accelerator array optimized for high throughput multiply accumulate (MAC) operations on floating point and fixed point vector data. We acknowledge prior research efforts that efficiently map generalized matrix matrix (GEMM) operations onto the AIE, such as *MaxEVA* [2] and *CHARM* [3]. However, these works primarily target throughput optimization in synthetic matrix multiplication scenarios, whereas our focus is on real time inference under stringent latency constraints. Furthermore, Recurrent Neural Networks (RNNs) are notoriously difficult to parallelize efficiently on hardware, making their low latency implementation a challenging yet valuable objective. In this work, we present a proof of concept implementation of a latency constrained GRU on the Versal AIE. The design explores several unconventional aspects of the architecture, including the use of free running kernels [4], row-wise matrix vector operations for efficient parallelization across hidden state dimensions, and a novel utilization of the AIE's interface tiles [5] as dynamic data aggregators. A direct comparison with GPU/CPU baselines is outside the scope of this work, which focuses on architectural trade offs within Versal.

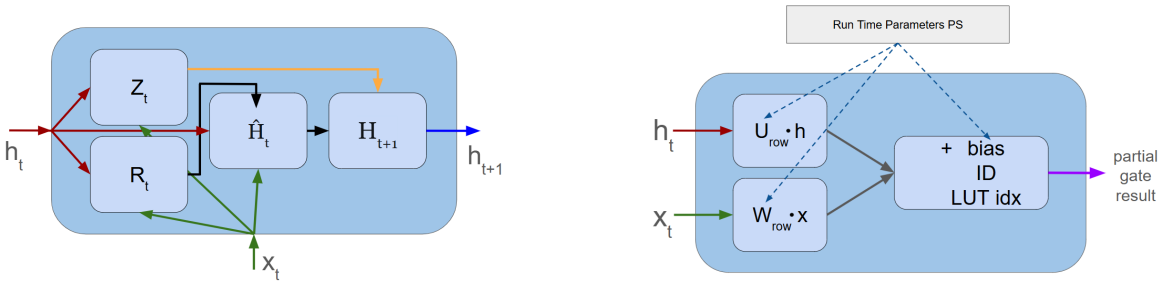
2 Recurrent Neural Networks-Gated Recurrent Unit

RNNs [6] are a specialized class of neural architectures designed to process sequential or time dependent data and incorporate temporal dependencies by maintaining a hidden state that evolves over time.

The Gated Recurrent Unit (GRU) [7], along with the Long Short-Term Memory (LSTM) [8], represents one of the most widely used RNN variants and use gating mechanisms as seen in the GRU Cell in figure 1(a). These gates are implemented via linear transformations followed by nonlinear activation functions. The Reset gate (R_t) determines how much of the previous hidden state should be forgotten, while the Update gate (Z_t) controls how much of the Candidate New Hidden State ($\tilde{H}_t$) should contribute to the current hidden state. Mathematically, the GRU can be expressed as:

$$
\begin{aligned}
Z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
R_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
\tilde{H}_t &= \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_{\tilde{h}}) \\
H_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

2026 JINST 21 C03015



(a) Data flow between gates inside the GRU "cell".

(b) Compute one element of a gate.

Figure 1. *(a)* A forward pass of the GRU. *(b)* Each of the Z_t, R_t, $\tilde{H}_t$ gates partial result needs three AI tiles to compute, each consuming a row of the matrix and the whole vector. The third tile adds the results together with the bias, assigns an ID, and transforms the floating point to a LUT index to be sent to the PL kernel (GRU HYBRID), or applies the LUT in the AIE and then sends it to a specialized aggregator/sorting AIE kernel via the packet merge construct (GRU AIE).

where $\sigma(\cdot)$ denotes the sigmoid activation, $\tanh(\cdot)$ the hyperbolic tangent, and $\odot$ the element-wise (Hadamard) product. Because each timestep depends on the previous hidden state, GRUs remain inherently sequential in computation, limiting parallelization. Moreover, within a single timestep, dependencies among the gates impose additional sequential constraints. Typical latencies for a forward pass on GPUs are in the order of milliseconds [9].

3 The Versal Adaptable Intelligent Engine

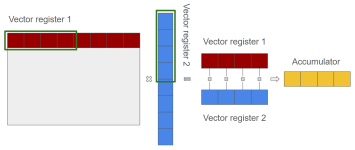
The AIE is a two dimensional tiled array composed of 400 RISC-V vector processors, interconnected through a high bandwidth network on chip. Data exchange between AIE and PL occurs via 39 specialized interface tiles that employ the AXI4-Stream protocol. These interfaces convert 128 bit data streams from the PL domain to 32 bit streams suitable for the AIE domain. Given that the AIE operates at a maximum clock frequency of 1.25 GHz, achieving peak throughput requires the PL domain to be clocked at one quarter of this frequency (312.5 MHz) to maintain synchronization. These are also the frequencies that we used for our implementation. Each AIE tile features a vector processor coupled with 32 KB of local memory. The vector unit supports both integer and floating point arithmetic, using two distinct execution pipelines. It can perform up to two 128 bit load operations and one store operation per clock cycle. To exploit this vectorized architecture, developers use C++ intrinsics or, more conveniently, the high level APIs provided in recent AMD toolchains [10]. Local memory within a tile typically serves as stack and heap storage, as well as a ping pong buffer for efficient double buffered data transfers. A particularly useful feature of the AIE architecture is the support for *runtime parameters*, which allow values to be dynamically loaded into the memory banks of specific AIE tiles by the processing system (PS). AIE tiles can communicate with one another through four principal mechanisms, two local: Cascade Streams and Input/Output Buffers, and two non local: AXI4-Streams and Packet Streams. We will make use the two latter. One of the main advantages of the AIE over traditional programmable logic is its native support for floating point arithmetic, which eliminates the need for quantization when deploying AI models. In addition, AIE computation is deterministic and does not require timing closure, a common challenge in FPGA design.

2026 JINST 21 C03015

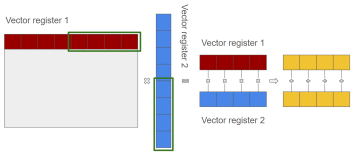
4 Model implementation

Our implementation aims to explore the capabilities of the AIE as a real time inference device *using the Single Precision Floating Point (32 bit) data type*. In this context, a forward pass of the model processes a *single batch*, and all AIE kernels execute within an *infinite* while *loop*. To implement the matrix vector multiplication, we leverage the vectorized Multiply Accumulate (MAC) operation available within each AIE tile. In this process, the input or hidden state vector is streamed into the vector registers, while the matrix parameters are loaded from the tile's local memory. The efficiency of this computation depends primarily on two factors: the data streaming bandwidth and the MAC pipeline. There are two principal approaches for executing this operation:

1. Column-wise Cascade. This method represents the most hardware compliant, or "lawful", implementation in that it fully exploits the AIE's most efficient data transfer mechanism; the cascade stream. The matrix parameters are partitioned into column blocks, and each tile performs MAC operations between a block of matrix columns and the corresponding elements of the input vector (which are broadcasted to match the vector width). The partial results of each block are then cascaded to the neighboring tile and added. The number of rows processed in each operation corresponds to the number of vector lanes of the chosen data type and each participating tile must process all the rows of the matrix which can create memory limitations.



(a) MAC the first part of the row with first part of the vector, store the result.



(b) Continues with the rest of the row with rest of the vector, accumulate on previous result.



(c) When the whole row and vector have been consumed, reduce the vector, Finally, apply activation function, the element that corresponds on the row of the matrix (here is the first) of the resulting vector is computed.

Figure 2. Row-wise implementation of the matrix vector multiplication within the AIE.

2. Row-wise Streams. This method addresses the limitations of the column-wise cascade by transposing the computation. Instead of performing the MAC operation along the columns, we now accumulate across the rows of the matrix. In this configuration, each tile MACs across an entire matrix row, progressing through the columns at a width determined by the vector lane width as shown in figure 2. This approach accelerates traversal through the matrix but requires the whole vector to be streamed to all participating tiles, rather than partitioned into smaller segments. In hindsight, this introduces certain inefficiencies in data movement since we cannot stream fast enough to exploit the best possible pipeline frequency of a tile. Thankfully, this limitation applies only for the first rows computed in a tile. All next rows can benefit by the vector being kept in memory and now we are only limited by how fast we can load from memory. To distribute the full vector efficiently, we exploit the streaming interface's capability to broadcast the same data stream from a single interface tile to multiple destinations. With this broadcast mechanism incurs a latency penalty. A data flow of three AIE tiles, as shown at figure 1(b) was used as template for the matrix vector multiplications of the Update and Reset gates. An AI Tile performs the $\mathbf{W}_{\mathbf{row}} \cdot x$ while another computes $\mathbf{U}_{\mathbf{row}} \cdot h$. A third

2026 JINST 21 C03015

tile accepts the dot product results, adds the bias, incorporates an ID to the packet and either sends it to the PL or applies a LUT for the activation function. Specialized $\tilde{H}_t$ AIE kernels are needed to interface with the R gate. This strategy allows for extensive utilization of AIE resources, enabling multiple matrix rows to be processed concurrently and using streams to move only one 32b numbers. Furthermore, it introduces the possibility of a "row reuse" factor, wherein each tile computes several consecutive rows. Using the template above the following formula calculates the AI tiles used for a single GRU layer: $3 \times HIDDEN \times 3 + 1$. It corresponds to three AI tiles, for every hidden state dimension, for three gates, plus one AI Tile for the New Hidden State Gate (H_t). In case of GRU AIE architecture another three kernels are added as aggregators.

Aggregation of partial results. The distributed partial results are then aggregated to reconstruct the output vector. One approach is to perform this reduction directly within the AIE using packet streams combined through the *merge* construct. However, the arrival order of the packets is difficult to predict before implementation. To manage this, we assign unique identifiers to each packet. These IDs are then used to sort the packets, reconstruct the output vector, and broadcast it to the subsequent processing stage (e.g., the next GRU gate). While functionally correct, this approach breaks pipeline continuity: downstream AIE kernels must wait for the entire vector to be reassembled before proceeding, thereby introducing a latency bottleneck. This leads to a natural question: can aggregation be achieved faster? We introduce a "hybrid" solution in which the interface tiles are used as a data path to a PL aggregation kernel. To the best of our knowledge, this is the first application of multiple interface tiles in such a role for a machine-learning workload. In this configuration, the partial results are streamed directly from the AIE into the PL fabric. From the AIE side, each interface tile connects to a three way packet-merge construct, allowing partial results from all three GRU gates to share the same interface tile. The corresponding PL kernel, implemented in High Level Synthesis (HLS) [11], operates as a finite state machine that executes the following sequence: 1) Perform blocking reads from all interface tiles; 2) Decode packet identifiers, extract LUT indices for the activation function; 3) Apply the appropriate activation function; 4) Write the output to the appropriate interface tile, padding with zeros when necessary to meet hardware alignment requirements. The timing and resources results for the PL kernel can be found in table 1. The column "AIE AGGR TILE LAT" measures only the latency of the specialized aggregator AI tile, to serve as a comparison with the PL kernel.

Table 1. Resource utilization and timing results for the PL kernel and the AIE.

INTERFACE TILES HIDDEN DIM	AIE GRU TILES USED	AIE HYBRID TILES USED	PL kernel				AIE AGGR TILE LAT
			II	DEPTH	FF	LUTS	
20	184	181	6	7 cc = 22.4 ns	5124	9250	229.6 ns
24	220	217	6	7 cc = 22.4 ns	6066	11834	273.6 ns
28	256	253	8	9 cc = 28.8 ns	7057	15663	319.2 ns
32	292	289	8	9 cc = 28.8 ns	8003	19198	363.2 ns

5 Latency measurements and conclusions

We focus exclusively on the row-wise implementation. Latency measurements are performed using Single Precision (32b) Floating Point (FP) 8-lane Vectors [12] on a single-layer GRU with hidden size

2026 JINST 21 C03015

up to 32 units. We are processing a single input vector per iteration (batch size = 1), using synthetic inputs and weights to characterize hardware latency independently of model accuracy. In order to scale the hidden and input sizes, it is not possible to use only one dataset, but we numerically tested the $H = 20$ and $X = 5$ with a GRU trained in a jet tagging dataset [13]. In this application, the GRU exhibits a computational dependency on its previous hidden state and thus, the overall latency of the model and the iteration interval should be identical, since each iteration must await the completion of the preceding one. The AI tiles responsible for performing the $\mathbf{W} \cdot x$ matrix vector multiplication are decoupled from the rest of the computational pipeline, allowing them to prefetch and process new input data as long as they do not experience backpressure from downstream tiles.

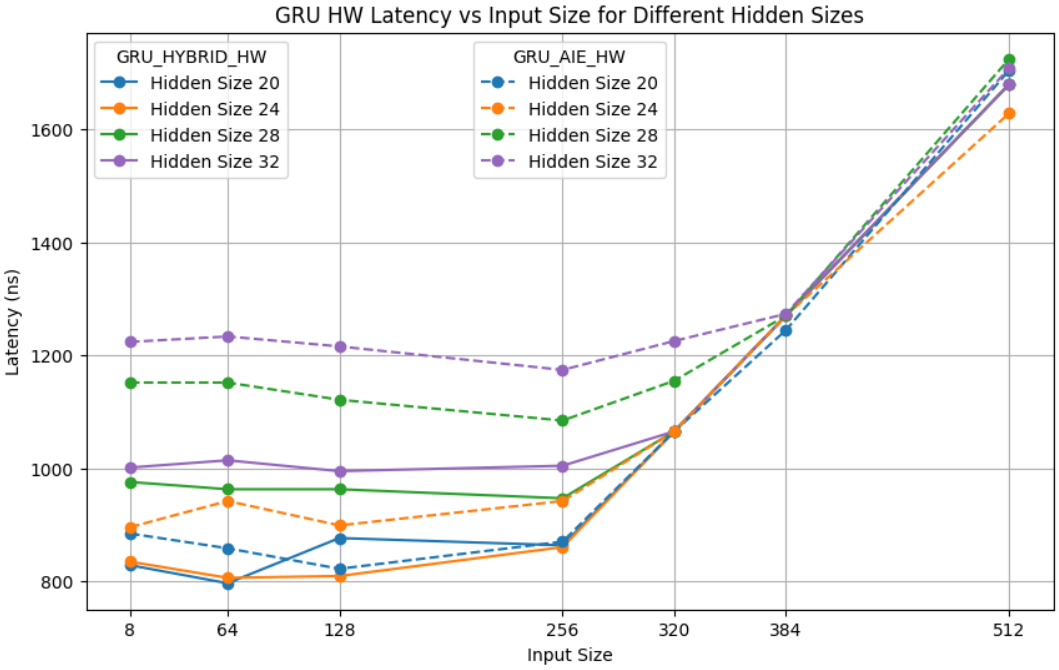


Figure 3. Latency results for a GRU using the PL kernel called "Hybrid" and the GRU implementation completely inside the AIE using specialized aggregation/sorting AI Tiles called "AIE".

Latency measurements are performed directly in hardware. We monitor two consecutive *TVALID* signal activations on the selected output stream interface, measuring the elapsed time between the assertion of the signal for the first valid datum and its subsequent activation for the next iteration. This interval corresponds to the true latency of a complete forward pass through the model. The key features of figure 3 are: 1) The better latency of the Hybrid model scaling up with the hidden state dimension which is 163 ns and 197 ns on average for hidden sizes 28 and 32. 2) The latency plateau with respect to the input size showcases the decoupling of the $\mathbf{W} \cdot x$ computing kernels. The dot product is in the shadow of the data transfers but after input dimension 256 it adds nanoseconds linearly as it starts dominating in latency. Finally, we suffer limitations in the number of available packet merge connections for the GRU AIE architecture (32). The GRU HYBRID architecture is also constrained by the number of interface tiles (39). In case we need to compute larger hidden states, the same architecture can be used by allowing the tiles to compute more matrix rows and reuse the same data transfer and aggregation paths, computing 32 outputs at a time. Therefore we expect that the PL kernel will accumulate latency gains over the AIE packet merge in the regime $H = 32$ and $X = [8, 256]$.

2026 JINST 21 C03015

References

[1] AMD, *Versal Adaptive SoCs*, https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal.html (2025).

[2] E. Taka, A. Arora, K.-C. Wu and D. Marculescu, *MaxEVA: Maximizing the Efficiency of Matrix Multiplication on Versal AI Engine*, arXiv:2311.04980.

[3] J. Zhuang et al., *CHARM: Composing Heterogeneous Accelerators for Matrix Multiply on Versal ACAP Architecture*, arXiv:2301.02359.

[4] AMD, *Free-Running AI Engine Kernel — UG1079 AI Engine Kernel and Graph Programming Guide*, https://docs.amd.com/r/2024.1-English/ug1079-ai-engine-kernel-coding/Free-Running-AI-Engine-Kernel (2024).

[5] AMD, *PL Interface Tile Capabilities — UG1079 AI Engine Kernel and Graph Programming Guide*, https://docs.amd.com/r/2024.1-English/ug1079-ai-engine-kernel-coding/PL-Interface-Tile-Capabilities (2024).

[6] R.M. Schmidt, *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*, *CoRR* **abs/1912.05911** (2019) [arXiv:1912.05911].

[7] J. Chung, C. Gulcehre, K.H. Cho and Y. Bengio, *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*, arXiv:1412.3555.

[8] S. Hochreiter and J. Schmidhuber, *Long Short-Term Memory*, *Neural Comput.* **9** (1997) 1735.

[9] C. Holmes et al., *GRNN: Low-Latency and Scalable RNN Inference on GPUs*, in the proceedings of the *Fourteenth EuroSys Conference 2019*, Dresden, Germany, 25–28 March 2019 [DOI:10.1145/3302424.3303949].

[10] Advanced Micro Devices, Inc., *AI Engine API User Guide (AIE-API) 2024.1 — API Reference*, https://download.amd.com/docnav/aiengine/xilinx2024_1/aiengine_api/aie_api/doc/modules.html (2024).

[11] Advanced Micro Devices, Inc., *Vitis High-Level Synthesis User Guide (UG1399) - HLS Programmer's Guide*, https://docs.amd.com/r/2024.1-English/ug1399-vitis-hls/HLS-Programmers-Guide (2024).

[12] Advanced Micro Devices, Inc., *Vector Data Types*, https://docs.amd.com/r/en-US/ug1079-ai-engine-kernel-coding/Vector-Data-Types (2025).

[13] E.E. Khoda et al., *Ultra-low latency recurrent neural network inference on FPGAs for physics applications with hls4ml*, *Mach. Learn. Sci. Tech.* **4** (2023) 025004 [arXiv:2207.00559].

2026 JINST 21 C03015