

PREPARED FOR SUBMISSION TO JINST

TWEPP 2025 TOPICAL WORKSHOP ON ELECTRONICS FOR PARTICLE PHYSICS OCTOBER
4, 2025 TO OCTOBER 10, 2025
RETHYMNO, CRETE, GREECE

A Latency-Constrained, Gated Recurrent Unit (GRU) Implementation in the Versal AI Engine

M. Sapkas¹, A. Triossi and M. Zanetti

*Università e INFN, 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 highlights the potential of deploying adaptable neural networks in real-time environments such as online preprocessing in the readout chain of a physics experiment, offering a flexible alternative to traditional fixed-function algorithms.

KEYWORDS: On-board data handling, Data processing, Trigger detectors, Instrumentation and hardware for accelerators

¹Corresponding author.

1 Introduction

With the introduction of the *Versal* family, AMD presents the *Adaptive Compute Acceleration Platform* (ACAP) [1], a heterogeneous computing architecture designed to provide high performance, low latency, and adaptability within a unified system-on-chip. One of the key innovations within this platform is the *AI Engine*, a specialized accelerator array optimized for high-throughput multiply-accumulate (MAC) operations on floating-point and fixed-point vector data.

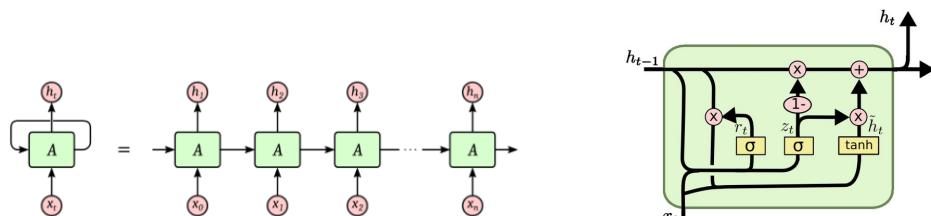
We acknowledge prior research efforts that efficiently map generalized matrix–matrix (GEMM) operations onto the AI Engine, such as *MaxEVA* [2] and *CHARM* [3]. However, these works primarily target throughput optimization in synthetic matrix–matrix computation scenarios, whereas our focus is on real-time 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 Gated Recurrent Unit on the Versal AI Engine. 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 AI Engine’s interface tiles [5] as dynamic data aggregators. These techniques collectively aim to demonstrate the feasibility of deploying state-of-the-art models under strict latency constraints, paving the way for future low-latency adaptive computing applications. We hope that these methods will inspire further exploration and refinement by the broader research community.

2 Recurrent Neural Networks - Gated Recurrent Unit

Recurrent Neural Networks (RNNs) [6] are a specialized class of neural architectures designed to process sequential or time-dependent data. Unlike feedforward networks, which assume independence among input samples, RNNs incorporate temporal dependencies by maintaining a hidden state that evolves over time.

Shown at Fig. 1, at each timestep t , the network receives an input feature vector x_t and updates its hidden state h_t . The dimensionality of this hidden state is a tunable hyperparameter of the model. Conceptually, the hidden state acts as a form of memory, encoding relevant information from previous timesteps and enabling the model to capture temporal context across the sequence.



(a) A Recurrent Neural Network operating in a feedback loop on the left and the equivalent "unroll in time" on the right.

(b) A graphical representation of the data flow between gates inside the GRU "cell"

Figure 1: Recurrent Neural Networks (RNNs) and the Gated Recurrent Unit (GRU)

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. Specifically, the Update gate (z_t) and the Reset gate (r_t), which regulate information flow through the network. These gates are implemented via linear transformations followed by nonlinear activation functions. The reset gate determines how much of the previous hidden state should be forgotten, while the update gate 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:

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

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 that can reduce computational efficiency on hardware. Typical latencies for a forward pass on GPUs are in the order of milliseconds[9].

3 Versal AI Engine

The AMD Versal family represents a new generation of adaptive computing platforms, integrating a heterogeneous System-on-Chip (SoC) architecture. Each device combines a real-time processing unit (RT), an application processing unit (APU), programmable logic (PL), and a novel acceleration fabric known as the *AI Engine* (AIE). The AI Engine is composed of a two-dimensional array of 400 RISC-V vector processors, interconnected through a high-bandwidth network-on-chip. Data exchange between the AIE and the 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 AI Engine domain. Given that the AI Engine 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.

Each AI Engine 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 can 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 AI Engine tiles by the processing system (PS).

AI Engine 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 use the two latter.

One of the main advantages of the AI Engine 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.

4 Model Implementation

The design of the computational pipeline is driven by three primary considerations: 1) the distribution of matrix–vector multiplications, which dominate the computational workload; 2) the implementation of activation functions; and 3) the scheduling of computations according to the model’s dataflow.

Our implementation aims to explore the capabilities of the AI Engine as a real-time inference device. In this context, a forward pass of the model processes a **single batch**, and all kernels execute within an **infinite while loop**.

To implement the matrix vector multiplication, we leverage the vectorized Multiply Accumulate (MAC) operation available within each AI Engine 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 AI Engine’s most efficient data transfer mechanism; the cascade stream. The matrix parameters are partitioned into column blocks, and each tile performs a MAC operation between a column of the matrix and the corresponding element of the input vector (which is broadcasted to match the vector width). The number of rows processed in each operation corresponds to the number of vector lanes of the chosen data type.

The distribution occurs along the columns, which correspond to the input-vector dimension \mathbf{x}_t in the GRU model. This can be problematic for architectures with large input feature vectors (on the order of thousands of features) and comparatively modest hidden state sizes (on the order of hundreds). Furthermore, unless significant implementation complexity is introduced, distributing computation in this fashion obliges each participating tile to process all assigned rows. In practice, once the columns corresponding to the input-vector dimension are consumed - each spanning a number of rows equal to the vector lane width - the operation must restart from the next block of rows and continue iteratively until the entire matrix has been processed.

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 Multiplies Accumulates across an entire row, progressing through the columns at a rate determined by the vector-lane width. This approach accelerates traversal through the matrix but requires the complete input (or hidden-state) vector to be streamed to all participating tiles, rather than partitioned into smaller segments.

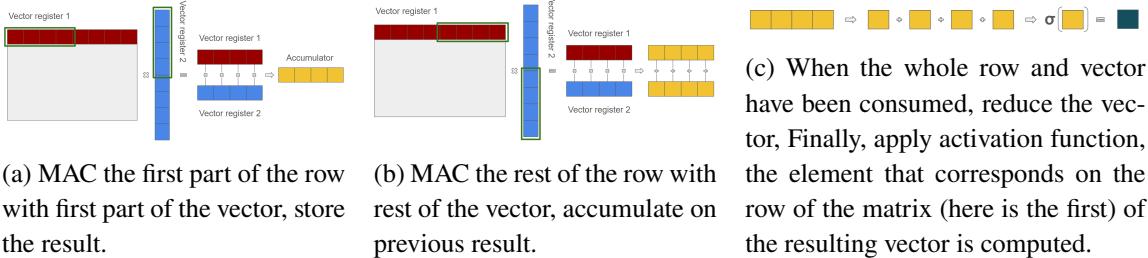


Figure 2: Column-wise implementation of the matrix–vector multiplication within the AI Engine.

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.

In operation, once an entire row has been processed, the resulting vector elements must be summed to produce a single scalar element of the output vector corresponding to that row. This strategy allows using streams to move only one 32b number and for extensive utilization of AI Engine resources, enabling multiple rows to be processed concurrently and improving latency through parallelization. Furthermore, it introduces the possibility of a “row reuse” factor, wherein each tile computes several consecutive rows, enhancing overall compute density and reducing idle cycles.

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 AI Engine 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 reorder 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 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 AI Engine into the programmable logic (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 performs 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 as a properly formatted vector, padding with zeros when necessary to meet hardware alignment requirements.

5 Latency Measurements and Conclusions

We focus exclusively on the row-wise implementation. In this application, the Gated Recurrent Unit (GRU) model exhibits a computational dependency on its previous hidden state and thus, the overall latency of the model and the iteration interval (the time between consecutive forward passes) should be identical, since each iteration must await the completion of the preceding one. The kernels 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 kernels.

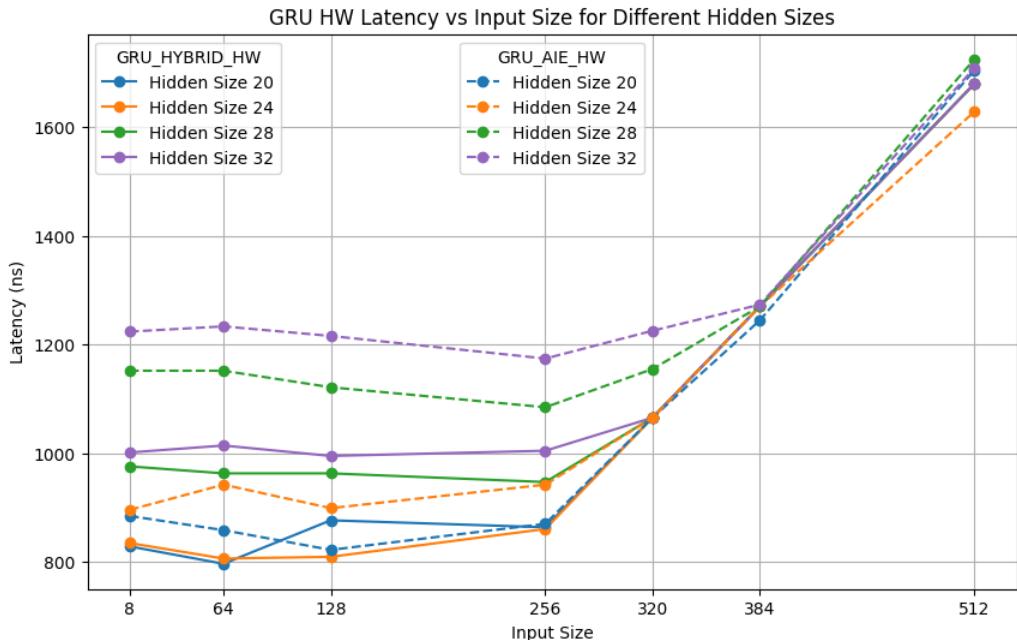


Figure 3: Latency results for a GRU using the PL kernel called "Hybrid" and the GRU implementation completely inside the AI Engine called "AIE".

The latency measurement is 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 Fig. 3 are: 1) The better latency of the Hybrid model scaling up with the hidden state dimension. 2) The plateau in latency with respect to the input size showcasing the decoupling of the $\mathbf{W} \cdot x$ computing kernels. Once the computation starts taking more time than the forward pass of the model itself, it adds latency linearly.

If we allow a hidden state dimension larger than 32 we suffer limitations in the number of available AIE tiles, interface tiles, and packet merge connections. In this case, the same architecture can be used by allowing the kernels to compute more matrix rows and reuse the same data transfer paths.

References

- [1] AMD, *Versal Adaptive SoCs*, 2025.
- [2] E. Taka, A. Arora, K.-C. Wu and D. Marculescu, *Maxeva: Maximizing the efficiency of matrix multiplication on versal ai engine*, 2023.
- [3] J. Zhuang, J. Lau, H. Ye, Z. Yang, Y. Du, J. Lo et al., *Charm: Composing heterogeneous accelerators for matrix multiply on versal acap architecture*, 2023.
- [4] AMD, *Free-Running AI Engine Kernel — UG1079 AI Engine Kernel and Graph Programming Guide*, 2024.
- [5] AMD, *PL Interface Tile Capabilities — UG1079 AI Engine Kernel and Graph Programming Guide*, 2024.
- [6] R.M. Schmidt, *Recurrent neural networks (rnns): A gentle introduction and overview*, CoRR [abs/1912.05911](https://arxiv.org/abs/1912.05911) (2019) [[1912.05911](https://arxiv.org/abs/1912.05911)].
- [7] J. Chung, C. Gulcehre, K. Cho and Y. Bengio, *Empirical evaluation of gated recurrent neural networks on sequence modeling*, 2014.
- [8] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, *Neural Computation* **9** (1997) 1735.
- [9] C. Holmes, D. Mawhirter, Y. He, F. Yan and B. Wu, *Grnn: Low-latency and scalable rnn inference on gpus*, in *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys ’19, (New York, NY, USA), Association for Computing Machinery, 2019, [DOI](#).
- [10] Advanced Micro Devices, Inc., *AI Engine API User Guide (AIE-API) 2024.1 – API Reference*. Advanced Micro Devices, Inc., 2024.
- [11] Advanced Micro Devices, Inc., *Vitis High-Level Synthesis User Guide (UG1399) – HLS Programmer’s Guide*. Advanced Micro Devices, Inc., 2024.