Department of Electrical Engineering, National Tsing Hua University
Special Topic on Implementation Research Abstract

High-Level Synthesis for Application Acceleration: Architectural Design of a Tensor Streaming Processor

高階合成實現應用加速：串流式張量處理器架構設計

Major Category: System

Group Number: B620

Advisor: Jiin Lai

Members: Bang Ruei Wang, Kai-Wen Lin, Yu-Tsen Chen, Ting-Wei Kung
Hsuan-Jung Lo, Tse-His Chang

Research Period: From (2025/08/15) to (2025/11/15).

Abstract

As deep learning models continue to expand, traditional CPU/GPU architectures face growing limitations in large-scale parallelism, latency predictability, and efficiency. Inspired by Groq's Tensor Streaming Processor (TSP) presented at ISCA, this project aims to design a hardware architecture optimized for deep learning inference. Our design adopts a dataflow-driven execution model with functional partitioning and external instruction control, while removing non-deterministic components such as caches and arbiters. This enables predictable latency, high utilization, and strong reconfigurability.

TSP's core idea is to model computation as deterministic producer–consumer data streams executed with fixed pipeline latency. The project consists of two stages:

1. **Hardware Architecture Design:** Build all functional modules and define their dataflow.
2. **Compiler Design:** Develop an instruction scheduler that issues precisely timed control signals to ensure stall-free streaming execution.

This report focuses on the hardware implementation and provides initial instruction scheduling and latency estimation for the compiler. We use the Vision Transformer (ViT) as our target workload and select *Normalize*—a frequent, regular, and computation-heavy operator—as the first module mapped to the TSP architecture.

The hardware system includes six major modules: **VXM**, **MXM**, **MEM**, **SXM**, **C2C**, and **ICU**. We describe the instruction format supporting external control and analyze each module's behavior and latency. In this phase, we perform a semi-automated scheduling of Normalize and estimate the total cycle count based on module delays. Full automation will be implemented once the compiler is completed.

We have successfully implemented and verified all six modules, achieving an Initiation Interval (II) of 1 across the system, enabling continuous, stall-free streaming. We also estimated the total cycles for Normalize and derived the supported Instructions Per Second (IPS), which will guide future compiler opimization.

1. Background

As deep learning models continue to grow in size and computational complexity, traditional CPU and GPU architectures increasingly struggle with large-scale parallelism, predictable latency, and energy efficiency. These limitations become more prominent in Vision Transformer (ViT) workloads, where repeated, structured operations intensify the need for deterministic and highly efficient execution.

The Tensor Streaming Processor (TSP), proposed by GROQ, introduces a dataflow-oriented, cacheless architecture controlled entirely through software-defined instructions. By eliminating nondeterministic components and enforcing a fixed-latency pipeline, the TSP achieves near-full hardware utilization and predictable performance.

Inspired by this design philosophy, our project develops a streaming tensor processor architecture featuring functional slicing, deterministic dataflow, and external instruction control. Among ViT operations, *Normalization* is chosen as the first implementation target due to its simple dataflow, high frequency of use, and repetitive instruction pattern—making it ideal for validating both the hardware structure and scheduling methodology.

Using HLS, we implement six key modules (VXM, MXM, MEM, SXM, ICU, and C2C) and successfully reach II = 1, for further integration with the compiler.

2. Purpose

The purpose of this project is to design and implement a streaming tensor processor architecture—inspired by the GROQ Tensor Streaming Processor—that provides deterministic, high-efficiency acceleration for deep learning workloads. By adopting a dataflow-oriented pipeline, functional slicing, and software-defined instruction control, the project aims to overcome the limitations of traditional CPU/GPU architectures in predictability and utilization.

To validate the proposed architecture, the Normalize operation in Vision Transformers (ViT) is selected as the initial implementation target due to its frequent use, simple dataflow, and repetitive computational pattern. The project further aims to develop and verify key hardware modules (VXM, MXM, MEM, SXM, ICU, and C2C) using HLS, demonstrating that the design can achieve predictable latency, stable dataflow behavior, and high computational efficiency.

3. Method

3-1. Design process

1. Define the operations required for normalization.
2. Develop hardware modules

3. Develop corresponding compiler to issue the right ICU control to each module at the right timing

3-2. Hardware design

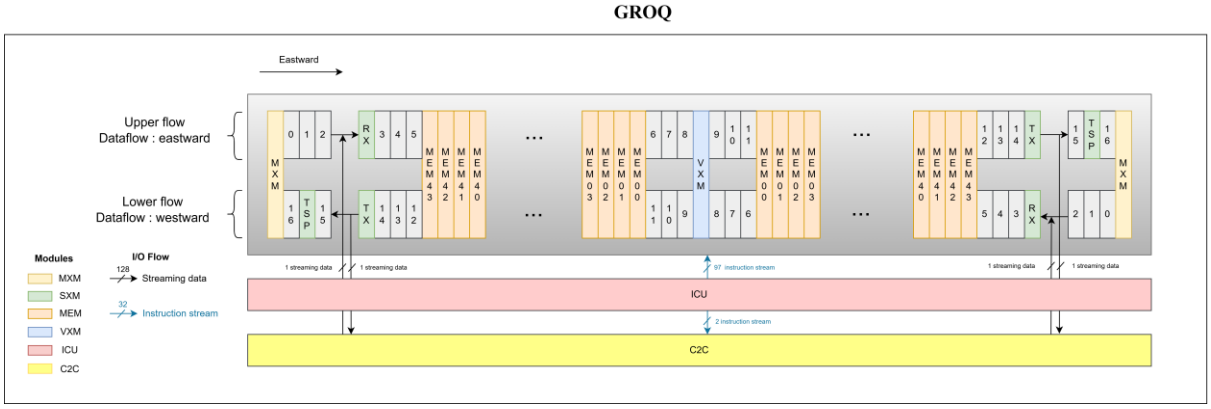

Fig. 3-0 Architecture overview

Our system is composed of six primary modules. The SXM module provides a bidirectional data path, receiving input data from the C2C interface and forwarding output data back to it. The VXM module performs linear operations, while the MXM module is responsible for matrix multiplication. MEM offers on-chip data storage. The ICU collects external instructions delivered through the C2C and distributes them to all compute modules. Finally, the C2C module manages external data and instruction transfers, forwarding instructions to the ICU and data streams to the SXM. In addition to these six submodules, the system has three key architectural features: **97 ICUs** that independently control the operations of their corresponding submodules; a SIMD-style superlane streaming dataflow composed of **32 streams**, each containing **16 lanes** for parallel processing; **64 MB** of globally shared SRAM for temporarily storing on-chip data to support subsequent computations.

3-3. Instruction control

Table 3-0

Instruction format

bit	**[31:30]**		**[29:2]**				**[1:0]**
ICU	ICU dst.		Function-specific Field				OPCODE
bit	**[31:30]**	**[29:25]**	**[24:20]**	**[19]**	**[18:6]**	**[5:4]**	**[3:0]**
MEM	ICU dst.	reserved	src_dst	side	addr	opcode	dskew
bit	**[31:30]**	**[29:27]**	**[26:22]**	**[21:17]**	**[16]**	**[15:13]**	**[12:9]**	**[8:0]**
VXM	ICU dst.	reserved	dstID	srcID	NOP	opcode	dskew	dfunc
bit	**[31:30]**	**[29:27]**	**[26:22]**	**[21:17]**	**[16:14]**	**[13:8]**	**[7:0]**
MXM	ICU dst.	size	dstID	srcID	opcode	dskew	dfunc
bit	**[31:30]**	**[29:17]**	**[17:14]**	**[13:9]**	**[8:7]**	**[6:3]**	**[2:0]**
TSP	ICU dst.	reserved	opcode	str	reserved	dskew	dfunc
bit	**[31:30]**	**[29:15]**	**[14]**	**[13:9]**	**[8:7]**	**[6:3]**	**[2:0]**

RX/TX	ICU dst.	reserved	pathsel	str	reserved	dskew	dfunc
bit	**[31:30]**	**[29]**	**[28]**	**[27:21]**	**[20:0]**		
C2C	ICU dst.	R/W	R_instr/R_data	len	Base address		

Following the instruction format, we construct a compiler capable of precisely determining the cycle at which each instruction should be issued to its corresponding module, thereby using the instructions to accurately control the overall dataflow.

3-4. Dataflow of normalization

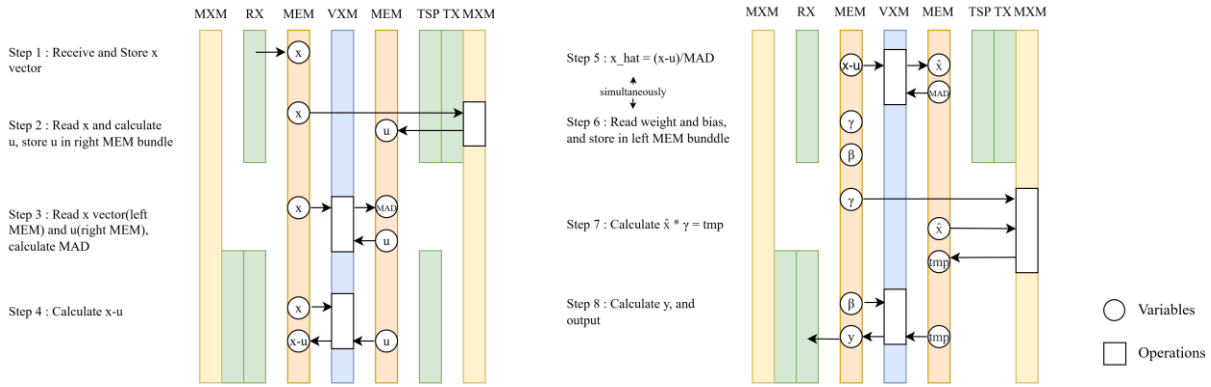

Fig. 3-1 Dataflow -- Normalization

For normalization to be implemented on our hardware architecture, we design the data flow as Fig. 3-1. And estimate the cycle count based on the above figure.

4. Results

Table 4-0

Performance for each module

Resource	II	LUT	DSP	BRAM	FF
VXM	1	957591	960	5760	564504
MXM	4	711250	960	0	1797515
SXM	1	819635	0	0	8199
MEM	1	212111	0	232	95729
ICU	1	104868	0	0	41616
C2C	1	4449	0	0	1600
Total		2809904	1920	5992	2509163

The estimated performance of Normalization on TSP is shown in Table 4.0. The cycle time is based on the synthesis result of each module; each can meet timing constraints under 10ns. And the cycle time and operation are estimated based on Fig. 3-1

Table 4-1

Estimated Performance of timing

Total cycle	1768

Total operation	2464
Cycle time	10 ns
IPS	1.39×10^8 operations/sec

5. Conclusion and Future Work

We successfully optimize the submodule to II = 1, meeting the feature of streaming interface. Future work will transform our prototype into a configurable compiler supporting the full Vision Transformer (ViT) workflow on the TSP architecture. We will extend the operator-lowering pipeline to map key ViT components — QKV projection, multi-head attention, Softmax, GELU, LayerNorm, and residuals—into hardware-level instructions (VXM, MXM, MEM, SXM, C2C, ICU). Unified templates for dataflow and scheduling will enable the entire ViT model to be expressed in our instruction set. We will also develop a latency-aware scheduler for conflict-free, predictable execution and refine MEM, ICU, and C2C formats, moving toward a fully configurable, timing-aware compiler for end-to-end ViT workloads.

6. Reference

[1] D. Abts et al., A software-defined tensor streaming multiprocessor for large-scale machine learning, unpublished.

[2] I. Ahmed et al., "Answer Fast: Accelerating BERT on the Tensor Streaming Processor," in Proc. IEEE 33rd Int. Conf. Application-Specific Systems, Architectures and Processors (ASAP), 2022, pp. 80–87.

[3] J. Devlin et al., "BERT: Pre-training of deep bidirectional transformers for language understanding," in Proc. NAACL, 2019, pp. 4171–4186, doi: 10.18653/v1/N19-1423.

[4] D. Abts et al., "Think Fast: A Tensor Streaming Processor (TSP) for accelerating deep learning workloads," in Proc. 47th Int. Symp. Computer Architecture (ISCA), 2020, pp. 145–158, doi: 10.1109/ISCA45697.2020.00023.

7. Reflection

Our project was inspired by the Groq Tensor Streaming Processor (TSP). We set out to implement a small-scale streaming tensor processor architecture using high-level synthesis (HLS) to explore deterministic execution and application acceleration. By studying the ISCA-TSP paper, we learned how superlanes, functional slices, and compiler-scheduled streams can be composed into a deterministic "stream engine," and we tried to capture these ideas in our

own HLS-based prototype with C2C, ICU, VXM, MXM, SXM, and MEM modules.

On the implementation side, we used HLS to build AXI and streaming interfaces, tune pipelines and pragmas, and integrate the compute-blocks and data-movement blocks into a working system. In practice, we encountered issues such as unexpected AXI burst behavior, stream stalls, and difficulty achieving II = 1, which forced us to read schedule reports and co-simulation waveforms carefully and to think in terms of timing and hardware structure rather than just C code. Collaborating on module ownership, agreeing on interface contracts, and debugging integration together gave us concrete experience in co-designing and validating a non-trivial hardware system as a team.

Finally, we would like to sincerely thank Professor Jiin Lai for his invaluable guidance and support throughout this project.