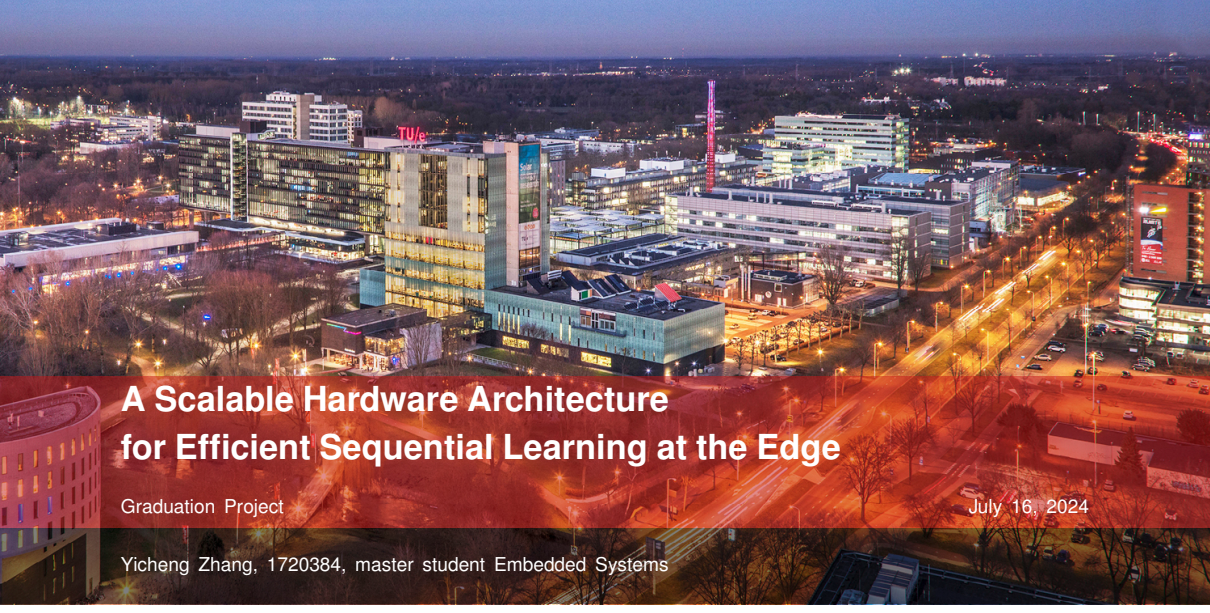


A Scalable Hardware Architecture for Efficient Sequential Learning at the Edge

Graduation Project

July 16, 2024

Yicheng Zhang, 1720384, master student Embedded Systems

Department of Electrical Engineering, Electronic Systems Group

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Introduction: Learning at the Edge

Edge Learning, which means the AI model learns on edge devices, is a solid need.

Why more learning? AI models are expected to perform better by

- Personalization for the individual user;
- Continuous adaptation to data collected from the ever-changing environment; ...

Why not still GPU? It is feasible, however

- Privacy, Latency, Energy;
- Under-utilization; ...

Learning locally on edge devices is a safe, prompt and efficient way to make AI models perform better.

Introduction: Sequential Learning

Definition

AI models learn the sequence of data points of a specific order.

e.g. speech commands, machine translation, weather forecasting, ...

A particular challenge of Sequential Learning at the Edge with Recurrent Neural Networks by traditional BPTT algorithm:

Memory Footprint

Problem Statement

A novel RNN training algorithm is promising for sequential learning at the edge:

FORWARD PROPAGATION THROUGH TIME (FPTT)

Prospects

- Less memory footprint - online formulation
- Comparable performance to BPTT

Concerns

- Inherits computations from BPTT
- Extra operations - regularization term

No edge hardware architecture has been proposed to accelerate FPTT.

Research Questions

Objective: Efficient **hardware acceleration** of FPTT for edge Sequential Learning

RQ-1: What is the computation flow of the FPTT-based training process?

i.e., neural network to use, forward/backward pass, optimizer, bottleneck

RQ-2: How to efficiently accelerate FPTT computations by designing an edge hardware architecture?

i.e., type of architecture, optimization, scalability

Literature Review

Researchers are focusing on two challenges:

- Latency of heavy computation load
- Memory footprint can not fit

through these techniques:

- Customize hardware for computation
- Exploit parallelism
- Skip unnecessary computations
- Trade more computations for less memory footprint
- Use compact datatypes

Methodology Overview

- Computation Design
  - Brief Introduction to FPTT
  - Neural Network Choice and Implementation
  - Training Process
  - Validation and Profiling

- Hardware Acceleration Design
  - Architecture Proposal
  - HW/SW Co-design Flow
  - Customization 1: Compression
  - Customization 2: System Scaleup

1 - Understanding FPTT

Forward Propagation Through Time is a training algorithm based on BPTT for Recurrent Neural Networks, which intuitively implements **sequence partition**, inspired by

- Online Gradient Descent (OGD),
- Follow-The-Regularized-Leader (FTRL),
- Truncated Back Propagation Through Time (T-BPTT),

1 - Understanding FPTT

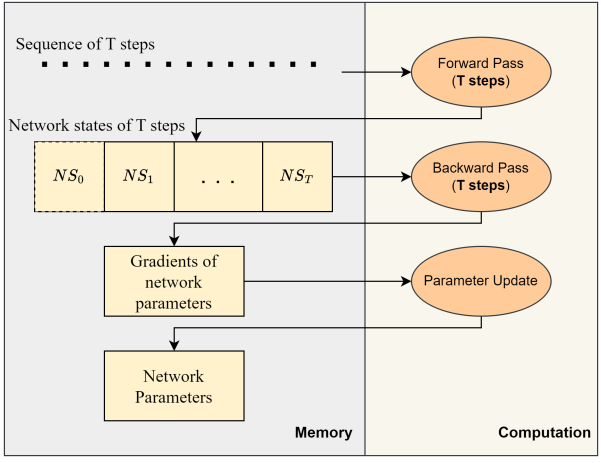


Figure: Visualization of training by BPTT

TU/e

1 - Understanding FPTT

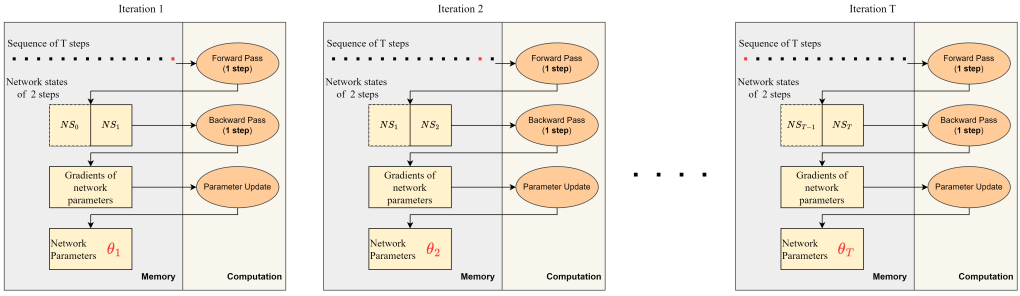


Figure: Visualization of training by FPTT

Network parameter θ is updated very frequently: $\theta_1, \theta_2, \theta_3, ..., \theta_T$
$\rightarrow$ Stability?

1 - Understanding FPTT

$$L_{(t)} = l(y_{(t)}, \overline{y_{(t)}}) + R_{(t)}$$

Including $R_{(t)}$ into the loss function stabilize the trace $[\theta_1, \theta_2, \theta_3, ..., \theta_T]$,
by penalizing the drastic difference between θ and $\overline{\theta}$

$$R_{(t)} = \frac{\alpha}{2} ||\theta - \overline{\theta}_{(t)} - \frac{1}{2\alpha} \nabla_{l_{t-1}}(\theta_{(t)})||^2 \quad (1)$$

α: hyper-parameter
θ: network parameter, i.e., the weight or the biases
$\overline{\theta}$: running average of θ
$\nabla_{l_{t-1}}(\theta_{(t)})$: a correction term, will be approximated by a running estimate $\lambda_{(t)}$

More Granularities:

Definition

K, being the partition factor, means a sequence of T is partitioned into K parts, each has a stride of $\frac{T}{K}$. (ignoring cases with reminders)

e.g. T=784, K=112, $\frac{T}{K} = 7$: partition a sequence of 784 steps into 112 parts, each has 7 steps

Trade more memory footprint for less parameter update.

Summary

Forward Propagation Through Time is a training algorithm based on BPTT for Recurrent Neural Networks, which intuitively implements **sequence partition**, inspired by

- Online Gradient Descent (OGD) $\rightarrow$ interleaved forward and backward pass
- Follow-The-Regularized-Leader (FTRL) $\rightarrow$ regularization term stabilizes $\theta_{(t)}$
- Truncated BPTT $\rightarrow$ back propagation within the current stride

2 - Neural Network Choice and Implementation

FPTT is proved to be effective on Recurrent Neural Networks with gate structure, especially Long Short-Term Memory (LSTM).

Two-layer structure:

LSTM layer - Fully Connected (FC) Layer

for terminal prediction problems.

2 - Neural Network Choice and Implementation

$$xc_{(t)} = [x_{(t)}, h^{l1}_{(t-1)}] \quad (2)$$

$$W^{\star}_{(t)} = [W^{x\star}_{(t)}, W^{h\star}_{(t)}] \quad (\star = i, f, g, o) \quad (3)$$

$$g_{(t)} = tanh(g\_input_{(t)}); \quad g\_input_{(t)} = \mathbf{W}^{g}_{(t)} \cdot xc_{(t)} + b^{g}_{(t)} \quad (4)$$

$$i_{(t)} = sigmoid(i\_input_{(t)}); \quad i\_input_{(t)} = \mathbf{W}^{i}_{(t)} \cdot xc_{(t)} + b^{i}_{(t)} \quad (5)$$

$$f_{(t)} = sigmoid(f\_input_{(t)}); \quad f\_input_{(t)} = \mathbf{W}^{f}_{(t)} \cdot xc_{(t)} + b^{f}_{(t)} \quad (6)$$

$$o_{(t)} = sigmoid(o\_input_{(t)}); \quad o\_input_{(t)} = \mathbf{W}^{o}_{(t)} \cdot xc_{(t)} + b^{o}_{(t)} \quad (7)$$

$$s_{(t)} = g_{(t)} * i_{(t)} + s_{(t-1)} * f_{(t)} \quad (8)$$

$$h^{l1}_{(t)} = tanh(s_{(t)}) * o_{(t)} \quad (9)$$

$$h^{l2}_{(t)} = \mathbf{W}^{l2}_{(t)} \cdot h^{l1}_{(t)} + b^{l2}_{(t)} \quad (10)$$

$$\hat{y}_{(t)} = softmax(h^{l2}_{(t)}) \quad (11)$$

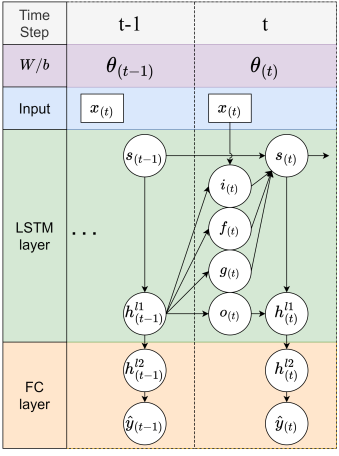


Figure: Network Forward Pass

2 - Neural Network Choice and Implementation

Backward Pass - $l'_{(t)}$ and $R'_{(t)}$

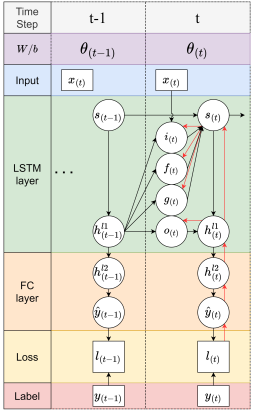


Figure: Network backward Pass for $l'_{(t)}$

$$R_{(t)} = \frac{\alpha}{2}||\theta - \overline{\theta}_{(t)} - \frac{1}{2\alpha}\nabla_{l_{t-1}}(\theta_{(t)})||^2 \quad (12)$$

$$R'_{(t)} = \alpha(\theta - \overline{\theta}_{(t)}) - \frac{1}{2}\nabla_{l_{t-1}}(\theta_{(t)}) \quad (13)$$

$$R'_{(t)} = \alpha(\theta - \overline{\theta}_{(t)}) - \frac{1}{2}\lambda_{(t)} \quad (14)$$

$$R'_{(t)}|_{\theta=\theta_{(t)}} = \alpha(\theta_{(t)} - \overline{\theta}_{(t)}) - \frac{1}{2}\lambda_{(t)} \quad (15)$$

$$\theta_{(t+1)} = \theta_{(t)} - \eta(l'_{(t)} + R'_{(t)}) \tag{16}$$

$$\lambda_{(t+1)} = \lambda_{(t)} - \alpha(\theta_{(t+1)} - \overline{\theta}_{(t)}) \tag{17}$$

$$\overline{\theta}_{(t+1)} = \frac{1}{2}(\overline{\theta}_{(t)} + \theta_{(t+1)}) - \frac{1}{2\alpha}\lambda_{(t+1)} \tag{18}$$

3 - Training Process
Integrate all components

Input: Data samples of S-MNIST, $B = \{x^i, y^i\}_{i=1}^{4}$, Timesteps $T = 784$, network size 128x10 (LSTMxFC)
Input: Learning rate η, Hyper-parameter α
Input: Partition factor set $KS = \{1, 2, 7, 14, 28, 56, 112, 392, 784\}$
Input: the truncation of h^{l1}: *trunc_h*
Input: the truncation of *s*: *trunc_s*
Initialize: $K \in KS$, $stride = \frac{T}{K}$
Initialize: θ_1 randomly in the domain θ
Initialize: $\overline{\theta_1} = \theta_1$ $\lambda_1 = 0$
Initialize: $h^{l1}_{(0)}$, $s_{(0)}$ to zeroes
for $p = 1$ **to** K **do**
    $subsequence = \{x^i\}_{i=1}^{4}$ **from** $(p-1) \cdot stride + 1$ **to** $p \cdot stride$
    NetworkStates = Forward(θ_p, *subsequence*, *NetworkStates*)
    NormalGrad, $R'_{(t)}$ = Backward(θ_p, λ_p, $\overline{\theta_p}$, $\{y^i\}_{i=1}^{4}$, *NetworkStates*, *trunc_h*, *trunc_s*) $(t = p \cdot stride)$
    $\theta_{(p+1)}, \lambda_{(p+1)}, \overline{\theta}_{(p+1)}$ = ParameterUpdate(*NormalGrad*, $R'_{(t)}, \theta_{(p)}, \lambda_{(p)}, \overline{\theta}_{(p)}$) $(t = p \cdot stride)$
end for
Return: $\theta_{(K+1)}$, $CrossEntropy(\hat{y}_{(T)}, y_{(T)})$

3 - Training Process

Type		Function Name
MatMul[1]	regular	mat_mul
	A transpose	mat_mul_a_T
	B transpose	mat_mul_b_T
EWO[2]	basic	element_wise_mac
		element_wise_mul
		element_wise_sub
	compound	find_ds
		find_d_l1_ifgo_input
	FPTT	FPTT_SGD
Activation[3]	sigmoid	sigmoid_on_matrix
	tanh	tanhf_on_matrix
	softmax	softmax
Data Movement		load_sub_seq_to_xc
		relay_network_states
		fill_l1_h_into_xc
Matrix Squeeze		mat2vec_avr_squeeze

[1] Matrix Multiplications
[2] Element-wise Operations (linear) between matrices
[3] are also element-wise operations, though being non-linear

4 - Validation and Profiling

Table: Accuracy on MNIST by designed network

Network Version	streaming by rows	streaming by pixels
self-designed	94%	78%
PyTorch	98%	88%
Gap	4%	10%

Table: Percentages taken by most time-consuming function types for nine K values [1]

Function Type	K-001	K-002	K-007	K-014	K-028	K-056	K-112	K-392	K-784
MatMul	98.58	98.33	99.07	97.26	95.31	95.7	95.36	80.13	70.2
EWO & Activation	0.6	1.5	0.55	2.4	4.39	4.37	4.7	19.94	28.44
Others	<1								

[1] Due to the up rounding by `gprof`, the sum of components may be slightly greater than 100%

Computation Design

Summary

- FPTT implements sequence partition
- 2-layer network LSTM - FC: forward, backward and parameter update
- Training Process with scalable partition factor K
- Dominance in computations: Matrix Multiplications and Element-wise operations

Methodology Overview

- Computation Design
  - Brief Introduction to FPTT
  - Neural Network Choice and Implementation
  - Training Process
  - Validation and Profiling

- Hardware Acceleration Design
  - Architecture Proposal
  - HW/SW Co-design Flow
  - Customization 1: Compression
  - Customization 2: System Scaleup

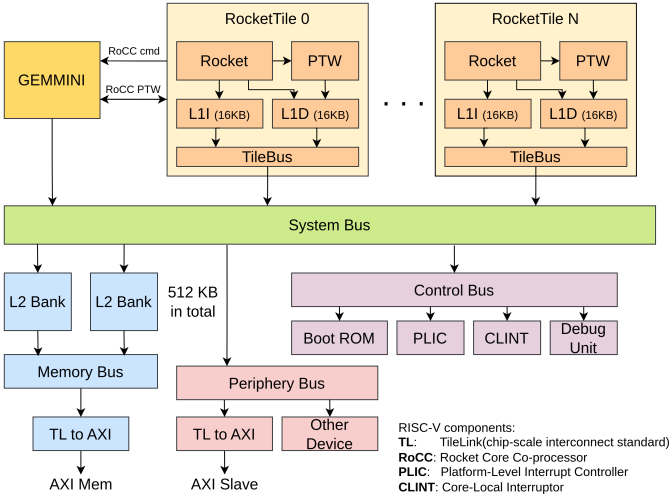
5 - System Proposal

To accelerate FPTT computations, we use the **Chipyard** framework to perform agile SoC development.

- Matrix Multiplications (MM) $\rightarrow$ A MM accelerator, Gemmini
- Element-wise Operations $\rightarrow$ Mutliple RISC-V CPU cores, Rocket Core

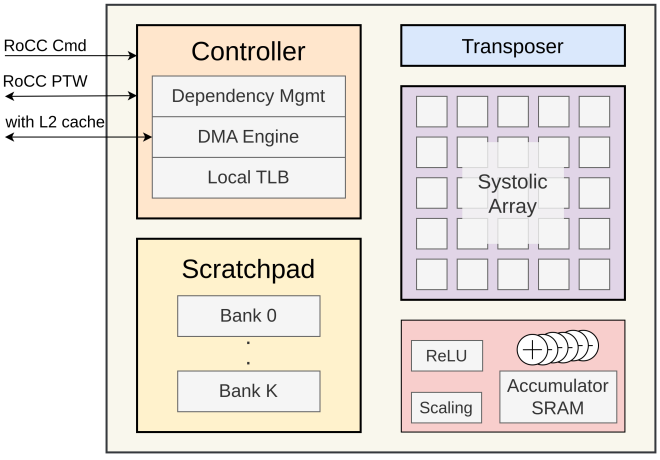
considering efficiency, flexibility and feasibility.

5 - System Proposal



5 - System Proposal

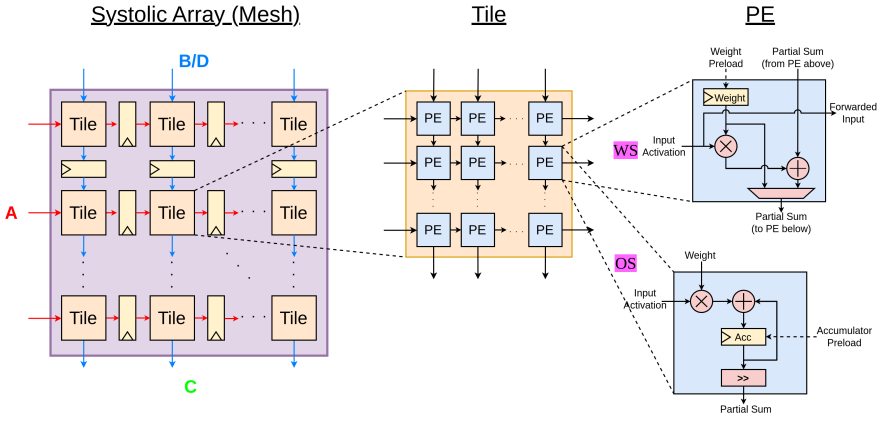
Gemmini Architecture



TU/e

5 - System Proposal

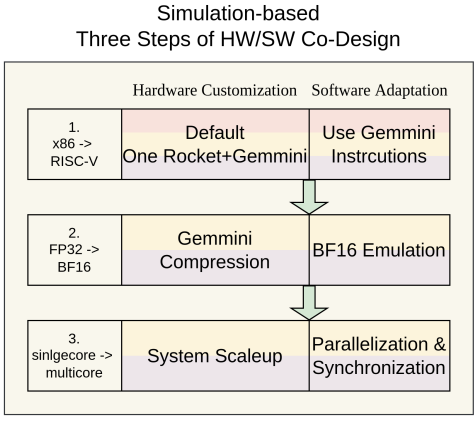
$$C = A \cdot B + D$$



TU/e

6 - HW/SW Co-design Flow

To manipulate, optimize and explore the architecture:



Instruction Simulation **Spike**

RTL Simulation **Verilator**

FPGA-Accelerated Simulation **FireSim**

Specialization → Efficiency → Scalability

7 - Gemmini Architecture Compression

Table: Hardware parameter change for Gemmini architecture compression

Division	Parameter	Default		Custom BF16
		FP32	BF16	
Arithmetic	inputType	Float(8,24)	Float(8,8)	Float(8,8)
	spatialArrayOutputType	Float(8,24)	Float(8,8)	Float(8,8)
	accType	Float(8,24)		Float(8,8)
Controllers	ex_read_from_acc	true		false
	ex_write_to_spad	true		false
Data Scaling	mvin_scale_args.multiplicand_t	Float(8,24)		Float(8,8)
	mvin_scale_acc_args.multiplicand_t	Float(8,24)		Float(8,8)
	acc_scale_args.multiplicand_t	Float(8,24)		Float(8,8)
Systolic Array	dataflow	Dataflow.BOTH		Dataflow.WS
Scratchpad	sp_capacity	256 KB		128 KB
Accumulator	acc_capacity	64 KB		32 KB
Periphery	has_max_pool	true		false
	has_dw_convs	true		false
	has_first_layer_optimizations	true		false

8 - System Scaleup

Table: Design points

Index	$SIZE_{mesh} - N_{cores}$
1	4x4-1
2	4x4-2
3	4x4-4
4	4x4-6
5	4x4-8
6	4x4-10
7	8x8-1
8	8x8-2
9	8x8-4
10	8x8-6
11	8x8-8
12	8x8-10

Programming

Programming methods in parallel to hardware changes:

- ISA switching: C libraries of Gemmini and RISC-V
- Conversion between FP32 and BF16: bit shifting
- Parallelism: core ID-based code allocation
- Synchronization: barrier

Results - Gemmini Compression
Resource Utilization & Precision

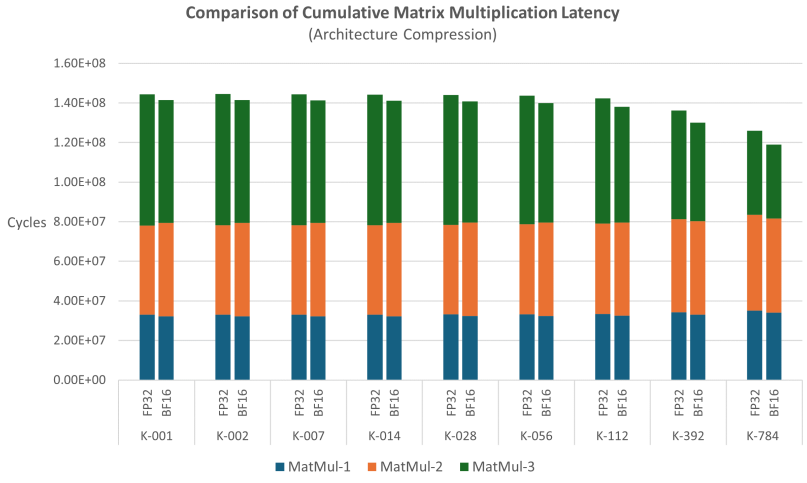
Table: Comparison of Gemmini Resource Utilization

Architecture	LUT	Register	BRAM	DSP
default-FP32	94969	31067	80	256
custom-BF16	42022	23818	40	120
change	-56%	-23%	-50%	-61%

Table: Comparison of Loss Function at final time step

Type	K-001	K-002	K-007	K-014	K-028	K-056	K-112	K-392	K-784
FP32	0.058755	0.058257	0.057592	0.056168	0.053817	0.051874	0.044308	0.027446	0.019935
BF16	0.071089	0.062993	0.070962	0.055088	0.079186	0.047183	0.071025	0.023592	0.023592

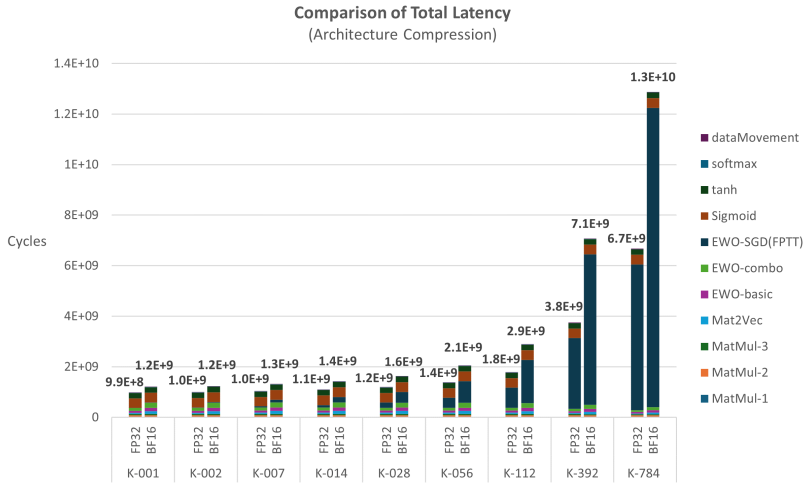
Results - Gemmini Compression
Matrix Multiplication Latency



TU/e

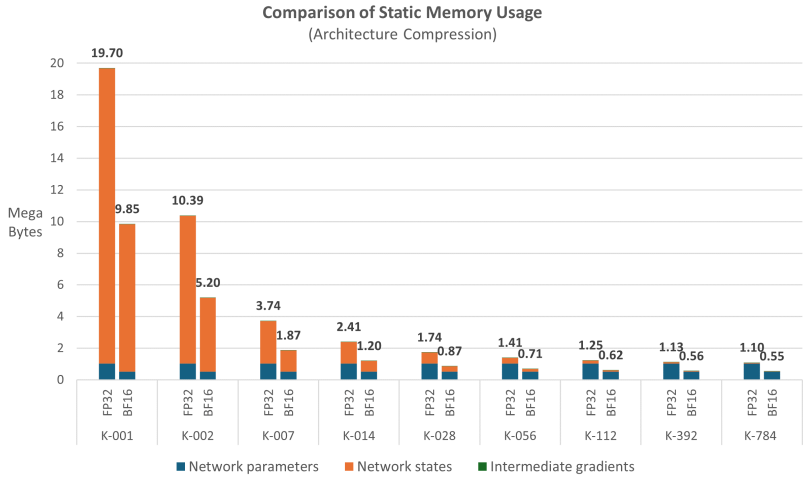
Results - Gemmini Compression

Total Latency - 20% to 50% conversion overhead in latency



TU/e

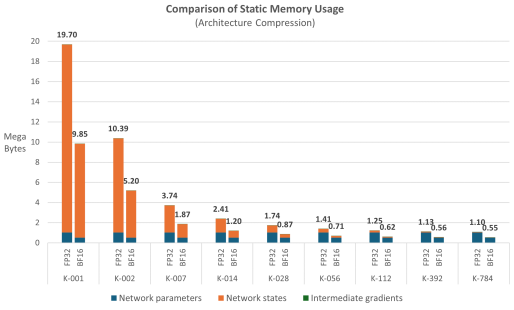
Results - Gemmini Compression
Memory - Halved Memory footprint

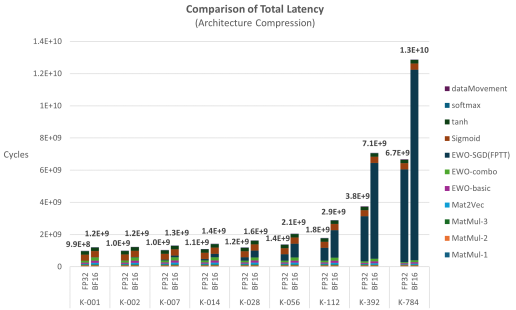


TU/e

Results - Gemmini Compression
trade-off in partition factors

K-028, compared with K-001, brings $\frac{19.7}{0.87} \approx 22.6$ times save in memory footprint, at the cost of $\frac{1.6}{0.99} \approx 1.6$ times total latency, including the conversion overhead





Results - Gemmini Compression

Summary

- 50% decrease in resource utilization
- Very close trainability (Cross-Entropy Loss)
- Negligible decrease in matrix multiplication latency
- About 20% to 50% increase in total latency (conversion overhead BF16-FP32)
- Halved Memory footprint

Results - System Scaleup

Table: Resource Utilization of Design Points

$SIZE_{mesh}$-N_{core}	LUT		Register		BRAM		DSP	
	Abs	Increase	Abs	Increase	Abs	Increase	Abs	Increase
4x4-1	86235	**1.0**	46766	**1.0**	168	**1.0**	135	**1.0**
4x4-2	115257	**1.3**	60224	**1.3**	180	**1.1**	150	**1.1**
4x4-4	172876	**2.0**	87117	**1.9**	196	**1.2**	180	**1.3**
4x4-6	230855	**2.7**	114008	**2.4**	212	**1.3**	210	**1.6**
4x4-8	288814	**3.3**	140893	**3.0**	228	**1.4**	240	**1.8**
4x4-10	346827	**4.0**	167839	**3.6**	244	**1.5**	270	**2.0**
8x8-1	123864	**1.4**	61105	**1.3**	168	**1.0**	135	**1.0**
8x8-2	152860	**1.8**	74561	**1.6**	180	**1.1**	150	**1.1**
8x8-4	210569	**2.4**	101454	**2.2**	196	**1.2**	180	**1.3**
8x8-6	268190	**3.1**	128347	**2.7**	212	**1.3**	210	**1.6**
8x8-8	326148	**3.8**	155229	**3.3**	228	**1.4**	240	**1.8**
8x8-10	384080	**4.5**	182179	**3.9**	244	**1.5**	270	**2.0**

Results - System Scaleup

Table: Workload Latency in seconds by design points (assuming 500MHz)

$SIZE_{mesh}$-N_{core}	K-001	K-002	K-007	K-014	K-028	K-056	K-112	K-392	K-784
4x4-1	2.47	2.50	2.68	2.89	3.30	4.15	5.82	14.17	25.77
4x4-2	1.71	1.74	1.88	2.05	2.38	2.66	3.60	8.02	13.96
4x4-4	1.15	1.16	1.21	1.27	1.39	1.66	2.13	4.65	7.86
4x4-6	0.91	0.91	0.95	0.99	1.07	1.28	1.59	3.30	5.57
4x4-8	0.80	0.80	0.83	0.86	0.93	1.11	1.33	2.70	4.46
4x4-10	0.73	0.74	0.76	0.79	0.84	1.01	1.18	2.33	3.84
8x8-1	2.32	2.36	2.54	2.75	3.16	4.00	5.68	14.05	25.66
8x8-2	1.57	1.60	1.72	1.90	2.23	2.52	3.47	7.89	13.84
8x8-4	1.00	1.01	1.06	1.12	1.24	1.52	1.99	4.51	7.74
8x8-6	0.76	0.77	0.80	0.84	0.93	1.14	1.45	3.18	5.46
8x8-8	0.66	0.66	0.69	0.72	0.79	0.97	1.20	2.58	4.37
8x8-10	0.59	0.60	0.62	0.65	0.70	0.86	1.05	2.20	3.72
Max Speedup	4.19	4.20	4.34	4.47	4.70	4.80	5.55	6.45	6.93

Results - System Scaleup

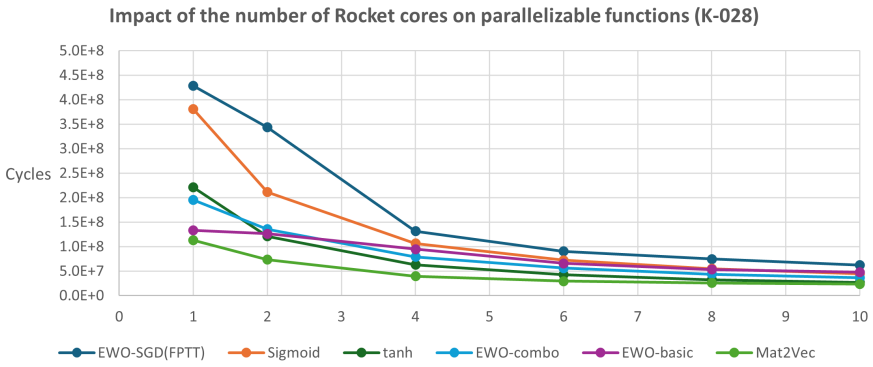


Figure: Impact of the number of Rocket cores on parallelizable functions

Results - System Scaleup

Summary

- 4-7 times speedup on the workload of various partition factors (computation intensity)
- at the cost of up to 4.5 times increase in resource utilization

Conclusion

In summary, we

- Figure out the computations of the FPTT-based training process
- Propose a scalable architecture of one Accelerator + Mutlicores for acceleration, leveraging Chipyard framework
- Tweaking the microarchitecture for efficiency and scalability