



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria - DISI

Master's Degree in Computer Engineering

Spiking Transformer Acceleration on RISC-V Streaming Processors

Supervisor:
Prof. Andrea Bartolini

Presented By:
Marco Crisafulli

Co-Supervisors:
Dr. Simone Manoni
Hiari Pizzini Cavagna

October Session 2026
Academic Year 2025/2026

Abstract

Transformers built on conventional *Artificial Neural Networks* have represented the key innovation leading to the powerful AI models we have access to today. *Spiking Neural Networks* offer a more energy-efficient option than standard neural networks because the neurons inside such networks exchange spikes, which are binary signals. Recent work has shown that spiking Transformers obtain strong results thanks to sparse binary computation and generally present better energy efficiency. A consequence of operating with sparse binary spike tensors is, for instance, the elimination of multiplication, that can be replaced by addition and mask operations. However, this sparsity is unstructured and known only at run time. Dedicated neuromorphic chips exploit it but are tied to specific models, and despite it, CPUs and GPUs execute spike tensors as dense operations.

In this work, we profile the attention module of the *Spiking Transformer with Spatial-Temporal Attention* on CIFAR-100 inputs and find its query, key and value spike tensors are about 96.87% sparse. These findings allow us to design accelerated kernels for the Snitch cluster, an eight-core RISC-V cluster with hardware streams for indirect accesses and vectorization support. They cover every operation of the attention block, and we evaluate them in cycle-accurate simulation, both individually and chained into the complete block.

Over a plain-C reference, the *Sparse Matrix-Vector Addition* outperforms the baseline for up to 34.02× at fp16x4. At fp16x4, the sparse engine is faster than a dense matrix multiplication of the same lane width up to a density of 0.75, and slower above it.

A programmable streaming cluster can thus turn the sparsity of a spiking Transformer into measured speedups without fixing the model in hardware, so it can follow the model as it evolves.

Contents

Abstract	i
List of Figures	vii
List of Tables	ix
Nomenclature	xi
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	3
2 Background	5
2.1 Spiking Neural Networks	5
2.2 The LIF Neuron	6
2.3 Transformer Architectures	7
2.4 Spiking Transformers	8
2.5 RISC-V Architecture	9
2.6 RISC-V Streaming Processors: the Snitch Cluster	10
2.7 SIMD, MIMD and Vectorization	12
2.8 The Challenge of Exploiting Sparsity	13
2.9 Sparse Data Structures and Energy Efficiency	15
3 Methodology	17
3.1 Profiling STAtten	17
3.1.1 Profiling Methodology	17
3.1.2 Model Architecture	17
3.1.3 Profiling Results	18
3.2 Implementation Details	20
3.2.1 Data Generation and Vectorization	20
3.3 Kernels	20
3.3.1 Sparse Matrix-Vector Addition (SpMVA)	21
3.3.2 Convolution: from SpMVA to SpMMA	25

3.3.3	Convolution: Indirect SpMMA	28
3.3.4	Batch Normalization (BN)	31
3.3.5	Fused Kernels	33
3.3.6	LIF Neuron Update	35
3.3.7	View, Reshape, and Sparse Compaction	37
3.3.8	A Fused Attention Sub-Block	42
3.3.9	The Attention Block Application	44
4	Results and Discussion	47
4.1	Measurements	47
4.1.1	Cycles and Speedup	47
4.1.2	Core Scaling	47
4.1.3	Software and Hardware Setup	47
4.1.4	Tensor Sizes	48
4.2	Plots	50
4.2.1	SpMVA: Speedup Against Sparsity and FPU Utilization	50
4.2.2	SpMVA: Core Scaling	53
4.2.3	Convolution: SpMMA Speedup and FPU Utilization	55
4.2.4	Convolution: Core Scaling	58
4.2.5	Batch Normalization: Speedup Against Sparsity	61
4.2.6	Batch Normalization: Core Scaling	62
4.2.7	LIF: Speedup and FPU Occupancy Across Lane Widths	65
4.2.8	LIF: Core Scaling	66
4.2.9	Reshape: View Materialization	67
4.2.10	Reshape: Sparse Compaction	69
4.2.11	Attention Block	71
5	Conclusions	75
5.1	Summary of Contributions	75
5.2	Limitations and Threats to Validity	76
5.3	Future Work	78
5.3.1	Short-Term Directions	78
5.3.2	Long-Term Research Vision	79
Appendix A	Development Environment	81
A.1	Code Repositories	81
Appendix B	Kernel Selection and Verification	83
B.1	Configuration and Data Generation	83
B.2	Compile-Time Kernel Selection	83

B.3 Verification	84
Bibliography	B1

List of Figures

2.1	Action potential and membrane conductance of the squid giant axon, as computed by Hodgkin and Huxley [1]	6
2.2	The LIF neuron abstraction [2]	6
2.3	Transformer architecture [3]	7
2.4	<i>Vanilla Self-Attention</i> (VSA) and SDSA [4]	9
2.5	The STAtten mechanism [5]	10
2.6	Read-after-write dependency between two instructions	13
2.7	CSR format of a sparse matrix [6]	15
3.1	STAtten architecture reconstructed from the released code	18
3.2	STAtten attention module with tensor shapes for a CIFAR-100 input	18
3.3	Sparsity of <code>attention_QKV</code> and <code>attention_QKV_postLIF</code> against timestep and encoder execution, CIFAR-10 image #202	19
3.4	Instrumented tensors of the SDSA module and their mean sparsity	19
3.5	One CSR row of $A \cdot B$: general SpMV against binary SpMVA	21
3.6	Interleaved column layout of B for fp32x2 and fp16x4	24
3.7	One row of $C = A \cdot B$ at fp64, fp32x2 and fp16x4	26
3.8	Classic 3×3 convolution against 1×1 convolution	26
3.9	Offline prescaling of the CSR column index, <code>indir</code> against <code>spmva</code>	28
3.10	The <code>indir</code> engine on one active channel	31
3.11	Folding BN into two per-channel coefficients	32
3.12	Dense and CSR BN kernel bodies	34
3.13	Hard-reset LIF update computed from sign bits	36
3.14	A 2×3 view and its permutation	38
3.15	<code>sla_to_spike_csr_fp16x4</code> on one row of a packed fp16x4 window	40
3.16	<code>sla_contiguous</code> materializing a permuted view	40
3.17	<code>sla_to_sparse_csr</code> on a 2×3 example	41
4.1	fp64 SpMVA speedup over plain C and FPU utilization against input density	51
4.2	fp32x2 SpMVA speedup per output column over plain C and FPU utilization against input density	51
4.3	fp16x4 SpMVA speedup per output column over plain C and FPU utilization against input density	52

4.4	fp64 SpMVA speedup over one core against core count	54
4.5	fp32x2 SpMVA speedup over one core against core count	54
4.6	fp16x4 SpMVA speedup over one core against core count	55
4.7	1 × 1 convolution speedup of <code>spmva</code> and <code>indir</code> over a dense GEMM and FPU utilization against input density	56
4.8	Speedup of <code>spmva</code> over dense GEMM and FPU utilization against input density	58
4.9	fp64 conv1x1 speedup over one core against core count	59
4.10	fp32x2 conv1x1 speedup over one core against core count	60
4.11	fp16x4 conv1x1 speedup over one core against core count	60
4.12	BN speedup of the CSR kernel over the dense apply against input density	63
4.13	BN speedup over one core against core count	64
4.14	LIF cycles, speedup over fp64 and over fp32x2, and FPU issue occupancy . . .	65
4.15	LIF cycles against neuron count with least-squares fit	66
4.16	LIF speedup over one core against core count	67
4.17	fp64 cycles per element of <code>sla_to_sparse_csr</code> against input density . . .	70
4.18	Cycles per element of the fp16x4 spike compaction and of <code>sla_to_sparse_</code> <code>csr</code> against input density	70
4.19	Stages of the accelerated STAtten attention block	72
4.20	Accelerated attention block execution on 8 cores fp16x4	73

List of Tables

3.1	Sparsity (%) of the instrumented encoder tensors against encoder execution . . .	20
3.2	Arguments of the fp64 ISSR-based SpMVA kernel	22
3.3	Binding of the SpMVA arguments in the <code>conv_dm_*</code> loop, iteration w	27
3.4	Kernels and support routines of the 1×1 convolution	29
3.5	BN kernel variants by role	33
3.6	Variants of the LIF kernel	36
3.7	Shape operations ported from PyTorch	41
3.8	Stages of the fused attention sub-block	43
4.1	Tensor shapes used in the kernel evaluation	48
4.2	The nine density points of the generated tensors	49

Nomenclature

Abbreviation	Meaning
AI	<i>Artificial Intelligence</i>
ASIC	<i>Application-Specific Integrated Circuit</i>
BN	<i>Batch Normalization</i>
CPU	<i>Central Processing Unit</i>
CSR	<i>Compressed Sparse Row</i>
DMA	<i>Direct Memory Access</i>
DRAM	<i>Dynamic Random-Access Memory</i>
FIFO	<i>First-In, First-Out</i>
fp16	16-bit (half-precision) floating-point format
fp16x4	four fp16 lanes packed in one 64-bit word
fp32	32-bit (single-precision) floating-point format
fp32x2	two fp32 lanes packed in one 64-bit word
fp64	64-bit (double-precision) floating-point format
FPGA	<i>Field-Programmable Gate Array</i>
FPU	<i>Floating-Point Unit</i>
FREP	<i>Floating-Point Repetition instruction</i>
GEMM	<i>General Matrix Multiplication</i>
GPU	<i>Graphics Processing Unit</i>
HBM	<i>High Bandwidth Memory</i>
ISA	<i>Instruction Set Architecture</i>
kGE	<i>kilo Gate Equivalents</i>
ISSR	<i>Indirection Stream Semantic Register</i>
LIF	<i>Leaky Integrate-and-Fire</i>
MAC	<i>Multiply-Accumulate</i>
MIMD	<i>Multiple Instruction, Multiple Data</i>
MLP	<i>Multi-Layer Perceptron</i>
RAW	<i>Read-After-Write</i>
RISC	<i>Reduced Instruction Set Computer</i>

Abbreviation	Meaning
RTL	<i>Register-Transfer Level</i>
SDSA	<i>Spike-Driven Self-Attention</i>
SIMD	<i>Single Instruction, Multiple Data</i>
SIMT	<i>Single Instruction, Multiple Threads</i>
SNN	<i>Spiking Neural Network</i>
SpMM	<i>Sparse Matrix-Matrix Multiplication</i>
SpMMA	<i>Sparse Matrix-Matrix Addition</i>
SpMV	<i>Sparse Matrix-Vector Multiplication</i>
SpMVA	<i>Sparse Matrix-Vector Addition</i>
SSR	<i>Stream Semantic Register</i>
STAtten	<i>Spiking Transformer with Spatial-Temporal Attention</i>
TCDM	<i>Tightly Coupled Data Memory</i>
VSA	<i>Vanilla Self-Attention</i>
WAR	<i>Write-After-Read</i>
WAW	<i>Write-After-Write</i>

Chapter 1

Introduction

1.1 Motivation

In recent years *Artificial Intelligence* (AI) has moved from research laboratories into everyday use, and conversational assistants such as ChatGPT and Claude are now widely deployed and accessible by everyone. Their capability rests on two factors. The first is the Transformer architecture [3]. The Transformer introduces an attention mechanism, called *self-attention* [3], that relates every element of a sequence to every other one in a single operation, allowing the model to draw on long contexts when producing each output [3, 7]. The second is scale: Kaplan *et al.* [7] show that the performance of Transformer language models improves as a power law of the number of parameters, the size of the dataset and the amount of training compute, and that, for compute-efficient training, most of any increase in compute should go towards larger models.

The consequence is a steadily growing demand for computing resources. Sevilla *et al.* [8] find that the compute used to train milestone machine learning models grew in line with Moore’s law before 2010, doubling roughly every 20 months, and that it has doubled approximately every 6 months since the advent of deep learning. Demand is therefore growing faster than the hardware scaling that historically supplied it, and the resulting models must then be executed every time they serve a request. Further progress thus depends, at least in part, on using the available hardware more efficiently.

One way to improve efficiency is to change how the model itself computes. *Spiking Neural Networks* (SNNs) borrow their computational primitive from biology: instead of exchanging continuous-valued activations, their neurons accumulate incoming current into a membrane potential and emit a discrete event, a spike, only when that potential crosses a threshold. Computation is spread over a sequence of timesteps, and networks built on this principle have reached high accuracy on image classification [9] and object recognition [10] while promising better energy efficiency. The Transformer architecture has also been rebuilt from spiking neurons, and the *Spike-Driven Transformer* [4], presented in Section 2.4, is the model on which this thesis builds. This work addresses inference only, that is, the execution of an already trained network,

and not training. Training is performed once or at intervals, whereas inference runs continuously and at large scale. Recent estimates reported by Jegham *et al.* [11] attribute up to 90% of a model’s total life-cycle energy use to inference.

Because a neuron below its threshold emits nothing, the tensors exchanged between spiking layers are binary and consist mostly of zeros. In principle every operation on a zero operand can be skipped, and this is the source of the efficiency SNNs promise. In practice, skipping these operations has a cost. Whether a neuron fires depends on its state at the previous timestep, so the positions of the nonzeros are known only at run time and change from one input to the next. Locating them requires additional indexing work, and the resulting memory accesses lack the regularity that processor pipelines and caches are designed for. Exploiting sparsity is therefore a trade-off between the arithmetic saved and the indexing overhead introduced, analyzed in Section 2.8.

For this reason, SNNs have so far been executed mostly on dedicated hardware. Neuromorphic *Application-Specific Integrated Circuits* (ASICs), such as IBM’s TrueNorth [12], Intel’s Loihi [13] and SpiNNaker [14], exploit sparsity directly and achieve high energy efficiency, but they are limited to specific models and workloads, which makes them inflexible and prone to obsolescence in a field where models evolve rapidly. Accelerators synthesized on *Field-Programmable Gate Arrays* (FPGAs), such as FireFly [15], Cerebron [16], SyncNN [17] and Spiker [18], are in practice hand-tuned, layer-by-layer, dataflow-mapped implementations of one fixed network topology and neuron model. *Graphics Processing Units* (GPUs), in contrast, remain flexible, but SNNs are mostly executed on them as ordinary dense tensor operations, using frameworks such as SpikingJelly [19], so that operations on zero operands consume cycles like any other. Even the sparsity support added to GPUs is restricted to fixed patterns: the Sparse Tensor Cores of NVIDIA’s Ampere architecture accelerate only structured sparsity [20], a pattern that the unstructured, data-dependent zeros of spike tensors do not follow.

Neither approach meets the needs of a spiking Transformer. Dedicated designs obtain their efficiency by committing to a single model, while GPUs obtain their generality by treating every tensor as dense. What this workload requires is a third option: a programmable platform in which the indexing demanded by sparsity is performed in hardware. RISC-V makes this option practical. It is an open and extensible *Instruction Set Architecture* (ISA), described in Section 2.5, that reserves part of its opcode space for custom instructions [21]. Operations that are frequent in a target domain can therefore be supported directly in hardware without abandoning the base ISA and its compiler toolchain.

The Snitch cluster [22], the platform adopted in this thesis, follows this approach. It groups eight small in-order RISC-V cores, each paired with a comparatively large *Floating-Point Unit* (FPU), around a shared scratchpad memory. Its ISA extensions let the hardware stream operands from memory and repeat loop bodies without executing additional instructions. Scheffler *et al.* [23] extended these streams to indirect accesses, moving the index lookups of sparse linear

algebra into hardware, and Manoni *et al.* [24] applied them to SNN inference with SpikeStream. The architecture of the cluster is described in Section 2.6.

In this work, we study the *Spike-Driven Transformer* with the attention of the *Spiking Transformer with Spatial-Temporal Attention* (STAtten) [5], both presented in Section 2.4. We measure how much of the sparsity of SNNs is retained in this attention, and we design sparse kernels that exploit it for inference on the Snitch cluster. We show that the query, key and value spike tensors and the attention output are more than 96% sparse, that the resulting kernels reach speedups of up to 34.02 $\times$ over a plain-C reference, and that sparse computation outperforms dense computation below a density threshold that we measure.

1.2 Contributions

The main contributions of this thesis, which extend the work of Manoni *et al.* [24] and Scheffler *et al.* [23], can be summarized as follows:

1. **Sparsity Analysis of a State-of-the-Art Spiking Transformer.** Analysis of the sparsity of the tensors inside the attention module of the *Spike-Driven Transformer* with STAtten attention, measured on CIFAR-100 inputs: the query, key and value spike tensors are about 96.87% sparse, the attention output 97.59%, and the output of the *Leaky Integrate-and-Fire* (LIF) neurons that follow it 99.86% (Chapter 3).
2. **Accelerated Computation Kernels for Spiking Attention on a State-of-the-Art RISC-V Platform.** Analysis and implementation, on the Snitch cluster, of kernels that perform the *Sparse Matrix-Vector Addition* (SpMVA), which replaces each multiplication by a binary spike with an addition, implemented in the 1×1 convolutions. Moreover, the kernels implemented cover *Batch Normalization* (BN), the LIF neuron update, and the reshape and *Compressed Sparse Row* (CSR) conversion operations. All of the aforementioned operations are implemented in fp64 and, where applicable, in packed *Single Instruction, Multiple Data* (SIMD) fp32x2 and fp16x4 variants. Leveraging vectorization raises the SpMVA speedup over a plain-C reference from up to 8.50 $\times$ at fp64 to 17.31 $\times$ at fp32x2 and 34.02 $\times$ at fp16x4. Using vector extensions also speeds up the LIF operation by up to 1.95 $\times$ and 3.71 $\times$ when compared to baseline fp64. The biggest speedup registered is 50.30 $\times$, achieved by the `indir` engine of the 1×1 convolution.
3. **Where Sparsity Pays Off.** Measurement, in a cycle-accurate simulation, of the overheads that sparse data structures add, quantified explicitly in the *Sparse Matrix-Vector Addition* (SpMVA) kernels, and implicitly for each kernel that uses CSR format as the cost of conversion to CSR, ranging from 10.07 cycles per element on the sparsest input to 20.65 cycles per element on a fully dense one. Analyzed the density below which the `spmva` engine becomes faster than dense computation. Against a dense matrix multiplication

of the same lane width, the threshold is a density of 0.29 at fp64, 0.39 at fp32x2 and 0.75 at fp16x4 on the 8×96 attention tile with a square weight matrix. The fixed cost of separating the packed outputs is amortized only on sufficiently dense inputs: the net packed SpMVA speedup exceeds $1\times$ at every shape from density 0.1 upward against a plain C implementation of the same kernel (Chapter 4).

The rest of this thesis follows the same order. Chapter 2 introduces the background the later chapters rely on: SNNs and the LIF neuron, self-attention and its spiking variants, the RISC-V ISA and the Snitch cluster, and the cost of exploiting unstructured sparsity on a programmable core. Chapter 3 profiles STAtten to measure the sparsity of its tensors, and derives from those measurements the kernels developed in this work. Chapter 4 evaluates those kernels across the tensor shapes and densities the profiling suggested, including the cases where they lose to a dense implementation, and Chapter 5 draws the conclusions and the limitations of this work.

Chapter 2

Background

Chapter 1 introduced spiking Transformers and the Snitch cluster only as far as needed to state the problem. This chapter provides the background on which the rest of the thesis relies. It begins with SNNs (Section 2.1) and the LIF neuron (Section 2.2), since the sparsity exploited in this thesis originates in the LIF neuron model. It then presents the Transformer and its self-attention (Section 2.3), followed by the spiking Transformers built on it (Section 2.4): the *Spike-Driven Transformer* and STAtten, the workload under study, whose self-attention is where that sparsity concentrates. The hardware side follows: RISC-V as the ISA (Section 2.5), the Snitch cluster as a RISC-V streaming processor (Section 2.6), and the SIMD and *Multiple Instruction, Multiple Data* (MIMD) execution models (Section 2.7), which together explain how a programmable core can be organized to exploit unstructured sparsity. The chapter closes with the challenges of exploiting sparsity (Section 2.8) and with the CSR format (Section 2.9).

2.1 Spiking Neural Networks

SNNs originate in the study of biological neurons and of the mechanisms through which they communicate. The mathematical description of spiking behavior goes back to Hodgkin and Huxley [1], whose 1952 model of the squid giant axon reproduced the generation and propagation of action potentials from the underlying ionic currents (Figure 2.1). The model is highly accurate but computationally expensive, since it requires the numerical integration of four coupled nonlinear differential equations per neuron. This motivated a long line of simplified neuron models that trade some biological detail for tractability. The most notable are the LIF neuron, sketched in Figure 2.2, which reduces neuronal dynamics to a single first-order differential equation, and the more expressive model of Izhikevich [25], which reproduces a wide range of observed firing patterns at a computational cost comparable to that of the LIF model. Building on these neuron models, Maass [26] formally argued that networks of spiking neurons constitute a *third generation* of neural network models, distinct from the perceptron-based first generation and from the sigmoidal, rate-based second generation to which most modern deep

artificial neural networks belong. Maass also showed that they are computationally at least as powerful, while operating on fundamentally different principles.

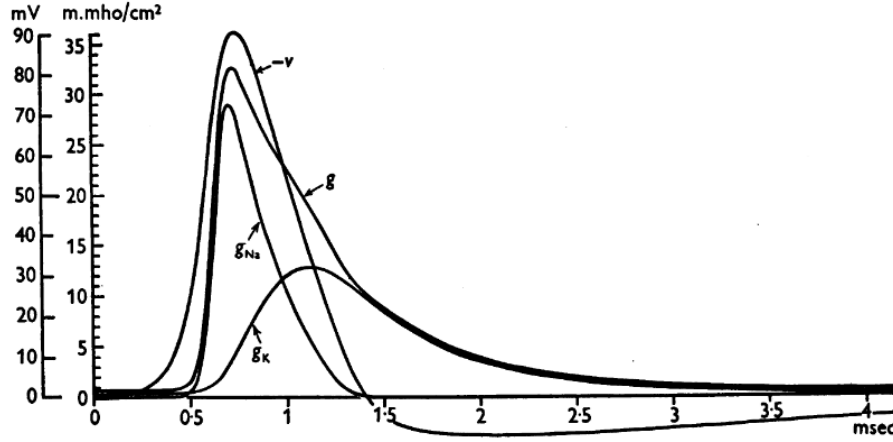


Figure 2.1: Action potential and membrane conductance of the squid giant axon, as computed by Hodgkin and Huxley [1]

Spiking neurons communicate through discrete, asynchronous events called *spikes* (or *action potentials*), whose timing and rate carry the information the network processes. A spiking neuron integrates its input over time in an internal state variable, generally referred to as its *membrane potential*. Only when this potential crosses a threshold does the neuron emit a spike and reset, and no further information is exchanged until the next spike. SNNs are therefore inherently temporal, event-driven and sparse: a neuron that never crosses its threshold consumes no output bandwidth and, on suitable hardware, negligible energy. An SNN thus acts as a dynamic mapping between input and output spike trains, characterized by a set of synaptic weights learned through training.

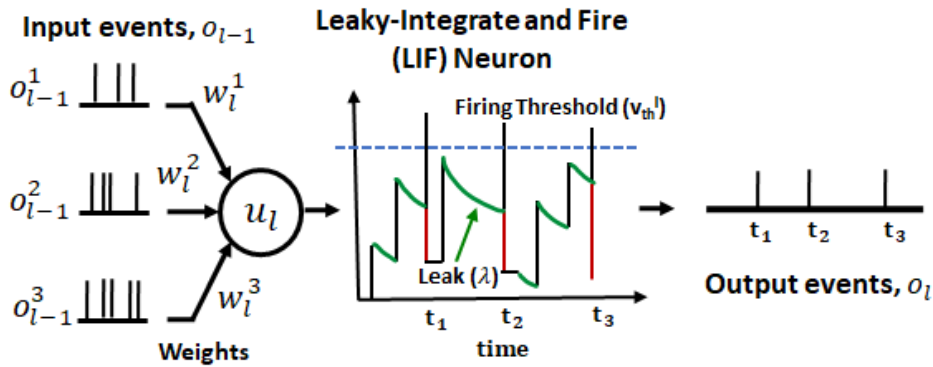


Figure 2.2: The LIF neuron abstraction [2]

2.2 The LIF Neuron

Among the simplified neuron models of Section 2.1, this thesis relies on the LIF model, the neuron model of the spiking Transformers studied here. The model is called “leaky” because the

membrane potential decays over time at a rate set by a constant, and “integrate-and-fire” because the neuron integrates incoming spikes over time and fires once the threshold is reached.

Following the formulation of Lee *et al.* [5], the membrane potential of a LIF neuron in layer l evolves at each timestep t as

$$\mathbf{u}[t+1]^l = \tau \mathbf{u}[t]^l + \mathbf{W}^l f(\mathbf{u}[t]^{l-1}), \quad (2.1)$$

$$f(\mathbf{u}[t]^l) = \begin{cases} 1 & \text{if } \mathbf{u}[t]^l > V_{th}, \\ 0 & \text{otherwise,} \end{cases} \quad (2.2)$$

where $\mathbf{u}[t]^l$ is the membrane potential of layer l at timestep t , $\tau \in (0, 1]$ is the leaky factor controlling the decay of the membrane potential, $\mathbf{W}^l$ is the weight matrix of layer l , and $f(\cdot)$ is the LIF firing function with threshold V_{th} . Whenever $\mathbf{u}[t]^l$ exceeds V_{th} , the neuron fires a spike and its membrane potential is reset to zero.

2.3 Transformer Architectures

The Transformer architecture introduced by Vaswani *et al.* [3] is a deep learning model that processes sequential data through self-attention. As Figure 2.3 shows, it consists of a stack of encoder and decoder layers, each of which contains a multi-head self-attention mechanism and a feed-forward layer.

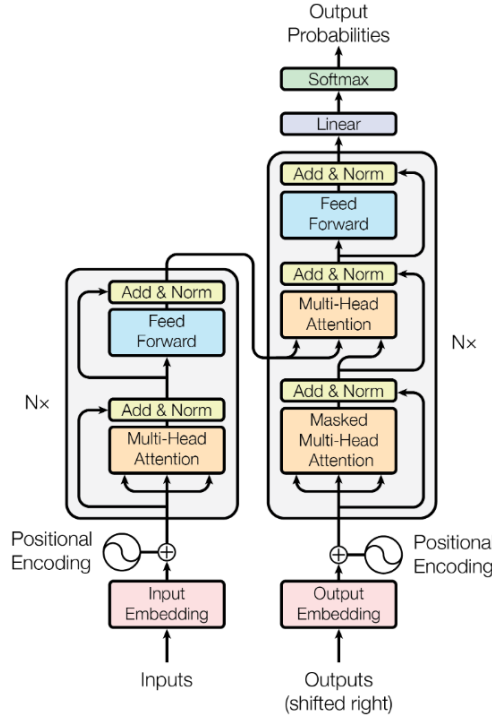


Figure 2.3: Transformer architecture [3]

Inside the self-attention mechanism, the input sequence is transformed into three matrices:

- **Query (Q):** a projection of the current token, encoding the information it seeks in the rest of the sequence.
- **Key (K):** a projection of every token (including the current one), encoding the information each token provides, which is matched against a query through a similarity score (a dot product).
- **Value (V):** a projection of every token carrying its content; the attention output is the sum of these value vectors, weighted by how closely each token's key matches the query.

The attention scores are the dot products of queries and keys. $QK^\top$ is a full matrix multiplication that produces an $N \times N$ similarity map, in which every token has its own row of scores against every other token. The map is normalized with a softmax, which requires exponentials and a per-row division, and the output is the product of the normalized map with V . Both matrix products consist of floating-point *Multiply-Accumulate* (MAC) operations, for a cost of $O(N^2D)$.

2.4 Spiking Transformers

This thesis builds on two spiking versions of self-attention. The first is *Spike-Driven Self-Attention* (SDSA), proposed by Yao *et al.* [4] as the attention of the *Spike-Driven Transformer*; the second is its STAtten extension. This section details how they compute attention, since that computation is what the kernels of Chapter 3 implement.

In SDSA, Q , K and V are binary spike tensors, each produced by a linear projection followed by a spiking (LIF) neuron: $Q, K, V = \text{LIF}(\mathbf{W}_Q X), \text{LIF}(\mathbf{W}_K X), \text{LIF}(\mathbf{W}_V X)$. SDSA does not compute the matrix multiplication $QK^\top$. It computes instead the Hadamard (element-wise) product of the spike tensors Q and K , sums the result column-wise, and passes that sum through a further spiking neuron. Because Q and K are binary, the column-wise sum collapses the token dimension, so the result is a single D -dimensional binary vector instead of an $N \times N$ attention map. This vector is applied to V as a mask through a broadcast element-wise product, as Figure 2.4 shows alongside vanilla self-attention. Yao *et al.* also redesign the residual connections to propagate membrane potentials instead of spikes, so that every neuron transmits binary spikes.

This form of attention requires neither a matrix multiplication nor a softmax, which makes it considerably cheaper to compute. Yao *et al.* estimate the computation energy of SDSA, excluding the generation of Q , K and V , to be up to $87.2\times$ lower than that of vanilla self-attention. The reduction has two sources: additions are cheaper than multiplications, and the complexity of SDSA is $O(ND)$, linear in both the number of tokens and the embedding dimension, against $O(N^2D)$ for vanilla self-attention. With this attention, the *Spike-Driven Transformer* reaches 77.1% top-1 accuracy on ImageNet-1K, a state-of-the-art result for SNNs at the time of publication.

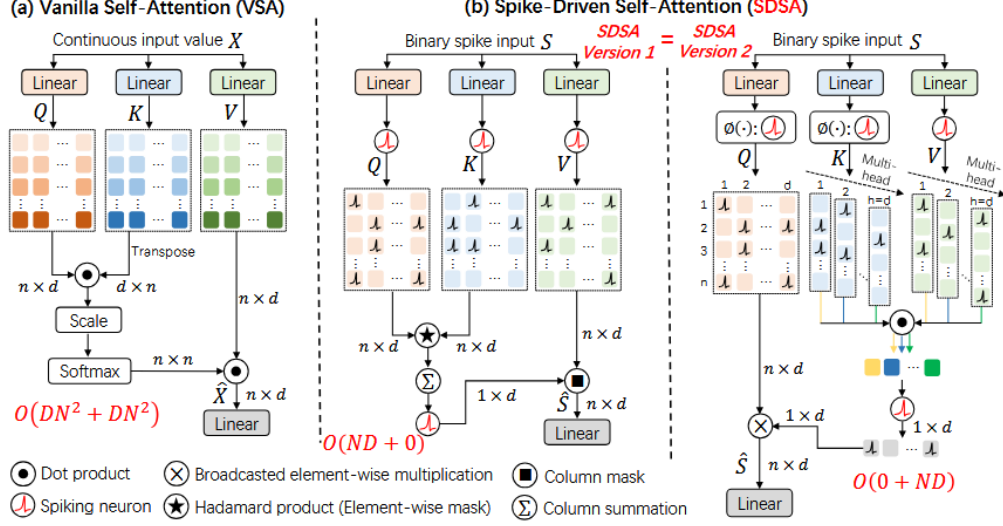


Figure 2.4: Vanilla Self-Attention (VSA) and SDSA [4]

STAtten, proposed by Lee *et al.* [5], extends SDSA to capture both spatial and temporal relationships between tokens. In SDSA, tokens attend to other tokens only within a single timestep, and no mechanism relates the attention computed at timestep t to that computed at timestep $t + 1$. Since SNNs process their input over T timesteps and carry state across them through the leak and decay of the membrane potential, this discards temporal structure that is present in the spike train.

The two mechanisms therefore address different questions. SDSA makes the attention computation spike-driven and inexpensive, which concerns arithmetic and hardware cost. STAtten changes which timesteps each attention computation compares: in SDSA, tokens are compared only with tokens of the same timestep, whereas STAtten groups the timesteps into blocks and compares tokens across all the timesteps of a block. The two can therefore be combined without conflict.

STAtten uses block-wise spatio-temporal attention: it partitions the $T \times N$ sequence into local spatio-temporal chunks and computes attention jointly across space and time within each block, as Figure 2.5 shows. The representation of a token can then depend on neighboring timesteps as well, while the complexity remains $O(TND^2)$ by construction, the same as that of spatial-only spiking self-attention. Since STAtten changes only which slices of Q , K and V are compared with each other, its authors describe it as “plug-and-play”. Lee *et al.* show that it consistently improves accuracy over spatial-only attention on both static (CIFAR-10, CIFAR-100, ImageNet) and neuromorphic (CIFAR10-DVS, N-Caltech101) benchmarks.

2.5 RISC-V Architecture

RISC-V is an open ISA originally developed at the University of California, Berkeley [21]. It is freely available to academia and industry, so anyone can implement a RISC-V core without

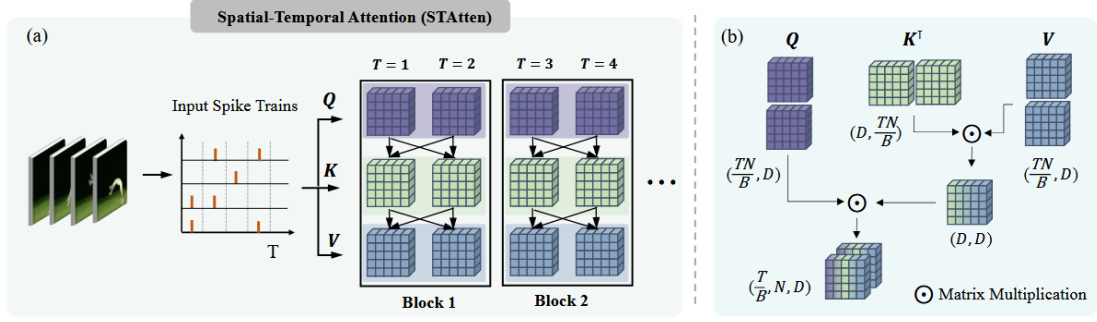


Figure 2.5: The STatten mechanism [5]

paying licensing fees. Its name derives from *Reduced Instruction Set Computer* (RISC), a design philosophy that favors a small set of simple instructions over the large and elaborate instruction sets of *Complex Instruction Set Computer* architectures.

The ISA is built around a small mandatory base integer instruction set, available in 32-, 64- or 128-bit variants. The base is usable by itself as the foundation of a customized accelerator, while general-purpose software is supported through a set of optional standard extensions [21]. Each extension adds a capability on top of the base and is identified by a single letter: M adds integer multiplication and division, A adds atomic memory operations, F and D add single- and double-precision floating-point support, and C adds a compressed 16-bit instruction encoding that reduces code size. A processor designer therefore implements only the base integer ISA and the extensions the target workload needs, instead of a single fixed instruction set meant to serve every possible use case. Crucially, the specification also reserves part of the opcode space for custom, non-standard extensions. A designer can thus add new instructions tailored to a specific application domain without breaking compatibility with the base ISA or with the standard compiler toolchain.

2.6 RISC-V Streaming Processors: the Snitch Cluster

As Zaruba *et al.* [22] observe, general-purpose processors, RISC-V ones included, devote most of their silicon to functions other than computation, such as instruction fetch, decode, branch prediction, and a register file sized to serve a wide variety of programs. On a fully featured out-of-order core such as the open-source BOOM *Central Processing Unit* (CPU), only about 2.7% of the overall core area is dedicated to the FPU, the unit that performs the useful computation in a data-parallel, floating-point intensive workload such as the matrix multiplications of a Transformer. The remaining area is spent on supplying that small fraction of logic with instructions and data.

Snitch follows the opposite design principle. The usual way to improve FPU utilization on superscalar and out-of-order processors is to make the control core wider or more speculative. Zaruba *et al.* [22] instead shrink the control core to a small, single-issue, in-order RISC-V core

of about 10 *kilo Gate Equivalents* (kGE), and pair it with a comparatively large double-precision FPU. Two minimally intrusive ISA extensions then restore the balance between compute and control. The first, the *Stream Semantic Register* (SSR), lets the core implicitly encode a memory access stream as a register: reading from or writing to a designated register triggers a load or store to the next element of the stream. This removes the explicit address computation and the load and store instructions that a loop would otherwise need for every element. The second, the *Floating-Point Repetition* instruction (FREP), decouples the floating-point pipeline from the integer one: a single instruction reissues a short sequence of floating-point instructions from a micro-loop buffer, so the control core does not fetch and decode them again at every iteration. Together, these two extensions let the Snitch core behave as if it were dual-issue, at an incremental area cost of only 3.2% over the baseline core. Zaruba *et al.* report more than a 6× multi-core speedup and a 3.5× gain in energy efficiency on parallel microkernels, measured on an octa-core cluster implemented in 22 nm technology.

A Snitch cluster groups several of these lightweight cores around a shared scratchpad memory and interconnect. Because each core is small, a cluster fits many of them within the area and power budget that a conventional design would spend on one wide, out-of-order core. The cluster thus trades single-thread performance for aggregate floating-point throughput and energy efficiency on data-parallel and streaming workloads.

This streaming approach has since been extended to sparse computation. Scheffler *et al.* [23] propose *Indirection Stream Semantic Registers* (ISSR), which first fetch an index from memory and then stream the element that index points to. They remove from the software loop the address computation that dominates a CSR product on a simple core. Manoni *et al.* [24] then applied these streams to SNN inference on RISC-V clusters with SpikeStream, mapping weight accesses to affine and indirect streams. This thesis examines whether the same mechanisms apply to the attention of a spiking Transformer, whose sparsity we measure in Chapter 3.

The octa-core cluster used in this thesis has eight worker cores and a *Direct Memory Access* (DMA) core that moves data in and out of the cluster. They share a 128 KiB L1 scratchpad memory, the *Tightly Coupled Data Memory* (TCDM), organized as 32 banks of 4 KiB each [22, 24]. The TCDM is a software-managed scratchpad, not a hardware cache, so there is no cache hierarchy inside the cluster. Everything outside it, which the rest of this thesis refers to as L3 main memory, is reached only through explicit DMA transfers. Larger Snitch-based systems tile several of these clusters together and back them with an off-cluster *High Bandwidth Memory* (HBM) or *Dynamic Random-Access Memory* (DRAM) interface. The TCDM capacity constrains the tensor sizes evaluated in Chapter 4: every operand a kernel touches must fit inside the 128 KiB TCDM at once, and there is no fallback if it does not.

2.7 SIMD, MIMD and Vectorization

The cluster described above combines two forms of parallelism that the classic taxonomy of parallel architectures keeps separate. Each Snitch core fetches and executes its own instruction stream independently of its neighbors, so a cluster of eight such cores is an MIMD machine at the cluster level. Each core can be assigned a different row of a sparse matrix, with its own length and its own nonzero pattern, without the lockstep that a single shared instruction stream would impose. This makes the cluster well suited to sparse computation, since MIMD execution across cores accommodates the irregularity that defeats a lockstep SIMD machine.

Within each core, a packed-SIMD extension of the FPU [24] adds a second, finer-grained form of parallelism: one instruction processes two fp32 or four fp16 lanes packed in a 64-bit register. This benefit materializes only when the lanes are mutually independent and packed accordingly. When a data dependency forces them to be processed one after another, the benefit vanishes, or the pipeline stalls. The two levels combine: a sparse row that runs independently on its own core through MIMD execution can also exploit SIMD internally, once its data is interleaved as shown in Chapter 3. Parallelism is then extracted both from the irregular structure of the workload across cores and from the regular structure within each row.

The data dependency mentioned above takes three forms, since the classic taxonomy distinguishes three hazards between two instructions that access the same storage location. A *Read-After-Write* (RAW) hazard, the true data dependency, occurs when an instruction reads a value that a preceding instruction has not yet written, so the second instruction must wait for the result. A *Write-After-Write* (WAW) hazard, an output dependency, occurs when two instructions write the same location: their writes must complete in program order, or the wrong value survives. A *Write-After-Read* (WAR) hazard, an anti-dependency, occurs when an instruction writes a location that a preceding instruction still needs to read: moving the write ahead of the read would corrupt the value the first instruction depends on. WAW and WAR arise from reusing the same register or memory location for two logically unrelated values, and a compiler or a hardware renamer can often remove them by giving the two values separate storage. RAW reflects an order that the computation itself imposes, and no renaming can remove it.

Figure 2.6 shows the RAW case, which is the most relevant to the kernels of this thesis and the one encountered by the SpMV example of Section 2.8. Instruction B reads register R1, which instruction A has not yet written, so B cannot be issued in parallel with A and must stall until A completes. Two instructions that could have been issued in parallel become a strictly sequential pair. Chapter 3 returns to this point: the accumulation chain inside a row of a sparse matrix-vector product and the data-dependent addressing of the CSR format are both RAW hazards that the kernel design must work around.

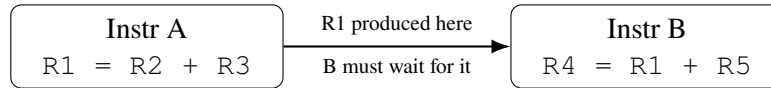


Figure 2.6: Read-after-write dependency between two instructions

2.8 The Challenge of Exploiting Sparsity

Sparsity is a property of the computation, not of the hardware that runs it: a spike train is equally sparse whether a neuromorphic ASIC or a GPU processes it. Turning its zeros into savings requires a representation that allows the machine to skip them, and every such representation has a cost. The indices and pointers that record where the nonzeros are must themselves be stored, loaded and followed. If handled carelessly, this cost can consume the entire saving that skipping the zeros was meant to deliver. The overhead of managing a sparse data structure must therefore be measured rather than assumed, and weighed against the energy that the sparsity saves.

As an example, we consider a *Sparse Matrix-Vector Multiplication* (SpMV), of which the attention kernels developed in Chapter 3 are a specialized variant. In a dense matrix-vector multiplication, every element of the matrix is multiplied by the corresponding element of the vector, and the products are summed to produce the output vector. In an SpMV, only the nonzero elements of the matrix are multiplied by the corresponding elements of the vector before being summed into the output vector. Without dedicated support, iterating over the nonzero elements of a sparse matrix is expensive: it requires many indirect memory accesses and conditional branches, which may cause cache misses and pipeline stalls. A general-purpose processor executing such an operation naively proceeds as follows:

1. The instructions of the loop body are fetched from memory and decoded.
2. The processor reads, from the row pointers, where the next row of the sparse matrix begins and ends.
3. For each nonzero entry in that row, it loads the column index stored alongside the entry, which identifies the element of the input vector the entry must be multiplied with.
4. From that column index it computes the memory address of the corresponding vector element, an additional address computation that a dense loop does not require, and issues an indirect load to fetch it.
5. It loads the nonzero value itself from memory.
6. It multiplies the matrix value by the vector element just loaded and adds the product to a running sum for the row.
7. It increments the pointers that track its position in the row and tests whether further nonzero entries remain, branching back to the third step if so.

8. Once every nonzero entry in the row has been processed, it stores the accumulated sum as the corresponding element of the output vector and proceeds to the next row.

Apart from the first, each of these steps corresponds to one or more instructions that must be fetched, decoded and executed for every nonzero element, and some of them are particularly difficult for a general-purpose core to hide. The address computed in the fourth step depends on the column index loaded in the third, so the vector load cannot start until the index load has completed. Neither load can be prefetched, since the address it touches is data-dependent rather than fixed. The branch in the seventh step must be predicted correctly at every iteration; otherwise the pipeline is flushed and refilled, which cancels the performance a deeper pipeline was meant to provide. Dedicated hardware support for sparsity, such as the ISSRs proposed by Scheffler *et al.* [23], is designed to remove this per-element control overhead. It turns the indirect load of the fourth step into an implicit side effect of reading a register, instead of a separate instruction that must be fetched and decoded.

A dense matrix-vector multiplication, by contrast, is a simple loop: each iteration loads a matrix element and the matching vector element and accumulates their product, with no indirect or conditional memory accesses. The processor can prefetch the next row of the matrix and the corresponding elements of the vector while still working on the current row, and the loop can be unrolled to amortize instruction fetch and decode over several iterations. The drawback of this simplicity appears when the matrix is sparse. For a matrix that is 90% sparse, the dense loop spends nine of every ten multiply-adds on zeros. Since every instruction and every fetched memory address consumes energy, 90% of the energy spent on fetching instructions and memory addresses is wasted on useless work. The sparse loop above skips those zeros, but each useful multiply-add then costs the several control instructions enumerated above. Sparsity is therefore beneficial only when the work it skips outweighs the overhead it introduces.

Westerink *et al.* [27] quantify this trade-off on real neuromorphic cores. They show that unstructured weight sparsity burdens lockstep SIMD execution with control, metadata and memory overhead that erodes most of the theoretical energy savings. Only a design in the style of *Single Instruction, Multiple Threads* (SIMT) execution, with per-lane address generation, recovers substantial energy scaling at high sparsity, at the price of load imbalance and sparsity-independent overhead phases. Yoon *et al.* [28] reach a related conclusion from the opposite direction, on dedicated dense-array accelerators instead of SIMD or SIMT cores. They show that the index-matching logic and load-balancing buffers that a processing element needs to skip zeros make it substantially larger than a plain dense processing element: the scatter network and *First-In, First-Out* (FIFO) buffers of a design such as SCNN occupy 3.75× the area of its own multiplier array. Their answer is to give up sparse compute at the processing-element level entirely. The sparse operand is decompressed back into a dense format and fed to an ordinary dense systolic array, so that only the memory-bandwidth savings of compressed storage are retained. The kernels developed in this thesis take neither of these positions. They keep the computation sparse, skipping every zero instead of restoring it, but avoid the area cost of a

dedicated matching network by moving the indirection into the stream registers of the Snitch cluster instead of into a custom processing element. Chapter 3 develops this choice.

2.9 Sparse Data Structures and Energy Efficiency

The cost of exploiting sparsity also depends on the data structures used to represent sparse data. When a sparse matrix is stored in a compressed format such as CSR, every step of the computation must be designed around that data structure. Compressed formats do not store the zero elements, but they store additional information to keep track of the nonzero elements. CSR, shown in Figure 2.7, keeps three arrays: VAL holds the value of each nonzero, COL its column index, and PTR, for each row, the number of nonzeros stored before that row begins. In the example of the figure, the first element of PTR is 0 and the second is 1, because one nonzero lies between the start of row 0 and the start of row 1. That nonzero sits in column 3, counting from 0, and its value, *a*, is the first element of VAL.

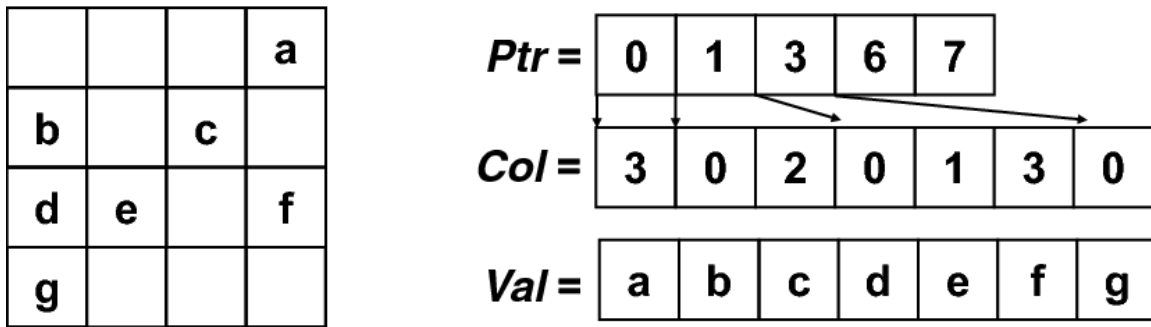


Figure 2.7: CSR format of a sparse matrix [6]

Kernels that handle this data structure efficiently are therefore required. The conversion from a dense to a sparse format, or vice versa, is itself an expensive operation, whose cost Section 4.2.10 measures. Converting a dense matrix to CSR for a single operation, such as one matrix multiplication, would waste both time and energy: to benefit from sparsity, the system as a whole must be designed around it. Even a system designed this way is not guaranteed a payoff. Wang [29] shows that how much energy sparsity can save is a property of the task rather than of the architecture. A feed-forward perception network can be driven down to a 5% firing rate with no accuracy loss, while a recurrent language model cannot be pushed below about 50%, held there by an information-theoretic firing floor. A spiking Transformer, on the other hand, sparsifies to 2% firing at no extra quality cost. The data structures and kernels described above therefore pay off only within the ceiling that the workload imposes, and Chapter 3 begins by measuring how much sparsity the STAtten workload offers, before designing any kernel around it.

Chapter 3

Methodology

Chapter 2 concluded that sparsity pays off only within the ceiling the workload imposes, and that the overhead of exploiting it must be measured rather than assumed. This chapter therefore begins by measuring the sparsity of the tensors of STAtten (Section 3.1). It then sets out the data types and layouts the kernels share (Section 3.2), and derives from those measurements the kernels developed in this work (Section 3.3): the SpMVA (Section 3.3.1), the 1×1 convolution with its two engines (Sections 3.3.2 and 3.3.3), BN (Section 3.3.4) and its fusion into the convolution pipeline (Section 3.3.5), the LIF update (Section 3.3.6), the view, reshape and sparse compaction operations (Section 3.3.7), a fused attention sub-block (Section 3.3.8), and the application that runs the whole attention block (Section 3.3.9).

3.1 Profiling STAtten

This section presents the profiling of the STAtten model introduced in Section 2.3.

3.1.1 Profiling Methodology

We profiled the model on a machine kindly provided by the University of Bologna, equipped with an NVIDIA H100 GPU, using the PyTorch implementation of STAtten released by its authors. The GPU serves here only as the platform on which the reference model runs; it is not a performance baseline for the kernels of this thesis. To measure sparsity, we dumped the tensors produced by each operation, counted their nonzero elements, and report sparsity as the fraction of elements equal to zero. The model was profiled on CIFAR-10 and CIFAR-100, two widely used image classification benchmarks.

3.1.2 Model Architecture

We first verified that the STAtten implementation matches the architecture described in the original paper [5]. To this end, we dumped the shapes of the tensors at each layer of the model for

the datasets above, and reconstructed the architecture by reading the code released in the authors' repository line by line. Figure 3.1 shows the resulting architecture, and Figure 3.2 details the attention module together with the tensor shapes it produces for a CIFAR-100 input image.

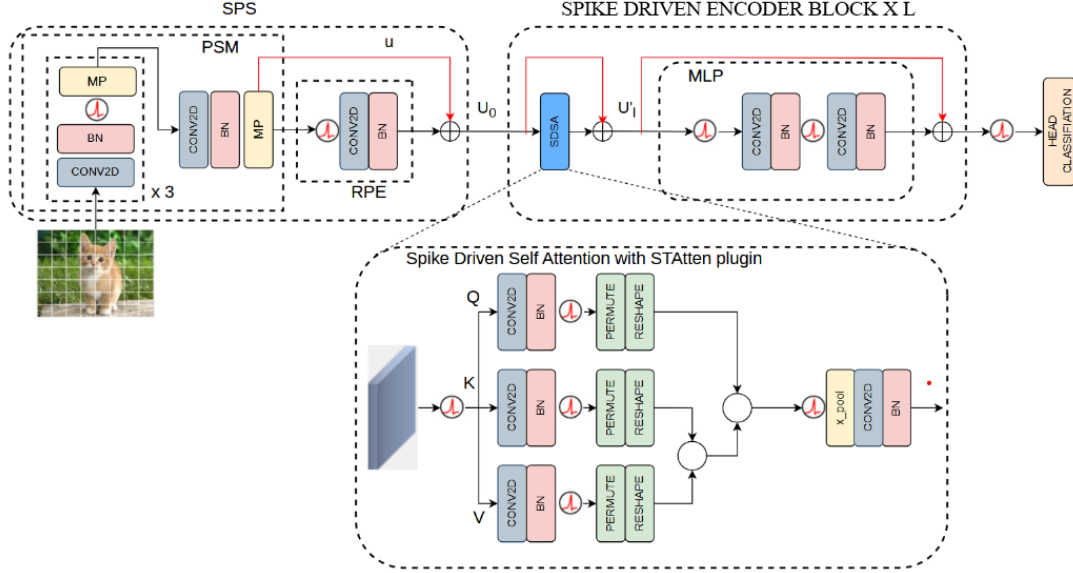


Figure 3.1: STAtten architecture reconstructed from the released code

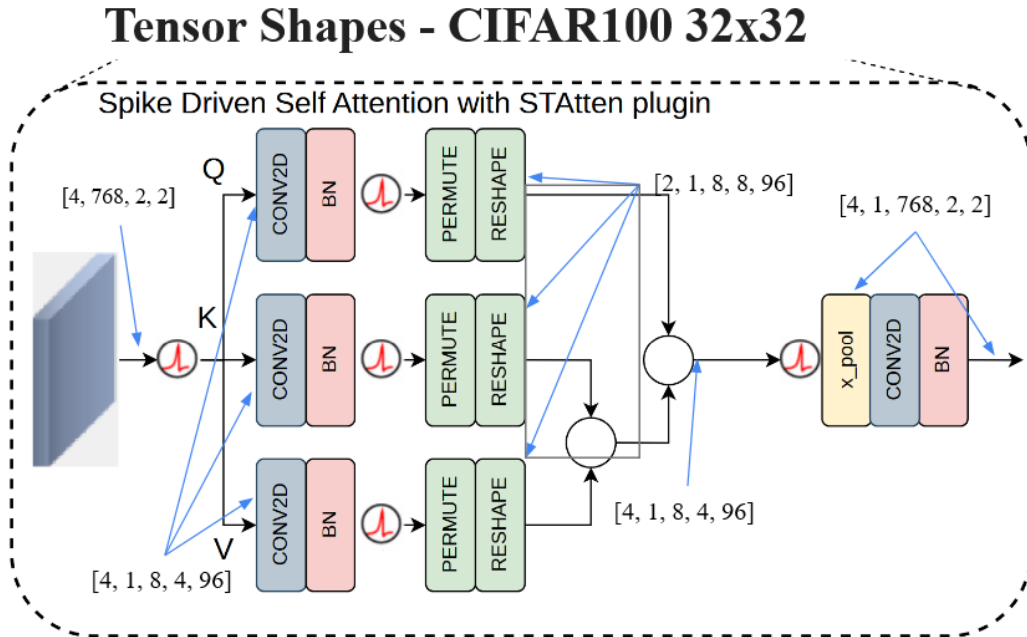


Figure 3.2: STAtten attention module with tensor shapes for a CIFAR-100 input

3.1.3 Profiling Results

Figure 3.4 locates the six tensors we instrumented inside the SDSA module and reports the sparsity measured at each of them. `x_for_QKV` is read after the input LIF, on the tensor that

feeds all three convolutions, and Q , K and V are read after the LIF that closes each of the three branches. `attention_QKV` is read at the output of the attention computation, immediately before the last LIF of the module, and `attention_QKV_postLIF` immediately after that LIF. Each value is the mean over 30 CIFAR-100 input images and the eight encoder executions of each forward pass. From Q onward the attention tensors stay within one percentage point across the eight executions, while `x_for_QKV` falls by about four points, from 68.28% at the first execution to 64.37% at the last.

Table 3.1 gives the value recorded at each of the eight encoder executions. The six rows above the rule are the attention tensors of Figure 3.4. The two rows below it are measured in the *Multi-Layer Perceptron* (MLP) block that follows the attention module, after its LIF layer and after its second fully connected layer; they are reported because they close the encoder block that the attention module opens. Figure 3.3 follows the two tensors at the end of the module, `attention_QKV` and `attention_QKV_postLIF`, over the four timesteps and the eight encoder executions for a single CIFAR-10 input, image #202.

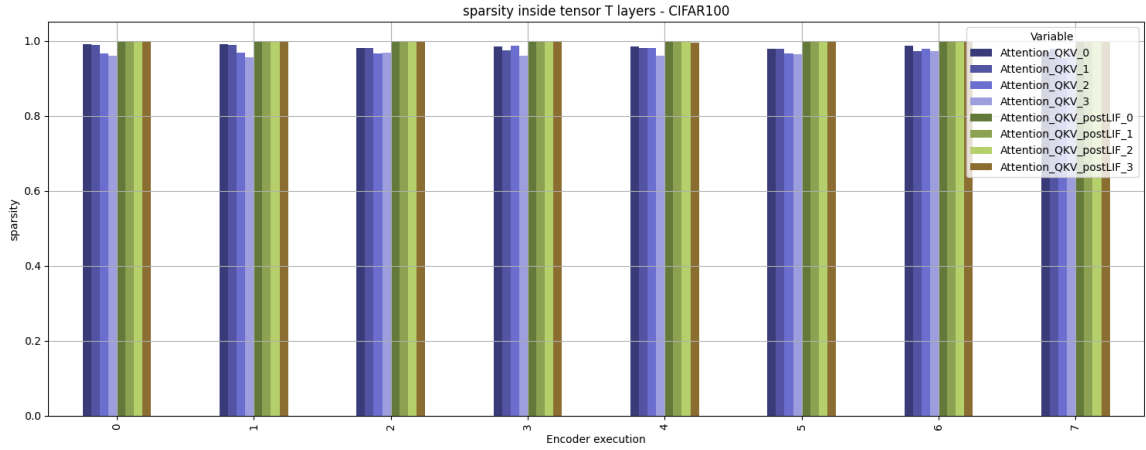
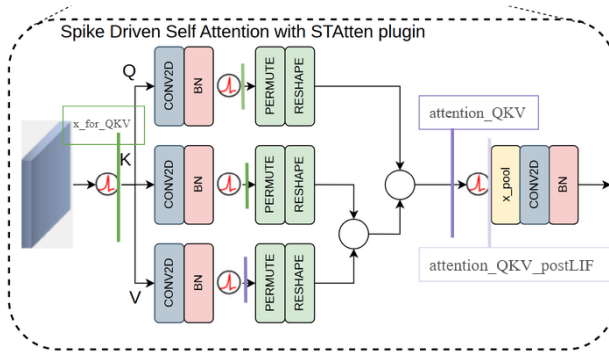


Figure 3.3: Sparsity of `attention_QKV` and `attention_QKV_postLIF` against timestep and encoder execution, CIFAR-10 image #202



Variable	Sparsity (%)
<code>x_for_QKV</code>	65.89
Q	96.87
K	96.86
V	96.87
<code>attention_QKV</code>	97.59
<code>attention_QKV_postLIF</code>	99.86

Figure 3.4: Instrumented tensors of the SDSA module and their mean sparsity

Variable	Encoder execution							
	1	2	3	4	5	6	7	8
x_for_QKV	68.28	67.33	66.45	65.81	65.33	64.94	64.60	64.37
Q	97.12	96.96	96.90	96.85	96.78	96.77	96.82	96.77
K	97.09	96.96	96.93	96.85	96.78	96.79	96.75	96.77
V	97.15	97.00	96.89	96.84	96.83	96.76	96.75	96.72
attention_QKV	97.95	97.80	97.70	97.62	97.49	97.34	97.40	97.41
attention_QKV_postLIF	99.89	99.87	99.86	99.86	99.86	99.84	99.85	99.85
MLP_postLIF	68.35	67.12	66.25	65.61	65.20	64.81	64.53	64.24
MLP_postfc2	97.06	96.96	96.91	96.83	96.79	96.76	96.75	96.74

Table 3.1: Sparsity (%) of the instrumented encoder tensors against encoder execution

3.2 Implementation Details

This section describes the design choices that guided the development of the kernels that execute STAtten on the Snitch cluster.

3.2.1 Data Generation and Vectorization

The kernels proposed in this thesis support multi-core (MIMD) execution and, where possible, SIMD execution. This choice affects mainly data types and memory access patterns: every input on which the kernels are evaluated is generated in fp64, fp32 and fp16 format. The inputs are generated, not dumped from the model. The standalone kernels run at the shapes of Table 4.1 and the density points of Table 4.2, which Section 4.1.4 derives from the profiling above; the attention block of Section 3.3.9 runs at the densities measured on CIFAR-100.

To leverage the SIMD capabilities of the Snitch cluster, the data must be packed into the 64-bit words on which the cluster operates. To leverage its MIMD capabilities, the work must be divided into independent portions, rows or channels, that the eight cores can process without synchronizing with each other.

Data dependency, in its RAW, WAW and WAR forms introduced in Section 2.7, is the main threat to both. In the kernels of this thesis, the accumulation into a single output element (summing partial products into one entry of the result vector in an SpMVA) is itself a chain of RAW-dependent additions. In addition, the indirect, data-dependent addressing of CSR access patterns can turn loads that appear independent into a dependency the compiler cannot resolve statically, the case illustrated in Figure 2.6.

3.3 Kernels

This section describes the kernels proposed in this thesis. Appendix B describes how a particular variant of each kernel is selected at build time, and how every result is verified.

3.3.1 Sparse Matrix-Vector Addition (SpMVA)

The first kernel we developed is the SpMVA, based on the kernels of Scheffler *et al.* [23]. The SpMVA is an inner-loop operation that an outer loop can repeat to compose a full matrix-matrix multiplication. The kernel takes a sparse matrix A stored in CSR format and a dense vector B , computes the matrix-vector product $C = AB$ over all rows of A , and stores it in the dense vector C . The sparse input matrix A is binary, meaning that all its nonzero entries are equal to 1.0. This property simplifies the kernel arithmetically: the multiplication by 1.0 can be omitted, and only the elements of B selected by the nonzero entries of A need to be accumulated. Dropping the multiplication by itself saves little time: Section 4.2.1 measures it at a median of about 5%, and attributes the speedup of the kernel to its indirect gather, its loop structure and its lane packing.

Figure 3.5 compares the two cases for the same row of A , stored in CSR as a value array and a matching column-index array. In both cases the column index drives the gather of the matching element of B . In the general case, on the left, A holds arbitrary nonzero values, so each gathered element requires a multiplication before it is accumulated into c . In the binary case this kernel implements, on the right, every nonzero of A is equal to 1, so the value array reduces to a sequence of ones that is never read. Each gathered element is then accumulated directly, with no multiplication.

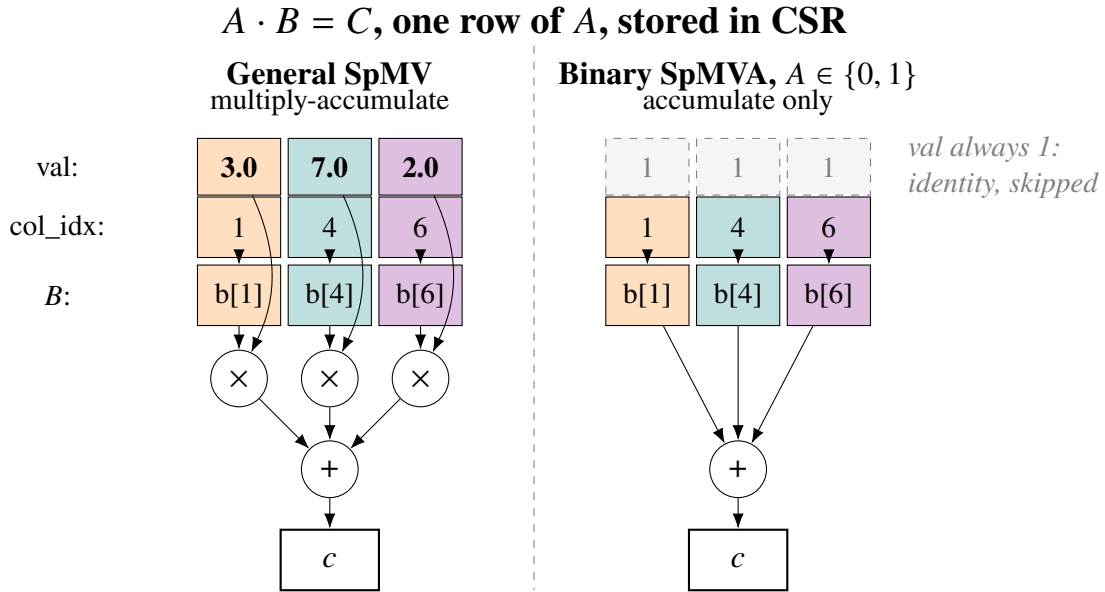


Figure 3.5: One CSR row of $A \cdot B$: general SpMV against binary SpMVA

The fp64 variant of this kernel is `__sla_inl_issr_sm16_dv_innerp_d_fadd`, and Table 3.2 lists its signature argument by argument. The two SIMD variants introduced below share the same signature and differ only in the packed floating-point instructions they use internally.

Each component of the name records one design choice: `issr` (indirect SSR gather), `sm16` (sparse matrix with 16-bit column indices), `dv` (dense vector operand), `innerp` (row-at-a-time inner-product formulation), `d` (double precision), and `fadd` (accumulation only, as opposed to

Argument	Type	Role
<code>a_vals</code>	<code>double *</code>	CSR nonzero values of A ; never dereferenced, since A is binary
<code>a_idcs</code>	<code>uint16_t *</code>	CSR column indices of A , walked by the indirect SSR
<code>a_ptrs</code>	<code>uint32_t *</code>	CSR row-pointer array, delimiting each row's slice of <code>a_idcs</code>
<code>a_rows</code>	<code>uint32_t</code>	number of rows of A , the outer row-loop trip count
<code>b_vals</code>	<code>double *</code>	base address of the dense operand B being gathered
<code>b_shift</code>	<code>uint32_t</code>	programmable shift applied to each gathered index before it addresses <code>b_vals</code> , letting the same indirection reach a specific column of a larger power-of-two-strided dense structure [23]
<code>res</code>	<code>double *</code>	base address of the dense result vector C
<code>res_stride</code>	<code>uint32_t</code>	stride, in elements, between consecutive rows' results in <code>res</code> (bound to <code>COUT_WORDS</code> in <code>conv1x1</code> 's reuse of this kernel, Table 3.3)

Table 3.2: Arguments of the fp64 ISSR-based SpMVA kernel

a fused multiply-add). In this pipeline, A is the spike matrix K^T , produced by a LIF layer and therefore binary. The kernel thus never streams or reads `a_vals`, and its inner accumulation is a plain `fadd.d`. The column indices of the nonzeros are gathered through the ISSR hardware of the Snitch cluster. One SSR channel is configured once, before the row loop starts: its index bound is set to the total nonzero count of the matrix minus one, and its index-register pointer walks `a_ids`. From that point on, every `fadd.d ft4, ft1, ...` instruction inside the loop consumes one element of `b_vals[a_ids[i]]` from the stream register `ft1`. No address computation, load instruction or integer register is involved, so the floating-point pipeline can sustain one addition per cycle.

The row loop adapts to the row length. Rows of one to three nonzeros are accumulated into a single register by an explicitly unrolled `fadd.d` chain and stored directly, since for such short rows the setup and drain latency of the FREP hardware loop would not be amortized. Every longer row enters a fixed-trip FREP prologue of three iterations. Together with the addition that precedes it, the prologue spreads the first four elements of the row across four independent accumulator registers, `ft4` through `ft7`, so that the data dependency between successive additions does not stall the pipeline. A row of exactly four nonzeros ends there, and its four partial sums are collapsed at once by `SLA_REDUCE_FT4_4`. A row of five or more continues into a second, variable-trip FREP over the remaining $z - 4$ elements, which rotates the destination across five registers: the four already filled plus `ft3`, zeroed before the row loop began so that it can join the rotation as an empty accumulator. The five partial sums are then collapsed by a small adder tree, `SLA_REDUCE_FT3_5`, before the result of the row is written back. The same structure is implemented once per SIMD width: the fp64 form shown here, `__sla_inl_issr_sm16_dv_innerp_d_fadd_simd32` (two fp32 lanes packed per 64-bit word, `vfadd.s`), and `__sla_inl_issr_sm16_dv_innerp_d_fadd_simd16` (four fp16 lanes, `vfadd.h`). The three variants share the same signature and control structure, and differ only in the packed floating-point instructions and in the packed-zero constant used to seed the accumulators. Which of them runs is fixed at build time, as described in Appendix B. Chapter 4 refers to the three variants as `issr_fadd`, and its plots label them `ISSR-issr_fadd`. Their reference is the plain-C kernel `__sla_inl_c_sm16_dv_innerp_d`, labeled `plain_c`, a general CSR SpMV that loads each nonzero value of A and multiplies it by the gathered element of B . A third fp64 kernel, `__sla_inl_issr_sm16_dv_innerp_d`, labeled `issr_fmadd`, keeps the ISSR structure but multiplies each gathered element by its value of A , so that comparing it with `issr_fadd` isolates what dropping the multiplication saves.

On the cluster, the kernel runs inside the block pipeline of `cluster_spmv.h`. The DMA core double-buffers the CSR operand into the TCDM one block of whole rows at a time, each buffer holding at most 4096 nonzeros (`ELEM_BUF_SIZE`), while the worker cores compute on the block already resident. Within a block, each worker claims its next group of rows by an atomic increment of a shared row index, and returns for another until the block is exhausted. A block of R rows is divided into groups of $\lfloor R/32 \rfloor + 1$ rows, at most 32 groups per block, so a

group is a single row whenever the block holds fewer than 32 rows. The claims are first-come, first-served, so a core can take more than its share.

With SIMD, the elements that one instruction processes together must first be packed into a single 64-bit word, and the kernel itself performs no packing. This step is avoided by generating the dense operand already interleaved in fp32 or fp16, as Figure 3.6 shows. The period of the interleave is the number of elements per column of B , that is, the sequence length, here 8. Position $j = 0, \dots, 7$ therefore collects the elements $j, j + 8, j + 16, \dots$ of a column-major B : the same row j of two or four *different* columns. A single 8-byte load then reads that group directly as a packed SIMD word, two lanes for fp32 or four for fp16, with no separate pack or unpack instruction. The SpMVA thus operates on a sparse matrix and on two or four interleaved dense vectors at once. The result is interleaved in the same way, so a separate un-interleave stage, a permutation inside the TCDM performed by `snn_unpack_fp32x2` or `snn_unpack_fp16x4`, separates the two or four output columns; Chapter 4 measures its cost.

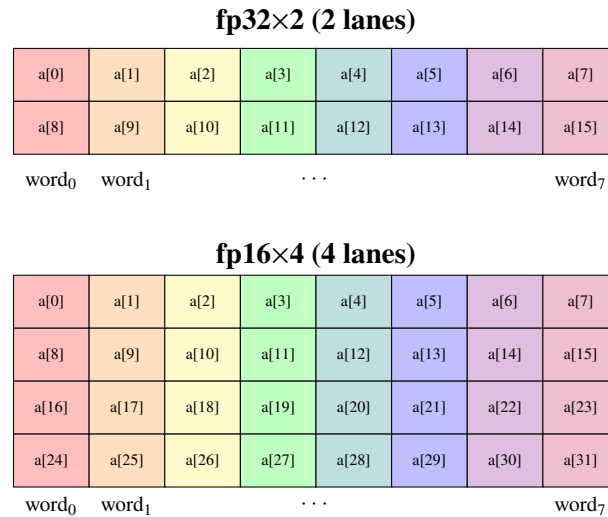


Figure 3.6: Interleaved column layout of B for fp32x2 and fp16x4

With SIMD, the `fadd.d` instruction becomes a `vfadd.s` or `vfadd.h` instruction, a vector addition over two fp32 or four fp16 lanes. The interleaved layout is particularly useful for matrix multiplication. A matrix-matrix multiplication is a matrix-vector multiplication repeated once per column of the second operand, so the same nonzero pattern of a sparse row must be traversed again for every output column. If those columns are stored interleaved, one indirect gather fetches a packed word carrying one element from each column, and the sparse row is traversed once for two or four output columns.

Figure 3.7 shows one row of $C = A \cdot B$ computed at each of the three widths. All three panels walk the same nonzero pattern of A : a nonzero at column j triggers a gather of $B[j]$, while a zero triggers no access at all. The fp64 case gathers one scalar per nonzero and accumulates it into the single element c_0 . The fp32x2 case gathers, at the same indices, a packed word holding $b[j]$ and $b[j + 8]$ and accumulates into two elements of C . The fp16x4 case gathers $b[j]$, $b[j + 8]$,

$b[j + 16]$ and $b[j + 24]$ with a load of the same width and accumulates into four. Since A is traversed in the same number of steps whatever the lane count, the cost of walking the sparse operand is amortized over one, two or four output columns.

3.3.2 Convolution: from SpMVA to SpMMA

The `MS_SSA_Conv` module contains four 1×1 convolutions. Three produce the projections $\mathbf{W}_Q X$, $\mathbf{W}_K X$ and $\mathbf{W}_V X$ introduced in Section 2.3, and the fourth, `proj_conv`, projects the attention output back at the exit of the module. None of the four is a convolution in the spatial sense: with kernel size 1, each one is a pointwise matrix multiplication over the channel axis,

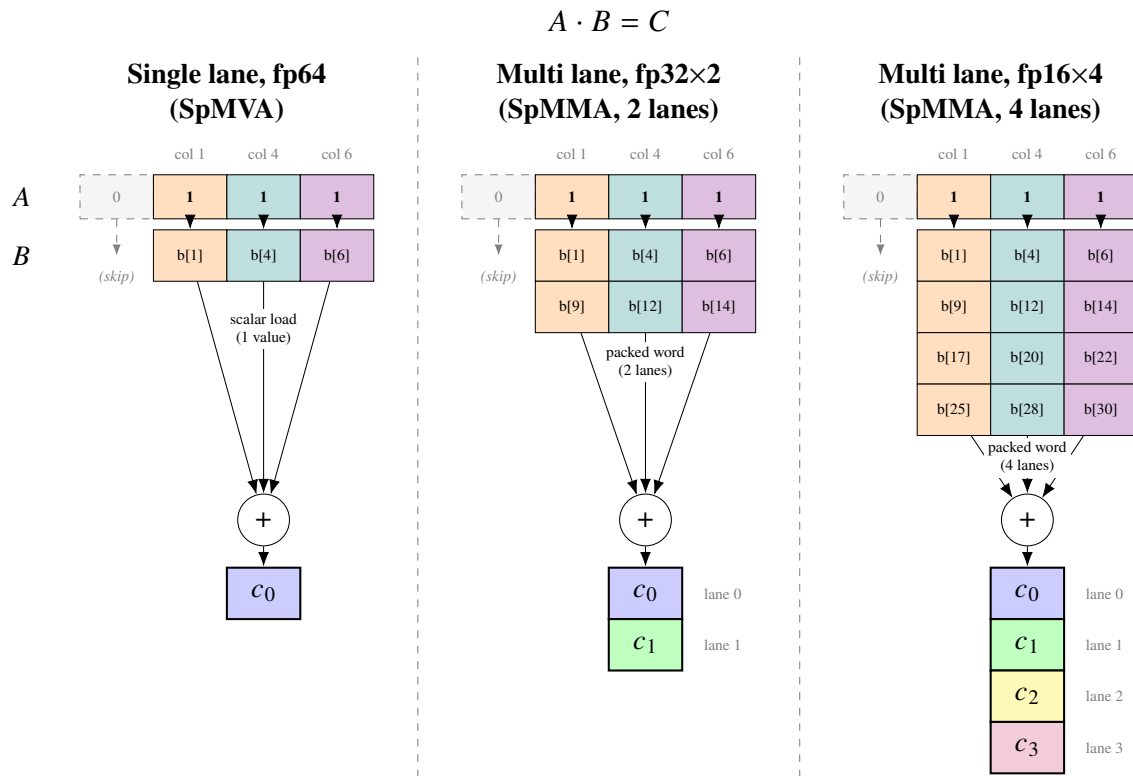
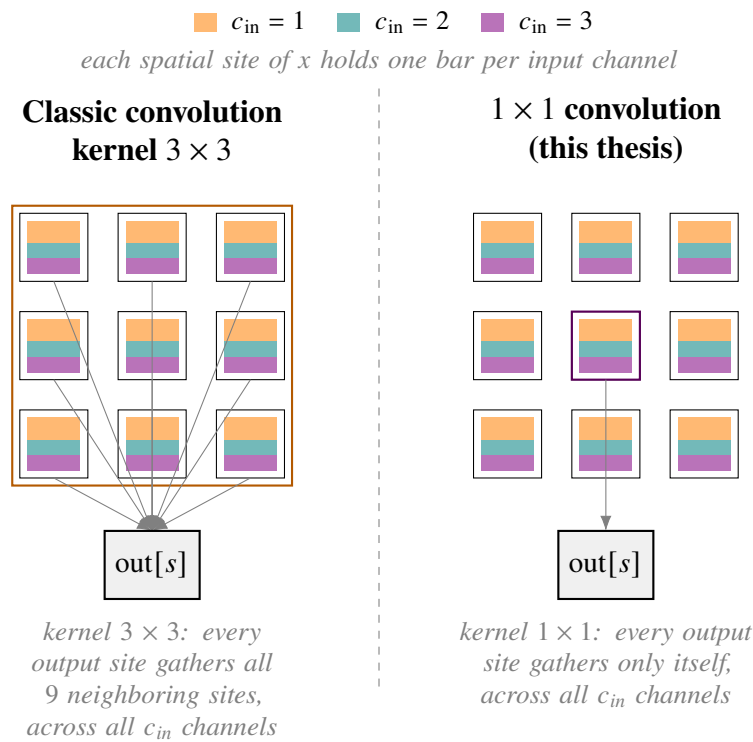
$$\text{out}[s, c_{\text{out}}] = \sum_{c_{\text{in}}} x[s, c_{\text{in}}] \mathbf{W}[c_{\text{in}}, c_{\text{out}}], \quad (3.1)$$

evaluated independently at every site $s = (t, b, n)$ of the timestep, batch and token axes. The input x is the output of a LIF layer and is therefore a binary spike tensor. The product $x[s, c_{\text{in}}] \mathbf{W}[c_{\text{in}}, c_{\text{out}}]$ thus never requires a multiplication, and the output row at a site is the sum of the weight rows selected by the active input channels of that site,

$$\text{out}[s] = \sum_{c_{\text{in}} \in \text{active}(s)} \mathbf{W}[c_{\text{in}}]. \quad (3.2)$$

With x held as a CSR matrix, the whole convolution is $\text{out} = \text{CSR}(x) \cdot \mathbf{W}$, a *Sparse Matrix-Matrix Multiplication* (SpMM) between a sparse and a dense operand. With a binary sparse operand, it reduces to a *Sparse Matrix-Matrix Addition* (SpMMA), in which the sparse operand is $[\text{sites}, c_{\text{in}}]$ and carries only ones, and the dense operand $\mathbf{W}$ is $[c_{\text{in}}, c_{\text{out}}]$ stored c_{in} -major. This is the multi-lane case of Figure 3.7 in general form, in which the same nonzero pattern of a sparse row is traversed once per column-word of $\mathbf{W}$, that is, once for every one, two or four output columns at fp64, fp32x2 or fp16x4. No new arithmetic kernel is therefore required, since the SpMVA kernel of Section 3.3.1 already computes this reduction, one dense column at a time.

Figure 3.8 shows why these convolutions are not spatial, comparing a classic convolution with the pointwise one over the same 3×3 grid of sites, each site holding one value per input channel. With a 3×3 kernel, the value written at one output site depends on all nine neighboring sites and on every input channel, so a spatial convolution cannot be separated from a gather over neighboring positions. With a 1×1 kernel, the receptive field reduces to the single site below the output, so the operation mixes channels but never positions. This is the pointwise reduction of the equation above, and the reason why the SpMVA kernel that traverses a CSR row in the attention stage can compute it unmodified.

Figure 3.7: One row of $C = A \cdot B$ at fp64, fp32x2 and fp16x4Figure 3.8: Classic 3×3 convolution against 1×1 convolution

The convolution reuses the fp64 inner-product kernel `__sla_inl_issr_sm16_dv_innerp_d_fadd` of Table 3.2, or one of its two SIMD variants, unmodified, invoked once per output column-word. For a single dense column vector b , the kernel computes

$$\text{res}[r] = \sum_{k \in \text{row}(r)} b[\text{idx}[k]], \quad (3.3)$$

reducing each CSR row through an indirect SSR that gathers from b and a FREP chain of `vfadd` instructions, with `a_vals` never read, as in the attention stage. A single 64-bit floating-point register holds `SIMD_COLS` lanes, so the dense column b is in practice one column-word of $\mathbf{W}$: `SIMD_COLS` adjacent output channels produced together in one pass over the sparse structure.

The extension from this SpMVA to the full SpMMA is done entirely at the call layer, by the family of `conv_dm_*` wrapper functions in `conv1x1.h`. They loop the engine over the `COUT_WORDS` column-words of one output row. Only the arguments that carry the dense operand and the output are rebound, while the sparse-side arguments are left untouched, as summarized in Table 3.3.

SpMVA argument	Bound to	Per call
<code>a_ptrs, a_idcs, a_rows</code>	site row pointers, scaled channel indices, site count	fixed, the same CSR traversal of x every iteration
<code>a_vals</code>	<code>nullptr</code>	never dereferenced (x is binary)
<code>b_vals</code>	<code>W + w</code>	advances one column-word
<code>res, res_stride</code>	<code>out + w, COUT_WORDS</code>	writes column-word w down the output

Table 3.3: Binding of the SpMVA arguments in the `conv_dm_*` loop, iteration w

For iteration $w = 0, \dots, \text{COUT_WORDS} - 1$, the loop sets the gather base to the w -th column-word of $\mathbf{W}$ and the store base to the w -th word of the output row, then traverses the same row pointers and indices of x again. After `COUT_WORDS` iterations the whole $[\text{sites}, c_{\text{out}}]$ result is written. Only these two base addresses change between iterations: every other argument, including the entire CSR structure of x , is passed unchanged, so the sparse operand is streamed `COUT_WORDS` times while only two base pointers move.

The inner-product engine expects each column index to already include the weight-row stride, so the index array is prescaled once, offline, on the host. Each index is stored as `channel[k] * COUT_WORDS` in a `uint16_t`, which is valid as long as $(c_{\text{in}} - 1) \cdot \text{COUT_WORDS} \leq 0\text{x}\text{FFFF}$. Figure 3.9 works this out for one nonzero of x at channel 5 with `COUT_WORDS = 3`: the stored index becomes 15, and both kernels reach the same address, 16, for column-word $w = 1$. The explicit `indir` kernel of Section 3.3.3 keeps the raw index and computes `idx * COUT_WORDS + w` in software before every gather, a shift-and-add on the integer datapath. The `spmva` kernel

receives the prescaled index, so the indirect SSR produces the same address as a plain $\text{idx} + w$, and no address instruction is issued.

A site with no active input channel takes the empty-row path of the engine, which stores a single zero word and performs no arithmetic. An empty site still has a cost, however: the SSR is configured and drained once per column-word regardless of what the rows hold. A pass over the sparse structure therefore costs at least `COUT_WORDS` SSR setups, whatever the sparsity of x . Section 3.3.3 returns to this point, because it is where the explicit engine has a structural advantage, beyond a different instruction mix. The `spmva` convolution kernels keep the same signature as the explicit ones, so the two engines are interchangeable at the call site (Appendix B).

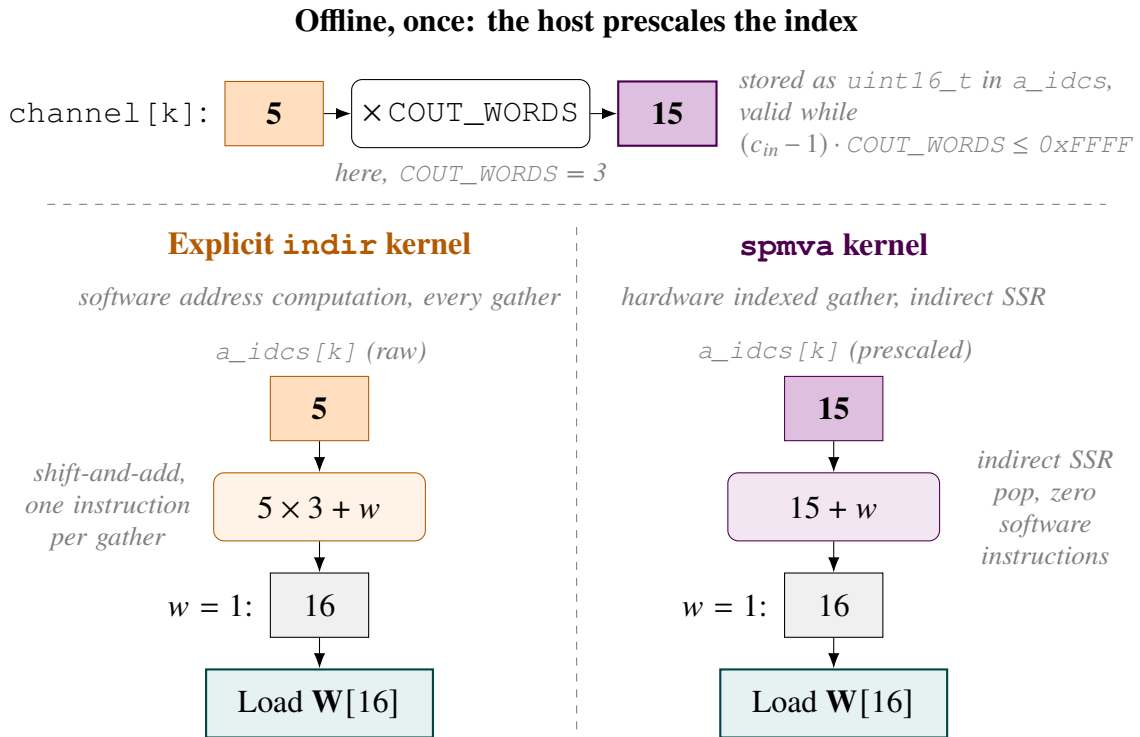


Figure 3.9: Offline prescaling of the CSR column index, `indir` against `spmva`

At the density measured for `x_for_QKV` in Table 3.1, approximately 34% over the 768 input channels, a CSR row in this convolution holds on the order of $z \approx 262$ nonzeros. Unlike the attention stage, whose $K^\top$ rows hold about three nonzeros on average, the convolution therefore always takes the FREP accumulation path, with an accumulation chain hundreds of `vfadd` instructions long. The tensor this accumulation produces is the input of the next kernel, BN, in the same `[site, channel]` layout.

3.3.3 Convolution: Indirect SpMMA

Section 3.3.2 described `conv1x1` as a reduction over the active input channels of a site,

$$\text{out}[s] = \sum_{c_{\text{in}} \in \text{active}(s)} \mathbf{W}[c_{\text{in}}], \quad (3.4)$$

and showed one way of computing it, reusing the SpMVA inner-product kernel of Section 3.3.1 unmodified, once per output column-word. The convolution kernels vary along two independent axes: the lane width, and the engine, which determines how a channel index becomes a read of a row of $\mathbf{W}$. The engine axis, not the lane width, separates the two kernel families: the `spmva` engine of Section 3.3.2 and the `indir` engine described in this subsection. Table 3.4 lists the kernels of the module along this split, together with the plain-C reference and the compaction routine that every configuration shares. With either engine, a site with no active input channel, $z = 0$, short-circuits to a zeroed output tile without any arithmetic, the same empty-row saving already established for the SpMVA kernels. An empty site is nevertheless cheaper under `indir`, whose empty row skips the loop, than under `spmva`, which still pays the per-column-word SSR setup of Section 3.3.2.

Family	Variants	Role
<code>indir engine</code>	<code>conv1x1_fp16x4</code> , <code>conv1x1_fp32x2</code> , <code>conv1x1_fp64</code>	hand-written asm, explicit <code>slli</code> address computation per gather, an 8-wide persistent accumulator bank swept over <code>NTILES</code> column tiles; expressible only where a $\mathbf{W}$ row is a power-of-two number of bytes
<code>spmva engine</code>	<code>conv1x1_innerp_fp16x4</code> , <code>conv1x1_innerp_fp32x2</code> , <code>conv1x1_innerp_fp64</code>	the <code>sla/snn</code> inner-product kernel of Section 3.3.1, reused unmodified once per <code>COUT_WORDS</code> column-word (Table 3.3)
Reference	<code>conv1x1_c</code>	untiled plain-C baseline, used to cross-check the tiling arithmetic bit-exactly
Pipeline utilities	<code>conv_compact_csr</code>	reformats <code>conv1x1</code> 's dense output into a row-sparse CSR container

Table 3.4: Kernels and support routines of the 1×1 convolution

We refer to the two engines as `spmva` and `indir`; the plots of Chapter 4 label them `ISSR-innerp_spmva` and `innerp_indir`. The `indir engine` takes its name from computing that address explicitly, in hand-written assembly, for every gather, as already contrasted with `spmva` in Figure 3.9. It loads a raw channel index, shifts it left into a byte offset with `slli` by a fixed amount, `ROW_SHIFT`, and adds that offset to the base of $\mathbf{W}$. Whereas `spmva` performs that multiplication once, offline, by prescaling the index, `indir` repeats a shift-and-add on the integer datapath at every gather. In exchange, it obtains a wider, purpose-built inner loop. Once the row address is known, the kernel issues an `fld` for eight consecutive words of that row

of $\mathbf{W}$. It accumulates them with `vfadd` in the SIMD variants, or `fadd` in `conv1x1_fp64`, into eight persistent registers, `fs0` through `fs7`, that stay live across the whole row. Figure 3.10 works this out for one active channel, channel 4. Its index is shifted into a byte offset, and the eight words of row 4 of $\mathbf{W}$ are loaded and accumulated into the eight registers shared by every active channel of the row. Once all active channels have been accumulated, the registers are stored as one 8-word tile of the output row. Eight registers cover only eight output words at a time, which is eight channels at fp64 and $8 \cdot \text{SIMD_COLS}$ under SIMD, and c_{out} is usually wider than that. The same accumulator bank is therefore swept across `NTILES` column tiles to cover the full output width, repeating the address computation, the gather and the accumulation for each tile. This sweep is the `indir` counterpart of the sweep of `spmva` over `COUT_WORDS` column-words: `indir` gives up exact kernel reuse in exchange for fewer, wider passes over a purpose-built accumulator bank.

The `slli` instruction also imposes a constraint unrelated to cycle count. A shift can express only a multiplication by a power of two, so `indir` is applicable only where one row of $\mathbf{W}$ occupies a power-of-two number of bytes, and the build rejects any other shape. In the model, each of the Q , K and V convolutions has 768 output channels, which the attention splits into 8 heads of 96 channels. A row of $\mathbf{W}$ is 6144, 3072 or 1536 bytes at fp64, fp32x2 and fp16x4 over the full width, and 768, 384 or 192 bytes over the 96 columns of one head, so no lane width satisfies the rule at either width. The prescaled index of `spmva` lifts the restriction, since a multiplication performed once on the host is under no such constraint. It is subject to a different limit instead: the scaled index must fit a `uint16_t`, which is the $(c_{\text{in}} - 1) \cdot \text{COUT_WORDS} \leq 0\text{x}\text{FFFF}$ bound of Section 3.3.2. The full width of the model, $c_{\text{in}} = c_{\text{out}} = 768$, exceeds that bound at every lane width. The two engines are therefore interchangeable only in the part of the shape space where both are expressible. Section 3.3.8 meets the per-head width, and Section 3.3.9 the full width.

The lane-width axis inside each engine is the same fp16x4/fp32x2/fp64 SIMD split used throughout this chapter. `conv1x1_c` provides an untiled, scalar, plain-C reference implementation of the reduction, against which the tiled kernels are checked bit-exactly (Appendix B).

Beyond the convolution itself, the pipeline offers two optional stages: a CSR compaction and a fused BN pass. The second raises a question that is not specific to the convolution: what folding a second kernel into the same launch removes, and what it requires the host to prepare instead. Section 3.3.5 addresses it once Section 3.3.4 has introduced the BN kernels. The compaction belongs to the convolution itself.

When CSR output is requested, `conv_compact_csr` turns the dense `[site, cout]` result into a row-sparse container. It serves only as a format bridge, so that a downstream SpMVA kernel can consume the output of the convolution like any other CSR tensor. Every non-empty site becomes one fully dense, c_{out} -wide CSR row, whose indices are $0, \dots, c_{\text{out}} - 1$, and every empty site becomes an empty row. No element-level sparsity is introduced: the container only drops empty sites. Because `out_ptrs` is built as a running total across sites, this compaction runs on a single core, for the reason discussed in Section 3.3.7.

The indir engine: one active channel, one of NTILES tiles

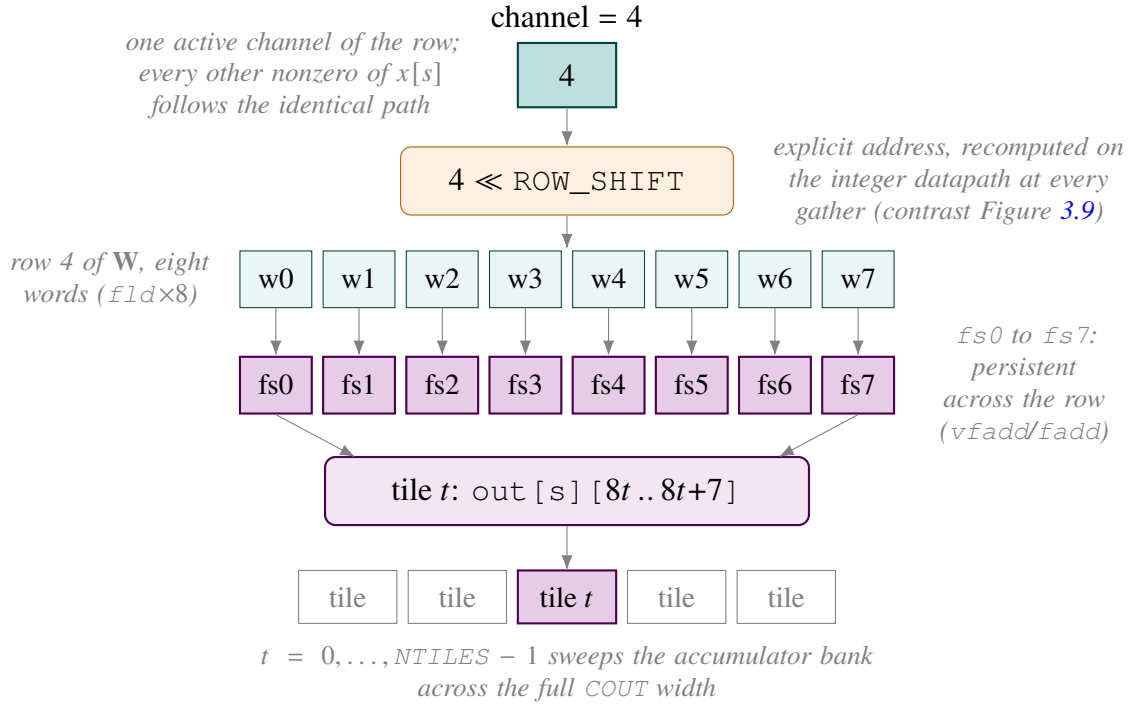


Figure 3.10: The indir engine on one active channel

3.3.4 Batch Normalization (BN)

BN runs only at inference in this pipeline, and only in its folded form, an affine map

$$y[s, c] = \kappa[c] x[s, c] + \lambda[c] \quad (3.5)$$

applied per channel c over the same site axis s along which the convolution produces its output. Figure 3.11 shows how this map is obtained. At inference, BN ordinarily needs five per-channel quantities carried over from training, the scale $\gamma[c]$, the shift $\beta[c]$, the running mean $\mu[c]$, the running variance $\sigma^2[c]$, and the numerical-stability constant ϵ , combined as

$$y[s, c] = \gamma[c] \cdot \frac{x[s, c] - \mu[c]}{\sqrt{\sigma^2[c] + \epsilon}} + \beta[c]. \quad (3.6)$$

Because none of γ , β , μ , σ^2 or ϵ depends on the input, this whole expression is affine in $x[s, c]$, and it is folded once, offline, on the host, into two per-channel coefficients,

$$\kappa[c] = \frac{\gamma[c]}{\sqrt{\sigma^2[c] + \epsilon}}, \quad \lambda[c] = \beta[c] - \kappa[c] \mu[c], \quad (3.7)$$

and, where the convolution feeding BN carries a bias, that bias is folded into $\lambda[c]$ as well before the kernel runs. The kernel itself never reads γ , β or the running statistics, and computes no square root. It sees only κ and λ , and executes one multiplication and one addition per element.

After folding, BN is one streamed element-wise operation over the same [site, channel] layout in which the convolution leaves its output.

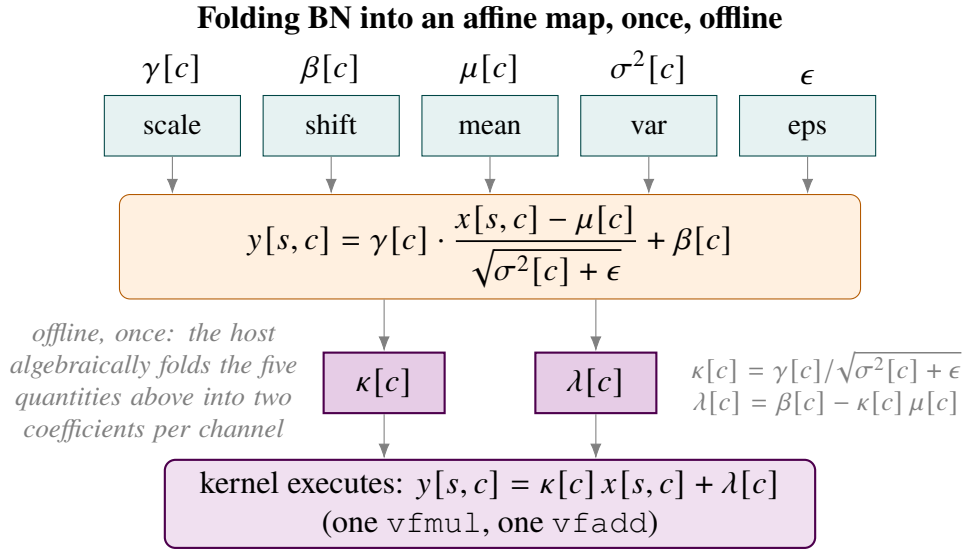


Figure 3.11: Folding BN into two per-channel coefficients

Two families of kernels implement this formula, summarized in Table 3.5 and laid out in Figure 3.12. The dense family operates over the full [site, channel] tensor, driven by SSR streams bounded by the number of channel words per core and the number of sites. Its three SIMD-packed variants, together with the plain-C `bn_apply_c` kept as a reference, read the same interleaved `kl` table, in which the κ and λ words of each channel group alternate.

The CSR family applies the same $\kappa \cdot x + \lambda$ map, but only over the stored nonzeros of a CSR tensor, and it preserves the sparsity pattern: `ptr` and `ids` pass through untouched. Since CSR values are stored as flat scalars rather than pre-packed SIMD lanes, this family needs a stage that the dense one does not. `bn_csr_expand_*` uses an indirect SSR, the same mechanism as in Figure 3.9, to gather `table[ids[k]]` and pack it into the same interleaved κ/λ stream the dense kernels read. `bn_csr_apply_*` then computes, over that gathered stream, the same `vfmul` followed by `vfadd` that `bn_apply_*` computes over the dense one. The plots of Chapter 4 label the two apply kernels `ssr_csr_apply` and `ssr_apply`, and the gather `kappa/lambda expand`. On a general CSR input the gather must run on chip, because the coefficient of each stored value depends on the column it occupies, while the dense kernels read a table that the host provides. For measurement, the gathered stream can also be precomputed on the host, which isolates the arithmetic of the CSR apply from the cost of the gather. Both settings give the apply kernel the same stream, so the two measurements are directly comparable.

The arithmetic is the same, but the two kernel bodies are not, and the difference appears as a cost in Chapter 4 rather than as a correctness issue. The dense body issues eight instructions

under one `frep`, four `vfmul` followed by four `vfadd`. Four independent $\kappa \cdot x + \lambda$ chains are thus in flight at once, and each multiplication has three other instructions between it and the addition that consumes it. The CSR body issues two, a `vfmul` whose result the next `vfadd` reads, with no independent work to hide that latency. The cause lies in the data structure rather than in the formula. The dense kernel traverses whole channel groups and can interleave four of them per core, while the CSR kernel traverses a flat run of stored nonzeros with no structure along which to unroll. The sparse form therefore performs less arithmetic in total but takes longer per element, and on a general CSR input it also pays for the gather. It is faster only on very sparse tensors. Section 4.2.5 finds the CSR kernel, gather included, faster than the dense apply at every lane width at $d = 0.0013$, the density of the `proj_conv` input, and slower at every shape above $d = 0.2$ at fp64, $d = 0.091$ at fp32x2 and $d = 0.068$ at fp16x4. In the general case, its overhead outweighs the arithmetic it saves.

Family	Variants	Operates over
Dense apply	<code>bn_apply_fp16x4</code> , <code>bn_apply_fp32x2</code> , <code>bn_apply_fp64</code> , <code>bn_apply_c</code>	the full [site, channel] tensor, reading the interleaved <code>kl</code> table directly
CSR gather	<code>bn_csr_expand_fp16x4</code> , <code>bn_csr_expand_fp32x2</code> , <code>bn_csr_expand_fp64</code>	the CSR index array, gathering <code>table[idcs[k]]</code> into an interleaved κ/λ stream
CSR apply	<code>bn_csr_apply_fp16x4</code> , <code>bn_csr_apply_fp32x2</code> , <code>bn_csr_apply_fp64</code> , <code>bn_csr_apply_c</code>	the CSR value array, over the stream the CSR gather stage (or the host) produced

Table 3.5: BN kernel variants by role

3.3.5 Fused Kernels

The BN kernel just described runs as a separate launch: it reads the buffer written by the previous kernel and writes a buffer that the next kernel reads. The convolution pipeline offers an alternative. It chains the convolution, with either the `spmva` or the `indir` engine (Sections 3.3.2 and 3.3.3), the CSR compaction that follows it, and a BN pass within one kernel launch over one set of TCDM buffers. When BN is fused, the CSR variant of the BN apply kernel of the matching lane width runs in place over the compacted values, immediately after `conv_compact_csr` produces them, split evenly across all eight workers. Fused BN is available only together with CSR output: the build rejects a configuration that requests it on the dense output.

The CSR output of the convolution also lets BN skip a stage that a general CSR input needs. On a general CSR input, the BN kernel gathers κ and λ for every nonzero with `bn_csr_expand_*`, because a general CSR pattern is irregular and the coefficient at a given position depends on the column that nonzero occupies. The CSR output of the convolution is not irregular in that sense.

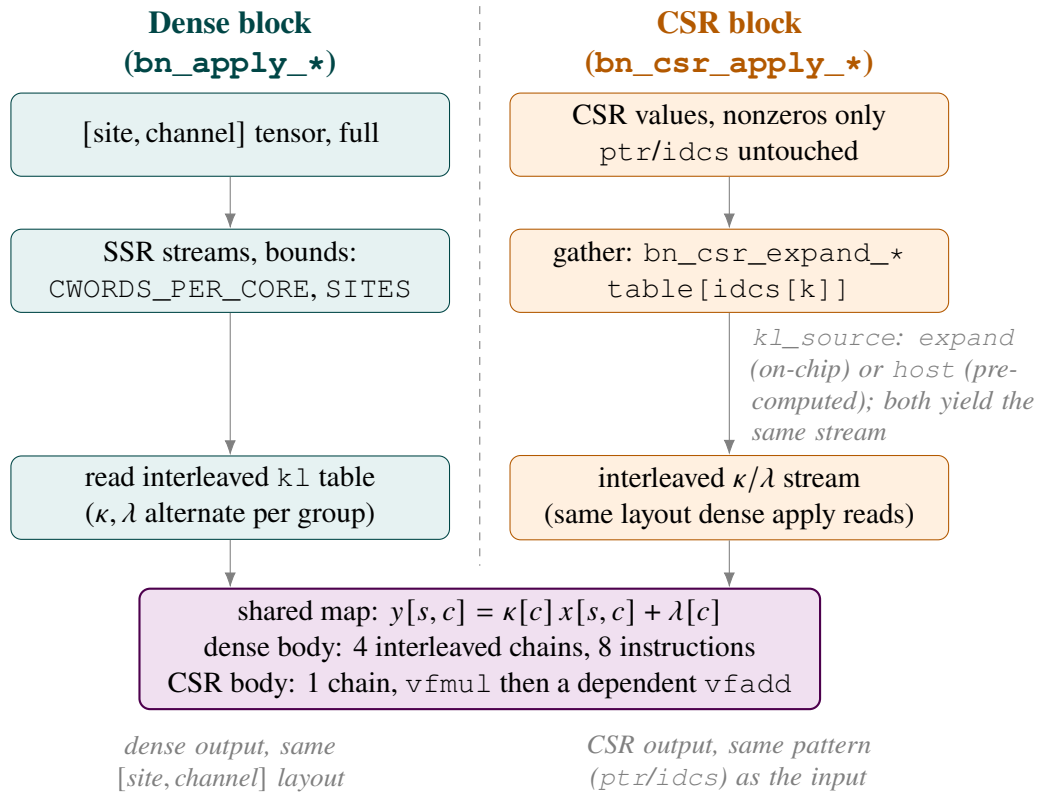


Figure 3.12: Dense and CSR BN kernel bodies

It is a row-sparse container that keeps a full COUT-wide row for every non-empty site, so the column indices of the stored values are $0, \dots, \text{COUT} - 1$ on every row present. The κ/λ pattern is therefore determined by column position alone and known before the convolution runs. The host folds it into one interleaved table, and the kernel proceeds directly to the SIMD apply body, skipping the gather that the general case requires. This follows from the container and not from fusion: a separate BN launch over the same container, given the same table, skips the gather as well.

Because the compaction and the fused BN both store fp16 values through the FPU, an explicit `__rt_fpu_fence_full()` must separate the last of those stores from the DMA core reading the buffer back out. Without it, the DMA transfer can start before the store retires, which produces wrong data without raising a fault.

Fusing a kernel does not remove the data BN needs; it only changes where that data must come from. A fused kernel requires that data to be present, in the layout the fused stage expects, before its launch begins. Here, the host must compute and transfer the interleaved $k1$ table, one word of κ followed by one word of λ per position, together with the weights of the convolution. Once resident in TCDM, the table needs no work at run time, because it depends only on the trained BN parameters and never on the spikes the convolution produces, but the host must still prepare and transfer it.

What fusion itself removes is therefore small: the round trip of the convolution output to memory and the barriers of a second launch. The fused apply runs the same body as the separate

one. Section 4.2.5 measures the saving at below 1% at the density of the Q , K and V input, so the fused pipeline is kept for its dataflow and not for its cycle count.

3.3.6 LIF Neuron Update

The LIF kernel implements the SpikingJelly LIF neuron update with a hard reset to zero, run over `TIMESTEPS` timesteps for `NEURONS` neurons. Per timestep,

$$H = VK_1 + XK_2, \quad S = \begin{cases} 1 & \text{if } H \geq V_{th}, \\ 0 & \text{otherwise,} \end{cases} \quad V \leftarrow H(1 - S), \quad (3.8)$$

matching the recurrence introduced in Section 2.2: K_1 plays the role of the leak factor τ , V_{th} is the same firing threshold, and X carries the weighted input already produced by the preceding kernels. Both constants derive from the membrane time constant τ_m of SpikingJelly, $K_1 = 1 - 1/\tau_m$ and $K_2 = 1/\tau_m$. The leak factor of the earlier formulation thus corresponds to $1 - 1/\tau_m$, and the input is scaled by $1/\tau_m$ before it is integrated. Two details refine that earlier definition. First, the reset to zero that Section 2.2 states as a consequence of firing is written here into the update itself, $V \leftarrow H(1 - S)$, and it is hard: a neuron that fires restarts the next timestep from exactly zero, with no residual membrane potential. Second, Section 2.2 fires on a strict inequality, $\mathbf{u}[t]^l > V_{th}$, whereas this kernel fires on $H \geq V_{th}$, the SpikingJelly convention, threshold included. This choice determines how the comparison itself is implemented.

The comparison is not implemented with a compare instruction. On the Snitch cluster, the SIMD compare instructions write their result, a lane bitmask, to the integer register file, and the FREP sequencer does not accept instructions that synchronize the integer and floating-point pipelines. A compare inside the timestep loop would therefore prevent that loop from being replayed by FREP. The Heaviside step is instead rebuilt from sign bits. The kernel forms $D = H - V_{th}$, then

$$S = \max(+\text{sign}(D), 0), \quad M = \max(-\text{sign}(D), 0) = 1 - S, \quad (3.9)$$

replacing a compare and a branch with two `fsgnj`-and-`fmax` sequences that remain in the floating-point pipeline. This reproduces the $\geq$ firing condition, threshold included, because IEEE 754 gives D a definite sign even when it is exactly zero, so a neuron with H numerically equal to V_{th} still resolves to $S = 1$. Figure 3.13 works through the whole step. V , K_1 , X and K_2 combine into the candidate membrane potential H , and the spike S is the positive part of the sign of D . Its complement M controls the reset, $V \leftarrow H \cdot M$, and the dashed box names the compare-and-branch sequence that this construction replaces.

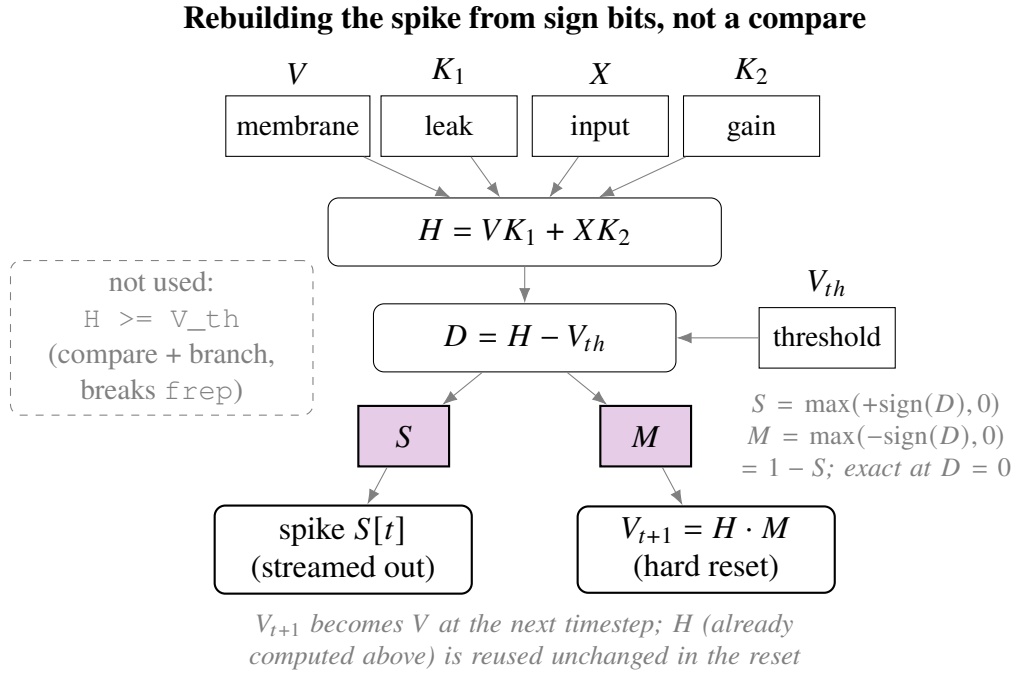


Figure 3.13: Hard-reset LIF update computed from sign bits

Every LIF kernel shares the same signature, $(x, s, words, k)$, and splits its work over neurons. The membrane state of each core is kept entirely in registers and never spilled to memory across timesteps. Two SSRs stream the input x in and the spikes s out, following the same three-dimensional access pattern, whose dimensions are, from the innermost, the chain within a block, the timestep, and the block. Neither operand therefore needs an explicit address computation inside the timestep loop. Table 3.6 lists the variants of the kernel.

Family	Variants	Role
Interleaved	<code>lif_update_fp16x4</code> , <code>lif_update_fp32x2</code> , <code>lif_update_fp64</code>	4 independent $V \rightarrow H \rightarrow V$ chains run per block, hiding FPU latency across chains
SIMD, packed constants	<code>lif_update_packed_fp16x4</code> , <code>lif_update_packed_fp32x2</code>	constants taken pre-packed from k rather than splatted; <code>frep</code> replays the whole 32-instruction timestep body, exactly filling the sequencer buffer
Reference	<code>lif_update_c</code>	plain C, generic over the lane type

Table 3.6: Variants of the LIF kernel

The variants differ along two axes. The first concerns latency hiding. `lif_update_fp16x4`, `lif_update_fp32x2` and `lif_update_fp64` each interleave four independent membrane chains inside one block, so that while one chain waits on the latency of its $H \rightarrow V$ dependency, the other three keep the FPU issuing. The plots of Chapter 4 label these three kernels by these names.

The second axis, the delivery of constants, applies only to the SIMD widths. The base `fp16x4/fp32x2` kernels splat their five constants, `ONES`, `ZEROS`, `K1`, `K2` and `V_th`, from scalar immediates with `vfcпка/vfcпkb` in the prologue. The packed variants instead take them already packed as `fp64` words from `k`, replicated once per worker to avoid TCDM bank conflicts. They trade the pack prologue and its branch overhead for a timestep body that `frep` replays whole. At 32 instructions, this body fills the instruction buffer of the sequencer with no headroom left. The plain-C kernel `lif_update_c` serves as a reference, against which every other variant is checked bit-exactly (Appendix B).

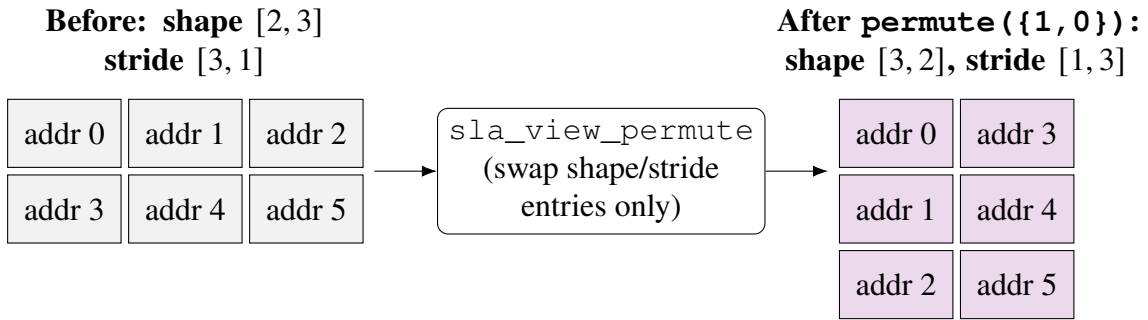
None of these kernels inspects its input, so their cost is independent of the density of the spikes they receive: every neuron is updated at every timestep, whether its input is zero or not. Sparsity could be exploited around the kernel instead of inside it, by *dead-neuron compaction*. A neuron whose input is zero at every timestep can never fire, since its membrane potential starts at zero and only decays, $H = VK_1$, without reaching V_{th} . Such neurons could be removed before the kernel runs, and the same dense kernel applied to the remaining ones. We do not adopt this approach, because the compaction is not a lightweight operation. Like the CSR conversion of Section 3.3.7, it is a scan that reads and tests every element of the input whatever the data holds. Performing it on the host, outside the measured kernel, would credit the LIF update with a saving whose cost is paid elsewhere. Chapter 4 therefore evaluates the LIF kernels in their dense layout, on the full input, where the gain is expected to come from lane width rather than from sparsity. Exploiting the sparsity of the LIF input, either through a compaction cheap enough to pay for itself or through a kernel that visits only the nonzero inputs, is left to future work (Section 5.3.1).

3.3.7 View, Reshape, and Sparse Compaction

Every kernel above expects its operand in one fixed layout. The SpMVA and the convolutions read CSR, the SpMVA reads its dense operand as a lane-interleaved word stream, and the LIF update reads a dense [timestep, neuron] array. Converting a tensor from the layout in which the model produces it to the layout a given kernel expects is itself a shape operation, which in PyTorch would be a single `.permute()`, `.reshape()` or `.to_sparse_csr()` call. On the Snitch cluster these operations are expressed through the SSR address generators by two headers, `sla_view.h` and `reshape.h`, divided in the same way PyTorch separates a view from a copy.

`sla_view.h` contains only metadata. A `sla_view_t` holds shape, stride and lane count in the PyTorch sense, together with the operations that never touch data. `sla_view_init` sets up contiguous strides, `sla_view_permute` and `sla_view_transpose` reorder axes, and `sla_view_narrow` slices one axis, which is how a core takes its share of a work split. `sla_view_reshape` is a direct port of the stride computation of PyTorch. It succeeds only when the new shape is stride-compatible, that is, exactly when the `.view()` of PyTorch would succeed instead of forcing a copy. `sla_view_merge` coalesces axes whose strides chain,

which allows a six-dimensional Transformer view to be reduced to the four dimensions an SSR can express. `sla_view_lower` turns a merged view into concrete bound and stride registers. It refuses to do so if the view still has more than four axes, or if it moves the SIMD-packed lane axis away from the innermost position, since such a permutation would require repacking the lanes. The header treats the latter case as a hard assertion rather than a silent slow path, because a stride error on this hardware produces wrong data without raising any fault. The principle the header encodes is that a permutation moves no data once it is folded into the SSR configuration of the consuming kernel. It becomes a different set of bound and stride registers, with no TCDM traffic, just as a PyTorch view moves no data. Building and lowering the view is not free, however: it is a fixed cost per call, independent of the tensor size, which Section 4.2.9 measures at 394 to 557 cycles. Figure 3.14 shows a permutation on a 2×3 view. After `sla_view_permute(v, {1, 0})` the same six words are read as a 3×2 view, visiting addresses 0, 3, 1, 4, 2, 5 instead of 0, 1, 2, 3, 4, 5, and only the shape and stride entries change.



Both grids describe the very same six 64-bit words; only which address each position (i, j) of the view maps to changes, since `sla_view_permute` swaps entries of the shape and stride arrays and touches no data. `sla_view_lower()` turns the right-hand stride pair into the SSR's bound and stride registers, so a permute reaching the hardware is a different pair of registers, never a different set of bytes.

Figure 3.14: A 2×3 view and its permutation

`reshape.h` is the materializing layer built on top of it. It is used when a view cannot be folded into the consuming kernel, either because the SSR streams of that kernel are already in use, or because the data must cross a memory boundary in any case. It is the fallback that performs an actual copy, as the `.contiguous()`, `.reshape()` and `.to_sparse_csr()` calls of PyTorch do. `sla_contiguous` is the dense-to-dense port of `.contiguous()`, together with a plain-C reference, `sla_contiguous_c`, that walks the view with an explicit multi-dimensional counter. As Figure 3.16 shows, one read SSR carries the permuted geometry of the view, up to four dimensions, and one write SSR carries the linear destination. A single `fsgnj.d` inside an `frep` loop moves each word from the first stream to the second, and the loop is split into chunks of at most 65536 words, the largest repeat count a single `frep` can encode. The address generators, not additional instructions, express the permutation, so the copy loop issues no instruction for it: besides building the view, only the copy itself costs cycles.

`sla_to_sparse_csr`, with its plain-C counterpart `sla_to_sparse_csr_c`, is the dense-to-CSR port of `.to_sparse_csr()`. It emits the same `a_vals/a_ids/a_ptrs` triple that the SpMVA kernels consume. It still streams the dense read through an SSR, but the compaction itself cannot be placed inside an `frep`. Testing each streamed value for zero involves the integer register file, and the sequencer does not accept instructions that synchronize the integer and floating-point pipelines, the same constraint that required the LIF spike function to be compare-free (Section 3.3.6). Figure 3.17 traces the kernel on a 2×3 example with rows `[0, 5, 0]` and `[3, 0, 0]`. The scan visits every element in row-major order and tests it for zero. Only the two nonzero entries advance the shared counter at which `a_vals` and `a_ids` are appended, and `a_ptrs` records the value of that counter at every row boundary.

This running counter is the reason why the compaction runs on a single core. Each output position depends on how many nonzeros preceded it, so position $k + 1$ cannot be written before the decision for position k is known. Only the running total, one addition per row, is inherently serial, however. The surrounding work is a scan in which every element is read and tested against zero, a test with no dependency across rows. A prefix sum over per-core counts would therefore parallelize the scan as every other stage of this chapter is parallelized, leaving only the row bookkeeping on one core. No compaction in this work implements that prefix sum. A sparse operand does not make the missing parallelization cheap, because converting to CSR is a scan and not a gather: every element of the dense input is read and tested whatever the data holds, and only the surviving elements pay the two stores. The cost is therefore set by the size of the dense operand, not by its sparsity. A tile at 2.8% density still costs about half as much to convert as the same tile fully dense, and the share of the conversion in a sparse pipeline is largest where the sparsity is most extreme. Chapter 4 measures this cost separately from the kernels that consume its output.

The attention block of Section 3.3.9 needs an efficient conversion to CSR of packed fp16x4 data, while `sla_to_sparse_csr` accepts only fp64 elements. `reshape.h` therefore also contains `sla_to_spike_csr_fp16x4`, a port written for that need, which the attention block runs as `ab_compact_fp16x4`. It is not a general fp16x4 conversion. It accepts only spike tensors, whose lanes hold either +1.0 or +0.0, and it takes its speed from that restriction. A general conversion of packed fp16x4 data, which must also store the values, is not part of this work. The port differs from `sla_to_sparse_csr` in four respects. Since every spike is exactly 1.0, it stores no values, only the indices. It tests the packed words as integers, without unpacking the lanes, since any set bit marks a spike. It reads a strided window instead of a view, `rows rows of words words placed stride words apart`, which is the shape of a (chunk, head) tile of a channels-last buffer. Finally, it emits the index of lane l of word w as $(4w + l) \cdot s$, where the scale s is what the consumer adds to its base address: the bytes of a weight row, the words of a row, or 1 for a plain column index.

Figure 3.15 traces one row of the window, where the test runs on three levels. Four words are loaded and combined with a bitwise OR under a single branch, so a silent group skips 16

lanes at once. Inside a live group each word is tested on its own, so a zero word costs one branch, and only a live word is split into its four lanes. Like `sla_to_sparse_csr`, the kernel cannot run inside an `frep`, because its test writes the integer register file. Unlike it, the kernel runs on several cores without a prefix sum: every call starts its own row pointers at zero, so the cores compact disjoint windows into disjoint lists. A capacity bounds each list; past it, the kernel keeps counting but stops storing, so an overflow is reported instead of being written over the list that follows.

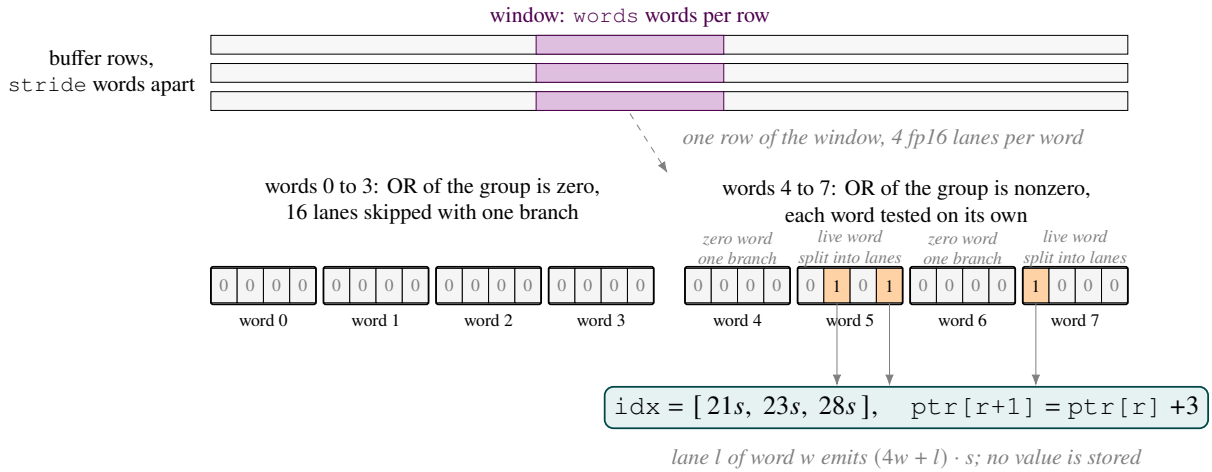
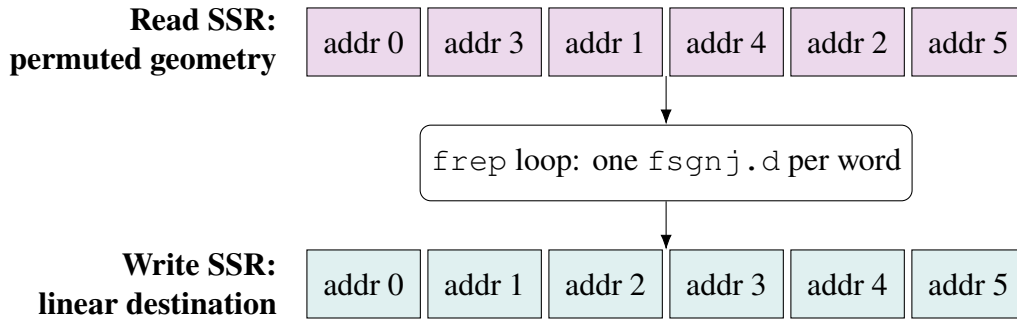


Figure 3.15: `sla_to_spike_csr_fp16x4` on one row of a packed `fp16x4` window

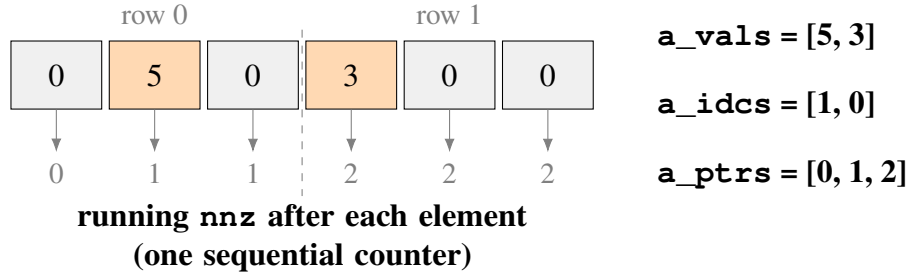


The read stream pops the six words in the permuted order of Figure 3.14, 0, 3, 1, 4, 2, 5; the write stream pushes them straight into a new, linear buffer, 0, 1, 2, 3, 4, 5. The permutation costs nothing beyond the SSR configuration already fixed when the view was built: the only instructions the `frep` loop issues here pay for the mandatory copy, chunked at `SLA_FREP_MAX` = 65536 words per `frep`.

Figure 3.16: `sla_contiguous` materializing a permuted view

Table 3.7 lists the four shape operations that the attention block requires in the PyTorch layout of the model, each a direct port of the PyTorch expression that previously ran on the host. The `chunk` case can be divided across cores only along its outermost axis, whose extent is `NUM_CHUNKS` = 2, so at most two of the eight cores are busy while it runs. This suggests moving that permutation into a DMA descriptor instead of a kernel, and Chapter 4 confirms the limit in its core-scaling measurements. The `bvals` case moves the lane axis away from the

Dense source, row-major scan order



Only the values 5 and 3 (highlighted above) are nonzero, so only they are appended to `a_vals/a_idcs`; `a_ptrs` snapshots the running count at each row boundary. Every output position is the running nnz at the moment its element is reached, so position $k + 1$ cannot be written before position k 's decision is known, the dependency that keeps this stage single-core.

Figure 3.17: `sla_to_sparse_csr` on a 2×3 example

Case	PyTorch expression	Consumer
chunk	<code>x.view(nc, cs, B, h, N, d) . permute(0, 2, 3, 1, 4, 5) . reshape(nc, B, h, cs*N, d)</code>	the windowed-attention chunking; the reshape after the permute is not stride-compatible, so this is the one case that genuinely pays for a copy
bridge	<code>x.transpose(4, 3) . reshape(T, B, C, N)</code>	the dense [timestep, neuron] layout <code>sla/lif</code> reads
bvals	<code>v.reshape(seq, hd/LANES, LANES) . permute(1, 0, 2)</code>	the lane-interleaved layout <code>sla/snn</code> reads
csr	<code>k.transpose(-2, -1) . to_sparse_csr()</code>	the <code>a_vals/a_idcs/a_ptrs</code> triple <code>sla/snn</code> and <code>sla/csrmv</code> read, with the transpose fused directly into the compaction so K^T is never materialized

Table 3.7: Shape operations ported from PyTorch

innermost position, which the view header refuses to lower silently, so it is guarded by an explicit check.

3.3.8 A Fused Attention Sub-Block

Section 3.3.5 fused two stages within the convolution pipeline. The same approach can be applied to a larger portion of the attention block. The kernel `sla/snn_merge` chains, within one cluster and one TCDM residency, the two 1×1 convolutions of the module that produce K and V , their BN layers, the two LIF updates they drive, the compaction of $K^\top$ to CSR, and the sparse product with V . It computes one (chunk, head) slice of the attention: two timesteps and four tokens, that is, eight sites, over the 96 channels of one head. Table 3.8 lists the stages. Stages A and B run twice, once for the K branch and once for the V branch, before the two LIF updates (0a, 0b) turn each branch into spikes; stages 1 to 2 then run once, over $K^\top$ and V together. Stages 1a and 1b move data rather than compute. Stage 1a rebuilds the column indices scaled by the row stride, which is necessary because the head dimension, 96, is not a power-of-two number of words at any lane width, and stage 1b packs V into the lanes the product consumes. Stage 2 exists in two interchangeable forms, a row-accumulating one and the inner-product kernel of Section 3.3.1 reused unchanged, so that the two can be compared within the same pipeline. A hardware barrier separates every pair of stages, since the work splits differ between stages. BN is split over channels, so that no two cores ever access the same κ/λ entry, while the convolution and the product are split over sites and rows respectively.

Two of the stages reuse the standalone kernels directly. The LIF and BN kernels are included from their own sources rather than copied, so `lif_update_fp64` and `bn_apply_fp64` are the same object code that the standalone kernels run. The convolution is not reused, because of the constraint discussed in Section 3.3.3. At the per-head width, $c_{\text{out}} = 96$, a row of $\mathbf{W}$ occupies 768 bytes at fp64, so the `indir` engine cannot be built at any lane width. The sub-block uses instead its own `sm_conv1x1_fp64`, which takes the index list already scaled to byte offsets and omits the shift entirely. This saves one instruction per gather with respect to `indir`, requires no padding of c_{out} to a power of two, and applies the same prescaling that the `spmva` engine uses. The compaction and the product are also local, for a similar reason: they require a strided read and a lane-packed result that the standalone kernels do not accept as arguments. The element-wise half of the block is thus reused unchanged, while every stage whose addressing the surrounding pipeline alters had to be rewritten.

Chaining these stages keeps every intermediate result in the TCDM, so that the dataflow is the only thing under test. The convolution, the BN and the LIF update divide across the eight workers like every other kernel in this chapter. The CSR compaction that turns K from a dense LIF output into the form the product expects, instead, runs on a single core, for the reason given in Section 3.3.7. Chaining does not make that compaction cheaper, but its input arrives from the LIF and its output goes to the product with no transfer to L3 on either side.

Stage	Kernel	Split
A (×2)	sm_conv1x1_fp64	sites, 8 workers
B (×2)	bn_apply_fp64, in place	channels, 8 workers
0a, 0b	lif_update_fp64	neuron words, 8 workers
1	sm_to_csr	single-core (a_ptrs is a running total)
1a	sm_scale_idcs	single-core, one integer pass over the nonzeros
1b	sm_pack_*	lane-packs V and the result, 8 workers
2	sm_spmv_* or sm_spmv_innerp_*	CSR rows, 8 workers

Table 3.8: Stages of the fused attention sub-block

Keeping one representation across five stages is affordable only because moving between those stages costs nothing. The natural output of the convolution is $[\text{site}, \text{COUT}]$ with COUT innermost. This is the $[\text{site}, \text{channel}]$ layout the subsequent in-place BN expects, and the $[T, \text{NEURONS}]$ layout the LIF update expects after it. One buffer per branch, K or V , therefore carries all three stages with no reshape and no copy between them. The transition into the CSR compaction rests on the same fact. The LIF writes $[T][\text{NEURONS}]$ with $\text{NEURONS} = N \cdot d$, and the compaction reads $K[\text{seq}][\text{hd}]$ with $\text{seq} = T \cdot N$ and $\text{hd} = d$, so in row-major order the two are the same bytes. The compaction reads K^T directly from that buffer, with row stride 1 and column stride hd , instead of materializing a transpose first. This is the zero-cost permutation of Section 3.3.7, applied where the two layouts already coincide, so that not even a view descriptor is needed. The pipeline thus keeps every stage feeding the next directly in TCDM, instead of staging intermediate results out to L3 and back between kernels.

As a result, the fused sub-block keeps the entire path from the shared x_for_QKV activation to the $K^T V$ product resident in the TCDM of one cluster. One DMA transfer brings in both weight matrices, the folded κ/λ tables and the single copy of x_for_QKV that both branches read, since Q , K and V share their input, and one DMA transfer writes the result back.

The sub-block is currently built with 32 input channels rather than the 768 of the deployed model, because both weight matrices must reside in TCDM at once and there is not yet room for the full width; every other dimension is that of the (chunk, head) slice of the model. Reaching the full width requires the streamed weight loading discussed as future work in Section 5.3.1. Two further limits apply. The weights are synthetic rather than trained: they are structured so that K and V have a closed form from which the reference output can be derived in exact rational arithmetic. In addition, the synthetic input activates 12 of the 32 input channels, a density of 0.375, at every site except the first, which is left empty to exercise the empty-row path, and each column of the synthetic weights holds 5 nonzeros in 32 rows. Each output element is therefore the sum of about two nonzero weights, rather than the roughly 262 weight rows of Section 3.3.2, so the sub-block says nothing about how a deep fp16 accumulation behaves near a firing threshold. For these reasons the sub-block is presented as a demonstration of dataflow and is not part of

the evaluation of Chapter 4. It nevertheless shows that a single kernel launch on this cluster can span an entire dynamically sparse sub-block, comprising two convolutions, two affine BN maps, two spiking nonlinearities, a format conversion and a sparse product, with a single round trip to memory.

3.3.9 The Attention Block Application

The application `sla/attention_block` runs the whole attention block of STAtten, MS_SSA_Conv, on one Snitch cluster, at fp16x4 on 8 cores. It chains the shortcut LIF, the Q , K and V convolutions with their BN and LIF, the products $K^T V$ and $Q(K^T V)$, the attention LIF, the output projection `proj_conv` with its BN, and the residual addition, in 26 stages separated by barriers. It runs at the shape of the profiled model, $T = 4$, $N = 4$ (16 sites) and 768 channels in 8 heads, except for the input channel count of the Q , K and V convolutions, which is reduced so that each weight matrix fits in the TCDM.

Making kernels work together is different from running each of them in isolation. Every kernel of this chapter was designed for its own shape and layout, and the attention block breaks those assumptions in four ways. First, every activation is stored channels-last, as [site, channel] with four fp16 values per word, so that every LIF and every BN sees a single geometry and none of the dense copies of Table 3.7 is needed. A (chunk, head) tile is then a window of that buffer, 24 words wide inside rows 192 words apart, while the standalone kernels assume that each row is contiguous and as wide as the step between rows. Second, at 768 channels a row of W occupies 1536 bytes, which `indir` cannot address, and the scaled index of `spmva` no longer fits a `uint16_t` (Section 3.3.3). Third, the spike tensors to compact are packed four lanes per word, while the fp64 compactions `sla_to_sparse_csr` and `sm_to_csr` of Sections 3.3.7 and 3.3.8 process one element at a time, and the first of them rejects packed input. Fourth, those compactions run on a single core, because of their running total.

The LIF and BN stages still run `lif_update_packed_fp16x4` and `bn_apply_fp16x4` unmodified. The other stages run five local kernels. `ab_compact_fp16x4` is the only new algorithm among them: it is the spike compaction `sla_to_spike_csr_fp16x4` of Section 3.3.7 (Figure 3.15), and it converts the convolution input, K , Q and the attention output to CSR, each core over its own sites or tiles. `ab_conv_fp16x4` is the convolution of the sub-block, with its byte-offset indices, at fp16x4. `ab_ktv_fp16x4` computes $K^T V$ by scattering over the row-major CSR of K : every nonzero $K[r][d]$ adds row r of V into row d of the result. This performs the same additions in the same order as a gather over the CSR of K^T , so the result is identical, and it avoids transposing packed data, which would move the lane axis. `ab_qktv_fp16x4` is the row product of the sub-block with word-offset indices, and it writes each output row directly to its (site, head) place in the attention output. `ab_add_fp16x4` computes the residual, a dense element-wise addition that no other kernel provided. The output

of every stage is checked bit for bit against a reference computed in fp16 in the operation order of the kernel, and Section 4.2.11 reports the cycles of each stage.

Three of these kernels are ports rather than new algorithms: `ab_conv_fp16x4`, `ab_qktv_fp16x4`, and `ab_ktv_fp16x4`, which reuses the same row-addition body. They exist because the originals fix two addressing choices, a row address formed by shifting the index and a row step equal to the row width, not because the computation required a new kernel. Generalizing each original kernel, with byte-offset indices for the convolution and a row step separate from the row width for the row product, would remove the ports, and it is left to future work (Section 5.3.1).

The kernels described in this chapter, together with the tensor shapes and densities suggested by the profiling of Section 3.1, define what Chapter 4 evaluates. For the SpMVA, the convolution and BN, it measures the speedup against density and the scaling with the number of cores, including the conditions under which a kernel loses to a dense implementation. It reports FPU utilization for the SpMVA and the convolution, and FPU issue occupancy for the LIF update, whose speedup comes from lane width rather than from density. The shape operations are measured against their plain-C references, and the attention block stage by stage.

Chapter 4

Results and Discussion

4.1 Measurements

This section defines how the results of this chapter are measured and on which configurations: the speedup metric (Section 4.1.1), core scaling (Section 4.1.2), the software and hardware setup (Section 4.1.3), and the tensor shapes and densities (Section 4.1.4). Section 4.2 then reports the results kernel by kernel.

4.1.1 Cycles and Speedup

Speedup is the ratio between the cycle count of a reference implementation and the cycle count of the implementation under test. Each figure states its reference explicitly. In some cases the reference is the dense baseline of the same kernel, to show how increasing sparsity affects the result. Cycle counts are exact, since they come from cycle-accurate simulation. The cycle count of a multi-core run is that of the slowest core, because the slowest core determines when the kernel completes. Not every kernel splits its work evenly across cores, and the balance depends on the input data.

4.1.2 Core Scaling

We also measure how the speedup scales with the number of cores, running each kernel on 1, 2, 4 and 8 cores where possible. Operations that run on a single core, such as the CSR compaction, are excluded from this metric.

4.1.3 Software and Hardware Setup

We developed and evaluated the kernels inside the cycle-accurate simulation container built around the Snitch cluster software stack, described in Appendix A.

On the hardware side, every design choice in this chapter is shaped by the constraint introduced in Section 2.6. The TCDM of the Snitch cluster holds only 128 KiB, is entirely software-managed,

and has no cache hierarchy behind it to absorb an operand that does not fit. The budget every kernel is written against is smaller still, 112 KiB, since the runtime reserves the rest. Each application states it as a `static_assert` over its whole TCDM structure, so that an operand that does not fit fails at compile time rather than at run time. We therefore limited the scope of our Transformer implementation to tensor sizes and orientations that fit inside that budget, as the rest of this section details.

4.1.4 Tensor Sizes

The sizes used by the full STAtten implementation depend on the input size and on the number of heads. With the chosen datasets, composed of 32×32 pixel images, the membrane of activations that is processed into Q, K and V is $[4, 768, 2, 2]$, and Q, K and V are $[4, 1, 8, 4, 96]$ each.

Given the memory limits discussed above, we scaled down the tensor sizes while keeping the tensors plausible and varied in both density and shape. Table 4.1 summarizes the tensor shapes used, which span three size classes and both standard and flipped dimension orderings.

Shape	Class	Orientation	Notes
8×96	small	standard	the STAtten attention tile; a few thousand cycles, so a full density grid is minutes not hours
8×256	medium	standard	the largest power-of-two input where a TCDM-resident weight matrix still fits for all three data types
8×1024	large	standard	deliberately past that limit: fp16x4 fits, fp32x2 and fp64 do not
16×128	medium	flipped	the element count of the medium case over twice as many rows
128×64	large	flipped	the element count of the large case with more rows than columns

Table 4.1: Tensor shapes used in the kernel evaluation

Each kernel maps these five shapes onto the axes it counts in: site count and input channel count for the convolution, sites and channels for BN, and, for the LIF update, which has one axis, 768, 1024 and 2048 neurons at four timesteps. A LIF holds two lane-word arrays of $[T, \text{neurons}]$ resident at once, so 8192 neurons exceed the budget in every data type. 1024 neurons is the largest size at which all three data types fit, and 2048 exceeds the budget for fp64 alone, the role that the 8×256 and 8×1024 shapes play for the other kernels. Where a configuration does not fit, we record the failure, since the position of that limit is itself a result. A CSR container carries

two words per stored value plus an index list and two per-channel tables, so at equal element count it is the larger of the two formats and reaches the budget first.

We include flipped tensors because the proposed kernels are not symmetric operations: some inputs are in CSR and others are dense, so flipping the input tensors does not yield an equivalent computation. In general, matrix multiplication is symmetric only under conditions that are not satisfied in our case.

Chapter 3 reports the sparsity of a tensor, the fraction of its elements that are zero. The results of this chapter use its complement, the density d , the fraction of elements that are nonzero, so that $d = 1 - \text{sparsity}$. For example, the Q tensor, 96.87% sparse on average in Figure 3.4, has a density of about 0.031.

All of the above tensors are generated at the nine density points of Table 4.2. Three of them stand for tensors we dumped while profiling the model: 0.0013 for the input of `proj_conv`, the spike output of the attention LIF; 0.028 for the spike tensors Q , K and V ; and 0.3711 for `x_for_QKV`, the input of the Q , K and V convolutions. On the CIFAR-100 inputs of Table 3.1 these tensors have mean densities of 0.0014, 0.031 and 0.34. The point 0.0117 stands for a K^T channel sparser than that tensor mean, about one nonzero in a row of 96. The other five points cover the range up to a fully dense matrix, so that the density at which a sparse kernel crosses its dense baseline can be located.

Density	0.0013	0.0117	0.028	0.0397	0.10	0.20	0.3711	0.60	1.00
Dumped tensor	•		•				•		
Regime	very sparse			moderately sparse			dense		

Table 4.2: The nine density points of the generated tensors

The shapes above are derived from those of the profiled model, and together with the density grid they describe the behavior of the kernels under different conditions, including conditions unfavorable to them. For example, we feed kernels designed for sparsity with dense matrices stored in CSR format, which are known in advance to perform poorly, and we measure this case as well.

In some cases only a subset of the nine points is used. The most common is the five-point subset 0.0013, 0.028, 0.1, 0.3711 and 0.6, which keeps every dumped density: the SpMVA figures report it, and the slowest plain-C references are measured on it. Core scaling is measured at one to three points among 0.028, 0.3711 and 1, and the LIF kernels, whose cost does not depend on density, are reported at a single density. Not every kernel is measured at every combination of Table 4.1 and Table 4.2: each kernel runs on the subset of shapes and densities that its operands allow. Some combinations do not fit the 112 KiB TCDM budget at all, and others, large tensors at high density fed to kernels designed for sparsity, take too long to simulate cycle by cycle to be swept in full. Each figure in Section 4.2 states the subset it reports.

4.2 Plots

This section reports the results kernel by kernel: the SpMVA (Sections 4.2.1 and 4.2.2), the convolution (Sections 4.2.3 and 4.2.4), BN (Sections 4.2.5 and 4.2.6), the LIF update (Sections 4.2.7 and 4.2.8), and the reshape operations (Sections 4.2.9 and 4.2.10). Section 4.2.11 then runs these kernels together as one attention block.

4.2.1 SpMVA: Speedup Against Sparsity and FPU Utilization

Figures 4.1 through 4.3 report the speedup of the ISSR-based SpMVA kernel, `issr_fmadd`, over the plain-C fp64 reference `plain_c` of Section 3.3.1, one figure per lane width. The reference does not depend on lane width, so the three figures share it. One pass of a packed kernel produces two or four output columns where the reference produces one, so the fp32x2 and fp16x4 speedups are computed per output column: the reference cycle count is multiplied by the lane count before the division. Bars are grouped by tensor shape from Table 4.1 and, within a group, cover five of the nine density points of Table 4.2: 0.0013, 0.028, 0.1, 0.3711 and 0.6. The right axis shows FPU utilization inside the same kernel window, that is, the fraction of cycles in which the FPU issues an arithmetic operation, FREP replays included, against a ceiling of 100% at one operation per cycle. It is read from the trace of the slowest worker and cross-checked against the cycle count recorded in the binary.

The multiplication accounts for little of the gain. At 8×96, replacing the MAC of `issr_fmadd` (Section 3.3.1) with the addition of `issr_fmadd` changes the cycle count by 0.98× to 1.10× across the density grid, a median saving of about 5%. The gain comes instead from three sources: the indirect SSR removes the per-element address computation and branch, the FREP prologue is amortized over long rows, and two or four lanes are packed per word.

Speedup grows with density up to $d = 0.3711$ at every shape and lane width. Plain C repeats the same per-element branch and address computation for every nonzero, so its cost per nonzero is roughly constant. The ISSR kernel pays a FREP prologue once per row and then streams the remaining nonzeros through the indirect SSR at close to one addition per cycle, so a longer row amortizes more of that prologue. The largest shape, 8×1024, leads at every density in all three figures, reaching 8.50× at fp64, 17.31× at fp32x2 and 34.02× at fp16x4, all at $d = 0.3711$. The 16×128 shape is at the other end. It spreads the element count of 8×256 over twice as many rows and holds about three nonzeros at $d = 0.0013$, so no row is long enough to enter the FREP loop, and fp64 falls to 0.92×, just under the plain-C baseline. It recovers with density and ranks third of the five shapes at $d = 0.6$. Between $d = 0.3711$ and $d = 0.6$ the speedup of the two largest shapes levels off at every lane width: at fp64, 8×1024 goes from 8.50× to 8.45× and 128×64 from 6.22× to 6.29×.

Dividing the three figures by each other isolates lane width, since the baseline is shared. At $d = 0.6$ the packed kernels span 12.49× to 16.83× at fp32x2 and 23.74× to 33.63× at fp16x4; on

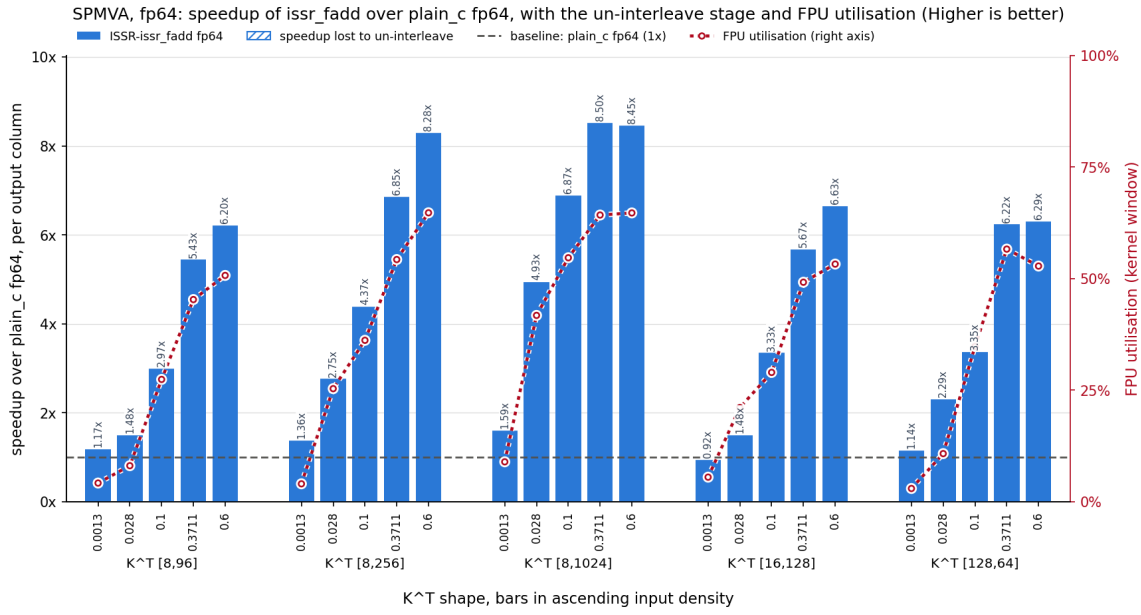


Figure 4.1: fp64 SpMVA speedup over plain C and FPU utilization against input density, K^T at 8×96 , 8×256 , 8×1024 , 16×128 and 128×64 . In the legend, ISSR-issr_fadd is the SpMVA kernel issr_fadd and plain_c its plain-C reference (Section 3.3.1).

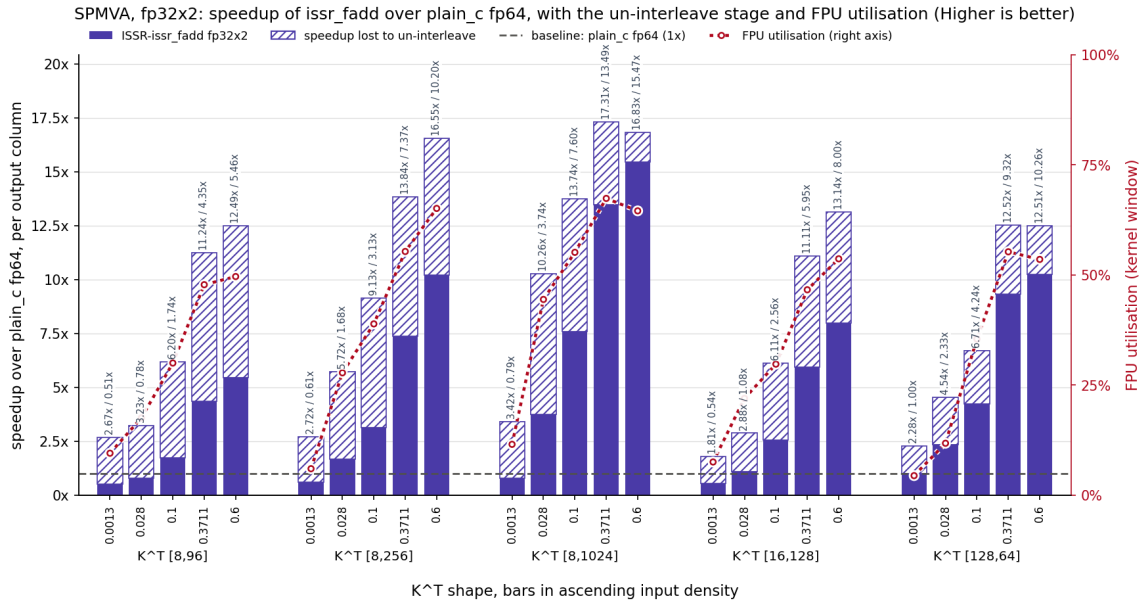


Figure 4.2: fp32x2 SpMVA speedup per output column over plain C, with and without the un-interleave stage, and FPU utilization against input density, K^T at 8×96 , 8×256 , 8×1024 , 16×128 and 128×64 . In the legend, ISSR-issr_fadd is the SpMVA kernel issr_fadd and plain_c its plain-C reference (Section 3.3.1); the hatched part, labeled speedup lost to un-interleave, is the cost of snn_unpack_fp32x2.

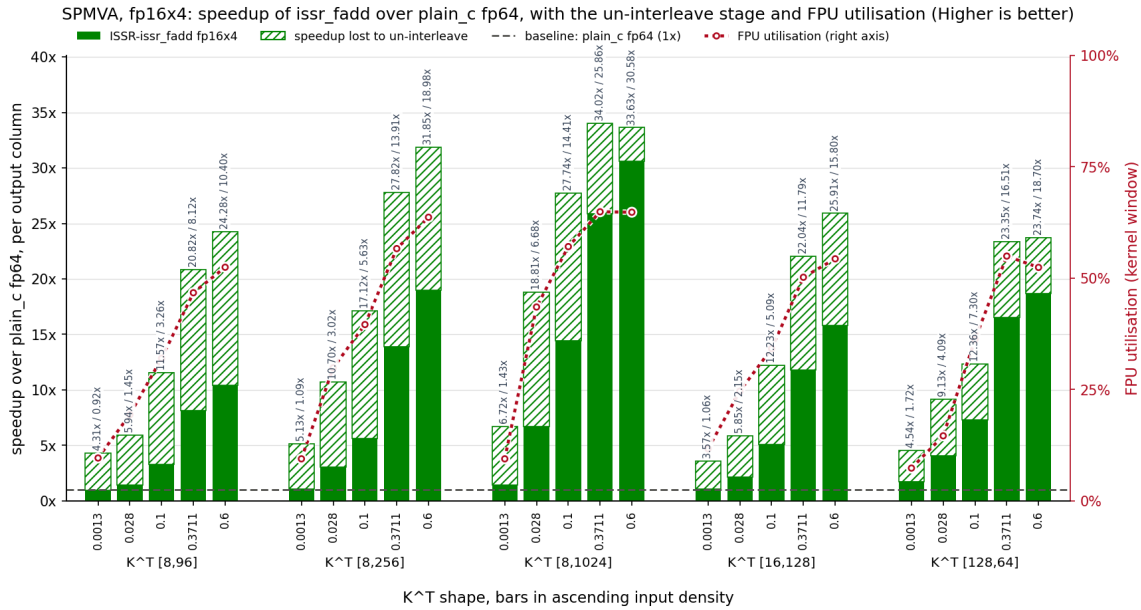


Figure 4.3: fp16x4 SpMVA speedup per output column over plain C, with and without the un-interleave stage, and FPU utilization against input density, K^T at 8×96 , 8×256 , 8×1024 , 16×128 and 128×64 . In the legend, ISSR-issr_fadd is the SpMVA kernel issr_fadd and plain_c its plain-C reference (Section 3.3.1); the hatched part, labeled speedup lost to un-interleave, is the cost of snn_unpack_fp16x4.

8×1024 , these are 1.99 $\times$ and 3.98 $\times$ the fp64 value of 8.45 $\times$. The ratios stay close to the nominal two and four lanes at every shape, so the gains of lane packing and of prologue amortization combine.

The hatched part of each bar in Figures 4.2 and 4.3 is the speedup lost to the un-interleave stage of Section 3.3.1, snn_unpack_fp32x2 or snn_unpack_fp16x4, timed with its staging transfers, and the solid part is what remains. The stage costs a fixed number of cycles at every density, so its share is largest where the kernel window is shortest. At $d = 0.0013$ on 8×96 it lowers fp32x2 from 2.67 $\times$ to 0.51 $\times$ and fp16x4 from 4.31 $\times$ to 0.92 $\times$, both under the plain-C baseline. At $d = 0.6$ on 8×1024 it lowers fp32x2 from 16.83 $\times$ to 15.47 $\times$ and fp16x4 from 33.63 $\times$ to 30.58 $\times$. Over the whole grid the net speedup per output column spans 0.51 $\times$ to 15.47 $\times$ at fp32x2 and 0.92 $\times$ to 30.58 $\times$ at fp16x4, and it is above 1 $\times$ at every shape from $d = 0.1$ upward.

Utilization rises with density up to $d = 0.3711$ at every shape, from 3% to 12% at $d = 0.0013$ across the three widths to 51% to 65% at $d = 0.6$ at fp64, 50% to 65% at fp32x2 and 52% to 65% at fp16x4. The three widths agree to within a few points, as expected of a measure taken against a one-operation-per-cycle ceiling that does not count how many lanes an operation packs. At the best point, 8×1024 at $d = 0.3711$ and fp32x2, utilization reaches 67% and the FPU is idle for 33% of the window, which is spent on the indirect gather, the row bookkeeping and the FREP prologue. FPU issue occupancy, the share of retired integer-core instructions that are floating-point with each frep counted once, is measured but not plotted. From $d = 0.028$

upward it stays near 28% at 8×256 , 8×1024 and 16×128 while utilization climbs, which places the residual cost in the number of instructions issued per row. It changes only at the sparsest point, where it drops to 15% to 19%, at 8×96 , where it rises from 17% to 27% over the lower densities, and at 128×64 , where it varies between 15% and 38%.

The 128×64 shape is the clearest case in which utilization does not follow density. It peaks near $d = 0.3711$ at 55% at fp32x2 and fp16x4 and decreases to 54% and 52% at $d = 0.6$, while its speedup levels off. With 64 columns per row the FREP loop saturates before $d = 0.6$, and the nonzeros added between $d = 0.3711$ and $d = 0.6$ add row bookkeeping without a proportional amount of replayed arithmetic. The conv1x1 kernel of Section 4.2.3 shows the same trend.

4.2.2 SpMVA: Core Scaling

`issr_fadd` runs inside the block pipeline of `cluster_spmv.h` (Section 3.3.1), where the workers claim rows first-come, first-served, so the slowest core sets the reported cycles. As in Section 4.2.1, the cycle counts cover only the kernel window, the kernel calls with their FPU fence, and exclude the transfers and barriers of the pipeline. The pipeline overhead is large: at 8×96 , fp64, $d = 0.3711$, the kernel occupies 117 of its 678 cycles, and at the two densities of the core-scaling runs, transfers and barriers take 51% to 91% of the cycles across the shapes and lane widths. The pipeline is also serial, so end to end 8 cores gain $1.67\times$ to $3.65\times$ over one at fp64. Figures 4.4, 4.5 and 4.6 report the speedup of the kernel window over a single core, one figure per lane width, at two densities: 0.028 and 0.3711.

Scaling is close to linear at every shape and density. On 8 cores every shape reaches $6.37\times$ to $7.12\times$ at fp64, $6.43\times$ to $7.21\times$ at fp32x2 and $6.44\times$ to $7.31\times$ at fp16x4. The lowest value at every lane width belongs to 128×64 at $d = 0.3711$.

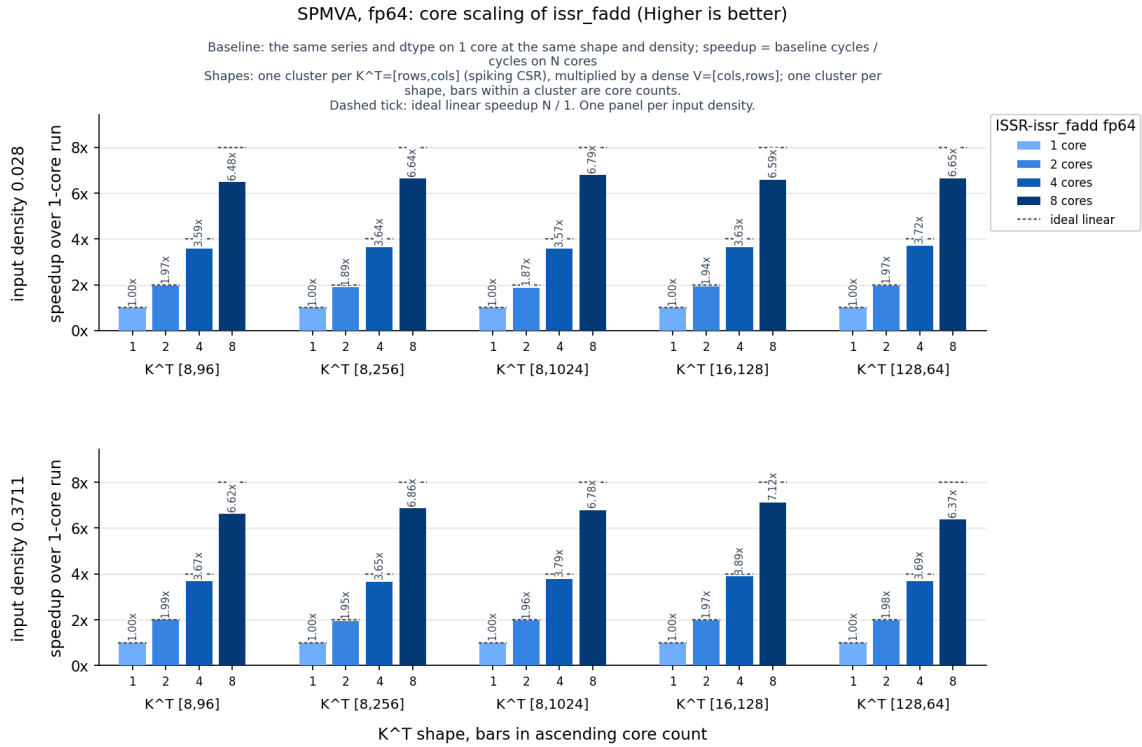


Figure 4.4: fp64 SpMVA speedup over one core against core count at densities 0.028 and 0.3711, K^T at 8×96 , 8×256 , 8×1024 , 16×128 and 128×64 . In the legend, ISSR-issr_fadd is the SpMVA kernel issr_fadd of Section 3.3.1.

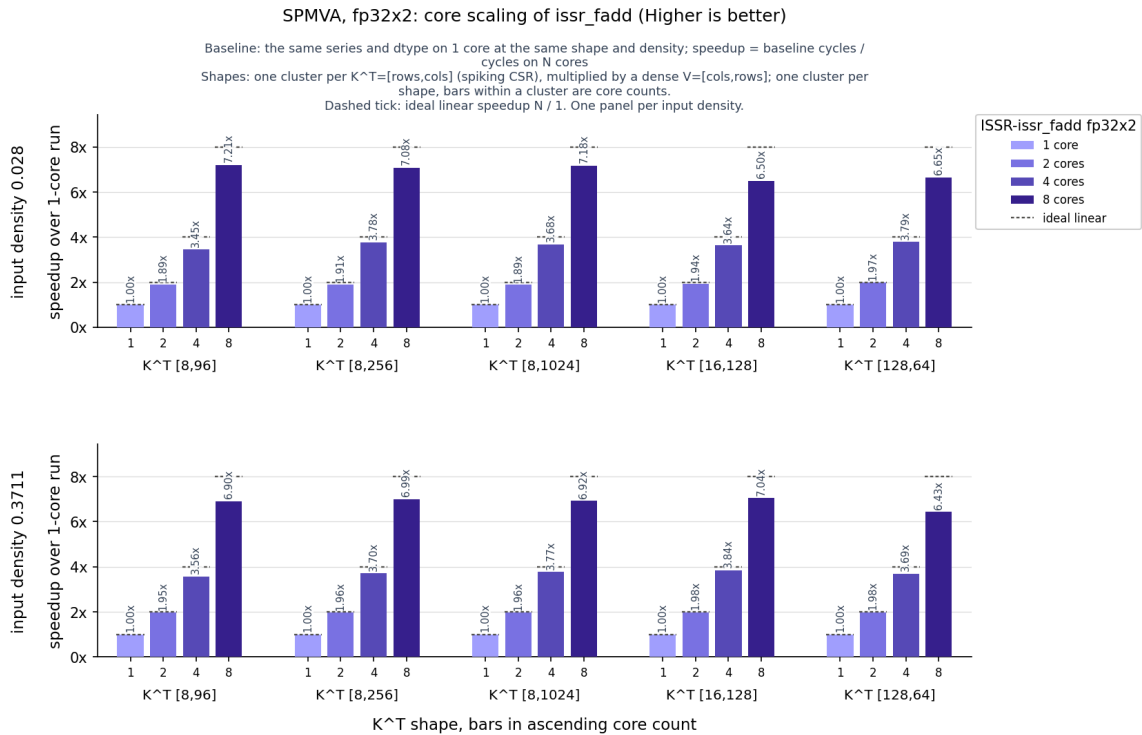


Figure 4.5: fp32x2 SpMVA speedup over one core against core count at densities 0.028 and 0.3711, K^T at 8×96 , 8×256 , 8×1024 , 16×128 and 128×64 . In the legend, ISSR-issr_fadd is the SpMVA kernel issr_fadd of Section 3.3.1.

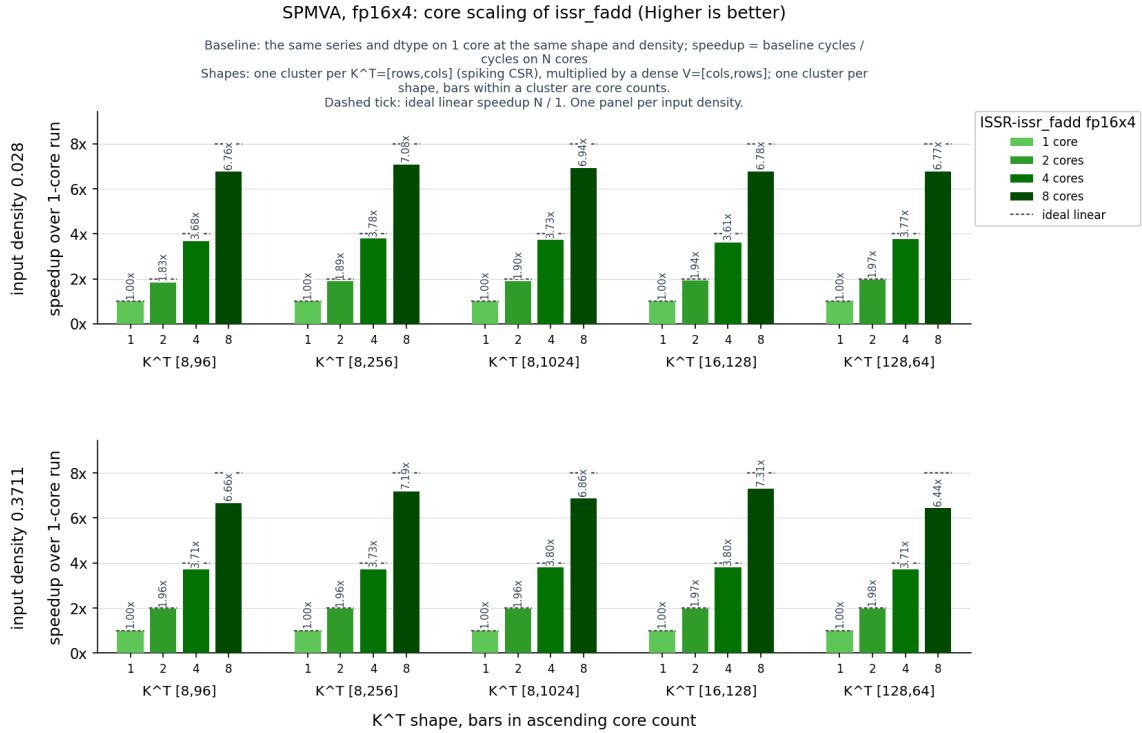


Figure 4.6: fp16x4 SpMVA speedup over one core against core count at densities 0.028 and 0.3711, K^T at 8×96 , 8×256 , 8×1024 , 16×128 and 128×64 . In the legend, ISSR-issr_fadd is the SpMVA kernel `issr_fadd` of Section 3.3.1.

4.2.3 Convolution: SpMMA Speedup and FPU Utilization

The convolution kernel extends SpMVA to SpMMA with an outer loop, the `spmva` engine of Section 3.3.2, and the `indir` engine of Section 3.3.3 is measured alongside it. `spmva` configures and drains one indirect SSR per column-word, `COUT_WORDS` times per pass over the sparse structure, and visits every row once per column-word whether it holds a nonzero or not. This cost is fixed per row, so it matters most when rows are short, and the comparison with `indir`, which skips an empty row outright, makes it visible.

Figure 4.7 reports the speedup of both engines over a dense *General Matrix Multiplication* (GEMM) of the same lane width, `gemm_fp64_opt`, `gemm_fp32_opt` and `gemm_fp16_opt`, one panel per lane width (Figures 4.7a, 4.7b and 4.7c). The right axis shows the FPU utilization of each engine, as defined in Section 4.2.1. The GEMM baseline is timing only: its built-in self-test needs `printf`, which does not link on this target, so its output is not verified. `indir` requires an element stride expressible as a single `slli`, and $A[8,1024] \times W[1024,32]$ fits the 112 KiB budget of Section 4.1.3 only at fp16x4. The fp64 and fp32x2 panels therefore show $A[8,96] \times W[96,64]$ and $A[8,256] \times W[256,32]$, and only the fp16x4 panel adds $A[8,1024] \times W[1024,32]$.

At $A[8,96] \times W[96,64]$, `indir` leads at every density until both engines fall under the dense baseline. At $d = 0.0013$ it reaches $14.04\times$ against $2.76\times$ for `spmva` at fp64, and $12.56\times$ against $2.83\times$ at fp32x2. By linear interpolation in log density, `spmva` crosses the baseline at $d = 0.27$

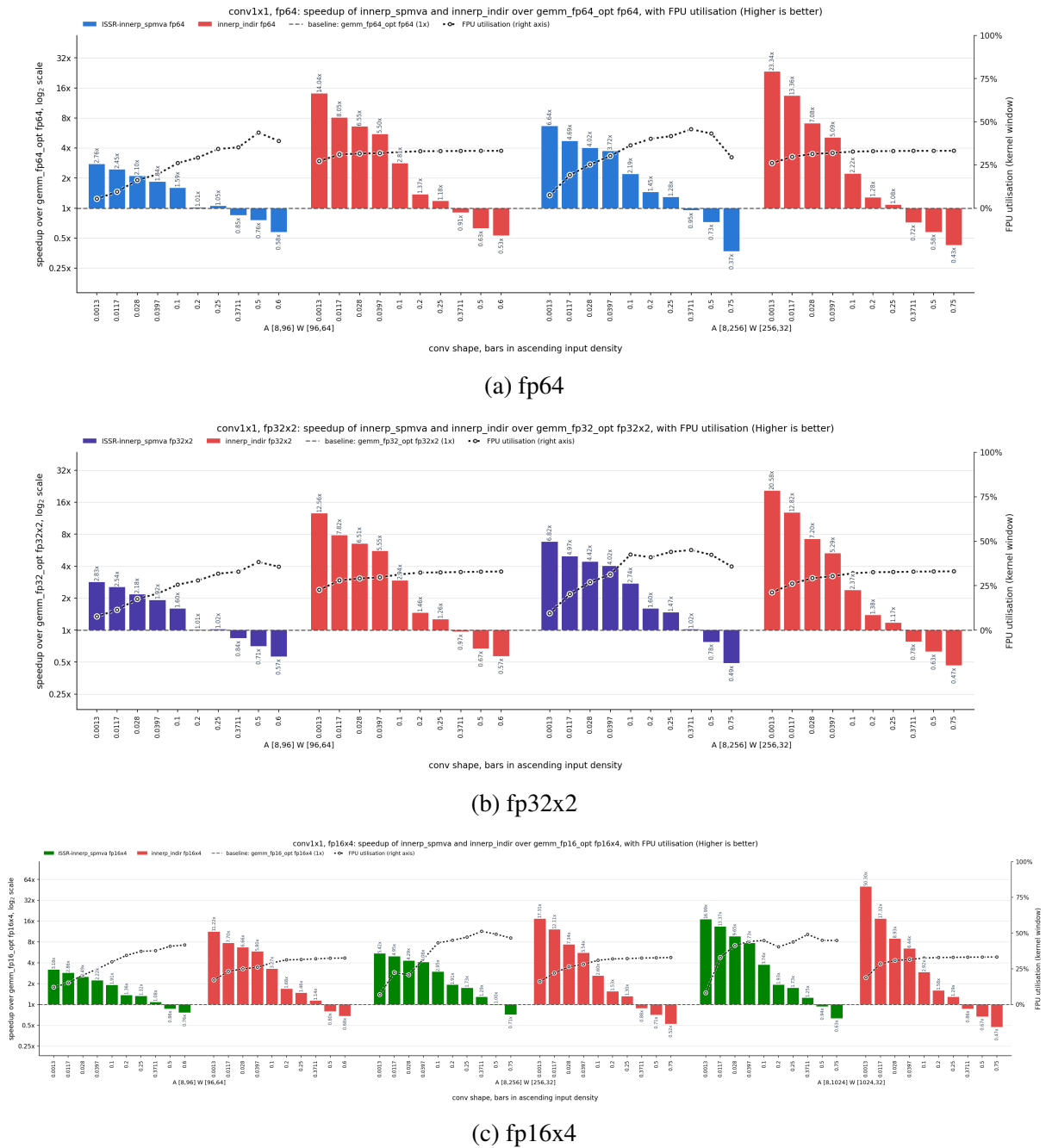


Figure 4.7: 1×1 convolution speedup of `spmva` and `indir` over a dense GEMM (left axis) and FPU utilization (right axis) against input density. In the legend, `ISSR-innerp_spmva` is the `spmva` engine and `innerp_indir` the `indir` engine (Section 3.3.3), and `gemm_fp64_opt`, `gemm_fp32_opt` and `gemm_fp16_opt` are the dense GEMM baselines.

and `indir` at $d = 0.32$, both at fp64. The lead of `indir` at the sparse end holds at the larger shapes, but it shrinks. At $d = 0.0013$, `indir` is 5.1 times as fast as `spmva` at $A[8,96] \times W[96,64]$ and fp64 (14.04 $\times$ against 2.76 $\times$), 3.5 times at $A[8,256] \times W[256,32]$ (23.34 $\times$ against 6.64 $\times$), and 3.0 times at $A[8,1024] \times W[1024,32]$, measured only at fp16x4 (50.30 $\times$ against 16.99 $\times$). The density above which `spmva` leads moves down as the shape grows. At $A[8,256] \times W[256,32]$ the two engines are level at $d = 0.1$: `indir` leads 2.22 $\times$ to 2.19 $\times$ at fp64, and `spmva` leads 2.74 $\times$ to 2.37 $\times$ at fp32x2 (Figure 4.7b). At $A[8,1024] \times W[1024,32]$, `spmva` leads from $d = 0.028$ upward, with 3.74 $\times$ against 2.92 $\times$ at $d = 0.1$ (Figure 4.7c). The three shapes share eight sites and differ in input channel count. The number of nonzeros per row at a given density therefore grows by a factor of 2.7 from the small shape to the medium one and by a further 4 to the large one, while c_{out} , and with it `COUT_WORDS`, halves from 64 to 32. Longer rows spread the fixed per-row cost of `spmva` over more work, so it overtakes `indir` at lower densities. The lead of `indir` comes entirely from skipping empty rows, so it remains only where most rows are empty.

The utilization of `spmva` rises with density and then decreases. At the small shape and fp64 it goes from 5% at $d = 0.0013$ to 44% at $d = 0.5$ and falls to 39% at $d = 0.6$; it peaks at 38% at $d = 0.5$ at fp32x2 and at 42% at $d = 0.6$ at fp16x4. At the medium shape it peaks at $d = 0.3711$, at 46% at fp64 and 45% at fp32x2, and falls to 29% and 36% at $d = 0.75$. At the two larger shapes at fp16x4 it stays between 40% and 51% from $d = 0.1$ upward. The utilization of `indir` is nearly flat, since every nonempty row issues the same gather-and-accumulate sequence. From $d = 0.028$ upward it stays between 25% and 33% at every shape and width, and at $d = 0.0013$ it falls to 26% at fp64, 21% at fp32x2 and 16% at fp16x4. Its FPU issue occupancy, measured but not plotted, ranges from 55% at the sparsest points to 70%, and stays above 66% from $d = 0.1$ upward, against a utilization of at most 33%. The issue occupancy of `spmva` stays between 27% and 33% from $d = 0.0397$ upward, while its utilization varies from 20% to 51% over the same densities.

`conv1x1` reuses each gather across `SIMD_COLS` output channels per column-word, so wider lanes cover more of c_{out} per pass over the sparse structure. We compare the cycle counts of `spmva`, measured but not plotted, across lane widths at the same shape and density, against a packing ceiling of 2 $\times$ at fp32x2 and 4 $\times$ at fp16x4. At $A[8,96] \times W[96,64]$, fp32x2 reaches 1.72 $\times$ to 1.92 $\times$ and fp16x4 3.49 $\times$ to 4.14 $\times$. At $A[8,256] \times W[256,32]$ both widths exceed the ceiling as density rises, with fp32x2 reaching 2.42 $\times$ and fp16x4 6.20 $\times$ at $d = 0.75$. The packed kernels also amortize the per-row overhead over fewer column-words, the same effect that Section 4.2.1 measured for SpMVA, and the two gains multiply.

The square shape $A[8,96] \times W[96,96]$ cannot host `indir` at any lane width: a row of **W** occupies 768, 384 and 192 bytes at fp64, fp32x2 and fp16x4, none of which can be expressed by the single `slli` that `indir` needs (Section 3.3.3). At this shape, `spmva` is measured only against a dense GEMM of the same lane width. Figure 4.8 gives one panel per width, from $d = 0.0013$ to $d = 0.75$, with the 1 $\times$ crossing marked by linear interpolation in log density and the utilization of `spmva` on the right axis. The crossover density rises with lane width: $d = 0.29$

at fp64, $d = 0.39$ at fp32x2 and $d = 0.75$ at fp16x4. At fp16x4 the two are level at the last point of the figure: at $d = 0.75$ spmva is still $1.02\times$ faster than the dense GEMM, 4,112 cycles against 4,181, and at $d = 1$, not plotted, it falls to $0.74\times$. Utilization rises as the bars fall. At fp64 it peaks at 42% at $d = 0.6$ and falls to 33% at $d = 0.75$, while at fp32x2 and fp16x4 it is still rising at $d = 0.75$, at 49% and 51%; with no indir run at this shape, these values belong to spmva alone. The cycle counts at this shape, not plotted, give lane-width ratios of $1.91\times$ to $2.51\times$ at fp32x2 and $3.52\times$ to $4.77\times$ at fp16x4 from $d = 0.0013$ to $d = 0.6$. At $d = 0.75$ the ratios reach $2.96\times$ and $6.03\times$, because the fp64 run slows down there, with 24,797 cycles against 15,908 at $d = 0.6$.

conv1x1 crossover: innerp_spmva against the dense gemm at every lane width, with the 1.00x crossing and FPU utilisation (Higher is better)

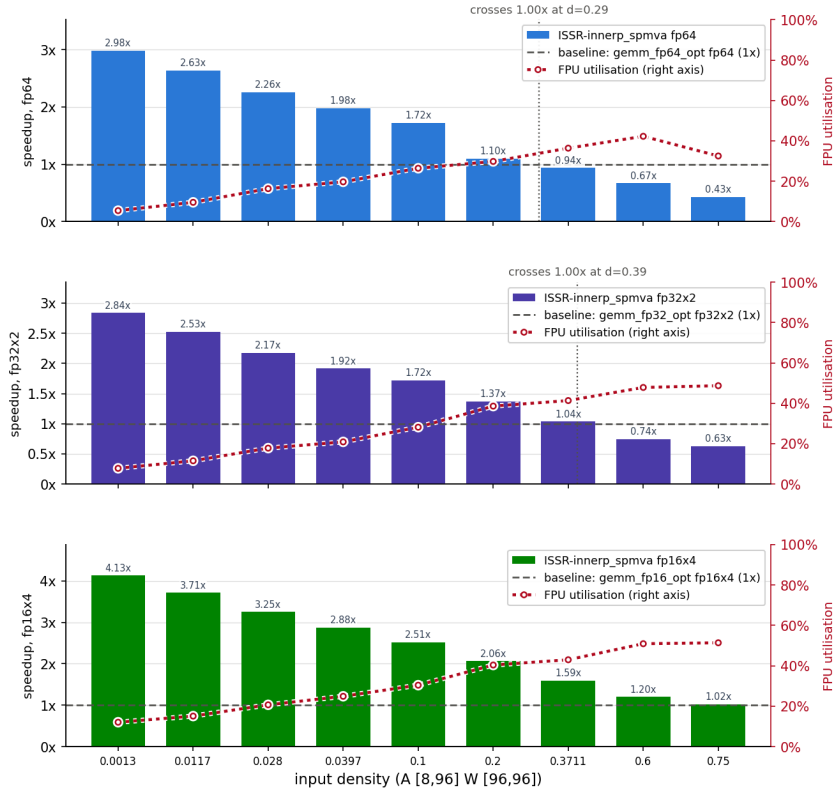


Figure 4.8: Speedup of spmva over dense GEMM and FPU utilization against input density at fp64, fp32x2 and fp16x4, A[8,96]×W[96,96]. In the legend, ISSR-innerp_spmva is the spmva engine (Section 3.3.2), and gemm_fp64_opt, gemm_fp32_opt and gemm_fp16_opt are the dense GEMM baselines.

4.2.4 Convolution: Core Scaling

Figures 4.9, 4.10 and 4.11 report the speedup of spmva and indir over their own single-core runs on 2, 4 and 8 cores at $d = 0.3711$ and $d = 1$, one figure per lane width. The shape is A[8,256]×W[256,32] at every width, with A[8,96]×W[96,64] added at fp16x4. Both engines split the output sites statically and evenly across the workers, so at the eight sites of these shapes each of eight cores computes one site, and the two engines differ only in how a site is computed.

`indir` scales close to linearly. On 8 cores at $d = 0.3711$ it reaches $6.23\times$ at fp64 (Figure 4.9), $6.21\times$ at fp32x2 (Figure 4.10) and $6.17\times$ at fp16x4 (Figure 4.11). At $d = 1$, where every site holds the same number of nonzeros, it reaches $7.97\times$, $7.95\times$ and $7.92\times$, and at $A[8,96]\times W[96,64]$ it reaches $6.79\times$ and $7.88\times$. `spmv` does not scale linearly. On 8 cores it stays between $3.79\times$ and $4.37\times$ at $d = 0.3711$; at $d = 1$ it reaches $2.25\times$ at fp64, $3.17\times$ at fp32x2 and $4.81\times$ at fp16x4 on the larger shape, and $3.00\times$ on the smaller one. The loss starts at two cores: at $d = 1$ and fp64, 2 and 4 cores give $1.27\times$ and $1.58\times$. Since the split is the same for both engines, the difference lies inside the `spmv` engine, and these measurements do not isolate its cause.

The engine comparisons of Section 4.2.3 were taken on 8 cores, so they already include this difference. On fewer cores `spmv` loses less of its single-core speed and compares better with `indir` than those figures show; conversely, the lead of `indir` at the sparse end is the one that holds as cores are added.

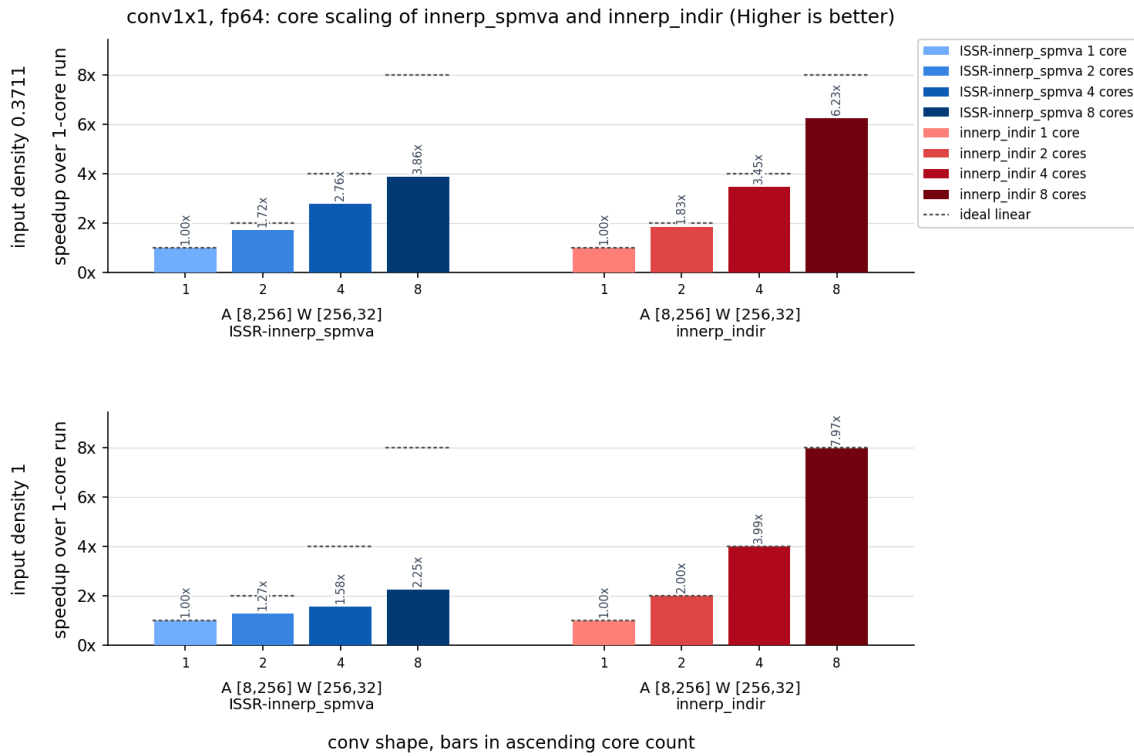


Figure 4.9: fp64 conv1x1 speedup over one core against core count, `spmv` and `indir`, at densities 0.3711 and 1, $A[8,256]\times W[256,32]$. In the legend, `ISSR-innerp_spmva` is the `spmv` engine and `innerp_indir` the `indir` engine (Section 3.3.3).

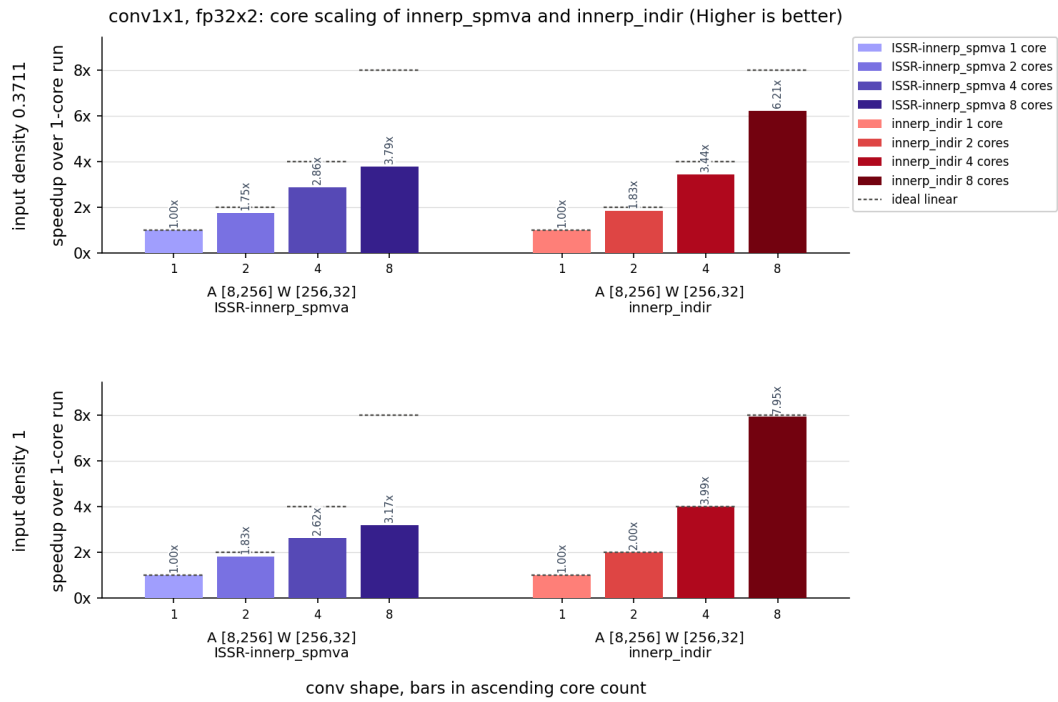


Figure 4.10: fp32x2 conv1x1 speedup over one core against core count, spmva and indir, at densities 0.3711 and 1, A[8,256]×W[256,32]. In the legend, ISSR-innerp_spmva is the spmva engine and innerp_indir the indir engine (Section 3.3.3).

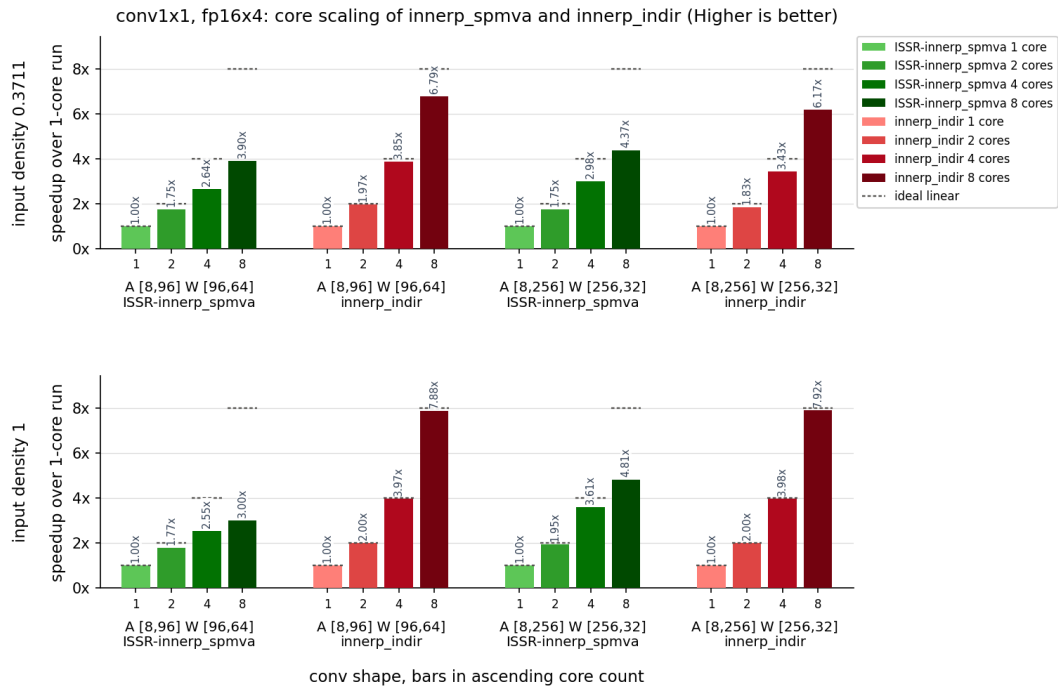


Figure 4.11: fp16x4 conv1x1 speedup over one core against core count, spmva and indir, at densities 0.3711 and 1, A[8,96]×W[96,64] and A[8,256]×W[256,32]. In the legend, ISSR-innerp_spmva is the spmva engine and innerp_indir the indir engine (Section 3.3.3).

4.2.5 Batch Normalization: Speedup Against Sparsity

The BN kernel of Section 3.3.4 is a folded affine map, one multiplication and one addition per element. It comes in a dense family that walks the full [site, channel] tensor and a CSR family that walks only the nonzeros left by `conv1x1`. The dense apply interleaves four independent $\kappa \cdot x + \lambda$ chains per `frep`, while the CSR apply runs a single chain. The standalone CSR kernel measured here also gathers κ and λ per nonzero at run time with `bn_csr_expand_*`, a cost that disappears when BN runs on the row-sparse output of `conv1x1`, whose κ/λ table the host folds once (Section 3.3.5).

Two limits determine which precisions are measured at each shape. The fp64 dense apply needs 144 KiB at the two 8192-element shapes, 8×1024 and 128×64 , against the 112 KiB budget of Section 4.1.3. The CSR container is the larger format at equal element count, and it also excludes fp32x2 at $d = 1$ on those shapes, where it needs 160 KiB. In addition, the four chains of the dense apply require the channel count divided by the lane count to be a multiple of $8 \times 4 = 32$, which 96 channels do not satisfy at fp32x2 and fp16x4, and 64 channels do not satisfy at fp16x4.

Figure 4.12 reports the speedup of the CSR kernel over the dense apply of the same lane width, one panel per lane width, at the eight densities up to $d = 0.6$, the densest point every shape reaches. The solid part of each bar is the speedup with the gather included, the value printed above it; the hatched part extends it to the CSR apply alone, as it runs on the output of `conv1x1`. Each legend gives the dense cycle count of every shape and the density at which the apply alone and the apply with the gather cross $1\times$.

At $d = 0.0013$ and fp64 the apply alone reaches $4.59\times$ (8×96), $12.24\times$ (8×256) and $10.26\times$ (16×128); with the gather included these become $1.94\times$, $4.32\times$ and $4.34\times$. At fp32x2 the peak of the apply alone grows with element count, from $6.73\times$ at 8×256 to $23.04\times$ at 8×1024 and $22.04\times$ at 128×64 . At fp16x4 it is lower at every matching shape, $3.90\times$ at 8×256 against $6.73\times$ at fp32x2, since the dense baseline also becomes cheaper with packing: at 8×256 it runs in 600, 330 and 191 cycles at fp64, fp32x2 and fp16x4. The apply alone falls under the dense apply between $d = 0.35$ and $d = 0.44$ at every shape and width, and at $d = 0.6$ it runs at $0.67\times$ to $0.75\times$, since per stored element the single-chain body costs about twice the four-chain body.

The gather moves the crossing to lower densities: between $d = 0.15$ and $d = 0.2$ at fp64, 0.055 and 0.091 at fp32x2, and 0.02 and 0.068 at fp16x4. At fp16x4 it removes almost all of the gain. At $d = 0.0013$ the CSR kernel is only $1.04\times$ and $1.02\times$ faster than the dense apply at 8×256 and 16×128 , and at 16×128 it is already slower at $d = 0.028$ ($0.82\times$). The gather costs one indirect-SSR read per nonzero plus its repacking at every width, while the apply body shrinks with lane width. The share of the gather in the CSR time therefore grows with packing: at $d = 0.6$ it is 41% to 43% at fp64, 75% to 76% at fp32x2 and 79% to 81% at fp16x4.

Fusing BN into the convolution pipeline of Section 3.3.5 brings a negligible gain. We measure it, but do not plot it, at the real input width of 768 channels, 8 sites and 8 cores, with the `spmva` engine and CSR output, at the densities of the Q , K and V input ($d = 0.3711$) and of the

`proj_conv` input ($d = 0.0013$). The fused launch is compared with the convolution followed by a separate BN launch over the same container and the same host-folded κ/λ table, so both arms run the same convolution, compaction and apply, and neither runs the gather. Every DMA transfer is timed, and the window runs from the first transfer in to the last transfer out. Fusion saves 133 to 153 cycles, the round trip of the convolution output to memory and the barriers of the second launch. The fused BN apply itself costs the same as the separate one. At $d = 0.3711$ the saving is $1.007\times$ to $1.009\times$ over the separate launches, since the convolution alone takes about 13,200 to 14,000 cycles. At $d = 0.0013$, where the convolution takes only about 750 cycles, it is $1.020\times$ to $1.037\times$. The raw totals at $d = 0.3711$ range from $0.998\times$ to $1.019\times$, because the convolution stage itself varies by up to 1.3% between the two builds, more than the saving. The case for fusion is therefore the dataflow it enables, not its cycle count.

4.2.6 Batch Normalization: Core Scaling

Figure 4.13 reports the speedup of the CSR apply and of the dense apply over their own single-core runs at 8×256 and $d = 0.3711$, one panel per lane width; the gather is not included. On 8 cores the CSR apply reaches $7.19\times$ at fp64 (Figure 4.13a), $6.51\times$ at fp32x2 (Figure 4.13b) and $5.52\times$ at fp16x4 (Figure 4.13c). The dense apply, which does not depend on density, reaches $6.93\times$, $6.41\times$ and $5.69\times$. Both bodies scale less well as the lane width grows, and the dense apply at fp16x4, 191 cycles on 8 cores, has the shortest window over which to spread the fixed per-core cost. The gather reverses the trend: with it included, which the figure does not show, the CSR kernel reaches $6.34\times$, $7.06\times$ and $6.54\times$ on 8 cores.

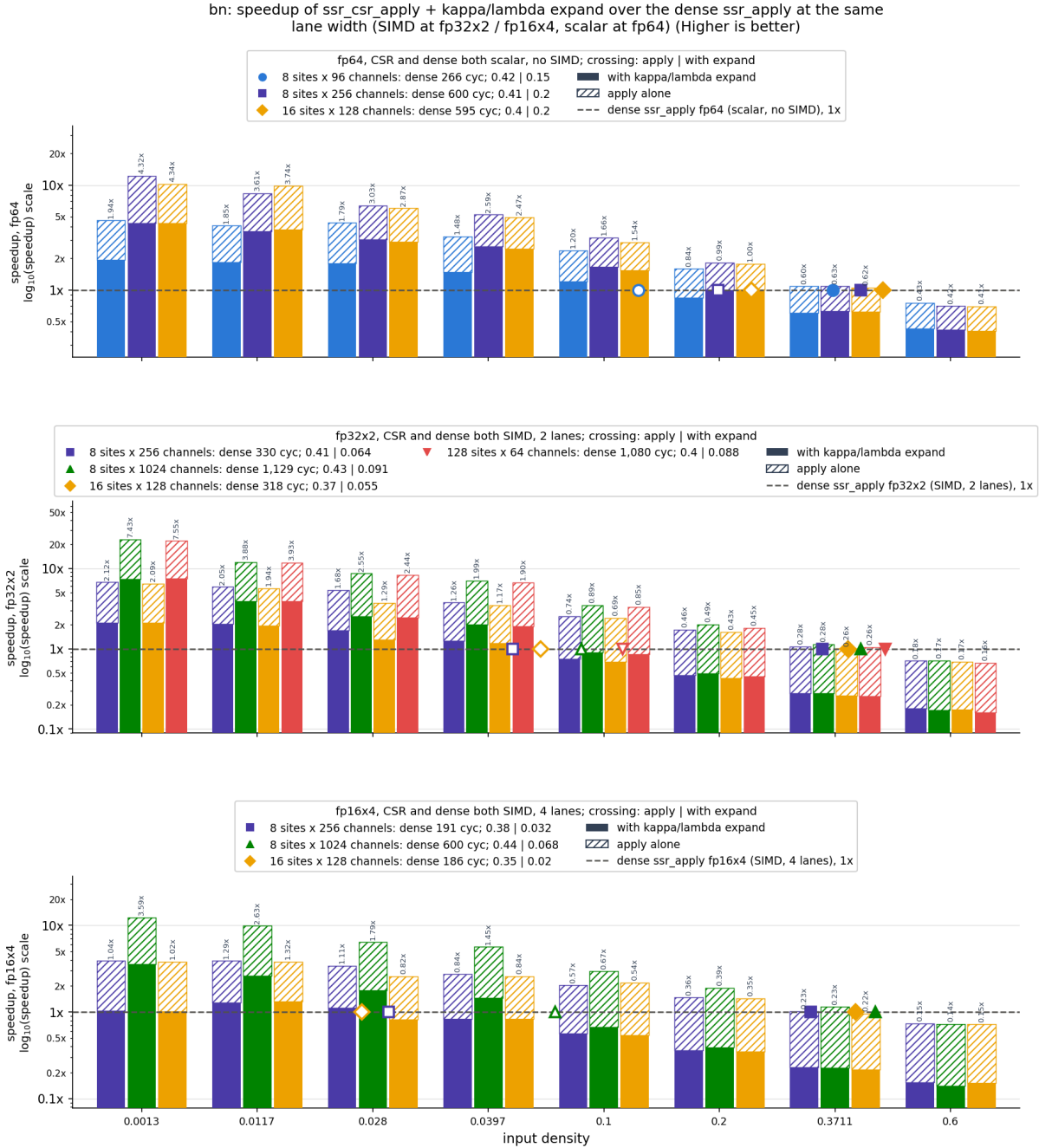
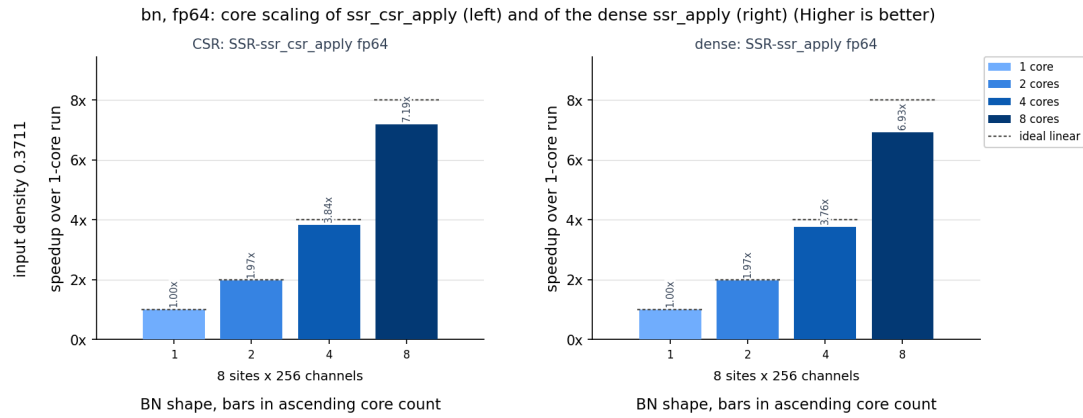
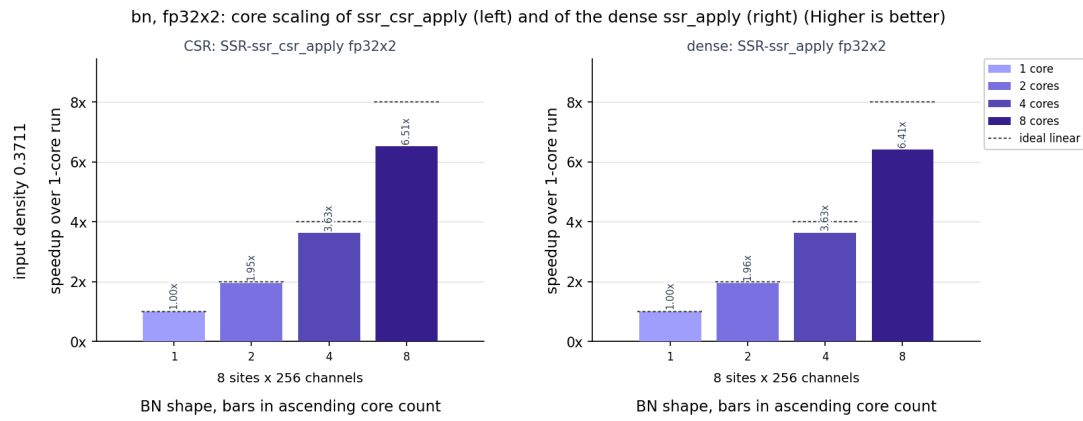


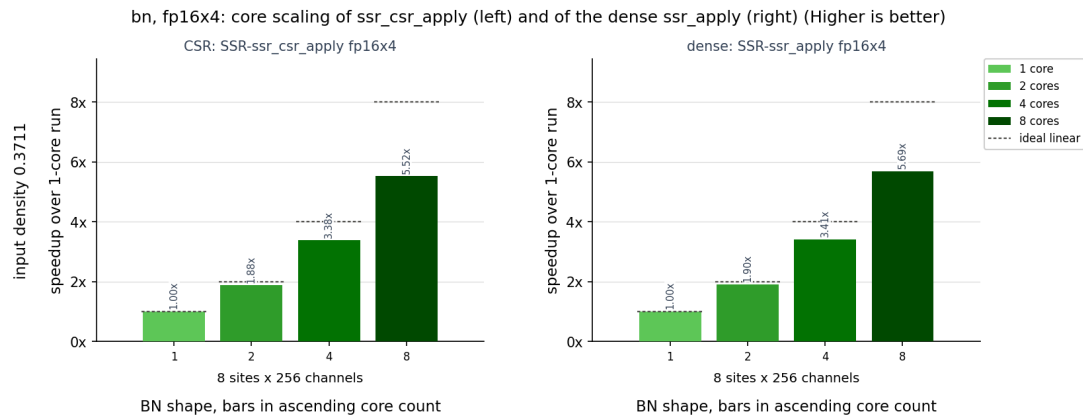
Figure 4.12: BN speedup of the CSR kernel over the dense apply, with and without the κ/λ gather, against input density, per lane width. In the legend, `ssr_csr_apply` is the CSR apply `bn_csr_apply_*`, `ssr_apply` the dense apply `bn_apply_*`, and κ/λ expand the gather `bn_csr_expand_*` (Table 3.5).



(a) fp64



(b) fp32x2



(c) fp16x4

Figure 4.13: BN speedup over one core against core count, CSR apply and dense apply, 8×256 at density 0.3711. In the panel titles, `SSR-ssr_csr_apply` is the CSR apply `bn_csr_apply_*` and `SSR-ssr_apply` the dense apply `bn_apply_*` (Table 3.5).

4.2.7 LIF: Speedup and FPU Occupancy Across Lane Widths

The LIF update is a dense operation: the kernels of Section 3.3.6 update every neuron at every timestep, whatever its input holds. Their cost therefore does not depend on density: at 768 neurons the fp64 kernel takes 3,547 cycles and the fp32x2 kernel 1,832 at all nine density points of Table 4.2, and fp16x4 takes 983 at all four densities measured. Each packed instruction updates two or four neurons, so fp32x2 and fp16x4 should approach 2 $\times$ and 4 $\times$ the throughput of fp64.

Figure 4.14 shows, on 8 cores and four timesteps, the cycle count of the interleaved kernels of Table 3.6 as bars, the speedup over fp64 and the speedup of fp16x4 over fp32x2 at the same neuron count on the outer right axis, and the FPU issue occupancy on the inner one. Since the density at which the update is tested does not matter, a single figure covers all three lane widths. Cycles fall from 3,547 to 1,832 to 983 at 768 neurons and from 4,697 to 2,406 to 1,266 at 1024, a speedup of 1.94 $\times$ and 1.95 $\times$ at fp32x2 and 3.61 $\times$ and 3.71 $\times$ at fp16x4. The prologue and the closing barrier cost the same at every width, so the widest lanes, with the shortest window, remain furthest from their ceiling, and both widths move closer to it from 768 to 1024 neurons. The 2048-neuron size has no fp64 bar: the fp64 kernel needs 128 KiB there, over the 112 KiB budget. Against fp32x2, the fp16x4 kernel is 1.86 $\times$ faster at 768 neurons, 1.90 $\times$ at 1024 and 1.94 $\times$ at 2048, where it takes 2,417 cycles against 4,698.

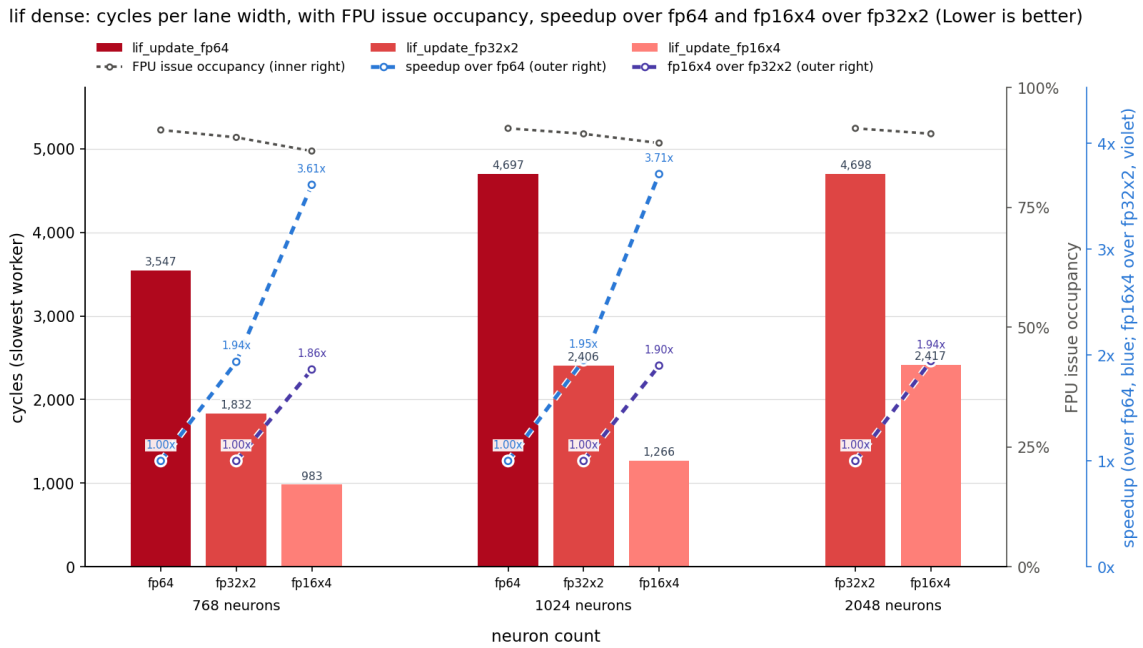


Figure 4.14: LIF cycles, speedup over fp64 and over fp32x2, and FPU issue occupancy at fp64, fp32x2 and fp16x4, 768, 1024 and 2048 neurons at $T = 4$, dense layout. The legend names the interleaved LIF kernels of Table 3.6.

Issue occupancy is between 0.87 and 0.92 at every width and size. With the threshold and reset computed from sign bits and no branch, the timestep body is almost entirely floating-point

work that `frep` keeps issuing. It declines slightly from fp64 to fp16x4, as the fixed prologue takes a larger share of a shorter window. Issue occupancy counts retired instructions, with `frep` replays appearing as floating-point rows of their own. It therefore ranks variants of one streaming kernel correctly, but it is not comparable with the FPU utilization of Sections 4.2.1 and 4.2.3.

Figure 4.15 fits the same cycle counts against neuron count. The least-squares lines give 4.49 cycles per neuron at fp64, 2.24 at fp32x2 and 1.12 at fp16x4, halving with every doubling of the lane count, and a fixed part of about 100 to 125 cycles at every width. This fixed part keeps the measured ratios below $2\times$ and $4\times$. fp32x2 at 2048 neurons takes 4,698 cycles, one cycle more than fp64 at 1024, both processing the same number of 64-bit words. The fp64 line extrapolates to 9,297 cycles at 2048 neurons, marked as not measured; refitting the packed widths without their 2048-neuron point predicts it within 0.1% at fp32x2 and 0.8% at fp16x4.

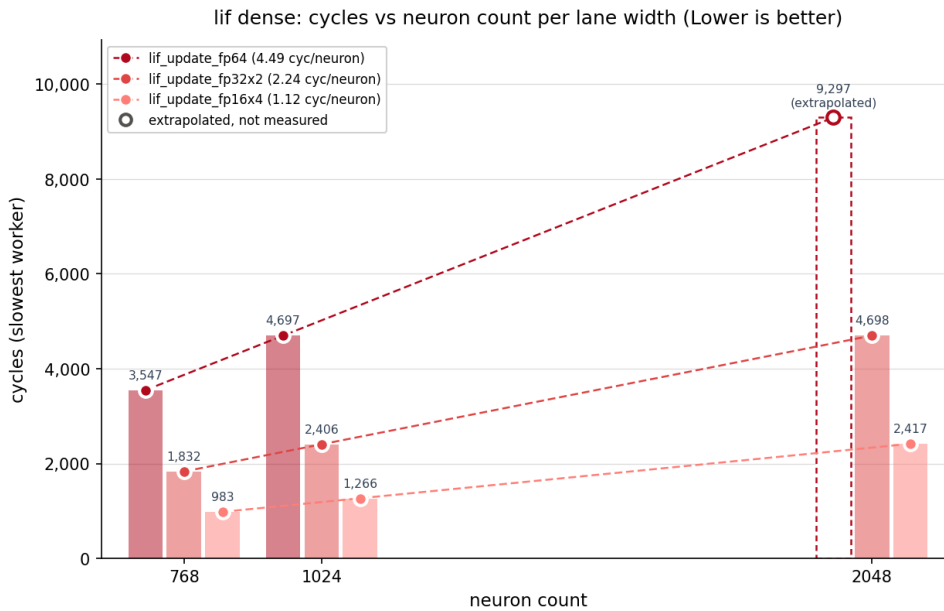
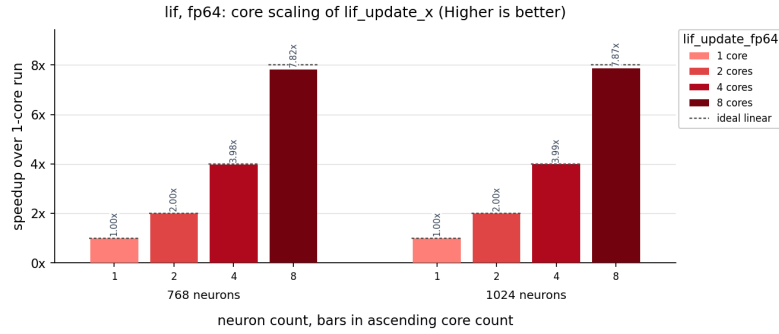


Figure 4.15: LIF cycles against neuron count with least-squares fit at fp64, fp32x2 and fp16x4, 768, 1024 and 2048 neurons at $T = 4$, dense layout. The legend names the interleaved LIF kernels of Table 3.6.

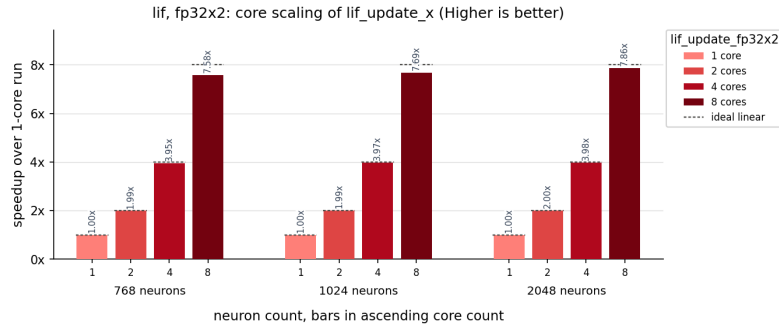
4.2.8 LIF: Core Scaling

Figure 4.16 reports the speedup of the dense LIF kernels of Figure 4.14 over their own single-core runs on 2, 4 and 8 cores, one panel per lane width. The LIF update is a dense operation, so the density at which these runs are tested does not matter. On 8 cores the fp64 kernel reaches $7.82\times$ at 768 neurons and $7.87\times$ at 1024 (Figure 4.16a); as in Section 4.2.7, the 2048-neuron size exceeds the TCDM budget at fp64. The fp32x2 kernel reaches $7.58\times$ to $7.86\times$ (Figure 4.16b) and the fp16x4 kernel $7.12\times$ to $7.66\times$ (Figure 4.16c). On 2 cores every configuration reaches $1.99\times$ or $2.00\times$. The work is split over neurons with no dependency between them, so the kernels scale almost linearly. The remaining shortfall grows with lane

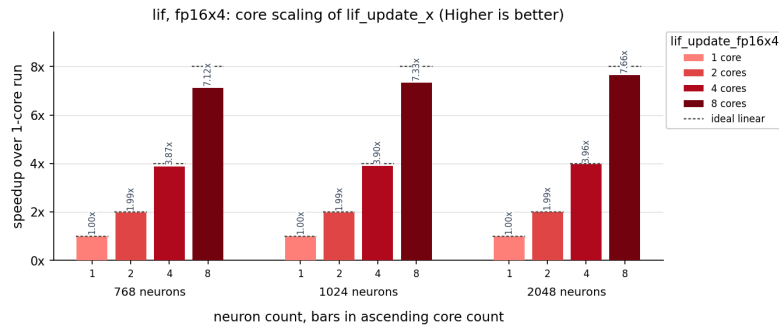
width and shrinks with neuron count. At the same neuron count, fp16x4 gives each core a quarter of the work of fp64 against the same fixed per-core cost, and doubling the neurons from 1024 to 2048 raises its 8-core speedup from 7.33 $\times$ to 7.66 $\times$.



(a) fp64



(b) fp32x2



(c) fp16x4

Figure 4.16: LIF speedup over one core against core count at fp64, fp32x2 and fp16x4, 768, 1024 and 2048 neurons at $T = 4$, dense layout. The legend titles name the interleaved LIF kernels of Table 3.6, and `lif_update_x` in the plot titles stands for the kernel of each panel.

4.2.9 Reshape: View Materialization

A shape operation of Table 3.7 is free when it can remain a view. A permutation, a transposition, a slice or a stride-compatible reshape rewrites only the shape and stride entries of a `sla_view_t` (Section 3.3.7). When the consuming kernel reads the view through its own SSR, as in Figure 3.14, no data moves, and the only cost is building and lowering the view once

per call. The operation is not free when the view must be materialized by `sla_contiguous`: when the SSR streams of the consumer are already in use, when the data must cross a memory boundary in any case, or when the new shape is not stride-compatible, as in the `chunk` case. The copy then adds its own cycles to those of the view.

We measure the three dense cases, `chunk`, `bridge` and `bvals`, in fp64 on 8 cores at 768 and 2048 elements, against their plain-C references `sla_contiguous_c`, and do not plot them. At fp64, with one lane, `bvals` reduces to a column-major copy of V . Building and lowering the view costs 551, 465 and 400 cycles for the three cases at 768 elements, and 557, 500 and 394 at 2048. This cost does not grow with the tensor, and it is the whole cost of a view that is not materialized. The copy of `chunk` takes 828 and 1,462 cycles against 8,588 and 22,047 for plain C, a speedup of $10.4\times$ and $15.1\times$. `bridge` takes 548 and 792 cycles, $6.1\times$ and $10.5\times$ faster than plain C, and `bvals` 423 and 521 cycles, $2.3\times$ and $3.9\times$ faster. The spread comes from the plain-C references, which walk the view with a general multi-dimensional counter. At 768 elements they range from 989 cycles for `bvals` to 8,588 for `chunk`, so the SSR gains most where that walk is most expensive. The gain also grows with element count in every case: from 768 to 2048 elements the plain-C `chunk` grows by a factor of 2.6 and the SSR `chunk` by a factor of 1.8. Summed over the three cases, with every view build included, `sla_contiguous` takes 3,215 cycles against 14,336 for `sla_contiguous_c` at 768 elements, and 4,226 against 33,834 at 2048, a $4.5\times$ and $8.0\times$ advantage. At 768 elements the three view builds account for 1,416 of those 3,215 cycles, so on small tensors building the views costs almost as much as moving the data.

The `chunk` copy does not scale beyond two cores. Measured on 1, 2, 4 and 8 cores, its speedup over one core is $1.49\times$ on 2 cores and $1.55\times$ on both 4 and 8 cores at 768 elements, and $1.75\times$ and $1.76\times$ at 2048. The case splits its work across cores along its outermost axis, `NUM_CHUNKS = 2`, which limits it to two-way parallelism (Section 3.3.7), so in the 8-core runs above only two of the cores copy.

The attention block of Section 4.2.11 uses none of these copies. It stores every activation channels-last, as `[site, channel]`, so the output of each stage is already in the layout the next stage reads, and the chunking of `chunk` becomes a two-dimensional window of the same buffer. The fused sub-block of Section 3.3.8 applies the same principle to the transition from the convolution to the LIF update. Of the four operations of Table 3.7, the block needs only the conversion to CSR, which the sparse kernels require in either layout. It runs that conversion with its own fp16x4 compaction, and folds the transpose of K into the product $K^T V$ rather than into the conversion. The other three operations were kept because each is a one-to-one port of a PyTorch expression: a model that keeps the PyTorch layout can use them unchanged, which may be useful in future work.

The conversion to CSR, `sla_to_sparse_csr`, was measured in the same runs at $d = 0.028$, the density of the spike tensor K , with the transpose of K fused into the conversion. It

gains less than any dense copy over its plain-C reference `sla_to_sparse_csr_c`, 2.5× at 768 elements and 2.6× at 2048, and its cost against density is the subject of the next section.

4.2.10 Reshape: Sparse Compaction

`sla_to_sparse_csr`, the `csr` case of Table 3.7 (`KT.to_sparse_csr()`), tests every dense element against zero, so its cost follows the size of the dense operand (Section 3.3.7). We measure it at every shape of Table 4.1, fp64, on a single core, since `a_ptrs` is a running total and the kernel has no core split. Figure 4.17 reports its cost in cycles per element of the dense operand against density. The right axis gives the same cost as a multiple of a dense SSR permute, the `chunk` copy at 2048 elements on one core, which takes 1.26 cycles per element.

The cost per element is nearly constant over the sparse part of the grid. Between $d = 0.0013$ and $d = 0.1$ it rises only from a range of 10.07 to 10.71 cycles per element across the five shapes to a range of 11.10 to 11.63, about eight times the dense permute. It then grows to 13.83 to 14.36 at $d = 0.3711$ and to 20.25 to 20.65 at $d = 1$, about sixteen times the permute; the two 8192-element shapes do not fit the TCDM budget at $d = 1$. This is the scan of Section 3.3.7: every element is read and tested whatever the data holds, and only the nonzeros pay the stores to `a_vals` and `a_idcs`. At the density of K , $d = 0.028$, converting the 2048-element 8×256 operand takes 21,670 cycles, against 41,466 for the same operand fully dense. The shape changes the cost per element by less than one cycle at every density. The smallest shape, 768 elements, is the most expensive per element throughout, since the fixed cost of a call is spread over the fewest elements.

The SSR helps the scan and not the stores. Measured against `sla_to_sparse_csr_c` but not plotted, the speedup is 2.47× to 2.65× at every shape and density below $d = 1$, and 1.78× to 1.80× at $d = 1$, where every element passes the test and both versions pay the full store cost. The stores to `a_vals` and `a_idcs` are scalar and cannot be streamed, and the zero test cannot run inside `frep` (Section 3.3.7).

The spike compaction `sla_to_spike_csr_fp16x4` of Section 3.3.7 is the port specialized to packed fp16x4 spikes that the attention block requires. We measure it on a single core at three windows: a (chunk, head) tile of K or Q , 8×96 or 768 lanes; the two sites of 768 channels that each core compacts from x or from the attention output, 1536 lanes; and a larger contiguous 16×384 window of 6144 lanes, which separates the fixed cost of a call from the cost per element. Figure 4.18 reports its cost in cycles per element against density, together with the fp64 curves of Figure 4.17.

Unlike that of `sla_to_sparse_csr`, its cost follows density. It takes 1.74 to 1.85 cycles per element at $d = 0.0013$ across the three windows, 2.56 to 3.00 at $d = 0.028$, and 9.38 to 10.07 at $d = 0.6$. At low density most groups of four words are silent and cost one branch each; as density rises, more groups and words are live, and every spike pays its own lane test and index store. At the same 768-element 8×96 shape, the spike compaction is 5.79× cheaper than

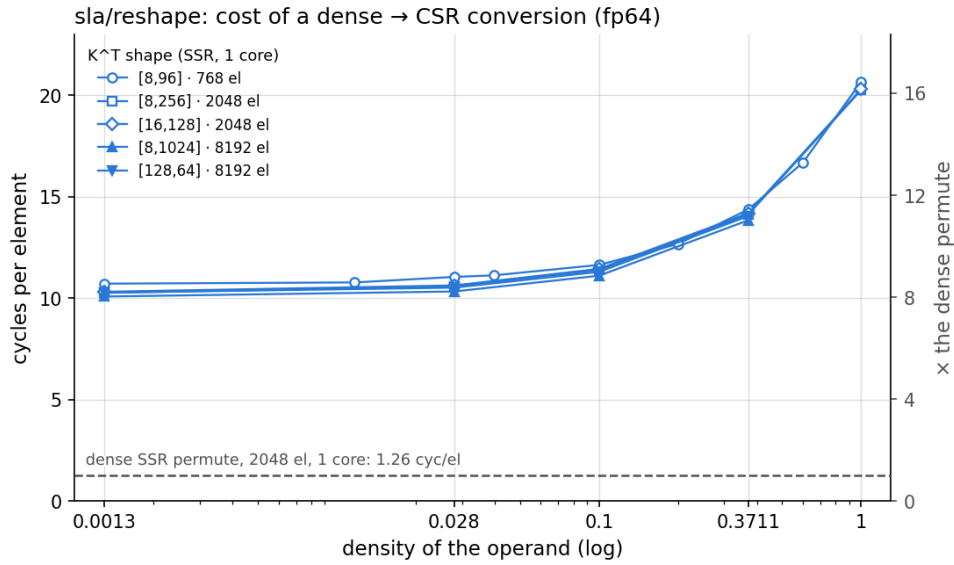


Figure 4.17: fp64 cycles per element of `sla_to_sparse_csr` against input density, single core. Each line is `sla_to_sparse_csr` at one K^T shape; the dense SSR permute is the chunk copy of `sla_contiguous` at 2048 elements on one core.

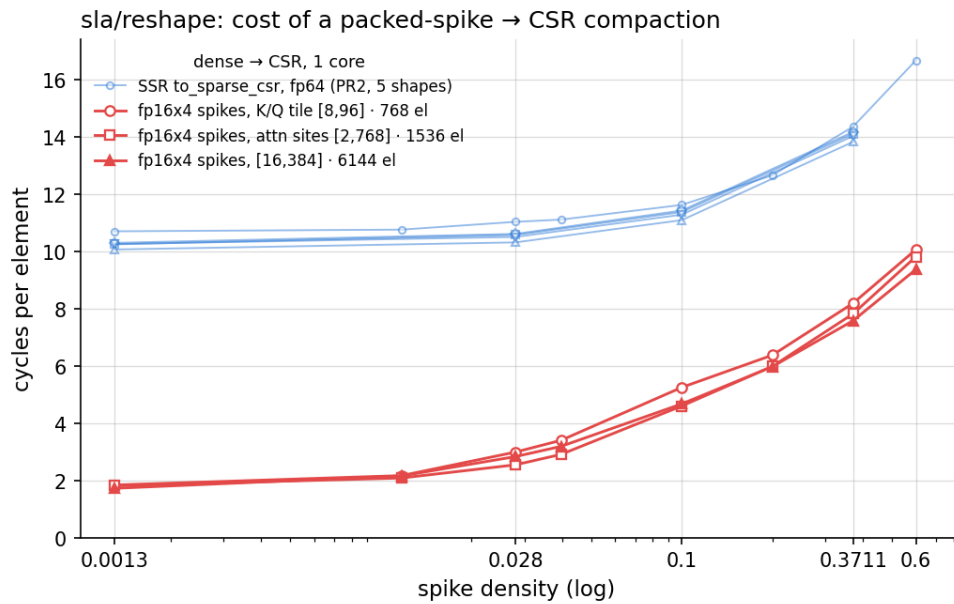


Figure 4.18: Cycles per element of the fp16x4 spike compaction and of `sla_to_sparse_csr` against input density, single core. In the legend, fp16x4 spikes is `sla_to_spike_csr_fp16x4` at one window, and SSR to_sparse_csr, fp64 (PR2, 5 shapes) is `sla_to_sparse_csr` at the five shapes of Figure 4.17.

`sla_to_sparse_csr` at $d = 0.0013$ (1,421 against 8,224 cycles), $3.68\times$ at the density of K , $d = 0.028$ (2,304 against 8,478), and $1.66\times$ at $d = 0.6$ (7,733 against 12,807). The comparison is between a general kernel and one specialized to spikes. The spike compaction accepts only binary input, reads a quarter of the bytes per element, and stores 32-bit indices but no values. The window changes the cost per element by at most 0.69 cycles at any density, and the 8×96 tile, whose rows are the shortest at 24 words, is the most expensive throughout. The attention block runs this kernel on all eight cores at once, each over its own windows, so these single-core figures do not include any contention between cores for the memory banks. Chapter 5 returns to the cost of the conversion to CSR among the limits of the approach.

4.2.11 Attention Block

The operations of the previous sections also run together, in the attention block application of Section 3.3.9: the whole attention block of STAtten on one Snitch cluster, at fp16x4 on 8 cores, in 26 stages separated by barriers, DMA transfers included. Figure 4.19 shows this chain, and Figure 4.20 names each stage. Only the Q , K and V convolutions and the compaction of their input run below the shape of the model. Their input channel count is scaled down to 24 so that each weight matrix fits in the TCDM; every measured value in this section comes from that run. Two further runs, at 12 and 16 input channels and not plotted, serve only to fit the cost of each convolution, which is then extrapolated to 768 input channels. The input data are generated at the densities measured on CIFAR-100 (Table 3.1).

Figure 4.20 reports the cycles of each stage, measured at 24 input channels in Figure 4.20a and extrapolated to 768 in Figure 4.20b. The measured block takes 101,407 cycles. Its largest stage is the attention product, $K^T V$ followed by $Q(K^T V)$, with 24,170 cycles (24%), ahead of the three convolutions at 10,081 to 10,136 cycles each, or 10,727 to 10,774 with their weight loads, the values printed in the figure. The extrapolation fits the cost of each convolution on the number of active input channels per site, at about 1,108 cycles per active channel with a worst residual of 21 cycles. The weight loads of the three convolutions and the compaction of their input also grow with the input channel count, and are fitted on the same runs; the compaction grows from 681 to 12,841 cycles. Every other stage already runs at its real shape and keeps its measured cost. At 768 input channels the block takes 1,010,255 cycles, and the three convolutions take 871,342 of them (86.2%). These convolution figures exclude the weight loads, which the figure draws as the hatched part of each convolution bar; with them, the three convolutions reach 92.0% of the block. The attention itself, meaning the compactions of K , Q and the attention output, both products and the attention LIF, takes 42,124 cycles, 4.2% of the block. At the real shape, the cost of the block is therefore set by the 1×1 convolutions and not by the attention they feed. The weight loads of the three convolutions, 57,601 cycles in the extrapolation, are counted without any overlap with computation. Section 5.2 discusses what this measurement leaves out for a full model.

Spike Driven Self Attention with STatten plugin (ACCELERATED VERSION)

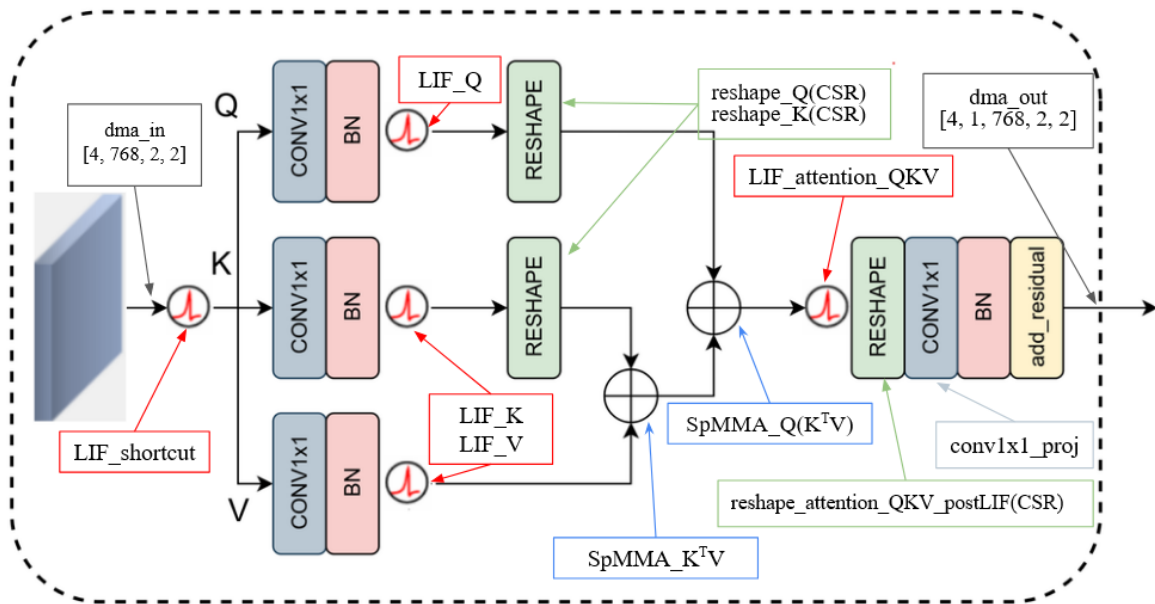
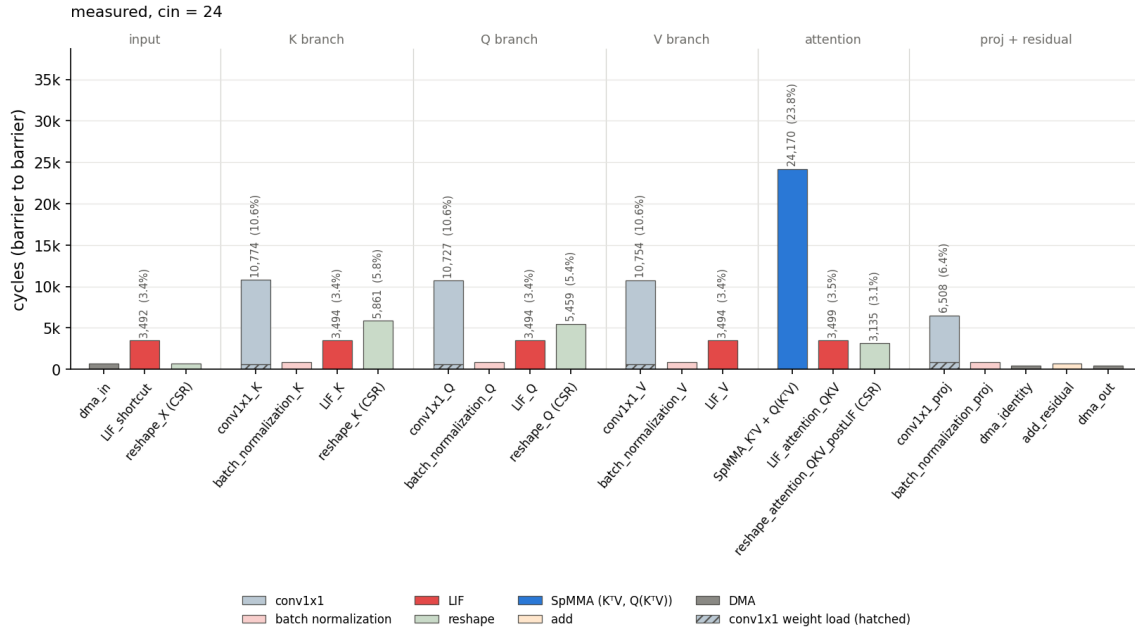


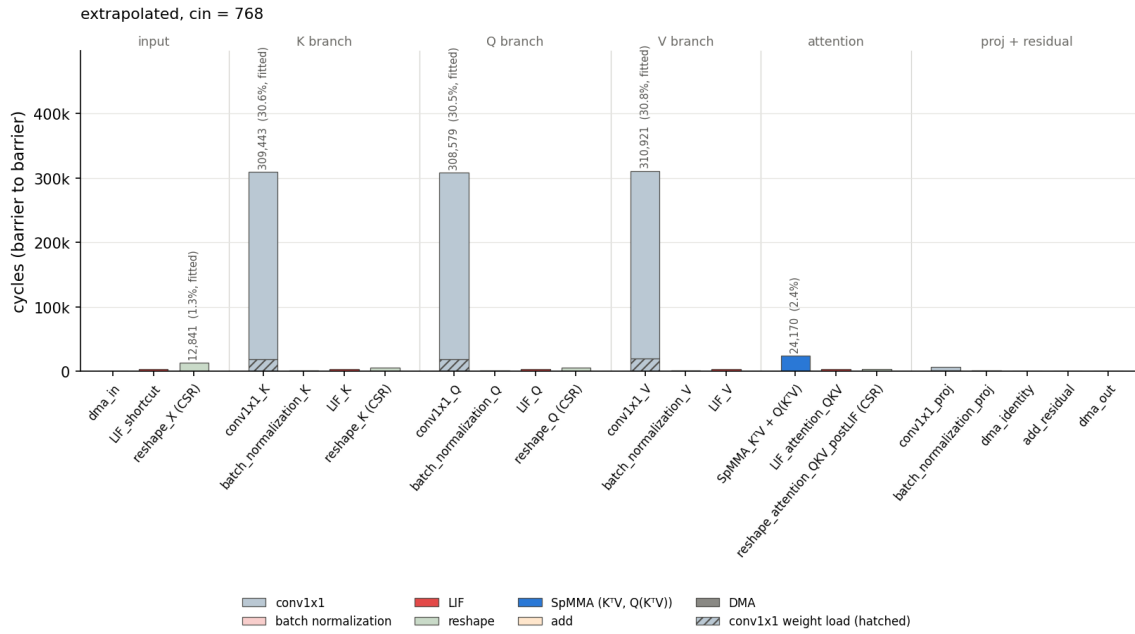
Figure 4.19: Stages of the accelerated STatten attention block

MS_SSA_Conv on one Snitch cluster, fp16x4: cycles per kernel (T = 4 timesteps, N = H·W = 4 tokens, C = 768 channels)



(a) Measured, cin=24

MS_SSA_Conv on one Snitch cluster, fp16x4: cycles per kernel (T = 4 timesteps, N = H·W = 4 tokens, C = 768 channels)



(b) Extrapolated, cin=768

Figure 4.20: Accelerated attention block execution on 8 cores fp16x4. Every operation labeled and colored on the x axis corresponds to a stage of Figure 4.19.

Chapter 5

Conclusions

5.1 Summary of Contributions

This thesis set out to measure how much of the sparsity of SNNs is retained in the attention of a spiking Transformer, and to determine whether a programmable RISC-V streaming cluster can turn that sparsity into faster inference. The results of Chapter 4, grouped by the three contributions listed in Section 1.2, answer these questions as follows.

The profiling of STAtten on CIFAR-100 (Section 3.1.3) confirms that the attention module is where sparsity concentrates. The query, key and value spike tensors are about 96.87% sparse, the attention output 97.59%, and the output of the LIF neurons that follow it 99.86%. The input of the Q , K and V convolutions, `x_for_QKV`, is denser, at 65.89% sparsity.

On the Snitch cluster we implemented kernels for the operations of the attention block: the SpMVA, the 1×1 convolutions with two engines, BN in dense and CSR form, the LIF update, and the reshape and CSR conversion operations, in fp64 and, where applicable, in packed fp32x2 and fp16x4 variants (Section 3.3). Replacing the multiplication with an addition contributes little to the speed by itself, a median saving of about 5%; the gain of the SpMVA comes from the indirect SSR, the FREP loop and lane packing (Section 4.2.1). The addition matters for energy instead. The SpMVA skips every zero operand and replaces each remaining multiplication by an addition, and Yao *et al.* [4] take an addition at 0.9 pJ against 4.6 pJ for a MAC, for 32-bit floating-point operations in 45 nm. We have not measured the energy this saves on the Snitch cluster. Over a plain-C reference, the SpMVA reaches up to 8.50 $\times$ at fp64, 17.31 $\times$ at fp32x2 and 34.02 $\times$ at fp16x4. Over their fp64 versions, packing speeds up the convolution by up to 2.42 $\times$ and 6.20 $\times$, and the LIF update by 1.94 $\times$ to 1.95 $\times$ and 3.61 $\times$ to 3.71 $\times$. The LIF update visits every neuron at every timestep, so its gain comes from lane width and not from sparsity; its work is split over neurons, while the timesteps of each neuron run in sequence. The cost of the dense BN apply depends only on the size of the tensor, while that of the CSR apply depends on both its size and its density. On eight cores the SpMVA reaches up to 7.31 $\times$ over a single core, the `indir` convolution 7.97 $\times$ and the fp64 LIF update 7.87 $\times$. The CSR conversion and the

view materialization are one-to-one ports of PyTorch operations. Of the two, the channels-last attention block uses only the conversion to CSR, in its own fp16x4 form. Below full density the conversion runs $2.47\times$ to $2.65\times$ faster than its plain-C reference, and the view materialization, with the construction of its views, $4.5\times$ to $8.0\times$.

The measurements also show where sparsity stops paying. Against a dense GEMM of the same lane width, the `spmva` engine is faster below a density of 0.29 at fp64, 0.39 at fp32x2 and 0.75 at fp16x4, on the 8×96 tile with a square weight matrix. The `indir` engine cannot run at that shape. At $A[8,96]\times W[96,64]$, where it can, it skips empty rows outright and reaches $14.04\times$ over the dense GEMM at $d = 0.0013$ and fp64 (Section 4.2.3). The overheads of the sparse structures are measured explicitly. The un-interleave stage of the packed SpMVA takes part of its gain. On the smallest and sparsest input it brings the net speedup under the plain-C baseline, and the net speedup is above $1\times$ at every shape only from $d = 0.1$ upward. On large and denser inputs the stage costs less: at 8×1024 and $d = 0.6$ the net speedup is still $15.47\times$ at fp32x2 and $30.58\times$ at fp16x4. A consumer that reads the interleaved layout directly would not pay this stage at all, and would keep the full speedups given above. The κ/λ gather of the CSR BN takes 41% to 81% of its time at $d = 0.6$ and removes almost all of its gain at fp16x4 (Section 4.2.5). The general CSR conversion runs on a single core and tests every element of the dense operand, so its cost per element is nearly constant below $d = 0.1$ and about twice as high at full density (Section 4.2.10). Scaling has two exceptions. `spmva` reaches only $2.25\times$ to $4.81\times$ on eight cores at $d = 1$, for a cause these measurements do not isolate. The `chunk` copy stops at two cores, because it splits its work along an outer axis of size two (Section 4.2.9).

Together, these results answer the question raised in Chapter 1. A programmable RISC-V streaming cluster turns the unstructured, run-time sparsity of a spiking Transformer into measured speedups without fixing the model in hardware: the same kernels run at every shape and density that fits the TCDM. They outperform dense computation on the sparsest tensors, such as the input of `proj_conv` at a density of 0.0014. The input of the Q , K and V convolutions, at a density of 0.34, lies close to the crossover at fp64 (0.29) and fp32x2 (0.39). At fp16x4, the precision of the attention block, the crossover lies even further, above 0.6 at 0.75, so at this density the sparse engine is faster than the dense one. This input decides the cost of the block. In the attention block of Section 4.2.11, extrapolated to 768 input channels, the three convolutions it feeds take 86.2% of the cycles, and the attention itself 4.2%. The third position is therefore reached, within the limits discussed in Section 5.2: the gain depends on density and shape, the bandwidth of the shared memory bounds the throughput of a full model, the conversion to CSR is still serial, and the evaluation is restricted to tensors that fit in a single cluster.

5.2 Limitations and Threats to Validity

Our scope was limited to the attention mechanism of the Transformer, since implementing a complete Transformer is a large task. The results come from one Snitch cluster and one attention

block. The block of Section 4.2.11 was verified bit for bit on generated data with the input of the Q , K and V convolutions reduced to 24 channels, and its cost at the real 768 channels is extrapolated, not measured. The patch embedding, the MLP and the classification head have not been ported. The patch embedding uses 3×3 convolutions and max-pooling, which no kernel in this work implements. Implementations at scale are possible with the kernels proposed in this thesis, but scaling to a full model on several clusters faces the limits discussed below.

The first limit is memory bandwidth. Yao *et al.* [4] report 66.34 M parameters for the *Spike-Driven Transformer* with 8 blocks and 768 channels, and the reference implementation stores them in fp32, about 265 MB. At the fp16 precision of our port, every inference at batch size 1 still streams about 133 MB of weights. The 16 sites reuse these weights very little. A convolution that gathers only the weight rows of active input channels uses each gathered row at about one site, and the Q , K and V convolutions, whose input is 34% dense, use each weight row at about $0.34 \times 16 \approx 5$ of the 16 sites. Throughput is therefore bounded by the bandwidth of the shared memory, not by the number of clusters. Double buffering can hide the latency of these transfers but cannot reduce the bytes moved. In this work it is implemented only for the SpMVA. The other operations process data differently, so each would need its own double-buffering implementation, and they were tested only with data that fits inside the TCDM.

The second limit is the serial CSR conversion. The general conversion, `sla_to_sparse_csr`, runs on a single core, because its row pointers are a running total and no parallel prefix sum exists in this work. Its zero test cannot run inside `frep`, so the SSR speeds it up by only $2.47\times$ to $2.65\times$ below full density (Section 4.2.10). The attention block works around this limit only for its own case, with the port specialized to fp16x4 spikes. Each core converts its own tiles into a separate CSR, so no running total crosses cores. Testing the packed words as integers also makes one conversion $3.68\times$ cheaper than the fp64 one at the density of K , on a single core. A single large matrix, or a tensor that is not binary, still depends on the serial conversion.

The third limit is the small amount of parallel work. With only 16 sites, fixed costs per call and the 26 barrier-separated stages of the block take a growing share of the time as the work per core shrinks. The `chunk` copy, for instance, stops scaling at two cores (Section 4.2.9).

The fourth limit comes from sparsity itself. The work per core depends on the spike pattern, so the balance between cores varies from input to input. The size of a sparse output is not known in advance, and the 112 KiB TCDM budget, together with the 16-bit indices, limits the tile shapes.

The last limit concerns timesteps and precision. The LIF neurons process timesteps in order, so time cannot be split across cores without streaming the weights again for each group of timesteps. Verification is bit-exact only against references computed in the fp16 arithmetic of the kernels themselves. The accuracy of the port against the fp32 reference, over all eight blocks, has not been assessed.

5.3 Future Work

5.3.1 Short-Term Directions

As anticipated in Section 5.2, several optimizations could still be implemented to further exploit sparsity. The most relevant is the parallelization of the CSR conversion, which proved to be a challenge and still represents the most expensive operation per element measured. It could be processed by many cores to hide memory latency, instead of by a single one. The attention block of Section 3.3.9 already runs its compactions on all eight cores, because each core converts its own tiles into a separate CSR; a single large matrix still needs a prefix sum over the counts of the cores. If the cost of this conversion dropped significantly, chaining the conversion before every operation on a sparse matrix would become a viable option.

SIMD and vectorization have been one of the main sources of performance in this thesis, and these gains could be extended further by implementing fp8 data parallelism, making each register an 8-lane register, a $2\times$ increase over fp16. Going from fp64 to fp32 to fp16 gave an almost linear increase in performance for the SpMVA and the LIF update, and a superlinear one for the convolution, up to $6.20\times$ at fp16x4, which a better outer loop could improve further. We expect a similar gain from fp8 wherever no per-element cost, such as the gather of the CSR BN, stays fixed across lane widths. Before fp8 is adopted, its accuracy must be assessed, a question that Section 5.2 leaves open even at fp16. Double buffering has been implemented for the SpMVA and can be extended to the other operations, hiding the latency of data transfers and allowing larger input sizes.

Each kernel could also be optimized individually. For instance, the `spmva` engine, which reuses the SpMVA kernel to cover a full matrix, could handle empty rows better. It pays a fixed cost for every row, empty rows included, which is why it performs better on larger matrices than on smaller ones: longer rows spread that cost over more work (Section 4.2.3). The `indir` engine could be extended to work with matrices of all sizes: it currently computes each row address with a single `slli`, so it can be built only where a row of $\mathbf{W}$ occupies a power-of-two number of bytes (Section 3.3.3). The BN and convolution kernels could be improved with weight streaming, to make them more efficient on the large matrices typical of Transformer workloads. Such a mechanism could be developed together with double buffering. Fusing kernels is not a source of speed: fusing BN into the convolution saves below 1% at the density of the Q , K and V input, DMA transfers included (Section 4.2.5). The fused sub-block of Section 3.3.8, which chains a whole sub-block within one TCDM residency, was built as a dataflow demonstration and is not evaluated in this thesis.

The attention block of Section 3.3.9 runs three kernels that are ports of existing ones, needed only because the originals form a row address by shifting the index and assume a row step equal to the row width. Generalizing the original convolution and row-product kernels, with byte-offset indices and a row step separate from the row width, would let the block reuse them directly. Two

costs of the block are also open. Its convolution was not compared with the other engines at the shape of the block, where it runs at about 23 cycles per nonzero and 32-channel tile, the same instruction-mix ceiling as `indir`. Its $K^T V$ stage writes zeros to every row of the product, while Q reads only the rows that its nonzeros select, so about three quarters of that stage could be avoided.

The LIF kernels leave the sparsity of their input unused, and two ways of exploiting it are worth exploring, provided their current limitations are overcome. The first is dead-neuron compaction: a neuron whose input is zero at every timestep can never fire, so it can be removed and the dense kernel run on the remaining neurons only. As Section 3.3.6 argued, the compaction is itself a scan over the whole input, so it pays off only once it runs on the cluster, parallelized like the CSR conversion discussed above, rather than on the host outside the measured kernel. The second is a sparse LIF kernel that stores the input in CSR form along the time axis and visits only its nonzero inputs, collapsing each run of empty timesteps into a single multiplication by a power of the leak factor. Such a kernel updates each membrane through a chain of dependent floating-point operations, so it needs a SIMD formulation of that chain to compete with the packed dense kernels.

5.3.2 Long-Term Research Vision

The long-term goal of this work is to implement the full Transformer with the kernels proposed in this thesis. This will require further work on porting the Transformer architecture, but it can build on all the conclusions and data presented in this work.

Appendix A

Development Environment

We developed the project inside a cycle-accurate simulation container provided by the developers of the Snitch cluster. The container is based on Ubuntu 20.04 and bundles the full Snitch cluster software stack, from the RISC-V toolchain to the cycle-accurate *Register-Transfer Level* (RTL) simulator used to collect every timing result reported in Chapter 4. Further details on the container and the environment are available at https://github.com/pulp-platform/snitch_cluster.

A.1 Code Repositories

The code used in this thesis is available in the repositories listed below.

- **Kernels and benchmarks.** The kernels of Chapter 3, their data generators and verification scripts, and the attention block of Section 4.2.11 are in a fork of the Snitch cluster repository, on the `snn-LIF` branch, under `sw/kernels/sla/`:
https://github.com/MarcoCrisafulli5/snitch_cluster
- **Profiled model.** The profiling of Chapter 3 runs a fork of the PyTorch implementation of STAtten released by Lee *et al.* [5], extended with the timing and sparsity measurements:
<https://github.com/MarcoCrisafulli5/STAtten>
The original implementation is available at
<https://github.com/Intelligent-Computing-Lab-Panda/STAtten>
- **Profiling results.** The results of the profiling as CSV files, the scripts that produce them from the text captures of the model, and a notebook that draws graphs from them:
https://github.com/MarcoCrisafulli5/STAtten_profiler

Appendix B

Kernel Selection and Verification

The kernels of Chapter 3 exist in several variants, which differ in engine, lane width, data layout and optional pipeline stages. This appendix describes how the data of an experiment is generated (Section B.1), how one variant is selected for it (Section B.2), and how its output is verified before any cycle count is recorded (Section B.3). Both mechanisms are shared by all the kernels of this thesis.

B.1 Configuration and Data Generation

Every kernel is built from a configuration file, `data/params.json`, which records the kernel family, the lane width (fp64, fp32x2 or fp16x4), the tensor shape, the input density, the number of worker cores, and any optional pipeline stage. A host-side Python script, `datagen.py`, reads this file and generates everything the kernel needs: the input tensors at the requested density, the reference (golden) outputs, and a C header, `data.h`, containing both the data arrays and a set of constants that describe the configuration.

This script performs all the offline transformations described in Chapter 3: the prescaling of the CSR column indices by the row stride of **W** (Section 3.3.2), the folding of BN into the coefficients κ and λ and their interleaved `kl` table (Section 3.3.4), and the interleaving of dense operands for the SIMD variants (Section 3.3.1). The same script rejects configurations that a kernel cannot express, for instance an `indir` convolution whose rows of **W** are not a power-of-two number of bytes, or a fused BN requested on a dense convolution output.

B.2 Compile-Time Kernel Selection

The `main.c` of each kernel contains a single preprocessor selection block that maps the constants of `data.h` to one concrete function. For the convolution, the constants `CONV_KERNEL` and `CONV_ENGINE` select the kernel, bound to the name `KTYPE`, and its matching index array, `KIDX`. For the LIF update, a single constant, `LIF_KERNEL`, selects among the variants of

Table 3.6. For the SpMVA, two constants act together: `SNN_ENGINE` selects the ISSR kernel or the plain-C reference used as the baseline of Chapter 4, and `SIMD_COLS`, the lane width, selects one of the three variants of Section 3.3.1. A combination that matches no variant stops the build with an explicit error.

Because every variant of a family shares the same signature, the call site itself is unconditional: the rest of `main.c` is identical for every variant, and a change of kernel changes a single binding. This allows the evaluation of Chapter 4 to compare variants within an otherwise identical program.

Optional pipeline stages are controlled differently. In the convolution pipeline, the constants `CONV_OUT_CSR` and `CONV_FUSE_BN` switch the CSR compaction and the fused BN on or off. They describe the shape of the pipeline, that is, what the data looks like at each stage, rather than which arithmetic runs, and the fused BN kernel is then selected by lane width alone. The fused attention sub-block of Section 3.3.8 follows the same structure, with the form of its final product selected by a separate parameter. In the standalone BN kernel, a parameter chooses whether the κ/λ gather of the CSR path runs on the cluster or is precomputed on the host. These are the two settings used in Chapter 4 to separate the cost of the gather from that of the apply kernel. The dense BN kernels are compiled only when the dense configuration is requested, so that a CSR-only user such as the fused convolution pipeline can include the BN header without the dense code.

Finally, each kernel states the size of its whole TCDM data structure in a compile-time assertion, so that an operand that does not fit the TCDM budget discussed in Chapter 4 fails at compile time rather than at run time. The fused sub-block also asserts, in the same way, that the output layout of its LIF stage coincides with the input layout of its CSR compaction, the identity on which Section 3.3.8 relies.

B.3 Verification

A cycle count is recorded only for an output that has been verified. The golden values are computed by `datagen.py` operation by operation in the lane type of the kernel, with the same rounding the hardware performs. For instance, the reference LIF step uses a fused multiply-add to match the `fmadd.d` the kernel issues, and the reference BN rounds the multiplication and the addition separately, as the `vfmul` and `vfadd` pair does. As a result, every output is checked for exact equality (`atol = 0`) rather than within a tolerance.

The checks are specific to each kernel. In the convolution pipeline, `verify_dense` compares the raw output of the convolution with its golden before BN, and, when CSR output is enabled, `verify_csr` also checks the compacted values, indices and row pointers after BN. The untiled plain-C convolution, `conv1x1_c`, must agree bit-exactly with the tiled kernels, since tiling changes only when a partial sum is written out and never the order in which nonzeros are summed. The LIF kernels are checked by `verify_spikes` against the plain-C reference `lif_update_c`. Because it reads every element from L3, `verify_spikes` dominates the

simulated time even though it runs outside the timed kernel window, so it is split across up to eight cores. The four shape operations of Table 3.7 are checked bit-exactly against a NumPy reference, since they move data without any arithmetic that could introduce rounding.

The view descriptors of Section 3.3.7 are validated separately. Before any shape operation runs, `sla_view_selftest` rebuilds from view descriptors a set of SSR configurations that were previously written by hand and are known to be correct, among them the three-dimensional input stream of the LIF kernel and the transpose of the standard kernel library. It then checks that both configurations visit the same addresses in the same order. Comparing address sequences rather than register contents is the stronger check, since a descriptor may merge axes differently from a hand-written configuration and still be correct. This validation matters because a stride error on this hardware raises no fault and produces wrong data.

Bibliography

- [1] A. L. Hodgkin and A. F. Huxley, “A quantitative description of membrane current and its application to conduction and excitation in nerve,” *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952.
- [2] A. K. Kosta and K. Roy, “Adaptive-SpikeNet: Event-based optical flow estimation using spiking neural networks with learnable neuronal dynamics.” arXiv preprint arXiv:2209.11741, 2022.
- [3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, pp. 5998–6008, 2017.
- [4] M. Yao, J. Hu, Z. Zhou, L. Yuan, Y. Tian, B. Xu, and G. Li, “Spike-driven Transformer.” arXiv preprint arXiv:2307.01694, 2023.
- [5] D. Lee, Y. Li, Y. Kim, S. Xiao, and P. Panda, “Spiking Transformer with spatial-temporal attention.” arXiv preprint arXiv:2409.19764, 2024.
- [6] J. Liu, X. He, W. Liu, and G. Tan, “Register-aware optimizations for parallel sparse matrix–matrix multiplication,” *International Journal of Parallel Programming*, vol. 47, no. 3, pp. 403–417, 2019.
- [7] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models.” arXiv preprint arXiv:2001.08361, 2020.
- [8] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos, “Compute trends across three eras of machine learning.” arXiv preprint arXiv:2202.05924, 2022.
- [9] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, “Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing,” in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2015.
- [10] P. Panda and K. Roy, “Unsupervised regenerative learning of hierarchical features in spiking deep networks for object recognition,” in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pp. 299–306, IEEE, 2016.

- [11] N. Jegham, M. Abdelatti, C. Y. Koh, L. Elmoubarki, and A. Hendawi, “How hungry is AI? Benchmarking energy, water, and carbon footprint of LLM inference.” arXiv preprint arXiv:2505.09598, 2025.
- [12] M. V. DeBole, B. Taba, A. Amir, F. Akopyan, A. Andreopoulos, W. P. Risk, J. Kusnitz, C. Ortega Otero, T. K. Nayak, R. Appuswamy, P. J. Carlson, A. S. Cassidy, P. Datta, S. K. Esser, G. J. Garreau, K. L. Holland, S. Lekuch, M. Mastro, J. McKinsty, C. di Nolfo, B. Paulovicks, J. Sawada, K. Schleupen, B. G. Shaw, J. L. Klamo, M. D. Flickner, J. V. Arthur, and D. S. Modha, “TrueNorth: Accelerating from zero to 64 million neurons in 10 years,” *Computer*, vol. 52, no. 5, pp. 20–29, 2019.
- [13] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, *et al.*, “Loihi: A neuromorphic manycore processor with on-chip learning,” *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.
- [14] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber, “SpiNNaker: A 1-W 18-core system-on-chip for massively-parallel neural network simulation,” *IEEE Journal of Solid-State Circuits*, vol. 48, no. 8, pp. 1943–1953, 2013.
- [15] J. Li, G. Shen, D. Zhao, Q. Zhang, and Y. Zeng, “FireFly: A high-throughput hardware accelerator for spiking neural networks with efficient DSP and memory optimization,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 8, pp. 1178–1191, 2023.
- [16] Q. Chen, C. Gao, and Y. Fu, “Cerebron: A reconfigurable architecture for spatiotemporal sparse spiking neural networks,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 10, pp. 1425–1437, 2022.
- [17] S. Panchapakesan, Z. Fang, and J. Li, “SyncNN: Evaluating and accelerating spiking neural networks on FPGAs,” *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 48:1–48:27, 2022.
- [18] A. Carpegna, A. Savino, and S. Di Carlo, “Spiker: An FPGA-optimized hardware accelerator for spiking neural networks,” in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 14–19, IEEE, 2022.
- [19] W. Fang, Y. Chen, J. Ding, Z. Yu, T. Masquelier, D. Chen, L. Huang, H. Zhou, G. Li, and Y. Tian, “SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence,” *Science Advances*, vol. 9, no. 40, p. eadi1480, 2023.
- [20] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micekevicius, “Accelerating sparse deep neural networks.” arXiv preprint arXiv:2104.08378, 2021.

- [21] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, “The RISC-V instruction set manual, volume I: User-level ISA, version 2.1,” Tech. Rep. UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.
- [22] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, “Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads.” arXiv preprint arXiv:2002.10143, 2020.
- [23] P. Scheffler, F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, “Indirection stream semantic register architecture for efficient sparse-dense linear algebra.” arXiv preprint arXiv:2011.08070, 2020.
- [24] S. Manoni, P. Scheffler, L. Zanatta, A. Acquaviva, L. Benini, and A. Bartolini, “SpikeStream: Accelerating spiking neural network inference on RISC-V clusters with sparse computation extensions.” arXiv preprint arXiv:2504.06134, 2025.
- [25] E. M. Izhikevich, “Simple model of spiking neurons,” *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003.
- [26] W. Maass, “Networks of spiking neurons: The third generation of neural network models,” *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.
- [27] M. Westerink, S. Sohail, B.-J. van der Zwaag, S. Gerez, and A. Yousefzadeh, “The sparsity tax: Weight sparsity trade-offs in event-driven SIMD and SIMT neuromorphic cores.” arXiv preprint arXiv:2607.22790, 2026.
- [28] H. Yoon, S. Ryu, and J.-J. Kim, “Sparse-on-dense: Area and energy-efficient computing of sparse neural networks on dense matrix multiplication accelerators.” arXiv preprint arXiv:2604.26587, 2026.
- [29] Z. Wang, “The sparsity ceiling: Where spiking networks can—and cannot—trade activity for energy.” arXiv preprint arXiv:2607.26648, 2026.