UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

MATHEUS ALMEIDA SILVA

Quantized Gemmini: A Design Space Exploration of Integer Precision in DNN Acceleration

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Antonio Carlos S. Beck

Porto Alegre
July 2026

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profª. Márcia Cristina Bernardes Barbosa
Vice-Reitor: Prof. Pedro de Almeida Costa
Pró-Reitora de Graduação: Profª. Nádya Pesce da Silveira
Diretor do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenador do Curso de Engenharia de Computação: Prof. Paulo Francisco Butzen
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

“It is important to draw wisdom from many different places. If you take it from only one place, it becomes rigid and stale. Understanding others, the other elements and the other nations will help you become whole.”

— Uncle Iroh

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Dr. Antonio Carlos S. Beck, for his guidance, patience, and support throughout my undergraduate years, and the development of this work. His mentorship was essential to my personal, academic, and professional growth.

I also acknowledge CADMicro (UFRGS, 2025) for providing the EDA tools and the computational environment used for the synthesis and characterization in this work.

A special acknowledgment goes to PET Computação and to Prof. Érika Fernandes Cota. Beyond the experience gained through several projects, it was the friendships I built and the encounters with different realities that taught me the empathy without which no science could truly be human.

I am deeply grateful to my family for their unconditional support in the pursuit of my dreams, despite all our limitations.

Last but not least, I thank my friends. Thank you for the endless support, for the laughter that fills ordinary days, and for walking beside me through every high and low. Your honesty and authenticity remind me, again and again, that becoming better is always a matter of choice.

ABSTRACT

The growing demand for efficient Deep Neural Network (DNN) inference has driven the adoption of Domain-Specific Architectures (DSAs), often built on open, parameterizable platforms such as the Gemmini accelerator generator. Among the levers available to such architectures, numerical precision strongly shapes the datapath, the on-chip memories, and therefore the area, power, and timing of the resulting circuit. This work presents a design space exploration on Gemmini centered on integer precision. Four configurations are studied along two architectural axes: datapath precision (INT4, INT8, and INT16) and, within INT4, accumulator data width (16 versus 32 bits). Each configuration is taken through a real ASIC logic-synthesis flow with Hammer and Cadence Genus targeting the ASAP7 predictive 7 nm node, and its throughput is measured through cycle-accurate simulation of ResNet-50 in Verilator. Classification accuracy is not re-measured but drawn from the quantization literature. The results show that precision is the dominant lever and acts on the compute: the spatial array grows to $1.93\times$ the baseline area and roughly twice its power at INT16 and shrinks to about half at INT4, while the byte-capacity-sized scratchpad stays essentially constant and damps the total-area swings. INT4 with a 16-bit accumulator emerges as the most efficient point, delivering $1.25\times$ the baseline throughput at $0.60\times$ its energy per inference, whereas INT16 is dominated on every hardware axis.

Keywords: Deep Neural Networks. Domain-Specific Architectures. Gemmini Accelerator. Quantization. Design Space Exploration.

Gemmini Quantizado: Uma Exploração do Espaço de Projeto da Precisão Inteira na Aceleração de DNNs

RESUMO

A crescente demanda por inferência eficiente de Redes Neurais Profundas (DNNs) tem impulsionado a adoção de Arquiteturas Específicas de Domínio (DSAs), frequentemente construídas sobre plataformas abertas e parametrizáveis como o gerador de aceleradores Gemmini. Entre os fatores disponíveis a essas arquiteturas, a precisão numérica molda fortemente o caminho de dados, as memórias on-chip e, portanto, a área, a potência e a temporização do circuito resultante. Este trabalho apresenta uma exploração do espaço de projeto no Gemmini centrada na precisão inteira. Quatro configurações são estudadas ao longo de dois eixos arquiteturais: a precisão do caminho de dados (INT4, INT8 e INT16) e, dentro do INT4, a largura de dados do acumulador (16 contra 32 bits). Cada configuração é submetida a um fluxo real de síntese lógica ASIC com Hammer e Cadence Genus, tendo como alvo o nó preditivo ASAP7 de 7 nm, e seu desempenho é medido por simulação ciclo-acurada do ResNet-50 no Verilator. A acurácia de classificação não é remedida, sendo extraída da literatura de quantização. Os resultados mostram que a precisão é o fator dominante e atua sobre a computação: o arranjo sistólico cresce para $1{,}93\times$ a área da base e cerca do dobro de sua potência em INT16, e encolhe para cerca de metade em INT4, enquanto a scratchpad, dimensionada por capacidade fixa em bytes, permanece essencialmente constante e amortece as variações de área total. O INT4 com acumulador de 16 bits surge como o ponto mais eficiente, entregando $1{,}25\times$ a vazão da base a $0{,}60\times$ sua energia por inferência, enquanto o INT16 é dominado em todos os eixos de hardware.

Palavras-chave: Redes Neurais Profundas, Arquiteturas Específicas de Domínio, Acelerador Gemmini, Quantização, Exploração do Espaço de Projeto.

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
BF16	Brain Floating-Point
CMOS	Complementary Metal-Oxide-Semiconductor
CNN	Convolutional Neural Network
CONV	Convolutional (Layer)
CPU	Central Processing Unit
DMA	Direct Memory Access
DNN	Deep Neural Network
DRAM	Dynamic Random-Access Memory
DSA	Domain-Specific Architecture
DSE	Design Space Exploration
EDP	Energy-Delay Product
FC	Fully Connected (Layer)
FP16	Half-Precision Floating-Point
FP32	Single-Precision Floating-Point
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HDL	Hardware Description Language
ICT	Information and Communications Technology
IS	Input-Stationary
LEF	Library Exchange Format
MAC	Multiply-Accumulate
NAS	Neural Architecture Search

ONNX	Open Neural Network Exchange
OS	Output-Stationary
PDK	Process Design Kit
PE	Processing Element
PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
RoCC	Rocket Chip Coprocessor
RTL	Register-Transfer Level
SDC	Synopsys Design Constraints
SoC	System-on-Chip
SRAM	Static Random-Access Memory
STA	Static Timing Analysis
TDP	Thermal Design Power
TLB	Translation Lookaside Buffer
TPU	Tensor Processing Unit
VHDL	VHSIC Hardware Description Language
VLSI	Very-Large-Scale Integration
WS	Weight-Stationary

LIST OF FIGURES

Figure 2.1 Energy Forecast. 16
Figure 2.2 Transistors per chip of Intel microprocessors vs. Moore's Law. 18
Figure 2.3 Transistors per chip and power per mm^2. 18
Figure 2.4 Functional organization of Google Tensor Processing Unit (TPU v1). 20
Figure 2.5 Basic principle of a systolic system. 21
Figure 2.6 Typical structure of a two-dimensional systolic array accelerator. 22
Figure 2.7 The im2col transformation, which recasts a convolution as a matrix multiplication. 23
Figure 2.8 Processing element structures for the output-stationary and weight-stationary dataflows. 24
Figure 2.9 Partitioning a workload into tiles for execution on a fixed-size systolic array. 25
Figure 2.10 Asymmetric quantization maps an offset float range, so the real zero maps to a nonzero integer zero-point ($z \neq 0$). 26
Figure 2.11 Symmetric quantization maps the float range onto a grid centered at zero, so the real zero maps exactly to the integer zero ($z = 0$) 27
Figure 2.12 Generalized ASIC design flow, partitioned into front end and back end. 30
Figure 2.13 Logic-synthesis flow: the RTL, the standard-cell library, and the timing constraints are combined to produce a gate-level netlist, from which area, timing, and power are reported. 31

Figure 3.1 Comparison of DNN accelerator generators 36

Figure 4.1 Gemmini integrated as a RoCC coprocessor in a Rocket Chip SoC, showing the controller, the banked scratchpad, the systolic array, the transposer, and the accumulator SRAM. 40
Figure 4.2 Hierarchy of the Gemmini spatial array—mesh of tiles, tile of PEs—and the PE datapath for the weight-stationary (WS) and output-stationary (OS) dataflows. 41
Figure 4.3 Delay registers skew the operand streams so that the correct elements meet in each PE, shown for a 2×2 matrix multiplication. 42
Figure 4.4 The transposer streams a matrix in along one axis and out along the other, transposing it; shown for a 2×2 matrix over five cycles. 43
Figure 4.5 Unified local address space: the scratchpad occupies the low addresses and the accumulator a high range (from `0x80000000`), with narrow scratchpad elements and wide accumulator elements. 45
Figure 4.6 The move-in (`mvin`) instruction transfers a two-dimensional tile from main memory into the scratchpad, using a main-memory stride and, for wide blocks, a local private stride. 46

Figure 6.1 Cell-area breakdown per module for the four configurations under the fixed 5 ns constraint. 60
Figure 6.2 Power across the four configurations under 5 ns clock period. 61
Figure 6.3 Normalized ResNet-50 throughput across the four configurations. 64
Figure 6.4 Normalized energy per inference across the four configurations. 65
Figure 6.5 Cited accuracy (ΔTop-1 on ResNet-50/ImageNet) against the measured cell area, normalized to `INT8`. 66

Figure 6.6 Cited accuracy (ΔTop-1 on ResNet-50/ImageNet) against the energy per inference, normalized to INT8. 67
Figure 6.7 Cited accuracy (ΔTop-1 on ResNet-50/ImageNet) against the throughput, normalized to INT8. 67

LIST OF TABLES

Table 4.1 Gemmini's numerical datatype parameters, separated into the storage types that size the on-chip memories and the datapath types that size the array's arithmetic.....42

Table 5.1 Configurations explored and the comparison each enables. Datatype widths are derived in Chapter 4 and given in Table 5.2.....49
Table 5.2 Datatype widths (in bits) of the configurations explored. `INT4` is the only precision for which the accumulator width is varied (Section 5.2.2).....50

Table 6.1 Synthesized cell area by module, in mm^2, under the fixed 5 ns constraint. The italicized rows normalize each module and the Gemmini total to the `INT8` baseline.....59
Table 6.2 Post-synthesis vectorless power, in mW, under the fixed 5 ns constraint. The italicized rows normalize each block and the Gemmini total to the `INT8` baseline.....61
Table 6.3 Minimum period and maximum frequency per configuration.....62
Table 6.4 ResNet-50 inference cycle counts, in 10^6 cycles. The last column normalizes the total to the `INT8` baseline.....63
Table 6.5 Latency and throughput for one ResNet-50 inference, each at the configuration's own $F_{\max}$. Throughput is normalized to the `INT8` baseline.....63
Table 6.6 Energy per inference, normalized to the `INT8` baseline. Power is taken at each configuration's $F_{\max}$.....64
Table 6.7 Cited accuracy (ΔTop-1 on ResNet-50/ImageNet) paired with the measured hardware cost, normalized to the `INT8` baseline. The calibration method (PTQ/QAT) is noted.....66

CONTENTS

1 INTRODUCTION **14**
2 BACKGROUND **16**
2.1 Sustainable Computing and the Environmental Impact of AI **16**
2.2 The Slowdown of Moore's Law and Dennard Scaling **17**
2.3 Domain-Specific Architectures (DSAs) **19**
2.4 Systolic Arrays **20**
2.5 Deep Neural Network Quantization **24**
2.6 ASIC Design and Logic Synthesis Flow **29**
2.7 Low-Power Hardware Design Foundations **32**
3 RELATED WORK **35**
3.1 Deep Neural Network Accelerators Landscape **35**
3.2 Design-Space Exploration on Gemmini **36**
3.3 Quantization-Aware Accelerator Co-Exploration **36**
3.4 Multi-Precision Arithmetic Hardware **37**
3.5 Positioning of This Work **38**
4 THE GEMMINI ACCELERATOR **39**
4.1 Architectural Template and SoC Integration **39**
4.2 Spatial Array: Mesh, Tile, and Processing Element **40**
4.3 Datatypes and Bit-Width Parameters **42**
4.4 Scratchpad and Accumulator Memory **44**
4.5 Scaling Pipelines: Mvin Scale and Accumulator Scale **47**
5 METHODOLOGY **48**
5.1 Design Space Definition **48**
5.2 Configuration Implementation **49**
5.2.1 Datatype Widths 50
5.2.2 INT4 Constraints 51
5.2.3 INT16 Constraints 51
5.2.4 Modified Mvin and Accumulator Scale 51
5.3 Synthesis Flow **52**
5.3.1 PDK Choice: ASAP7 versus SKY130 52
5.3.2 Memory Macro Handling 53
5.3.3 Clock Targets 54
5.4 Performance Evaluation **55**
5.5 Metrics and Extraction **56**
6 EXPERIMENTS AND RESULTS **58**
6.1 Area **58**
6.2 Power **60**
6.3 Maximum Frequency **62**
6.4 Throughput **62**
6.5 Energy **64**
6.6 Accuracy and Cost Trade-off **65**
6.7 Discussion **68**
7 CONCLUSION **70**
7.1 Limitations **71**
7.2 Future Work **72**
7.3 Code and Data Availability **72**
REFERENCES **73**

APPENDIX A — SYNTHESIS ENVIRONMENT AND CONFIGURATION.....77
A.1 Tool and License Configuration.....77
A.2 ASAP7 Technology Configuration.....77

1 INTRODUCTION

Deep Neural Networks (DNNs) have revolutionized the way we approach tasks in computer vision, natural language processing, autonomous systems, and beyond, due to their powerful capacity to learn complex patterns from large volumes of data (LIU et al., 2017). However, this computational prowess comes at a cost: significant energy consumption. According to Jones (2018), data centers could consume up to 8% of global electricity demand by the end of this decade, raising concerns regarding sustainability and environmental impact.

These increasing computational demands of DNNs have driven the development of specialized hardware accelerators, aligning with a broader shift in computer architecture toward Domain-Specific Architectures (DSAs), computer architectures tailored to specific application domains that trade generality for performance and efficiency. DSAs, particularly when combined with open instruction sets like RISC-V, are a cornerstone of a new golden age for computer architecture, enabling agile, full-stack innovation (HENNESSY; PATTERSON, 2019).

Within this landscape, various DNN accelerator strategies have emerged, including fixed-function designs (SIJSTERMANS, 2018), programmable architectures (MOREAU et al., 2019), and automated generator frameworks targeting FPGAs or ASICs (ZHANG et al., 2018). The Gemmini Berkeley's Chipyard accelerator (GENC et al., 2021) exemplifies this trend as an open source, parameterizable systolic array generator integrated into a complete System-on-Chip (SoC) environment. Its full-stack design enables systematic Design Space Exploration (DSE), making it a compelling platform for evaluating hardware-software co-design trade-offs and DNN optimization strategies.

To achieve efficiency, hardware-aware optimization strategies, such as quantization, are essential for deploying DNNs on accelerators. Representing weights and activations with low bit-width integers, instead of 32-bit floating point, shrinks both the storage footprint and the arithmetic logic that dominates DNN workloads, and enables the underlying multiply-accumulate operations to be executed with integer-only arithmetic (JACOB et al., 2018; NAGEL et al., 2021). These gains, however, are only realized when the hardware is designed to exploit the chosen precision: the bit-width directly shapes the datapath, the accumulator, and the on-chip memories, and therefore the area, power, and timing of the resulting circuit.

This work presents a design space exploration on the Gemmini accelerator centered on integer precision. The accelerator's design is modified to generate designs that operate at INT4, INT8, and INT16. Additionally, for the INT4 design, the accumulator data width is explored, varying between 16 and 32 bits, yielding a set of architectural configurations that share a common base microarchitecture but differ in numerical precision and accumulator organization. Each configuration is processed through an ASIC logic synthesis flow to extract area, power, and timing estimates, and its throughput is evaluated through cycle-accurate simulation of ResNet-50. To enable a fair comparison, all configurations are synthesized under a common clock constraint for the area and power analysis, while each is also synthesized at its own maximum frequency for the throughput analysis. DNN accuracy is not measured directly; instead, we focus on the hardware costs reported here, which are related to the accuracy trends already established in the quantization literature, allowing the trade-offs of each precision to be discussed without reproducing experiments that are covered by prior work.

The remainder of this thesis is organized as follows. Chapter 2 reviews the background on sustainable computing, domain-specific architectures, systolic arrays, quantization, and the ASIC synthesis flow. Chapter 3 surveys related work on DNN accelerators and quantization-aware design and motivates the choice of Gemmini. Chapter 4 then describes the Gemmini accelerator in detail, focusing on the components affected by the precision configurations explored in this work. Chapter 5 presents the methodology, detailing the configurations, their hardware implications, and the synthesis and simulation setup. Chapter 6 reports and discusses the results, and Chapter 7 concludes the work and outlines its limitations and future directions.

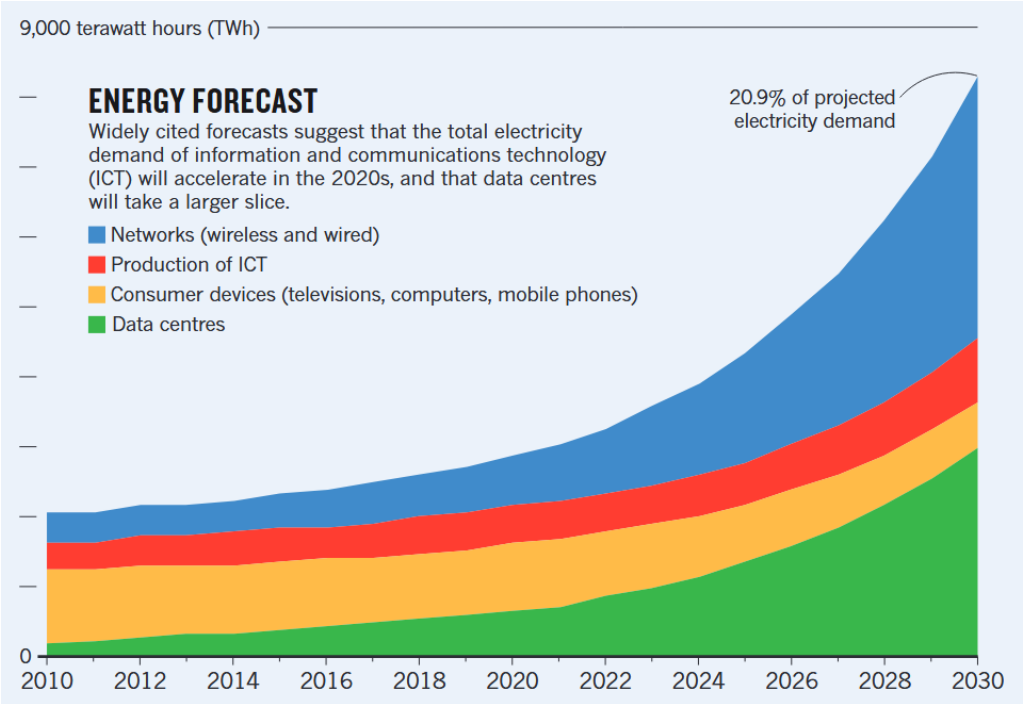
2 BACKGROUND

This chapter reviews the concepts and developments that underpin this work, progressing from the energy and sustainability pressures that motivate efficient deep-learning hardware, through the technological limits that drive architectural specialization, to the systolic arrays, quantization techniques, and ASIC design flow on which the Gemmini accelerator and our experiments rely.

2.1 Sustainable Computing and the Environmental Impact of AI

The accuracy improvements of modern neural networks depend heavily on exceptionally large computational resources, which in turn demand substantial energy. As the information and communications technology (ICT) ecosystem expands, data centers already consume on the order of 200 terawatt-hours (TWh) of electricity per year and worst-case projections suggest that data centers could account for more than 8% of global electricity demand by 2030 (JONES, 2018).

Figure 2.1: Energy Forecast.



Source: Adapted from (JONES, 2018)

The environmental cost of powering specialized hardware such as GPUs and TPUs for the days or weeks required to train large models is equally significant. Training

a single base model such as BERT on GPUs emits a carbon footprint comparable to a trans-American flight, while training a model through Neural Architecture Search (NAS) can require hundreds of thousands of GPU-hours and emit over 626,000 pounds of CO_2 equivalent—close to five times the lifetime emissions of an average American car (STRUBELL; GANESH; MCCALLUM, 2019).

Mitigating these environmental impacts requires progress on two complementary fronts: more computationally efficient algorithms, such as low-bit quantization (NAGEL et al., 2021); and low-power, domain-specific hardware accelerators that execute those algorithms efficiently. Shifting from massive, energy-hungry general-purpose processors toward specialized, scalable hardware is therefore central to keeping AI deployment both economically viable and environmentally sustainable.

The remainder of this chapter examines the technological pressures that motivate this shift and the architectural principles that underpin it.

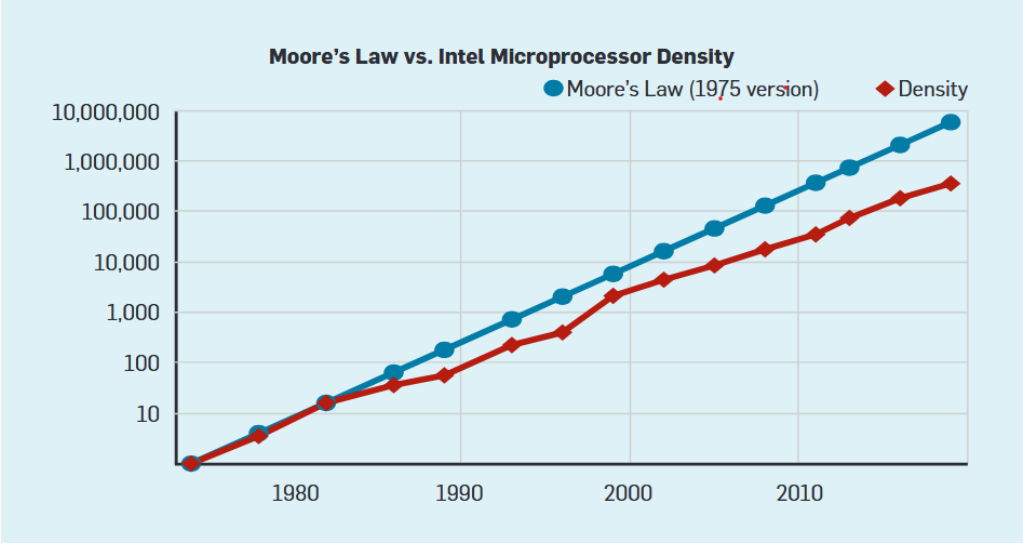
2.2 The Slowdown of Moore's Law and Dennard Scaling

For several decades, the sustained improvement of general-purpose processors rested on two complementary trends. The first, known as Moore's Law, is the empirical observation that the number of transistors that can be economically integrated on a chip doubles roughly every two years. The second, Dennard scaling, describes how transistor behavior improves as devices shrink: as linear dimensions are reduced by a given factor, supply voltage and current can be scaled down proportionally, keeping power density approximately constant while allowing clock frequency to increase (WESTE; HARRIS, 2011).

Together, these trends meant that each technology generation delivered more transistors, higher frequencies, and greater performance without a corresponding rise in power density. In recent generations, however, this cadence has faltered: transistor density still increases, but at a pace that increasingly trails the historical doubling rate, as the widening gap between the ideal projection and the measured density of Intel microprocessors in Figure 2.2 makes clear.

Dennard scaling, in turn, effectively ended in the mid-2000s. Supply voltage could no longer be reduced at the same pace, because lowering the transistor threshold voltage causes subthreshold leakage current to grow exponentially, making static (leakage) power a significant contributor alongside dynamic power. Since dynamic power is proportional

Figure 2.2: Transistors per chip of Intel microprocessors vs. Moore's Law.

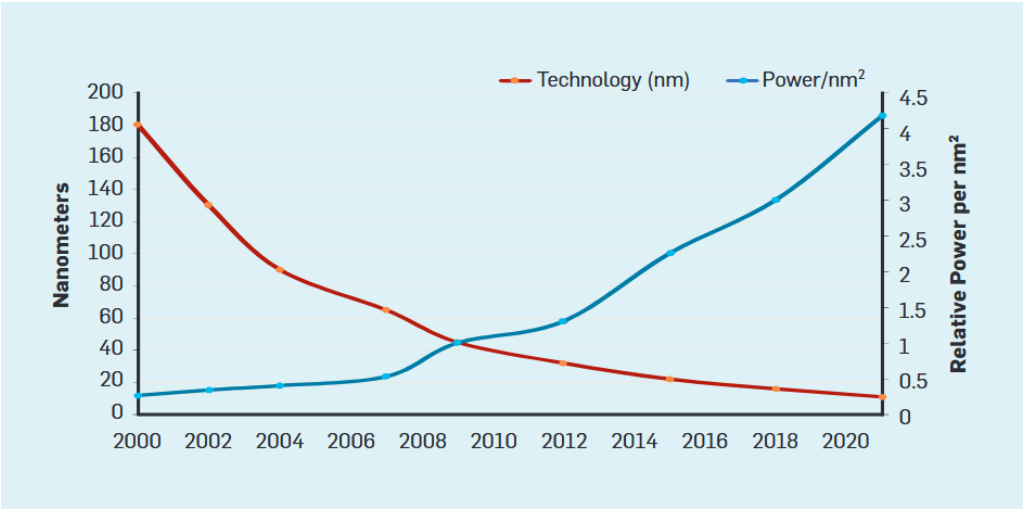


Source: Adapted from (HENNESSY; PATTERSON, 2019)

to the switched capacitance, the switching frequency, and the square of the supply voltage, a stalled supply voltage means that integrating more transistors and raising frequency now increases power density with every generation (HENNESSY; PATTERSON, 2019).

This effect is visible in Figure 2.3, where the relative power per unit area climbs sharply from the late 2000s onward even as the technology node continues to shrink. The result is the so-called power wall: clock frequencies plateaued, and power—rather than the number of available transistors—became the principal factor limiting further performance gains (HOROWITZ, 2014).

Figure 2.3: Transistors per chip and power per mm^2.



Source: Adapted from (HENNESSY; PATTERSON, 2019)

The immediate industry response was the shift to multicore processors, which use additional transistors to add cores rather than to raise frequency. However, multicore

processors are limited by the thermal design power (TDP), i.e., the average amount of power the package and cooling system can remove. These limitations led to the dark silicon phenomenon, whereby processor cores would reduce the clock rate and turn off idle cores to prevent overheating (HENNESSY; PATTERSON, 2019).

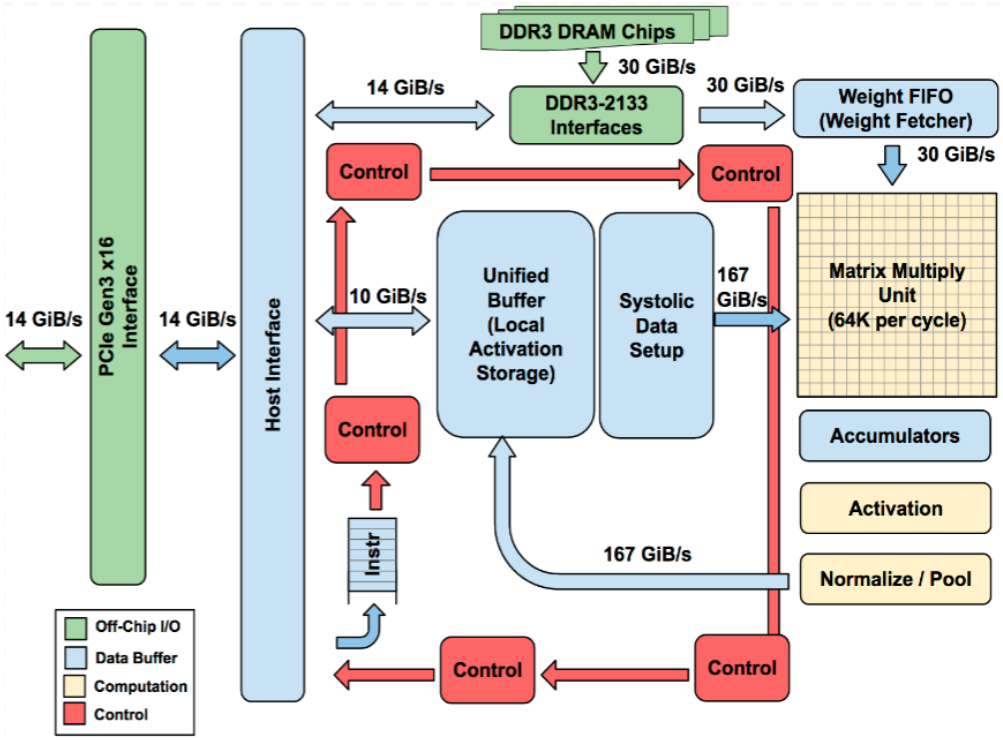
As the returns from transistor scaling diminish, energy efficiency, rather than die area, has become the most critical resource in processor design (HOROWITZ, 2014). This context motivates the turn toward hardware specialization: when generic scaling can no longer deliver efficiency, efficiency must instead come from architectures tailored to the workload (HENNESSY; PATTERSON, 2019).

2.3 Domain-Specific Architectures (DSAs)

With the slowdown of Moore's Law and the end of Dennard scaling, the computer architecture community has turned to Domain-Specific Architectures (DSAs) to sustain performance and efficiency improvements. DSAs are hardware accelerators designed for a narrow class of applications, trading the flexibility of general-purpose processors for substantial gains in performance per watt within their target domain (HENNESSY; PATTERSON, 2019).

These gains stem from design principles that a general-purpose processor cannot fully exploit. Rather than spending energy on speculative, out-of-order execution, DSAs employ simpler and more predictable forms of parallelism matched to the domain, such as the data-parallel and systolic structures used in matrix computation. They favor software-managed scratchpad memories over hardware-managed cache hierarchies, reducing the energy spent on data movement by keeping operands close to the compute units and reusing them deterministically. They also adopt the lowest numerical precision that the application can tolerate, since narrower data types shrink both the arithmetic logic and the memory footprint—a principle that is central to the present work. Finally, they are typically programmed through domain-specific abstractions that map applications onto the hardware with little overhead (HENNESSY; PATTERSON, 2019).

Figure 2.4: Functional organization of Google Tensor Processing Unit (TPU v1).



Source: Adapted from (JOUPPI et al., 2017)

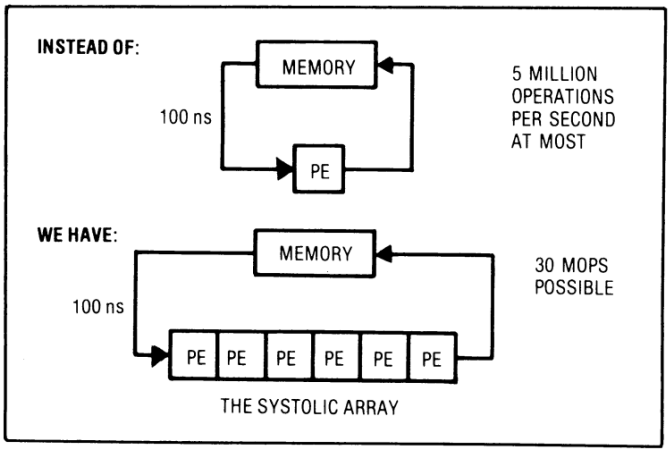
The Google Tensor Processing Unit (TPU) is a prominent example of this approach: its core is a large matrix-multiply unit built from a systolic array of multiply-accumulate cells operating on reduced-precision integers, whose functional organization is shown in Figure 2.4. This structure delivers throughput and energy efficiency far beyond contemporary CPUs and GPUs for neural-network inference (JOUPPI et al., 2017). The pairing of DSAs with open instruction sets such as RISC-V further amplifies their impact, enabling agile, full-stack innovation in which hardware, system software, and application are co-designed and evaluated together (HENNESSY; PATTERSON, 2019). The matrix-multiply engine at the heart of accelerators like the TPU—the systolic array—is the subject of the following section.

2.4 Systolic Arrays

Systolic arrays are a class of parallel architectures that perform computation through rhythmic, localized data movement. The term was coined by H. T. Kung by analogy with the circulatory system: just as the heart rhythmically pumps blood through the body, a systolic array pumps data through a network of processing elements in

synchronized steps (Kung, 1982). Introduced in the early 1980s for signal and image processing, the architecture saw renewed interest with the rise of deep learning, when Google's Tensor Processing Unit demonstrated that a systolic array could accelerate neural-network inference with throughput and energy efficiency far beyond CPUs and GPUs (XU et al., 2023; JOUPPI et al., 2017).

Figure 2.5: Basic principle of a systolic system.

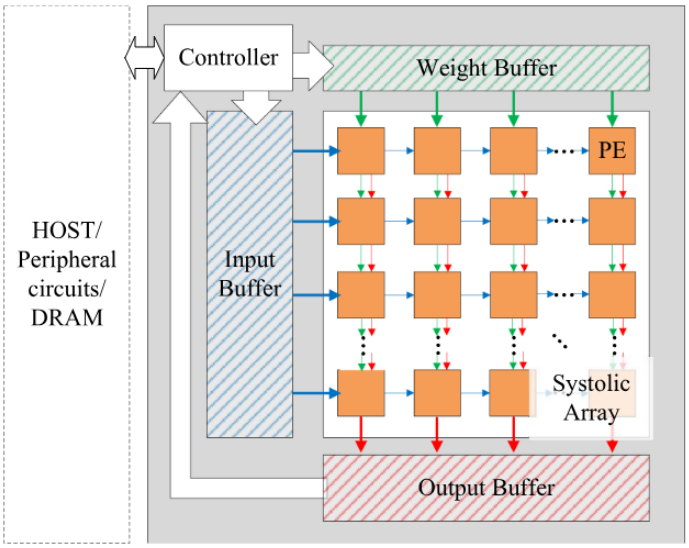


Source: Adapted from (Kung, 1982)

A systolic array consists of a regular grid of simple, homogeneous processing elements (PEs), together with a controller and on-chip buffers that feed data to the array and collect its results, as shown in Figure 2.6. Each PE performs a single multiply-accumulate (MAC) operation and communicates only with its immediate neighbors. Data flow synchronously through the grid in a wave-like fashion (Figure 2.5): an operand read once from the buffer is passed from PE to PE and reused at each step, rather than being fetched repeatedly from memory. Because communication with the buffers occurs only at the boundary of the array, a few I/O ports along one edge can keep an entire two-dimensional grid busy—n input ports can drive on the order of n^2 PEs (XU et al., 2023). Since off-chip memory accesses are far more energy-expensive than arithmetic operations (HOROWITZ, 2014), amortizing each memory access over many computations is what makes systolic arrays so energy-efficient.

Before turning to how these workloads map onto a systolic array, it is useful to fix the vocabulary of the layers involved. A convolutional (CONV) layer applies a set of learned *filters* (or *kernels*) to an input *feature map*: each filter is a small volume of weights spanning a spatial extent (e.g., 3×3) and the full *input-channel* depth C_{in} of the layer, and one filter produces one *output channel* by sliding across the input's spatial dimensions and computing a weighted sum at every position. A layer with C_{out} filters therefore produces

Figure 2.6: Typical structure of a two-dimensional systolic array accelerator.



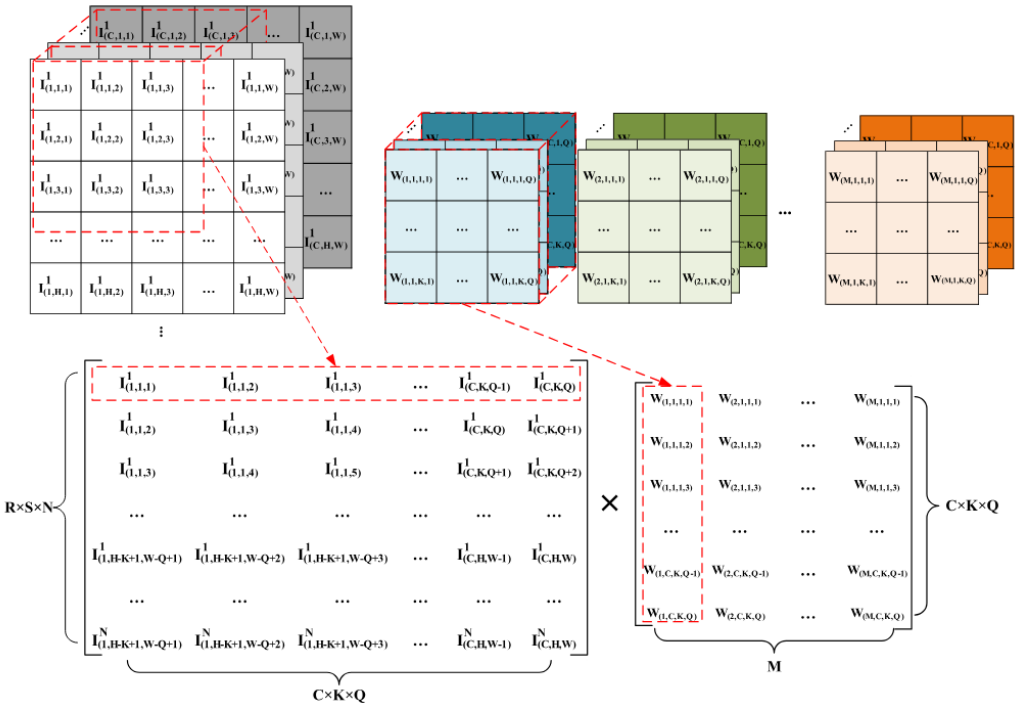
Source: Adapted from (XU et al., 2023)

C_{out} output channels, and every output value is a reduction over C_{in} input channels and the filter's spatial extent. A fully connected (FC) layer is the degenerate case in which the spatial extent collapses to a single point, so every output is a weighted sum over the entire input vector.

The workloads that dominate modern DNNs map naturally onto the systolic array from this description. Both FC and CONV layers reduce to matrix multiplication: an FC layer is itself a matrix multiplication between its weights and input activations, while the *im2col* transformation recasts a convolution into one large matrix multiplication by unrolling every receptive field — every spatial position at which a filter is applied, together with its full C_{in}-channel depth — into one column of a matrix, and by flattening each of the C_{out} filters into one row of a second matrix, as illustrated in Figure 2.7 (XU et al., 2023). The contraction (inner) dimension of this multiplication is therefore the size of one filter (C_{in} times its spatial extent), and the output dimension is C_{out}. This is the concrete link between the abstract convolution and the array: once im2col has expressed a CONV layer this way, the array reduces along its columns over exactly this contraction dimension, in the same way it would for any matrix product, and Chapter 4 shows how Gemmini executes this reduction physically across its mesh (Equation 4.1). Because the two-dimensional systolic array is, at its core, a matrix-multiplication engine, both layer types can be executed on the same hardware simply by streaming the appropriate matrices through the array.

Concretely, for a matrix product $C = A \times B$, the elements of A are streamed in

Figure 2.7: The im2col transformation, which recasts a convolution as a matrix multiplication.



Source: Adapted from (XU et al., 2023)

from one edge of the array while the elements of B enter from an orthogonal edge; each PE multiplies the operands that meet within it and accumulates a running partial sum for one element of C. Since every input value travels across an entire row or column, a single operand loaded into the array contributes to many MAC operations, maximizing reuse and minimizing the memory bandwidth required.

Systolic arrays differ in their *dataflow*, which determines which tensor remains stationary inside each PE and which ones move through the array (XU et al., 2023). In a weight-stationary (WS) dataflow, the weights are preloaded and held in the PEs while activations stream through and partial sums propagate along the columns; in an output-stationary (OS) dataflow, each PE accumulates one output element in place while both inputs flow past; an input-stationary (IS) dataflow instead pins the activations. These configurations change which data is reused, and therefore affect the energy and performance of a given workload, and they imply slightly different PE structures, as shown in Figure 2.8.

In practice, the dimensions of a DNN layer far exceed those of the physical array along every axis: a single ResNet-50 convolution can have hundreds of input and output channels, whereas the array used in this work is only 16×16 PEs (Chapter 4). The computation cannot be unrolled in a single pass and is instead partitioned into *tiles* sized to

Figure 2.8: Processing element structures for the output-stationary and weight-stationary dataflows.



(a) OS dataflow PE (b) WS dataflow PE

Source: Adapted from (XU et al., 2023)

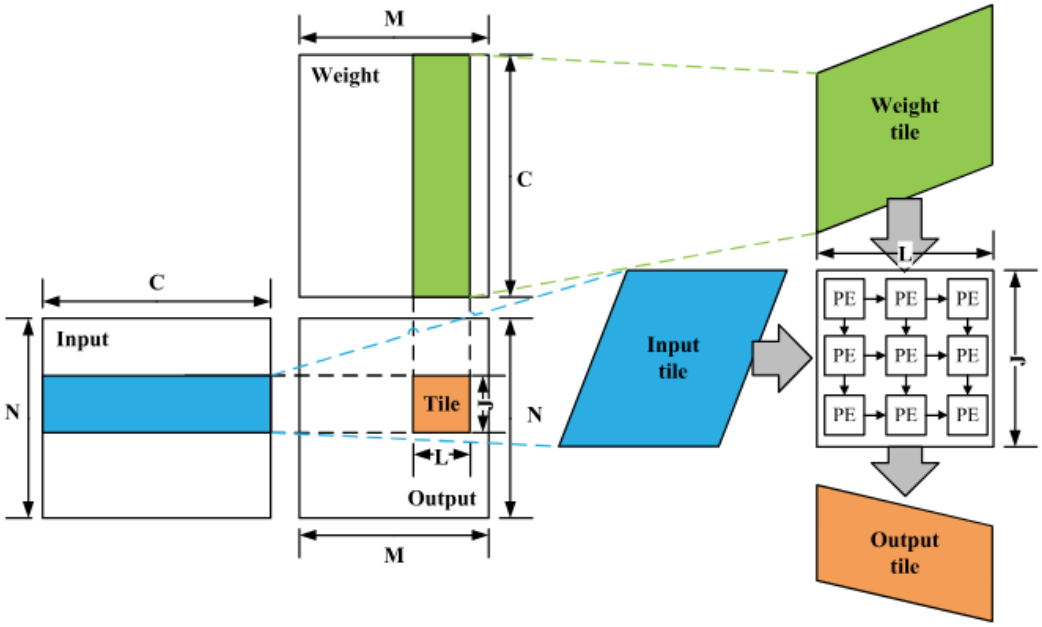
fit the array along each of the three matrix-multiplication dimensions: the output-channel dimension (C_{out}), the spatial/batch dimension of the im2col-transformed activations, and the contraction dimension (C_{in} times the kernel extent), the last of which is additionally bounded by how many rows the on-chip scratchpad and accumulator can hold. Each tile is streamed through the array in one pass and executed serially against the others, with its partial sum accumulated on-chip across passes along the contraction dimension until the corresponding output tile is complete, as illustrated for a generic systolic accelerator in Figure 2.9 (XU et al., 2023). This tiling is orchestrated by the accelerator's control logic and software stack and is central to how a fixed-size array such as Gemmini's processes networks of arbitrary size; Chapter 4 describes the concrete `mvin`/`mvout` instructions that move each tile on- and off-chip, and Section 6.4 shows that this tiling, not the numerical precision, is what determines Gemmini's cycle count for a given configuration.

The simple, regular, and locally connected structure of the systolic array makes it inexpensive to design and verify, straightforward to implement in VLSI, and easy to scale to larger dimensions, with cost growing roughly in proportion to the performance required (XU et al., 2023). These properties make it a canonical domain-specific architecture and explain its adoption in accelerators ranging from Google's TPU (JOUPPI et al., 2017) to the open-source Gemmini platform (GENC et al., 2021).

2.5 Deep Neural Network Quantization

Quantization is one of the most effective techniques for reducing the computational time and energy consumption of DNN inference (NAGEL et al., 2021). It replaces the 32-bit floating-point (FP32) representation of weights and activations with lower bit-width fixed-point integers, such as 16-bit, 8-bit, or 4-bit values. The hardware benefits are

Figure 2.9: Partitioning a workload into tiles for execution on a fixed-size systolic array.



Source: Adapted from (XU et al., 2023)

twofold: reducing the bit-width from 32 to 8 shrinks the storage footprint of every tensor by a factor of four, and, because the cost of a multiplier grows roughly with the product of its operand widths, the area and energy of each MAC fall even more steeply. These savings compound across the billions of MAC operations in a modern network.

The deeper motivation, however, is data movement. In modern technology nodes, moving data dominates the energy budget: accessing off-chip DRAM can cost up to three orders of magnitude more energy than a single arithmetic operation (HOROWITZ, 2014). By shrinking each operand, low-bit quantization reduces the volume of data transferred and increases the effective capacity of on-chip memories, allowing a larger share of the working set to remain on-chip and thereby mitigating the memory wall.

The mathematical basis of most hardware-friendly schemes is uniform affine quantization (JACOB et al., 2018; NAGEL et al., 2021). A real value x confined to a range $[\alpha, \beta]$ is mapped onto a grid of 2^b evenly spaced levels, for a target bit-width b, through a floating-point scale factor s and an integer zero-point z:

$$s = \frac{\beta - \alpha}{2^b - 1}, \qquad z = \operatorname{round}\left(-\frac{\alpha}{s}\right). \tag{2.1}$$

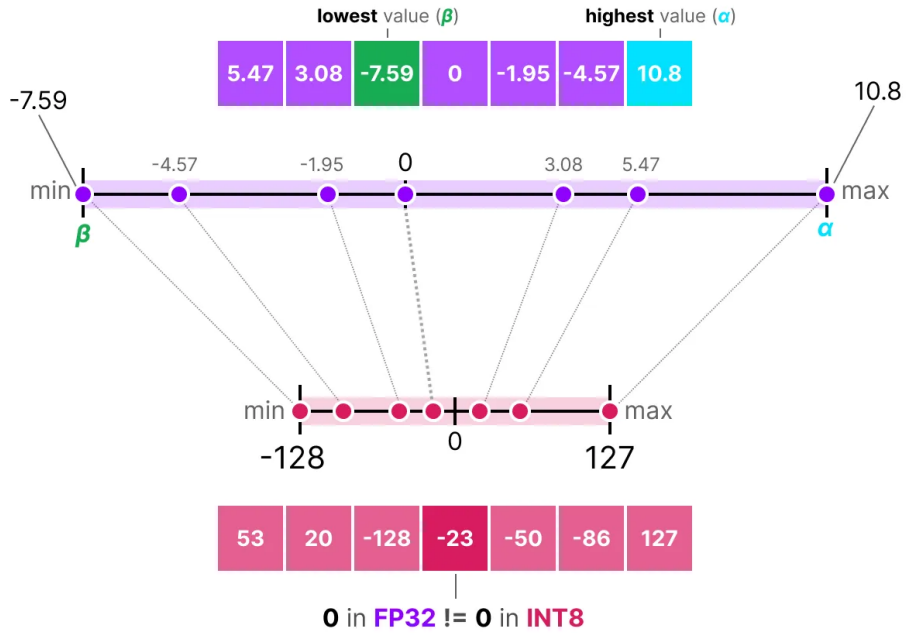
The scale s is the spacing between adjacent representable values—the quantization step—while the zero-point z is the integer onto which the real value zero maps exactly. Representing zero without error matters because it appears pervasively in padded and rectified

(ReLU) tensors. Quantization and its inverse, dequantization, are then

$$x_{\text{int}} = \text{clamp}\big(\,\text{round}(x/s) + z,\ 0,\ 2^b - 1\big), \qquad \hat{x} = s\,(x_{\text{int}} - z), \tag{2.2}$$

where clamp saturates values that fall outside the representable range and $\hat{x}$ denotes the recovered approximation of x. Figure 2.10 illustrates this mapping for an offset range, where the shifted float interval forces a nonzero zero-point.

Figure 2.10: Asymmetric quantization maps an offset float range, so the real zero maps to a nonzero integer zero-point ($z \neq 0$).



Source: Adapted from (GROOTENDORST, 2024)

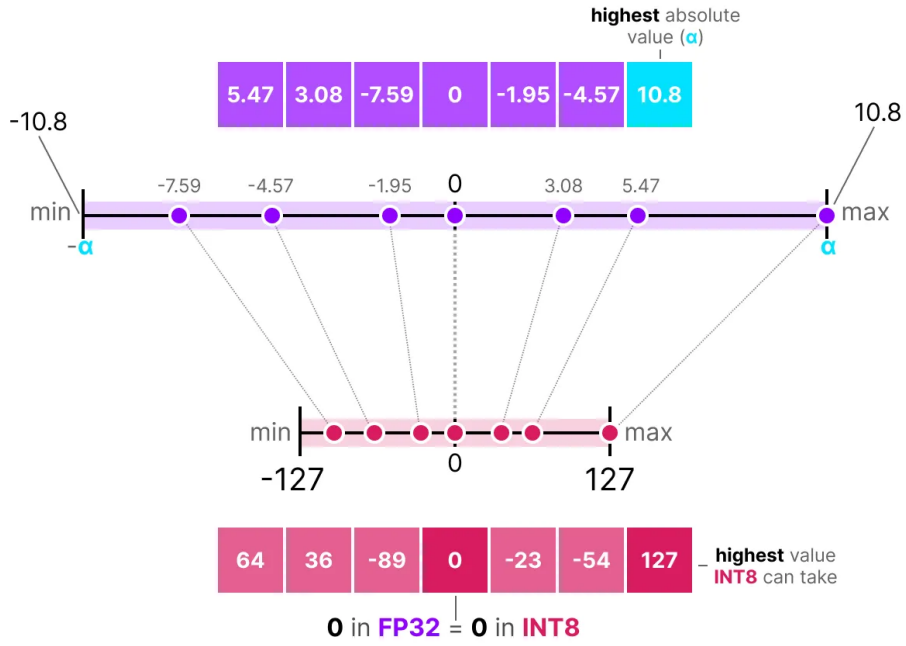
When the data is distributed symmetrically around zero, as is typical for weights, the zero-point is fixed at $z = 0$, yielding symmetric quantization (NAGEL et al., 2021). For a signed b-bit integer this reduces the mapping to

$$s = \frac{\max(|x|)}{2^{b-1} - 1}, \qquad x_{\text{int}} = \text{clamp}\big(\,\text{round}(x/s),\ -2^{b-1},\ 2^{b-1} - 1\big), \tag{2.3}$$

which eliminates the zero-point offset from the arithmetic and is therefore cheaper to implement in hardware.

Figure 2.11 illustrates this symmetric case, in which the real zero coincides with the integer zero. The associated rounding error is bounded by half a step, $|x - \hat{x}| \leq s/2$, so for a fixed range each additional bit halves both the quantization step and the worst-case error. This is the precise sense in which more bits buy more accuracy.

Figure 2.11: Symmetric quantization maps the float range onto a grid centered at zero, so the real zero maps exactly to the integer zero ($z = 0$).



Source: Adapted from (GROOTENDORST, 2024)

The reason the scale factor is central to a systolic accelerator becomes clear once quantized values enter a matrix multiplication. Using symmetric quantization for an activation x and a weight w, the dot product underlying each output element is

$$y = \sum_i x_i\, w_i \;\approx\; \sum_i (s_x\, x_{\text{int},i})(s_w\, w_{\text{int},i}) \;=\; s_x s_w \sum_i x_{\text{int},i}\, w_{\text{int},i}. \tag{2.4}$$

The array therefore performs a purely integer accumulation, $\sum_i x_{\text{int},i}\, w_{\text{int},i}$, and the real-valued result is recovered afterwards by multiplying by the combined scale $s_x s_w$. Two hardware consequences follow directly from Equation 2.4. First, the accumulator must be wide enough to hold this growing integer sum without overflow, which is why accumulators are provisioned with substantially more bits than the operands—a sizing question examined in detail in Chapter 4. Second, before the result can be stored as a low-bit activation it must be *requantized*: the accumulated integer is multiplied by a scaling factor $M = s_x s_w / s_y$, where s_y is the output scale, and then rounded and clamped to the output bit-width. In integer-only hardware, this multiplication by the typically sub-unitary factor M is realized as a fixed-point multiply followed by a bit shift (JACOB et al., 2018). Gemmini implements precisely this pattern through two scaling stages—one applied as data is moved into the array and one applied to the accumulated result on the way out—both of which are described and modified in this work.

The accumulator's bit-width is therefore not only a hardware cost but an accuracy-relevant design choice in its own right, distinct from the operand precision discussed above. Narrowing `accType` shrinks the accumulator's read–modify–write logic and its storage (Chapter 4), but it also shrinks the headroom available for the running sum of Equation 2.4: if the accumulator is too narrow for the contraction depth K of a given layer, the partial sum must saturate before the reduction completes, an accuracy-loss mechanism that is independent of the operand bit-width b and that the affine-quantization error bound derived above does not capture. This tension between accumulator width and cross-tile accumulation depth is why this work scopes its accuracy discussion the way it does for the narrow-accumulator configuration explored in Section 5.2.2.

Quantization unavoidably introduces clipping and rounding noise, and the resulting accuracy loss grows as the bit-width shrinks. Two families of techniques are used to contain it:

- **Post-Training Quantization (PTQ):** A lightweight, "push-button" approach that converts a pre-trained FP32 network directly into a fixed-point one without end-to-end retraining (NAGEL et al., 2021). It is fast and requires only a small calibration dataset, but can suffer noticeable accuracy drops when bit-widths are pushed to 4 bits or below.
- **Quantization-Aware Training (QAT):** A more robust method that models quantization noise during training by inserting simulated quantizer blocks into the forward pass, so the model learns to adapt to the rounding and clipping errors (NAGEL et al., 2021). QAT generally closes the accuracy gap to the full-precision baseline, enabling aggressive quantization (e.g., 4-bit weights and activations) with limited accuracy loss.

Beyond these two families, a range of complementary techniques narrows the PTQ accuracy gap without full retraining—for example, correcting the systematic bias that quantization induces in a layer's statistics (FINKELSTEIN; ALMOG; GROBMAN, 2019) or factorizing the per-channel scales of consecutive layers (MELLER et al., 2019)—while at the extreme end of the spectrum, binary networks push both weights and activations to a single bit (HUBARA et al., 2016).

The viable precision in practice depends strongly on the technique and the workload. For convolutional networks, 8-bit PTQ generally preserves accuracy within a fraction of a percent of the FP32 baseline, which has made INT8 the de facto standard for

inference (JACOB et al., 2018; NAGEL et al., 2021). Pushing to 4 bits substantially increases quantization noise, and PTQ alone can incur a significant accuracy drop; recovering near-baseline accuracy at this width typically requires QAT (NAGEL et al., 2021).

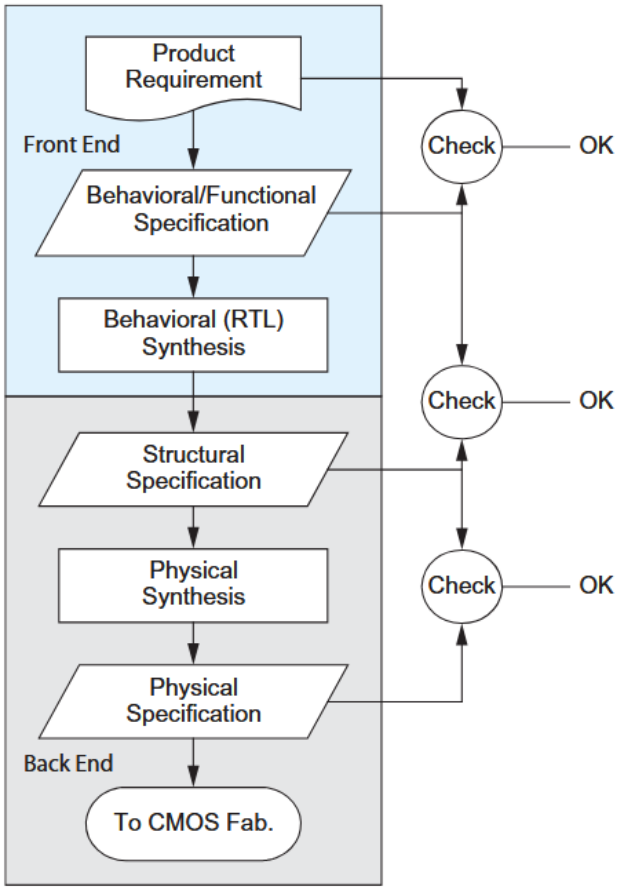
At the opposite end, 16-bit integers yield accuracy essentially indistinguishable from FP32 for these workloads, but at a markedly higher hardware cost: quantization-aware accelerator studies report performance-per-area and energy differences of several-fold across bit precisions, with lower-precision processing elements far outperforming a 16-bit integer baseline at comparable accuracy (INCI et al., 2023). These accuracy characteristics are well established in the literature; accordingly, this work does not re-measure them, focusing instead on the hardware cost of supporting each precision and relating that cost to the accuracy trends summarized above.

2.6 ASIC Design and Logic Synthesis Flow

The realization of a digital integrated circuit from a behavioral description follows a sequence of refinement steps known as the ASIC design flow, which is conventionally partitioned into a *front end* and a *back end* (WESTE; HARRIS, 2011). The front end captures the design at the register-transfer level (RTL) in a hardware description language (HDL) such as Verilog or VHDL and transforms it, through *logic synthesis*, into a gate-level netlist of standard cells. The back end then carries this netlist through physical synthesis—floorplanning, placement, clock-tree synthesis, and routing—to produce the manufacturable layout. This partitioning is what enables the ASIC model in which a design house develops and verifies the RTL while a largely automated downstream flow completes the transition to silicon (WESTE; HARRIS, 2011). The experiments in this work are conducted entirely in the front end, at the logic-synthesis stage, which already yields technology-accurate estimates of area, timing, and power and is therefore sufficient to compare architectural configurations without paying the cost of full physical design.

Logic synthesis consists of a sequence of steps (WESTE; HARRIS, 2011), and it proceeds in three conceptual stages. First, the RTL is *elaborated* into a technology-independent representation of generic gates and registers. Second, this representation is *optimized* through Boolean and structural transformations, together with datapath and finite-state-machine optimizations, to improve speed, area, and power. Third, the generic logic is *mapped* onto the cells of a standard-cell library, and the mapped netlist is refined further by sizing gates and inserting buffers so that the imposed constraints are met. The

Figure 2.12: Generalized ASIC design flow, partitioned into front end and back end.



Source: Adapted from (WESTE; HARRIS, 2011)

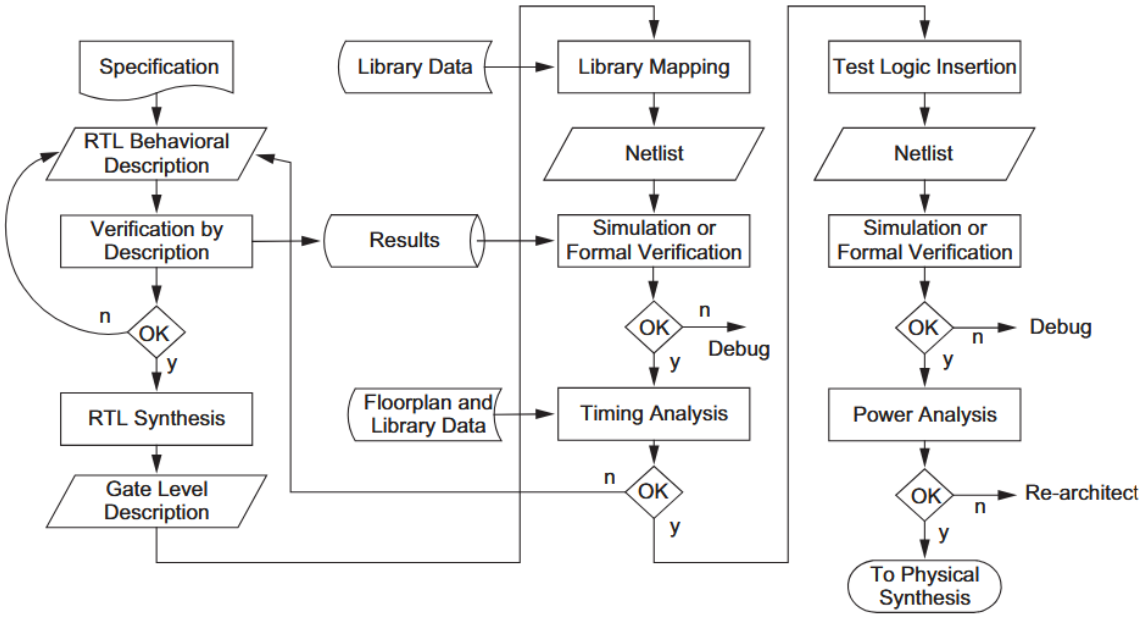
same Boolean function can be realized by many different cell networks with different costs, so throughout these stages the tool searches that space to minimize a cost function that trades off timing, area, and power (WESTE; HARRIS, 2011).

Three classes of input drive this process:

- the **RTL description**, the behavioral source to be synthesized;
- the **standard-cell library**, consulted in two complementary views: a logical and timing view (the Liberty `.lib` format), which gives each cell's Boolean function together with its delay and power characterized as functions of the input transition time and the output load, and a physical view (the Library Exchange Format, `.lef`), which gives cell dimensions and pin locations. The logical view governs synthesis and analysis, whereas the physical view is required for back-end placement and routing (WESTE; HARRIS, 2011);
- the **design constraints**, written in a Synopsys Design Constraints (SDC) file, which specify the target clock period, the input and output delays, the output loads, and any false or multicycle paths.

Typical synthesis tools include Cadence Genus and Synopsys Design Compiler (WESTE; HARRIS, 2011).

Figure 2.13: Logic-synthesis flow: the RTL, the standard-cell library, and the timing constraints are combined to produce a gate-level netlist, from which area, timing, and power are reported.



Source: Adapted from (WESTE; HARRIS, 2011)

Once the gate-level netlist exists, the synthesis tool reports the three quantities that this work uses as its cost metrics.

a) **Area.** The area is obtained by summing the layout area of every instantiated cell, taken from the library, and separates naturally into the standard-cell logic and any instantiated hard macros.

b) **Timing.** Timing is verified by *static timing analysis* (STA), which exhaustively evaluates every timing path in the design without requiring input vectors (WESTE; HARRIS, 2011). Using the cell delays from the library together with estimated interconnect loads, STA computes, for each path between registers, the signal arrival time and the time required to meet the clock, and reports their difference as *slack*. It checks both maximum-delay (setup) and minimum-delay (hold) constraints; the path with the worst slack is the critical path, and its delay sets the maximum frequency at which the design can operate. Because STA follows every structural path, it can also flag paths that are never exercised in practice—*false paths*—which the designer must explicitly exclude (WESTE; HARRIS, 2011).

c) **Power.** Power is estimated from the netlist by combining the per-cell power models

in the library with an estimate of how often each node switches. This switching activity is the key input: it depends on the data the circuit processes (WESTE; HARRIS, 2011) and can be supplied either from a simulation of representative stimuli (a vector-based analysis driven by a recorded activity file) or, when no stimuli are available, by propagating statistical activity assumptions from the primary inputs through the netlist (a *vectorless* analysis). A power-analysis engine—such as the Cadence Joules engine used in this work—combines either activity source with the library models to produce average-power estimates directly from the synthesized netlist. The components of power dissipation themselves are reviewed in Section 2.7, and the specific vectorless setup adopted here is detailed in Chapter 5.

Not all components are built from standard cells. Large on-chip memories, such as the scratchpad and accumulator SRAMs of an accelerator, are far denser and more efficient when generated as dedicated hard macros by a memory compiler and characterized separately, rather than synthesized from individual flip-flops. The synthesis tool then instantiates these memories as pre-characterized blocks placed among the standard-cell rows. As discussed in later chapters, the way these memory macros are generated and accounted for has a direct and substantial effect on the reports, a point that proves central to the methodology of this work.

2.7 Low-Power Hardware Design Foundations

The total power dissipated by a CMOS circuit has two contributions: *dynamic* power, consumed when the circuit switches, and *static* power, consumed continuously regardless of switching activity (WESTE; HARRIS, 2011):

$$P_{\text{total}} = P_{\text{dyn}} + P_{\text{stat}}, \qquad P_{\text{dyn}} = P_{\text{sw}} + P_{\text{sc}}, \tag{2.5}$$

where the dynamic term is itself the sum of a switching component P_{sw} and a short-circuit component P_{sc}.

The switching component dominates the dynamic power and arises from charging and discharging the load capacitances each time a node toggles; for a circuit operating at frequency f it is approximately

$$P_{\text{sw}} \approx \alpha\, C_L\, V_{\text{DD}}^2\, f, \tag{2.6}$$

where α is the activity factor (the average fraction of cycles in which a node toggles), C_L the switched capacitance, and V_{DD} the supply voltage (WESTE; HARRIS, 2011). The short-circuit component P_{sc}, dissipated during the brief interval in each transition when both the pull-up and pull-down networks conduct simultaneously, is comparatively small in well-designed gates and is usually a minor fraction of the dynamic total (WESTE; HARRIS, 2011).

Static power is dissipated even when the circuit is idle and the clock is stopped. It is the product of the supply voltage and the total leakage current,

$$P_{\mathrm{stat}} = I_{\mathrm{leak}}\, V_{\mathrm{DD}}, \tag{2.7}$$

where I_{leak} is dominated by *subthreshold leakage*—the current that flows through a transistor that is nominally off—with smaller contributions from gate-oxide and reverse-biased junction leakage (ROY; MUKHOPADHYAY; MAHMOODI-MEIMAND, 2003; WESTE; HARRIS, 2011). The subthreshold current depends exponentially on the gate overdrive,

$$I_{\mathrm{sub}} = I_0\, e^{\frac{V_{\mathrm{GS}}-V_T}{n\,V_\theta}} \left(1 - e^{-\frac{V_{\mathrm{DS}}}{V_\theta}}\right), \tag{2.8}$$

where $V_\theta = kT/q$ is the thermal voltage and n the subthreshold slope factor (ROY; MUKHOPADHYAY; MAHMOODI-MEIMAND, 2003). Because of this exponential dependence, subthreshold leakage rises sharply as the threshold voltage V_T is lowered—precisely what the aggressive supply-voltage scaling of past technology generations demanded. As a consequence, static power has grown from a negligible quantity in older nodes into a substantial share of the total in modern technologies, accumulating with every transistor integrated (KIM et al., 2003; HOROWITZ, 2014). Because leakage is paid continuously and grows with circuit area, reducing the size of a design—for instance, by narrowing its datapath through quantization—lowers static as well as dynamic power.

Two design techniques target these components directly:

- **Clock gating** reduces dynamic power by disabling the clock to registers or logic blocks that are momentarily idle, halting the switching activity they would otherwise propagate (WESTE; HARRIS, 2011).
- **Power gating** reduces static leakage by disconnecting the supply from inactive blocks through high-V_t header or footer switches during sleep modes (ROY; MUKHOPADHYAY; MAHMOODI-MEIMAND, 2003).

The dependence of switching power on frequency in Equation 2.6 carries a direct methodological consequence. Because P_{sw} scales linearly with f, comparing the power of two designs running at different frequencies conflates architectural differences with frequency differences. Synthesizing every configuration under a single, common clock constraint isolates the effect of the architecture itself, which is why the area and power comparisons in this work are all performed at a fixed clock period (Chapter 5). Gemmini additionally exposes an optional clock-gating feature, allowing this technique to be evaluated at the architectural level.

3 RELATED WORK

This chapter reviews the literature that frames the present work and motivates its central platform choice. It begins by surveying the broader landscape of DNN accelerators, establishing why Gemmini provides the groundwork for a precision-focused design space exploration. It then organizes the most relevant prior work into three groups: design-space exploration carried out on Gemmini itself, frameworks that jointly explore quantization and accelerator design, and hardware proposals for multi-precision arithmetic. The chapter closes by positioning this work with respect to those efforts, setting up the detailed treatment of the Gemmini architecture in Chapter 4.

3.1 Deep Neural Network Accelerators Landscape

The landscape of DNN accelerators is diverse, encompassing fixed-function ASIC, programmable architectures, and automated generation frameworks. Early efforts often resulted in hand-designed accelerators optimized for specific workloads. More recently, open-source hardware designs like NVIDIA's NVDLA (SIJSTERMANS, 2018) have emerged, providing a concrete, parameterizable DSA for CNN inference, enabling wider adoption and experimentation. Programmable accelerators like VTA (MOREAU et al., 2019) offer flexibility through specialized instruction sets and compilers.

However, the rapid evolution of DNN models and the need to deploy them across a wide range of platforms and performance targets necessitate approaches that facilitate rapid prototyping and systematic exploration of the hardware design space. This has fueled the development of automated accelerator generation platforms. FPGA-oriented tools such as DNNWeaver (SHARMA et al., 2016), DNNBuilder (ZHANG et al., 2018), and PolySA (CONG; WANG, 2018) generate high-performance accelerators or systolic-array mappings from high-level descriptions, but target a specific implementation rather than broad architectural exploration. MAGNet (VENKATESAN et al., 2019) represents a modular accelerator generator framework that produces synthesizable RTL for ASIC targets from parameterizable templates.

Amidst this landscape, Gemmini distinguishes itself from these generators by pairing generation with integration. From a flexible template supporting various data types and dataflows, it produces core accelerator IP for both ASIC and FPGA flows (PECCIA et al., 2024) and embeds it in a complete System-on-Chip (SoC), rather than emitting a

standalone FPGA implementation or context-free ASIC RTL, as compared in Figure 3.1.

Figure 3.1: Comparison of DNN accelerator generators.

	Property	NVDLA	VTA	PolySA	DNNBuilder	MAGNet	DNNWeaver	MAERI	Gemmini
Hardware Architecture Template	Datatypes	Int/Float	Int	Int	Int	Int	Int	Int	Int/Float
	Dataflows	✗	✗	✓	✓	✓	✓	✓	✓
	Spatial Array	vector	vector	systolic	systolic	vector	vector	vector	vector/systolic
	Direct Convolution	✓	✗	✗	✓	✓	✓	✓	✓
Programming Support	Software Ecosystem	Compiler	TVM	SDAccel	Caffe	C	Caffe	Custom	ONNX/C
	Virtual Memory	✗	✗	✗	✗	✗	✗	✗	✓
System Support	Full SoC	✗	✗	✗	✗	✗	✗	✗	✓
	OS Support	✓	✓	✗	✗	✗	✗	✗	✓

Source: Adapted from (GENC et al., 2021)

3.2 Design-Space Exploration on Gemmini

The closest neighbor to this work is the Gemmini case study of Gookyi et al., who sweep several hardware parameters (array size, on-chip memory capacity, the choice of dataflow, and the presence of a hardware im2col unit) and report their effect on area, frequency, and power (GOOKYI et al., 2023). Their results quantify, for instance, that the weight-stationary dataflow is about three times faster than the output-stationary one and that doubling the array size raises both area and power by roughly $3.3\times$. That study, however, is conducted on an FPGA, so its cost metrics are resource utilization and FPGA frequency rather than synthesized ASIC area and power.

Other studies use Gemmini as an evaluation target while optimizing the software side of the stack. Chen et al. (2022) derive new lower bounds on data movement for mixed-precision convolutions and propose tiling algorithms that outperform the conventional im2col approach, demonstrating improvements between 13% and 150% over a vendor-supplied algorithm on Gemmini. Peccia and Bringmann (2024) integrate Gemmini into the TVM compiler, providing a generic schedule to offload general matrix multiplication onto the accelerator and applying automatic kernel tuning to it. These efforts improve how workloads are mapped onto a fixed accelerator, whereas the present work varies the accelerator's numerical configuration and measures the resulting hardware cost.

3.3 Quantization-Aware Accelerator Co-Exploration

A second line of research explores quantization and accelerator design jointly. The framework most directly comparable to this work is QUIDAM, a highly parameter-

ized quantization-aware co-exploration framework that spans bit precision, processing-element type, buffer sizes, and the number of processing elements (INCI et al., 2023). QUIDAM reports that performance-per-area and energy vary by more than $5\times$ and $35\times$ across these choices, and that lightweight, lower-precision processing elements can match the accuracy of a 16-bit integer baseline while delivering up to $5.7\times$ better performance-per-area and energy. Crucially, QUIDAM attains its speed by relying on *pre-characterized* power, performance, and area models, which lets it explore designs three to four orders of magnitude faster precisely because it removes the need to synthesize and characterize each design individually. The present work occupies the opposite end of this trade-off: it sacrifices breadth of exploration to obtain the actual cost of each configuration through a real synthesis flow.

Bit Fusion approaches the precision question by making the hardware itself bit-flexible: it is composed of bit-level processing elements that dynamically fuse to match the bitwidth required by each individual DNN layer, minimizing computation and communication at fine granularity with no loss in accuracy (SHARMA et al., 2018). Implemented in Verilog and synthesized in a 45 nm technology, it reports sizable speedups and energy savings over fixed-bitwidth accelerators such as Eyeriss and Stripes. Bit Fusion is thus a bespoke precision-flexible architecture rather than a study of the cost of supporting different fixed precisions on a general-purpose accelerator.

3.4 Multi-Precision Arithmetic Hardware

A third group proposes new arithmetic units to support multiple precisions efficiently. Dai et al. introduce a bit-split-and-combination systolic accelerator for multi-precision CNNs produced by neural architecture search, combining a reconfigurable multiply-accumulate unit with a multi-precision dataflow; described in Chisel and synthesized in 28 nm, it reports up to $2.94\times$ area efficiency and $6.38\times$ energy savings over designs including Gemmini, Bit Fusion, and a bit-serial baseline (DAI et al., 2022). Ashar et al. propose QuantMAC, a quantize-enabled multiply-accumulate unit that uses a shift-and-add formulation with runtime bit-truncation to support variable precision, reporting roughly 37% area and 45% power reductions over a state-of-the-art MAC baseline on an FPGA target (ASHAR et al., 2024). Taking precision reduction to its limit, Li et al. present LUT-DLA, which uses vector quantization to convert network layers into look-up tables, achieving extreme low-bit inference; through a co-design

space exploration it reports power- and area-efficiency gains of 1.4–$7.0\times$ and 1.5–$146.1\times$ respectively, at a modest accuracy cost (LI et al., 2025).

These works contribute novel datapath primitives whose benefits are typically demonstrated on custom hardware. They establish that precision is a powerful lever for efficiency, but they do not quantify the cost of precision within a standard, full-stack accelerator obtained through a conventional synthesis flow.

3.5 Positioning of This Work

Across these three groups, two gaps emerge. Studies that explore Gemmini concentrate on array size, memory, dataflow, and mapping rather than on datapath precision, and the most directly comparable one reports FPGA resources rather than ASIC area and power. Studies that do center on precision either estimate cost with pre-characterized analytical models, as in (INCI et al., 2023), or propose bespoke precision-flexible datapaths evaluated in isolation, as in Bit Fusion (SHARMA et al., 2018), QuantMAC (ASHAR et al., 2024), and LUT-DLA (LI et al., 2025).

This work addresses these gaps through a real logic-synthesis flow with a modern technology library, characterizing the hardware cost of supporting concrete fixed precisions on a widely used, open-source, full-stack accelerator, broken down by the components that the precision actually affects. Rather than proposing a new precision-flexible unit or estimating cost from a model, it synthesizes concrete INT4, INT8, and INT16 configurations of Gemmini and reports their area, timing, and power, separating the contribution of the datapath from that of the on-chip memories.

4 THE GEMMINI ACCELERATOR

This chapter describes the Gemmini accelerator in the detail required to understand the design space explored in this work. After introducing how Gemmini is integrated into a complete system, it focuses on the components that the precision configurations directly affect: the spatial array, the numerical datatypes and their bit-widths, the on-chip memories, and the scaling pipelines that connect the integer datapath to the quantization scheme of this work.

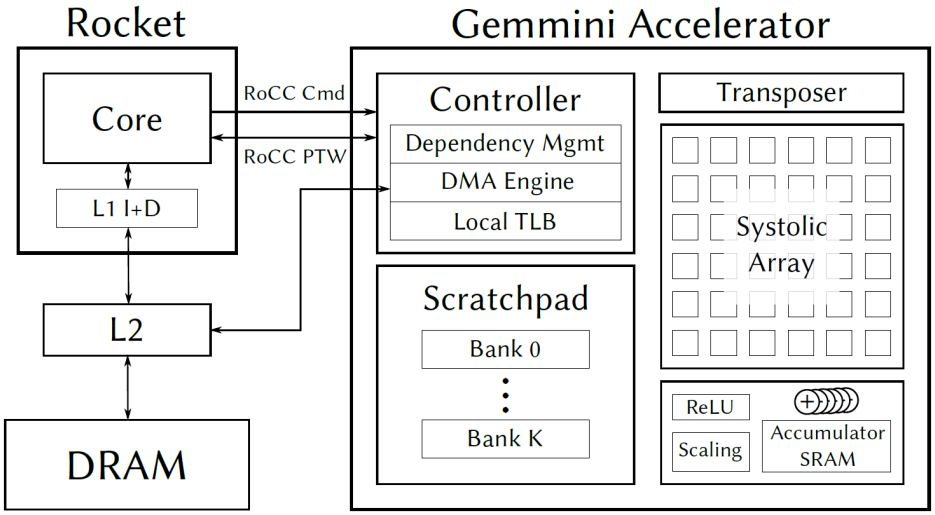
4.1 Architectural Template and SoC Integration

Gemmini is an open-source project that produces a parameterized systolic-array accelerator for deep-learning inference and training (GENC et al., 2021). Rather than a single fixed design, it is a hardware template written in the Chisel/Scala language, from which a concrete RTL instance is elaborated by choosing a set of parameters. The generated accelerator is integrated as a tightly coupled coprocessor into a Rocket Chip system-on-chip assembled with the Chipyard framework (AMID et al., 2020), which is what allows the same design to be both simulated and synthesized through a complete hardware flow.

The accelerator communicates with its host core through the Rocket Chip Coprocessor (RoCC) interface, as shown in Figure 4.1. Gemmini does not fetch its own instructions; instead, custom RISC-V instructions are interspersed with ordinary CPU instructions, and whenever the host core encounters a RoCC instruction it dispatches it, by opcode, to the accelerator and continues executing, with synchronization handled through memory fences (GENC et al., 2021). The accelerator shares the memory hierarchy of the host, including the L2 cache, the translation lookaside buffer, and the page-table walker, so that system-level effects are captured. Non-matrix operations that the array does not support are left to the host CPU.

Internally, Gemmini follows a decoupled access-execute organization built around three instruction classes, each with its own queue: load instructions move data from main memory into the scratchpad, execute instructions stream data from the scratchpad through the spatial array and back, and store instructions write results from the scratchpad back to main memory (GENC et al., 2021). A reservation station tracks dependencies between these instruction streams. This separation lets data movement and computation overlap,

Figure 4.1: Gemmini integrated as a RoCC coprocessor in a Rocket Chip SoC, showing the controller, the banked scratchpad, the systolic array, the transposer, and the accumulator SRAM.



Source:Adapted from (GENC et al., 2021)

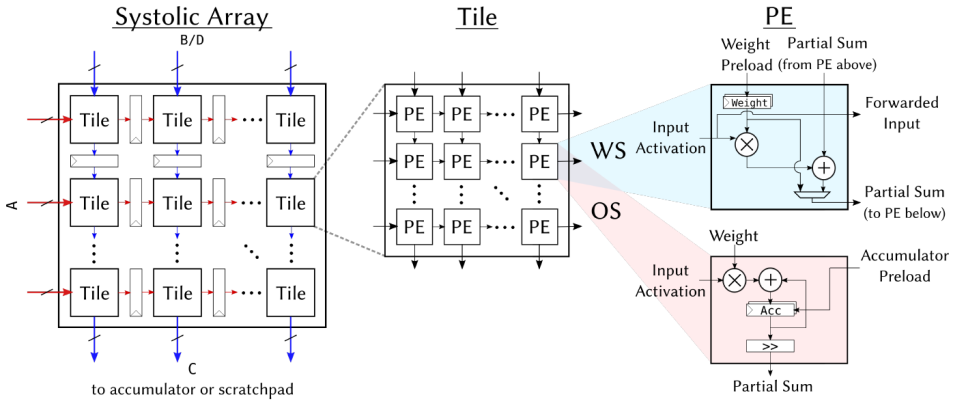
keeping the array busy. Besides the array and the on-chip memories, the controller hosts a DMA engine and a local TLB, and the accelerator includes a transposer that can transpose an operand matrix on its way into the array, as required by the dataflow.

4.2 Spatial Array: Mesh, Tile, and Processing Element

The computational core of Gemmini is a two-dimensional spatial array organized hierarchically: the array (or *mesh*) is a grid of *tiles*, and each tile is in turn a grid of *processing elements* (PEs), as shown in Figure 4.2 (GENC et al., 2021). The configuration studied in this work uses single-PE tiles (tileRows $=$ tileColumns $=$ 1) arranged in a 16×16 mesh (meshRows $=$ meshColumns $=$ 16), giving 256 PEs. This mesh size, together with the 256 KB four-bank scratchpad described below, corresponds to Gemmini's own `defaultConfig` in the project's Chisel sources (Berkeley Architecture Research, 2024) — the same baseline shipped with Chipyard's `GemminiRocketConfig` — rather than a size chosen specifically for this study. The two axes this work varies are the numerical datatypes layered on top of this fixed template (Chapter 5), not the array or memory dimensions themselves.

Each PE contains a multiply-accumulate (MAC) unit and a small number of registers, and communicates only with its immediate neighbors, exactly as described for systolic arrays in Section 2.4.

Figure 4.2: Hierarchy of the Gemmini spatial array—mesh of tiles, tile of PEs—and the PE datapath for the weight-stationary (WS) and output-stationary (OS) dataflows.



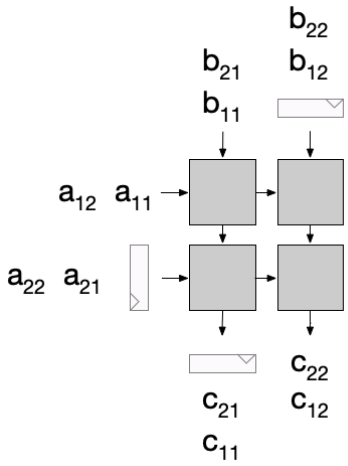
Source:Adapted from (GENC et al., 2021)

The array supports both weight-stationary (WS) and output-stationary (OS) dataflows, selectable at elaboration time or, when both are generated, at run time. In the weight-stationary dataflow used here, weights are first loaded into the PEs by a `preload` instruction and held stationary while activations stream through the array; each PE multiplies the incoming activation by its stored weight and accumulates the result into a partial sum that propagates down its column (GENC et al., 2021). The choice of dataflow has a measurable effect on performance: prior design-space studies on Gemmini report the weight-stationary dataflow outperforming the output-stationary one by roughly threefold on convolutional workloads (GOOKYI et al., 2023). The column dimension of the array therefore sets the number of products reduced in a single pass, which directly determines the width of the partial sums leaving the array, as derived next.

Because data enters the array from its edges and advances by one PE per cycle, the two operands of a given product reach their PE on different cycles. Gemmini skews the input streams with a triangle of delay registers at the array's edges—and a matching set at its outputs—so that the correct activation and weight meet in each PE at the right cycle and the results emerge aligned. Figure 4.3 illustrates this choreography for a 2×2 example computing $C = A \times B$.

The weight-stationary dataflow also requires one of the operand matrices to enter the array in transposed order. Gemmini performs this on-chip with a dedicated *transposer*: a small systolic array that takes a matrix in along one axis and streams it out along the orthogonal one, swapping its row and column indices. Figure 4.4 traces the operation for a 2×2 matrix over five cycles, with the elements entering from the left and emerging transposed from the top.

Figure 4.3: Delay registers skew the operand streams so that the correct elements meet in each PE, shown for a 2×2 matrix multiplication.



Source: Adapted from (Berkeley Architecture Research, 2024)

4.3 Datatypes and Bit-Width Parameters

A central feature of Gemmini for this work is that it separates the datatypes used to *store* operands from those used to *compute* on them. The storage types—`inputType`, `weightType`, and `accType`—define the element widths of the scratchpad and accumulator memories, whereas the datapath types—`spatialArrayInputType`, `spatialArrayWeightType`, and `spatialArrayOutputType`—define the operand and partial-sum widths inside the array (GENC et al., 2021; Berkeley Architecture Research, 2024). This separation, summarized in Table 4.1, is what makes a clean precision sweep possible, and it is also where the platform's two width constraints arise.

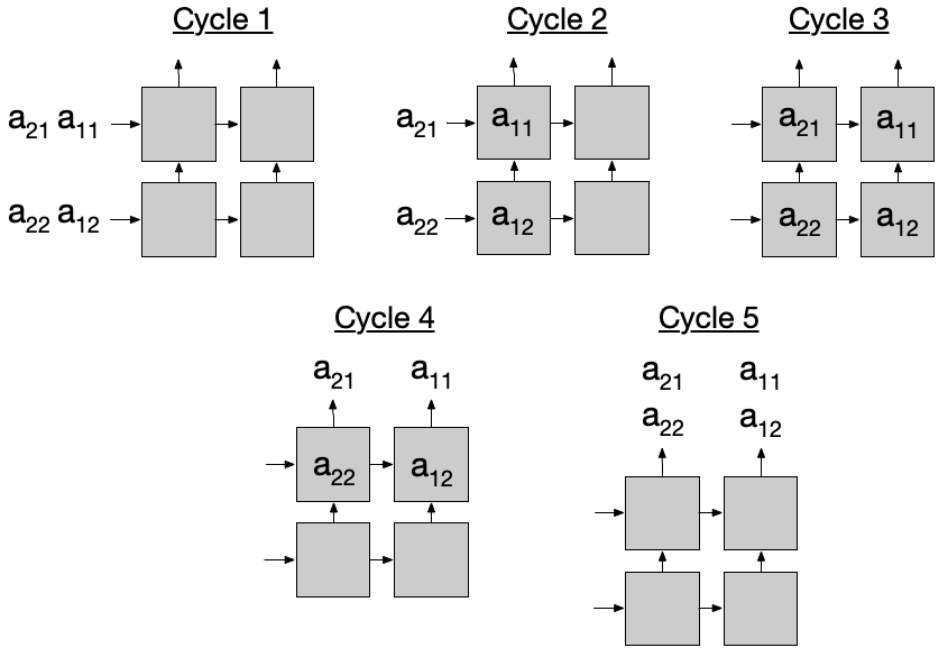
Table 4.1: Gemmini's numerical datatype parameters, separated into the storage types that size the on-chip memories and the datapath types that size the array's arithmetic.

Parameter	Domain	Width
`inputType`, `weightType`	Scratchpad storage	Set by configuration
`accType`	Accumulator storage	Set by configuration, bounded by the ISA
`spatialArrayInputType`, `spatialArrayWeightType`	Array datapath	Set by configuration
`spatialArrayOutputType`	Array datapath (partial sum)	Derived: $2n + \lceil \log_2 D \rceil$

Source: The Author.

The widths are not free parameters; they follow from the arithmetic of the MAC operation. The sizing rule is the classical one for multiply–accumulate hardware: in a conventional MAC unit that accumulates N products of two n-bit operands, the adder

Figure 4.4: The transposer streams a matrix in along one axis and out along the other, transposing it; shown for a 2×2 matrix over five cycles.



Source: Adapted from (Berkeley Architecture Research, 2024)

and accumulator must be at least $2n + \lceil \log_2 N \rceil$ bits wide to hold the result without overflow (MASADEH; HASAN; TAHAR, 2019). The two terms have a direct reading: multiplying two n-bit operands produces a product of up to $2n$ bits, and summing N of them adds $\lceil \log_2 N \rceil$ bits of headroom. Gemmini applies exactly this rule to the in-array reduction, where a single mesh column sums $D = \text{tileRows} \times \text{meshRows}$ products into one partial sum, so the width of the partial sum leaving the array is

$$\text{spatialArrayOutputType} = 2n + \lceil \log_2 D \rceil. \tag{4.1}$$

For the default configuration, with $n = 8$ and $D = 16$, this gives $2 \cdot 8 + \log_2 16 = 20$ bits, matching the value emitted by the generator—and coinciding with the canonical 8-bit, 16-deep MAC that the same rule sizes at 20 bits (MASADEH; HASAN; TAHAR, 2019).

The accumulator width is governed by a different and larger reduction. The accumulator does not hold a single in-array partial sum but accumulates results across the full contraction dimension K of the matrix multiplication, which is generally far larger than the physical array: an output tile is produced by streaming many array-sized chunks through the mesh over time and summing them in the accumulator. In the terms of the classical rule above, Equation 4.1 sizes only the D-deep reduction performed inside a column; the accumulator must additionally absorb the cross-tile accumulation over K, a re-

duction the single-array MAC model does not consider (MASADEH; HASAN; TAHAR, 2019). Because the worst-case magnitude of this running sum grows with K, the accumulator must be provisioned with substantial headroom; the default `accType` is a 32-bit signed integer, which comfortably accommodates 8-bit operands over realistic contraction depths. The accumulator width is thus decoupled from Equation 4.1: the partial-sum width is sized for the in-array reduction, while `accType` is sized for the full-matmul reduction.

Two of these parameters carry constraints that bound the precisions a configuration may adopt. First, the storage element types `inputType` and `weightType` cannot be reduced below one byte: sub-byte elements break the byte-aligned addressing assumed throughout the generator, where the bytes-per-element term $\lfloor \text{bit-width}/8 \rfloor$ would evaluate to zero and trigger a division-by-zero during elaboration (Berkeley Architecture Research, 2024). A precision finer than one byte must therefore be carried by the datapath types alone, with the operands stored byte-aligned and sign-extended. Second, `accType` cannot be made arbitrarily wide: its width is bounded by how the accumulator parameters are encoded in the ISA configuration instruction (`ConfigNormRs2`), independently of the headroom that the in-array derivation of Equation 4.1 would otherwise suggest (Berkeley Architecture Research, 2024). How a given configuration navigates these two constraints, and the specific precisions and accumulator widths explored in this work, are the subject of the methodology (Chapter 5).

4.4 Scratchpad and Accumulator Memory

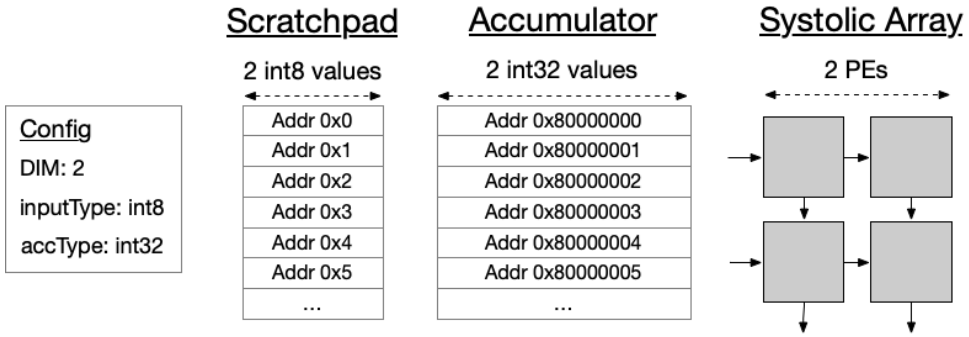
Gemmini provides two on-chip memories. The scratchpad holds the input, weight, and intermediate operand tiles, and is parameterized by its capacity in kilobytes, its number of banks, and whether it is single- or dual-ported (GENC et al., 2021; Berkeley Architecture Research, 2024); the configuration here uses a 256 KB, four-bank, single-ported scratchpad. Crucially, the scratchpad is sized by a fixed byte capacity, not by a number of elements. Its physical dimensions, and therefore its area, are independent of the operand bit-width—a property that, as shown in the results, makes the scratchpad area constant across the INT4, INT8, and INT16 configurations.

The accumulator stores the wide partial results produced by the array, with elements of width `accType`, and is similarly parameterized by capacity, banks, and porting. Because accumulation is a read-modify-write operation—each new partial sum must

be read, added, and written back—the accumulator requires concurrent read and write access and is therefore provisioned with multiple ports by default. Two of this work's design axes act directly on this memory: its element width (`accType`, 16 versus 32 bits) and, indirectly, the multi-port requirement, which constrains which SRAM macros can be generated for it and which is examined in the methodology.

The scratchpad and the accumulator share a single local address space (Berkeley Architecture Research, 2024), distinguished by the most significant address bit: scratchpad rows occupy the low addresses, while accumulator rows are mapped to a high range beginning at `0x80000000`, as illustrated in Figure 4.5. The figure also shows that each accumulator row holds wide (`accType`) elements, whereas the scratchpad holds narrow (`inputType`) elements, even though both are addressed one row at a time.

Figure 4.5: Unified local address space: the scratchpad occupies the low addresses and the accumulator a high range (from `0x80000000`), with narrow scratchpad elements and wide accumulator elements.

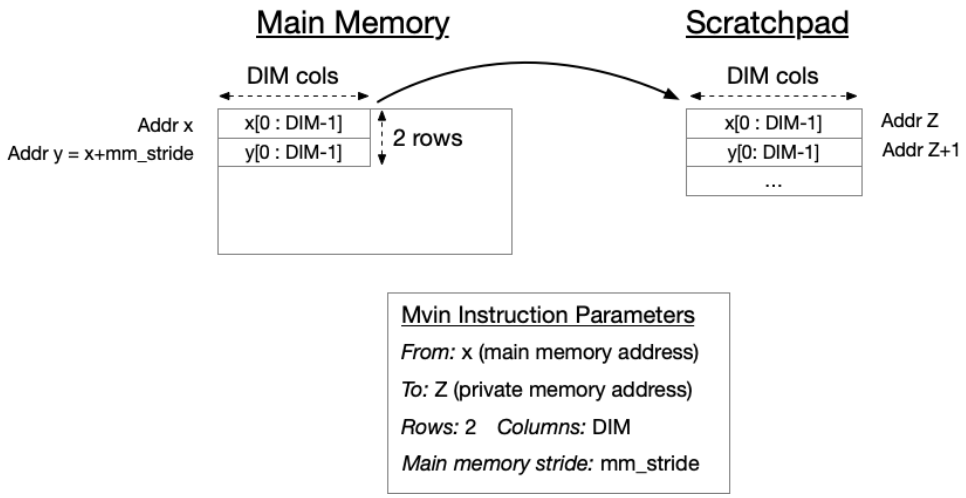


Source: Adapted from (Berkeley Architecture Research, 2024)

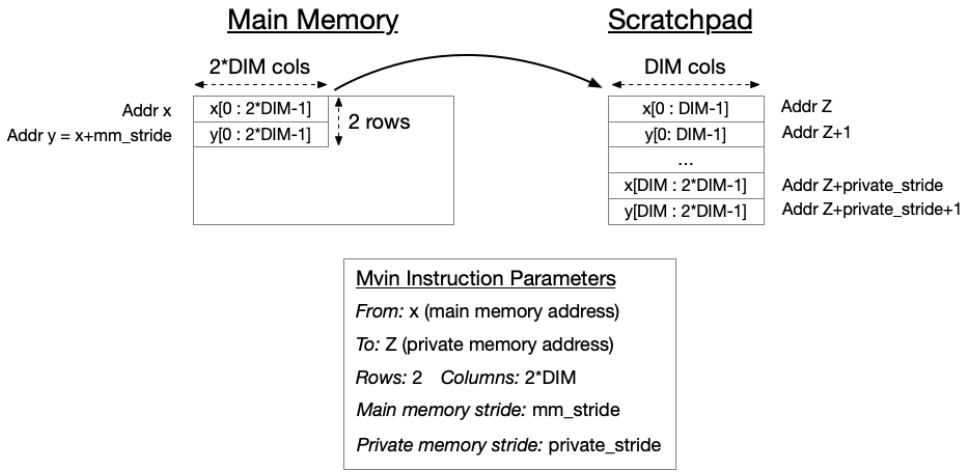
Data is brought on-chip by the move-in (`mvin`) instruction, which transfers a two-dimensional tile from main memory into the scratchpad (Berkeley Architecture Research, 2024). The instruction specifies a source address, a destination scratchpad row, the number of rows and columns, and a main-memory stride that lets it gather rows that are not contiguous in DRAM, as in Figure 4.6 (a). A single `mvin` can also move a block wider than the array, distributing the extra columns across the scratchpad with a local (*private*) stride, as in Figure 4.6 (b); this amortizes the instruction overhead when loading large operands. Results are scaled as they are read out of the accumulator, by the stage described next.

To keep the array fed while these transfers take place, Gemmini's software stack applies *double buffering* (ping-pong buffering) to the scratchpad and the accumulator: a CISC-style loop instruction such as `gemmini_loop_ws` is restricted to at most half of each memory's rows, so that one half can be computed on by the array while the other half

Figure 4.6: The move-in (`mvin`) instruction transfers a two-dimensional tile from main memory into the scratchpad, using a main-memory stride and, for wide blocks, a local private stride.



(a) Standard move-in of a DIM-wide tile.



(b) Block move-in of a wider tile, split across the scratchpad by a private stride.

Source: Adapted from (Berkeley Architecture Research, 2024)

is refilled by an in-flight `mvin`/`mvout`, overlapping data movement with computation across successive tiles (Berkeley Architecture Research, 2024).

Because half of the nominal capacity is reserved for the buffer currently in flight, the usable tile size for the outer-loop tiling of Section 2.4 is effectively half of the scratchpad's and accumulator's total row count, a constraint the tiling search for ResNet-50 must respect on every configuration (CHEN et al., 2022). Double buffering does not add memory area beyond the capacity already reported in Chapter 6: it is a scheduling policy applied by software and the reservation-station/DMA control logic over the existing banks, not an additional hardware structure, so it is not a separate line item in the area and power breakdown of Section 6.1.

4.5 Scaling Pipelines: Mvin Scale and Accumulator Scale

The integer datapath of the array is connected to the quantization scheme of Section 2.5 by two scaling stages, both of which this work modifies. Each stage multiplies an integer value by a floating-point scale factor—represented as an FP32 number—and converts the result back to a target integer type with rounding and saturation.

The first stage, the *mvin scale*, is applied as operands are moved from main memory into the scratchpad. It multiplies each loaded value by a per-tensor floating-point scale and clamps the result to the operand range, realizing in hardware the input quantization scaling of Section 2.5. The second stage, the *accumulator scale*, is applied to the accumulated result as it is read out of the accumulator. It multiplies the wide integer accumulator value by the output scale factor $M = s_x s_w / s_y$ and rounds and clamps it to the output bit-width, thereby performing the requantization step described for integer-only inference (JACOB et al., 2018).

Supporting INT4 and INT16 required adapting both scaling stages, since each must saturate to the range of its target precision. The saturation bounds were changed from the default 8-bit range to $[-8, 7]$ for INT4 and $[-32768, 32767]$ for INT16, applied consistently in both the Chisel hardware (the clamping logic of each scale unit) and the C reference expression used by the software stack to mirror the hardware behavior. These modifications, together with the accumulator width, constitute the configurable levers whose hardware cost is characterized in the remainder of this work.

5 METHODOLOGY

This chapter describes how the design space exploration is realized and evaluated, building on the Gemmini architecture introduced in Chapter 4. The goal is a *like-for-like* comparison of distinct hardware configurations of the Gemmini accelerator (GENC et al., 2021) through a real ASIC synthesis flow, rather than through analytical or pre-characterized models. Section 5.1 defines the design space and the two architectural axes under study. Section 5.2 explains how each configuration is derived from a common baseline in Gemmini's parameterized template, together with the constraints that bound the precision configurations and the RTL modifications introduced in the scaling pipelines. Section 5.3 details the synthesis flow built on top of Hammer (LIEW et al., 2022) and Cadence Genus (Cadence Design Systems,), the choice of process design kit (PDK), the handling of memory macros, and the clock targets. Section 5.4 describes the cycle-accurate throughput, and Section 5.5 the metrics extracted and how they are attributed to each axis.

5.1 Design Space Definition

The exploration is organized around two architectural axes. All other template parameters — the 16×16 weight-stationary mesh, the single-PE tiles, the 256 KB four-bank scratchpad, the host CPU, and the SoC integration described in Chapter 4 — are held fixed, so that the measured differences can be attributed unambiguously to the axis under variation. The two axes are:

1. **Datapath precision.** The integer bit-width of the spatial-array operands (`spatialArrayInputType` and `spatialArrayWeightType`). Three levels are studied: `INT4`, `INT8` (the baseline), and `INT16`. This axis primarily affects the arithmetic units inside each processing element (PE) and the width of the partial sums propagated through the mesh.
2. **Accumulator data width, within INT4.** The bit-width of the accumulator datatype (`accType`), set to either 16 bits (`acc16`) or 32 bits (`acc32`). This axis affects the accumulator read–modify–write adders, the accumulator scale unit, and the surrounding accumulator-memory logic; the accumulator SRAM itself is sized by a fixed byte capacity and does not change with `accType`.

The accumulator-width axis is deliberately confined to the `INT4` precision. As derived in Chapter 4 (Equation 4.1), the partial sum leaving the mesh is 20 bits wide for `INT8` and 36 bits wide for `INT16`, both of which exceed 16 bits; a 16-bit accumulator can therefore only hold a single partial sum for `INT4`, whose mesh output is 12 bits. Comparing `acc16` against `acc32` is thus meaningful only for the `INT4` datapath, where both widths are valid choices, and the exploration restricts it accordingly. For `INT8` the accumulator is fixed at 32 bits, and for `INT16` it is held at 32 bits by the ISA encoding (Section 5.2.3), narrower than the 36-bit partial sum it must absorb.

The precision axis is restricted to integer formats. Gemmini's template also supports floating-point element types, including half-precision floating point (FP16) and the brain floating-point format (BF16), which trade a wider dynamic range for a smaller mantissa and are common in training and mixed-precision inference. Exploring these floating-point datapaths is outside the scope of this work and is left as future work (Chapter 7).

The reference point of the exploration is a single *baseline* configuration — conventional `INT8` precision with a 32-bit accumulator — against which the variations along each axis are normalized. Normalizing against a common baseline removes the *constant* biases shared by every configuration (for example, fixed control and SoC overhead), isolating the contribution of each axis. Table 5.1 enumerates the four configurations and the comparison each one enables; their concrete datatype widths, which follow the derivation of Chapter 4, are collected in Table 5.2.

Table 5.1: Configurations explored and the comparison each enables. Datatype widths are derived in Chapter 4 and given in Table 5.2.

Configuration	Precision	`accType`	Comparison enabled
INT8	INT8	32	baseline
INT16	INT16	32	precision axis (vs. INT8)
INT4 / acc32	INT4	32	precision axis (vs. INT8); accumulator width (vs. INT4 / acc16)
INT4 / acc16	INT4	16	accumulator width (vs. INT4 / acc32)

Source: The Author.

5.2 Configuration Implementation

Each configuration is instantiated through Gemmini's Chisel template (GENC et al., 2021) within the Chipyard SoC framework (AMID et al., 2020), and selected at elaboration time, so that the entire SoC — including the RISC-V host core, the RoCC interface,

and the memory hierarchy — is regenerated for each design point.

To keep the configurations strictly comparable, all of them are derived from the same baseline by a single helper, `quantConfig(w, accW)`, which takes the datapath operand width `w` and the accumulator width `accW` and returns a configuration that copies the baseline (using Chisel's `.copy()`) and overrides only the affected datatype parameters. The four design points of Table 5.1 are then `quantConfig(8, 32)` for the baseline, `quantConfig(16, 32)` for `INT16`, and `quantConfig(4, 32)` and `quantConfig(4, 16)` for the two `INT4` variants. Deriving every variant from one helper guarantees that the configurations differ *only* in the parameters under study and share every other structural choice.

5.2.1 Datatype Widths

The datapath and storage widths are not free parameters: they follow from the arithmetic of the multiply–accumulate operation and are derived in detail in Chapter 4. In brief, multiplying two n-bit datapath operands produces a $2n$-bit product, and reducing D of them along a mesh column adds $\lceil \log_2 D \rceil$ bits, giving the partial-sum width `spatialArrayOutputType` of Equation 4.1; the accumulator width `accType` is sized separately, for the larger reduction over the full contraction dimension of the matrix multiplication. The resulting widths for the configurations explored are collected in Table 5.2. The two precision configurations that depart from this clean derivation, because of platform constraints, are described next.

Table 5.2: Datatype widths (in bits) of the configurations explored. `INT4` is the only precision for which the accumulator width is varied (Section 5.2.2).

Datatype (width in bits)	INT4	INT8	INT16
Storage operand (`inputType`)	8	8	16
Datapath operand (`spatialArrayInputType`)	4	8	16
Partial sum (`spatialArrayOutputType`)	12	20	36
Accumulator (`accType`)	16 or 32	32	32

Source: The Author.

5.2.2 INT4 Constraints

As established in Chapter 4, `INT4` carries its 4-bit precision only in the datapath types, while the scratchpad storage stays byte-aligned at 8 bits. The multipliers therefore operate at the true 4-bit width with the storage unchanged, so the area savings of `INT4` are expected in the array's arithmetic rather than in the scratchpad, as examined in Chapter 6.

The `INT4` precision is also the only one for which the accumulator width is itself a design variable. Its 12-bit mesh output (Table 5.2) fits within a 16-bit accumulator, so both `acc16` and `acc32` are valid, and the pair forms the accumulator-width axis of Section 5.1.

5.2.3 INT16 Constraints

As established in Chapter 4, the `INT16` accumulator is capped at 32 bits by the ISA encoding — narrower even than the 36-bit partial sum the mesh produces (Table 5.2), and far short of the headroom that cross-tile accumulation would require (the `INT8` accumulator, by contrast, carries 12 bits above its 20-bit partial sum). Each partial sum must therefore be saturated to fit the accumulator, so the clamping in the modified accumulator scale (Section 5.2.4) engages far more often than in the other configurations. The accuracy implications of this bound, and the broader question of whether `INT16` earns its additional hardware cost over `INT8` for inference, are taken up in the discussion of Chapter 6.

5.2.4 Modified Mvin and Accumulator Scale

The two scaling stages that connect the integer datapath to the quantization scheme — the *Mvin scale* on the way into the scratchpad and the *accumulator scale* on the way out of the accumulator — are described architecturally in Chapter 4. Supporting the `INT4` and `INT16` configurations required modifying both, and three points of that modification are methodological.

First, the accumulator scale saturates to the range of the *output element type*, not of the accumulator type, so the saturation bound is precision-dependent and is set through a custom clamping expression per configuration: $[-8, 7]$ for `INT4` and $[-32768, 32767]$ for

`INT16`, replacing the default 8-bit range. Second, the clip is applied *before* the datatype cast: casting first would let an out-of-range value wrap modulo the destination width, after which clipping can no longer recover the saturated result, whereas clipping first guarantees true saturation. This matches the affine integer quantization semantics of Section 2.5 (JACOB et al., 2018; NAGEL et al., 2021). Third, the same bounds are applied consistently in the Chisel hardware and in the C reference used by the software stack, so that the simulated and synthesized behavior agree. Supporting the 36-bit `INT16` datapath, in particular, required generalizing the width-clamping logic to operate beyond 32 bits and using arbitrary-precision arithmetic for the elaboration-time width computations, to avoid latent overflow of the 32-bit integers used internally by the generator.

5.3 Synthesis Flow

The RTL emitted by Gemmini is taken through an ASIC logic-synthesis flow driven by Hammer (LIEW et al., 2022), the modular ASIC flow design tool from the Berkeley Architecture Research group, which provides a tool- and PDK-agnostic front end. Hammer invokes Cadence Genus (Cadence Design Systems,) to perform synthesis and to produce the area, power, and timing reports analyzed in Chapter 6. The flow is fully scripted for reproducibility through Hammer's `make`-based driver: each configuration is described by a Hammer configuration that selects the ASAP7 technology and its SRAM compiler, the input RTL, and the clock constraints, and synthesis runs through the default flow. No custom Genus hooks for memory handling or timing-path cutting are required, because every on-chip memory maps to a real SRAM macro under ASAP7 (Section 5.3.2); the area, power, and timing reports are produced by Genus's standard reporting commands (Section 5.5).

The exact Hammer configuration files(tool, license, and PDK setup) are reproduced in Appendix A.

5.3.1 PDK Choice: ASAP7 versus SKY130

Two open PDKs were considered: the ASAP7 7 nm FinFET predictive kit (CLARK et al., 2016) and the SkyWater SKY130 130 nm open-source process (SkyWater Technology Foundry; Google, 2022). ASAP7 was selected as the synthesis target

because its SRAM generation flow produces the macros required by Gemmini's local memories, together with the corresponding LEF abstracts, so that both the scratchpad and the accumulator — including the dual-port memory the accumulator requires — are realized as real SRAM blocks. SKY130 could not: its SRAM generator catalog (`sram22`) is exclusively single-port, whereas the accumulator performs an accumulating read–modify–write every cycle and therefore needs a true dual-port (one-read/one-write) memory. Under SKY130 that memory would fall back to synthesized flip-flops (synflops), which would distort the area results for the reasons discussed in Section 5.3.2. ASAP7 is also the more representative target, being a 7 nm predictive node closer to the technology in which contemporary DNN accelerators are fabricated.

5.3.2 Memory Macro Handling

The accelerator instantiates two distinct on-chip memories: the *scratchpad*, which stores inputs and weights, and the *accumulator*, which stores partial sums.

If a memory cannot be mapped to a real SRAM macro, the synthesis tool implements it as an array of standard-cell flip-flops — so-called *synflops*. Synflops are pathological for this study: a flip-flop-based memory is roughly an order of magnitude larger than the equivalent SRAM and, crucially, this inflation *interacts with the variable under study*, since the accumulator size changes with `accType`. Normalizing against a baseline removes only the constant component of such a bias and cannot correct a configuration-dependent artifact, so it is essential that every memory map to a real macro rather than to synflops.

This is exactly what the ASAP7 SRAM compiler provides, and it is the reason the flow requires no special memory handling. The scratchpad, whose access pattern is a single read/write per cycle, maps to single-port (1RW) SRAM macros, and the accumulator, whose accumulating read–modify–write needs concurrent read and write, maps to a dual-port (2RW) macro (for example, `SRAM2RW128x8`). Chipyard's MacroCompiler performs this mapping as part of the standard flow, so no memory is blackboxed or expanded into synflops, no `set_false_path` or other timing-path hooks are introduced, and the configuration is left at its defaults.

Because both memories are realized as real SRAM blocks, their area is captured in full alongside the surrounding logic. The accumulator SRAM, however, is sized by a fixed byte capacity in the same way as the scratchpad, so halving `accType` doubles

the number of stored entries rather than shrinking the array. The accumulator-width axis (`acc16` versus `acc32`, within `INT4`) is therefore expected to move the accumulator *logic* — the read–modify–write adders and the accumulator scale unit — while leaving the storage area essentially unchanged, as the synthesized results confirm (Chapter 6).

5.3.3 Clock Targets

Two complementary synthesis sweeps are run for every configuration.

- **Fixed clock (iso-frequency).** A common target period of 5 ns is imposed on all configurations. Because every design is synthesized under the same timing constraint, the resulting area and power figures are directly comparable and reflect the cost of each configuration at the same operating frequency. This sweep underlies the area and power analyses of Chapter 6.
- **Maximum frequency ($F_{\mathbf{max}}$).** The clock period is tightened per configuration until timing slack is no longer met, yielding the maximum frequency each design can sustain. The resulting $F_{\max}$ is combined with the cycle counts of Section 5.4 to translate the per-configuration cycle counts into wall-clock latency and throughput.

Both sweeps additionally reserve a *clock uncertainty* margin, imposed through `set_clock_uncertainty`. Clock uncertainty models the jitter of the clock source together with the skew introduced by the clock distribution network (WESTE; HARRIS, 2011; STATIC. . . , 2009). At the logic-synthesis stage the clock tree has not yet been built, so its skew is still unknown and must be anticipated by the margin; for this reason the pre-layout uncertainty is set higher than the value that would suffice once the tree exists and only jitter remains (STATIC. . . , 2009).

The magnitude reserved here is 10% of the clock period—an engineering rule of thumb rather than a fixed standard, but one whose exact value is not critical to this study: because the same margin is imposed on every configuration, it shifts all of them by an identical amount and does not affect the relative area, power, and frequency comparisons the exploration draws, the same invariance that motivates the common clock period itself. Under the 5 ns fixed-clock constraint, for example, the margin leaves 4.5 ns for the logic and interconnect.

5.4 Performance Evaluation

Cycle-level throughput is evaluated through cycle-accurate RTL simulation in Verilator (SNYDER,), which executes the exact RTL that is fed to synthesis and reports the cycle counts of the simulated workload. The chosen workload is inference of ResNet-50, a representative convolutional network that is bundled with Gemmini's software and exercises the convolution, matrix-multiplication, residual-addition, and pooling paths of the accelerator.

A property of Gemmini that simplifies this measurement is that its cycle counts are *data-independent*: the number of cycles is determined entirely by the layer dimensions and the tiling chosen for a given configuration, not by the numerical values of the weights or activations. The throughput figures therefore depend on the architecture and its tiling, and not on the correctness of any particular set of quantized weights.

Quantization-induced classification accuracy is *not* re-measured here; instead, the accuracy implications of each precision level are drawn from the published quantization literature. This applies uniformly to every configuration of Table 5.1, including the `INT8` baseline: no classification run, with or without quantization-aware training (QAT), was performed for any configuration in this work, and the ΔTop-1 values reported in Section 6.6 are in every case the source's own reported baseline-versus-quantized accuracy, not a result reproduced here.

To keep that comparison consistent across sources, accuracy is reported as the loss in Top-1 accuracy relative to each work's own floating-point baseline (ΔTop-1), since absolute baselines vary across studies, and is taken for ResNet-50 on ImageNet — the same network used for the throughput measurements. These cited accuracies are then paired with the measured hardware cost in Section 6.6, keeping the focus of this work on hardware characterization while still grounding the precision axis in known accuracy behavior.

Per-configuration latency is computed by dividing the simulated cycle count by the configuration's F_{max} (Section 5.3.3), and throughput is reported in inferences per second. Because the cycle counts are data-independent, the `INT4` and `INT8` configurations reuse the quantized ResNet-50 weights bundled with Gemmini, whose 8-bit byte-aligned storage they share. The `INT16` configuration, whose 16-bit storage element departs from that byte-aligned format, instead uses weights regenerated by exporting the network to ONNX and re-quantizing into the 16-bit format Gemmini expects (JACOB et al., 2018).

In every case the resulting cycle counts reflect the tiling structure alone, not the numerical weight values.

5.5 Metrics and Extraction

Three classes of metric are extracted from the synthesized design, each from a standard Genus report produced by the flow.

Area. A hierarchical area report (`report_area -hierarchy`) is collected so that the contribution of each module can be isolated. A central methodological point is the separation of *SRAM* area from *logic* area: the scratchpad SRAM macro dominates the total cell area and, if left aggregated, masks the much smaller but more interesting differences in the compute and accumulator logic. Reporting the breakdown hierarchically maps each axis to the modules it affects:

- **Precision** is read from the spatial array (`mesh/Mesh` and `mesh/MeshWithDelays`), where the PE and MAC logic resides;
- **Accumulator width** (within `INT4`) is read from the accumulator adders (`acc_adders`), the accumulator scale unit (`acc_scale_unit`), and the accumulator memory (`acc_mems`); because the SRAM inside `acc_mems` is byte-capacity-sized and stays constant, the axis surfaces in the accumulator logic rather than in its storage.

Consistent with the byte-capacity sizing of the scratchpad described in Chapter 4, the scratchpad SRAM area is expected to remain constant across the precision configurations; whether the synthesized results bear this out is examined in Chapter 6.

Power. Post-synthesis power is estimated through a `report_power` command in *vectorless* mode, in which Genus (using its built-in Joules power engine) propagates default switching activities and static probabilities from the primary inputs through the synthesized netlist and combines them with the library power models, without requiring a simulation-derived activity file. Because every configuration is evaluated under the same fixed 5 ns constraint and the same vectorless assumptions, the resulting figures provide a consistent *relative* comparison of power across the architectures, which is what the design space exploration requires. This is consistent with the established observation that data

movement and memory accesses, rather than integer arithmetic, dominate the energy of these designs (HOROWITZ, 2014). The ASAP7 SRAM macros, however, are predictive timing-only models: their Liberty views provide setup/hold and propagation timing and declare the power and ground pins, but contain no power tables — neither dynamic `internal_power` groups nor a static `leakage_power` value. No power-analysis flow, vectorless or activity-driven, can therefore attribute internal or leakage power to these macros; both the scratchpad and the accumulator memories contribute only a negligible boundary term, so the reported totals characterize the logic and registers — the datapath and control — rather than the storage arrays. This is the relevant quantity for the precision axis, which changes the arithmetic while the dominant scratchpad access energy stays fixed across `INT4` and `INT8`; but it means the absolute totals understate full-chip power, a scope limitation revisited in Chapter 6.

Power is reported at two operating points. Under the common 5 ns constraint it gives the iso-frequency comparison used for the area and power analysis; for the per-inference energy of Chapter 6, power is additionally re-estimated at each configuration's own F_{max}, since dynamic power scales with the operating frequency (Equation 2.6).

Timing. Static timing analysis provides the critical-path delay, the worst-case slack, and the F_{max} used in Section 5.4. Because both on-chip memories are mapped to real SRAM macros with their own characterized timing models (Section 5.3.2), the paths to and from the scratchpad and accumulator are properly constrained, and no false-path exceptions or other timing hooks are required.

6 EXPERIMENTS AND RESULTS

This chapter reports the results of the design space exploration defined in Chapter 5. The four configurations of Table 5.1 — `INT8` (baseline), `INT16`, and the two `INT4` variants with a 32-bit and a 16-bit accumulator — are characterized along the two architectural axes: datapath precision and, within `INT4`, accumulator data width. Section 6.1 presents the synthesized area, Section 6.2 the post-synthesis power, and Section 6.3 the achievable clock frequency. These first three sections use the fixed 5 ns (iso-frequency) sweep, so that the figures are directly comparable across configurations. Section 6.4 reports the cycle-accurate ResNet-50 cycle counts and combines them with the per-design maximum frequency to obtain wall-clock latency and throughput, and Section 6.5 the energy per inference. Section 6.6 relates these costs to the classification accuracy reported in the quantization literature, and Section 6.7 draws the two axes together.

All area and power figures are extracted from Cadence Genus through the hierarchical reports described in Section 5.5. Area is reported as *cell area* (the standard-cell and macro area), which is the meaningful pre-layout quantity; the interconnect (net) area is a wireload estimate and is not analyzed here. Because the ASAP7 predictive kit draws its standard cells at a $4\times$ linear scale, the absolute area magnitudes should not be read as literal silicon area at a production 7 nm node; the analysis therefore emphasizes the *relative* comparisons, which are invariant to that scaling. Throughout, values are normalized to the `INT8` baseline.

6.1 Area

Table 6.1 breaks the synthesized cell area down by the modules that the two axes affect: the spatial array (`mesh/Mesh`), where the multiply–accumulate logic resides; the scratchpad SRAM (the four `ScratchpadBank` macros); the accumulator memory (`acc_mems`, both banks, which includes its mapped SRAM); the accumulator scale unit (`acc_scale_unit`); and the accumulator adders (`acc_adders`).

The precision axis is concentrated almost entirely in the mesh. The `INT16` mesh is $1.93\times$ the `INT8` mesh, while the `INT4` mesh is roughly half ($0.52\times$ for `acc32`), tracking the widening and narrowing of the multipliers and the partial-sum datapath. This is the expected behavior: the area of the arithmetic grows with the operand width, and the mesh is where that width lives.

Table 6.1: Synthesized cell area by module, in mm^2, under the fixed 5 ns constraint. The italicized rows normalize each module and the Gemmini total to the `INT8` baseline.

Module	INT8	INT16	INT4/acc32	INT4/acc16
Mesh (compute)	0.4924	0.9494	0.2569	0.2506
(normalized)	1.000	1.928	0.522	0.509
Scratchpad SRAM (4 banks)	1.6989	1.7387	1.6988	1.6988
(normalized)	1.000	1.023	1.000	1.000
Accumulator (`acc_mems`)	0.5407	0.5407	0.5407	0.5298
(normalized)	1.000	1.000	1.000	0.980
Accumulator scale	0.1283	0.1340	0.1251	0.0823
(normalized)	1.000	1.044	0.975	0.641
Accumulator adders	0.0085	0.0085	0.0085	0.0041
(normalized)	1.000	1.000	1.000	0.482
Gemmini total	3.4198	4.0145	3.1522	3.0695
(normalized)	1.000	1.174	0.922	0.898

Source: The Author.

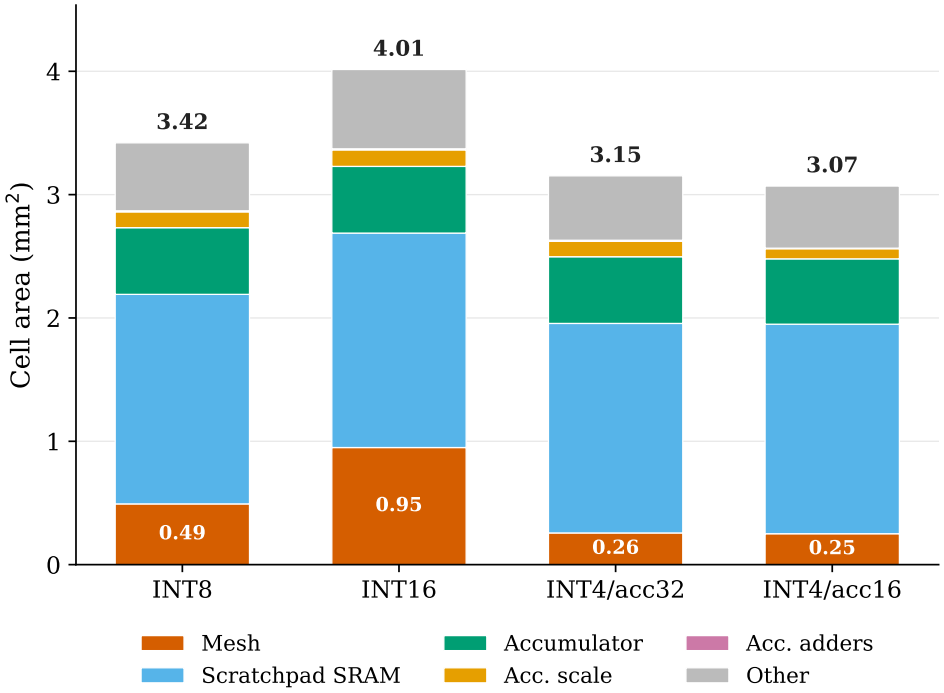
The scratchpad SRAM tells the complementary story. It is identical, to within 0.01%, for `INT8` and both `INT4` variants ($\approx 1.699\,mm^2$), because all three store 8-bit elements and the scratchpad is sized by a fixed byte capacity, as anticipated in Chapter 4. The only departure is `INT16`: its byte capacity is the same 256 KB, so the array stores the same number of bytes, but its 16-bit storage element changes the word organization of the mapped macros — wider words and the associated decode and I/O periphery — which raises the scratchpad area by 2.3%. The increase therefore reflects the memory's word geometry, not a change in stored capacity, leaving the byte-capacity argument intact.

Because that scratchpad SRAM accounts for about half of the `INT8` cell area and is essentially constant, it damps the total-area swings: the large $1.93\times$ and $0.52\times$ mesh ratios translate into a much milder $+17\%$ for `INT16` and -8% for `INT4/acc32` at the top level. A design that is dominated by on-chip storage hides much of the arithmetic cost of precision in its total area, which is why the per-module breakdown, rather than the aggregate, is the informative view.

Accumulator-width axis. Comparing `INT4/acc16` against `INT4/acc32` isolates the effect of halving the accumulator width. The saving is -2.6% of the Gemmini total, and — importantly — it comes entirely from the surrounding *logic*, not from the accumulator storage. The accumulator scale unit shrinks by 34% (0.1251 to $0.0823\,mm^2$) and the read–modify–write adders by 52% (0.0085 to 0.0041 mm^2), because both process accumulator-width operands. The accumulator SRAM itself (`mem_0_ext`), by contrast,

is unchanged: at $\approx 0.514\,\mathrm{mm}^2$ for both widths (a difference of 0.07%), it is byte-capacity-sized in exactly the way the scratchpad is, so narrowing `accType` buys twice as many entries rather than a smaller array. The accumulator-width axis is thus a *logic* optimization, and the storage is invariant to it.

Figure 6.1: Cell-area breakdown per module for the four configurations under the fixed 5 ns constraint.



Source: The Author

6.2 Power

Table 6.2 reports the vectorless power estimated under the same 5 ns constraint, for the whole accelerator and for the two blocks the axes move most: the mesh and the scratchpad subsystem (`spad`, which also contains the accumulator, its scale unit, and the DMA-side queues).

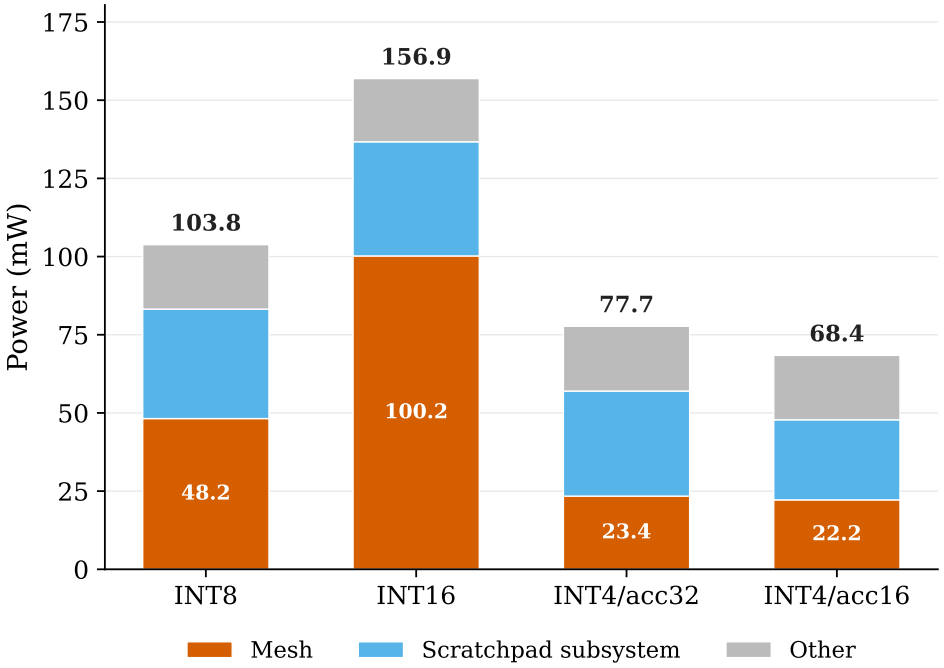
The precision axis dominates power just as it dominates mesh area: the `INT16` mesh draws about $2.08\times$ the `INT8` mesh power, the `INT4` mesh about $0.49\times$. At the top level, `INT16` draws $1.51\times$ and the two `INT4` variants $0.75\times$ and $0.66\times$ the baseline. The accumulator-width axis lowers total power by 12% (`acc16` vs. `acc32`), once more concentrated in the accumulator logic — the scale unit alone accounts for most of the difference.

Table 6.2: Post-synthesis vectorless power, in mW, under the fixed 5 ns constraint. The italicized rows normalize each block and the Gemmini total to the `INT8` baseline.

Block	INT8	INT16	INT4/acc32	INT4/acc16
Mesh	48.2	100.2	23.4	22.2
(normalized)	1.000	2.079	0.485	0.461
Scratchpad subsystem	35.0	36.5	33.6	25.6
(normalized)	1.000	1.043	0.960	0.731
Gemmini total	103.8	156.9	77.7	68.4
(normalized)	1.000	1.512	0.749	0.659

Source: The Author.

Figure 6.2: Power across the four configurations under 5 ns clock period.



Source: The Author

Two properties of these figures should be stated plainly, as they bound how the numbers may be read. First, the estimates are leakage-dominated: at `INT8`, the logic leakage is $\approx 40\%$ of the total, internal $\approx 36\%$, and switching $\approx 24\%$. A large static share is characteristic of the ASAP7 predictive node and means the comparison reflects standing as much as switched power. Second, and more important, the ASAP7 SRAM macros are predictive timing-only models that carry no power data — neither dynamic `internal_power` nor static `leakage_power` — so no power-analysis flow can attribute internal or leakage power to them; both the scratchpad and the accumulator memories contribute only a negligible boundary term, and the totals in Table 6.2 characterize the *logic and registers* — the datapath and control — rather than the storage arrays. This is

the relevant quantity for the precision study, since the arithmetic is exactly what the axes change and the dominant scratchpad access energy is precision-invariant across `INT4` and `INT8`; but the absolute totals understate full-chip power and should not be read as such.

This is consistent with the established observation that data movement and memory accesses, not integer arithmetic, dominate the energy of these designs (HOROWITZ, 2014), and it means the storage contribution that this caveat omits is precisely the part that the precision axis leaves unchanged.

6.3 Maximum Frequency

Table 6.3 reports the shortest clock period each configuration sustains, and the corresponding maximum frequency, from the $F_{\max}$ sweep of Section 5.3.3.

Table 6.3: Minimum period and maximum frequency per configuration.

Configuration	Period (ns)	$F_{\max}$ (MHz)
INT8	3.0	333
INT16	5.0	200
INT4/acc32	2.8	357
INT4/acc16	2.5	400

Source: The Author.

Frequency follows precision in the expected direction: the narrower the datapath, the shorter the critical path through the multipliers and the partial-sum reduction. `INT16` closes only at 5 ns (200 MHz), `INT8` at 3.0 ns (333 MHz), and the `INT4` variants are fastest, with `acc16` reaching 400 MHz. The accumulator-width axis also helps timing: the lighter 16-bit accumulator logic shortens the path enough to move `INT4` from 357 MHz to 400 MHz.

6.4 Throughput

Cycle counts for a full ResNet-50 inference, obtained from cycle-accurate Verilator simulation, are given in Table 6.4, broken down by the operation classes the workload exercises. All four runs pass functional validation.

The cycle counts confirm that they are governed by the tiling, which in turn follows from the storage widths. `INT8` and `INT4/acc32` take essentially the same number of cycles (within 0.01%): they store 8-bit elements and accumulate at 32 bits, so they

Table 6.4: ResNet-50 inference cycle counts, in 10^6 cycles. The last column normalizes the total to the `INT8` baseline.

Configuration	Matmul	Conv	Res-add	Other	Total	Norm.
INT8	35.5	53.4	12.0	0.14	101.0	1.000
INT16	66.4	63.6	15.8	0.34	146.1	1.446
INT4/acc32	35.5	53.4	12.0	0.14	101.0	1.000
INT4/acc16	31.1	55.4	10.6	0.13	97.2	0.962

Source: The Author.

tile identically. `INT16` needs 45% more cycles, because its 16-bit elements halve how many values fit in a scratchpad row, forcing more tiles and more passes. `INT4/acc16`, conversely, is 3.8% faster than `INT8`: the 16-bit accumulator holds twice as many entries for the same byte capacity (the invariance observed in Section 6.1), enabling a larger output tile and fewer accumulator passes. The accumulator-width axis, which barely touched the storage area, thus reappears here as a small but real throughput lever through the tiling it enables.

Combining the cycle counts with each design's own F_{max} gives the wall-clock latency and throughput of Table 6.5.

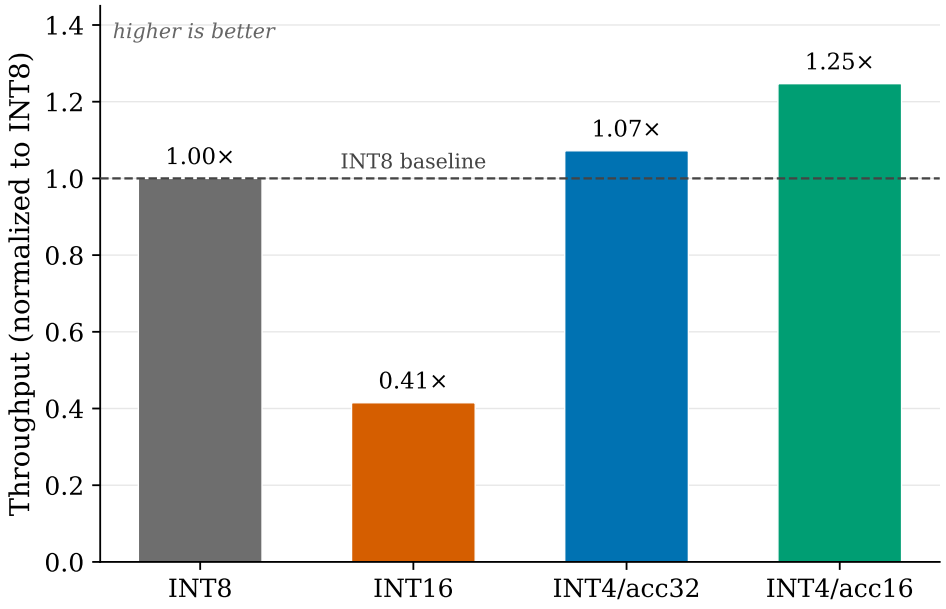
Table 6.5: Latency and throughput for one ResNet-50 inference, each at the configuration's own F_{max}. Throughput is normalized to the `INT8` baseline.

Configuration	F_{max} (MHz)	Cycles (10^6)	Latency (ms)	Throughput (norm.)
INT8	333	101.0	303.1	1.000
INT16	200	146.1	730.7	0.415
INT4/acc32	357	101.0	282.9	1.072
INT4/acc16	400	97.2	243.1	1.247

Source: The Author.

The two effects compound: `INT4/acc16` is both the fastest in cycles and the fastest in frequency, reaching $1.25\times$ the baseline throughput, while `INT16` is penalized on both and delivers only $0.42\times$, as shown in Figure 6.3

Figure 6.3: Normalized ResNet-50 throughput across the four configurations.



Source: The Author

6.5 Energy

Energy per inference is the product of the power drawn at each design's $F_{\max}$ and its latency at that frequency; it is given in Table 6.6. Since the power figures carry the SRAM-macro caveat of Section 6.2, the energy here is the energy of the logic and registers; the dominant scratchpad access energy, which is precision-invariant across `INT4` and `INT8`, is not included.

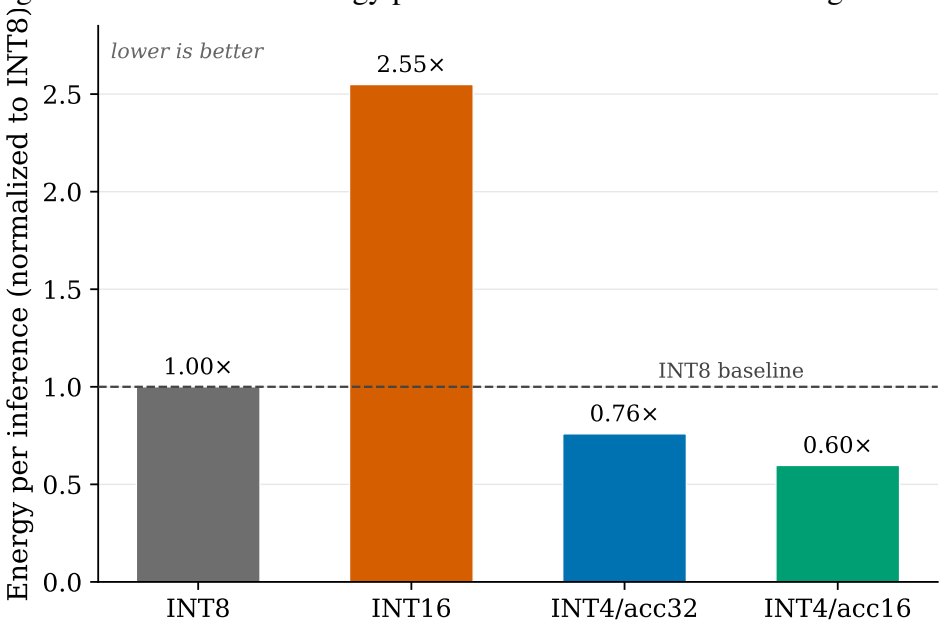
Table 6.6: Energy per inference, normalized to the `INT8` baseline. Power is taken at each configuration's $F_{\max}$.

Configuration	Power@$F_{\max}$ (mW)	Energy/inf. (mJ)	Energy (norm.)
INT8	148.4	45.0	1.000
INT16	156.9	114.6	2.549
INT4/acc32	120.7	34.1	0.759
INT4/acc16	110.5	26.9	0.597

Source: The Author.

Energy separates the configurations more sharply than any single metric, because precision moves power and latency in the same direction. `INT4/acc16` spends 40% less energy per inference than `INT8` and `INT4/acc32` 24% less, while `INT16` spends $2.5\times$ as much.

Figure 6.4: Normalized energy per inference across the four configurations.



Source: The Author

6.6 Accuracy and Cost Trade-off

The hardware costs of the previous sections are only half of the trade-off: their value depends on the classification accuracy each precision sustains. As set out in Section 5.4, this work does not re-measure accuracy but draws it from the quantization literature, expressed as the loss relative to the floating-point baseline (ΔTop-1) for ResNet-50 on ImageNet. This holds for `INT8` exactly as it does for `INT4` and `INT16`: no accuracy figure in this section was produced by running the synthesized configurations on a classification task; every value is a citation.

For convolutional inference, 8-bit integer quantization is essentially lossless: on ResNet-50/ImageNet it stays within a few tenths of a percentage point of the floating-point baseline with standard post-training quantization (JACOB et al., 2018; NAGEL et al., 2021), and 16-bit integer accuracy is indistinguishable from floating point (INCI et al., 2023; NAGEL et al., 2021).

The extra dynamic range of the 16-bit datapath therefore buys essentially no accuracy for this class of networks while incurring the hardware penalties measured here. At the other end, 4-bit quantization is more delicate and generally needs per-channel scaling and quantization-aware training, but with such care it is also near-lossless: learned step-size quantization reaches 76.7% Top-1 on ResNet-50/ImageNet against a 76.8% floating-

point baseline, a loss of only 0.1 pp (ESSER et al., 2020).

Table 6.7: Cited accuracy (ΔTop-1 on ResNet-50/ImageNet) paired with the measured hardware cost, normalized to the `INT8` baseline. The calibration method (PTQ/QAT) is noted.

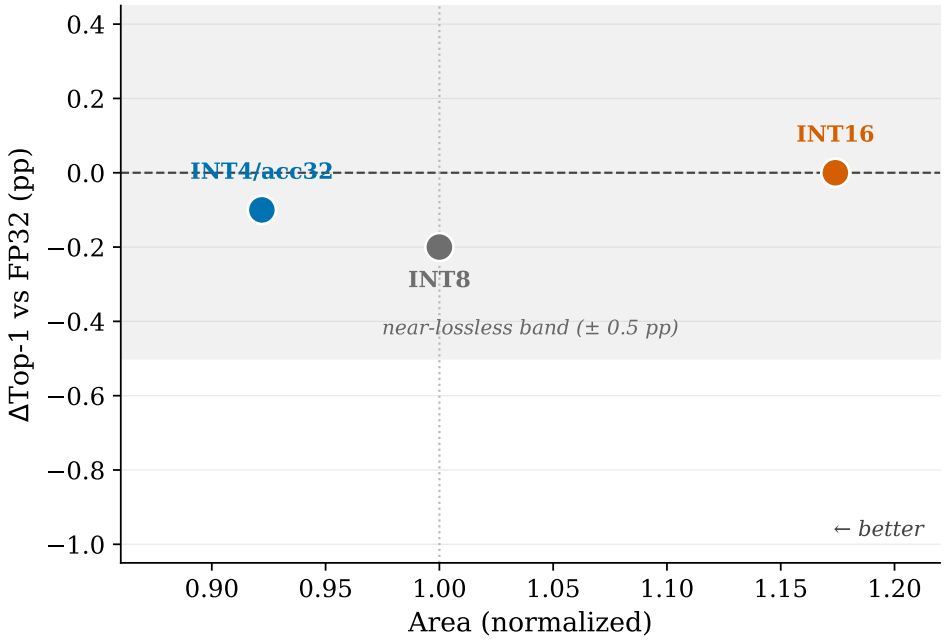
Configuration	Precision	ΔTop-1 (pp)	Area	Throughput	Energy
INT8	INT8	≈ -0.2 (PTQ)	1.000	1.000	1.000
INT16	INT16	$\approx$ 0.0 (lossless)	1.174	0.415	2.549
INT4 / acc32	INT4	-0.1 (QAT)	0.922	1.072	0.759

Source: The Author. Cited accuracies: `INT8` and `INT16` from (NAGEL et al., 2021; INCI et al., 2023); `INT4` from (ESSER et al., 2020).

Table 6.7 collects these cited accuracies alongside the measured hardware cost. The `INT4/acc16` point is deliberately left out of this accuracy comparison: its 16-bit accumulator holds only a single 12-bit in-array partial sum, with no headroom for the cross-tile accumulation over the contraction dimension K (which reaches 9×512 in ResNet-50's convolutions), so it may saturate in a regime the published INT4 accuracies do not cover. Its accuracy is therefore not characterized here, and only its hardware advantage (Sections 6.1–6.5) is retained.

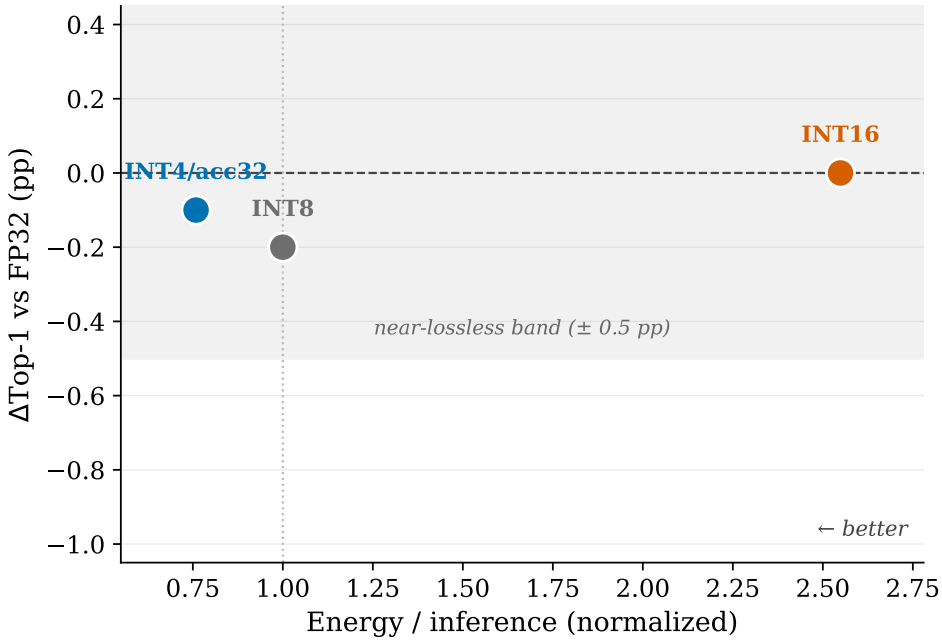
Figures 6.5, 6.6, and 6.7 plot ΔTop-1 against the area, energy, and throughput of each configuration.

Figure 6.5: Cited accuracy (ΔTop-1 on ResNet-50/ImageNet) against the measured cell area, normalized to `INT8`.



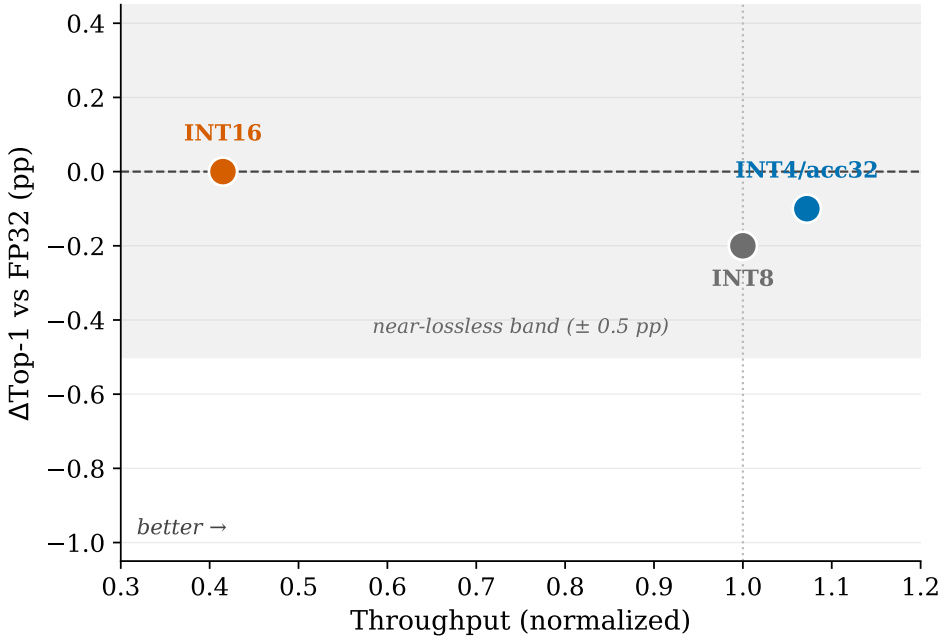
Source: The Author.

Figure 6.6: Cited accuracy (ΔTop-1 on ResNet-50/ImageNet) against the energy per inference, normalized to `INT8`.



Source: The Author.

Figure 6.7: Cited accuracy (ΔTop-1 on ResNet-50/ImageNet) against the throughput, normalized to `INT8`.



Source: The Author.

Across all three axes the picture is the same: the precisions fall within a single near-lossless band, so for ResNet-50 inference the choice between them is governed by hardware cost rather than by accuracy. At that near-equal accuracy the cost separates them

sharply — INT16 has the largest area, spends $2.5\times$ the energy, and delivers only $0.42\times$ the baseline throughput, whereas INT4/acc32 is the smallest, the most energy-efficient, and the highest-throughput of the three.

6.7 Discussion

Read across the two axes, the results form a consistent picture. The *precision* axis is the dominant lever, and it acts on the compute: the mesh area and power scale with operand width ($1.93\times$ area and $2.08\times$ power for INT16, roughly half for INT4), the critical path lengthens or shortens with it, and the storage width sets the tiling and hence the cycle count. Lowering precision from INT8 to INT4 improves every hardware metric at once — smaller, faster, lower power, and equal or better throughput — while raising it to INT16 worsens every one of them. It is worth stressing, however, that the *total*-area gains from precision are consistently smaller than the mesh's own gains, because the byte-capacity-sized scratchpad — roughly half of the total cell area — does not shrink with the datapath: a designer reading only the top-level area row of Table 6.1 would underestimate how much the compute itself actually shrinks at INT4.

The *accumulator-width* axis is a smaller and more localized lever. Its saving — -2.6% area, -12% power, and a 43 MHz frequency gain — comes from the accumulator *logic*: the scale unit shrinks by a third and the adders by half. The accumulator *storage* does not shrink, because it is byte-capacity-sized like the scratchpad; narrowing accType converts that fixed capacity into more entries rather than a smaller array, which is what gives INT4/acc16 its modest cycle-count and throughput edge. The axis is therefore best understood as trading accumulator headroom for lighter arithmetic and finer tiling, at no cost in storage.

The clearest design conclusion concerns INT16. It is dominated on every hardware axis: it is the largest, the most power-hungry, the slowest to close timing, the slowest per inference, and it spends $2.5\times$ the energy of the baseline. Whether that cost is ever justified depends on accuracy, and the accuracy analysis of Section 6.6 shows that it is not: INT16 buys no accuracy over INT8 for this class of networks, so its additional hardware cost is a poor trade for inference.

Taken together, the hardware results and the cited accuracy of Section 6.6 place the inference sweet spot at the low-precision end of the explored space: INT4. With a 32-bit accumulator its near-lossless accuracy is supported directly by the literature, while the

`acc16` variant pushes the hardware metrics further still — the smallest area, the highest throughput, and the lowest energy of any configuration — at an accumulator-saturation risk this work does not quantify and that is therefore left to future verification. `INT8` remains the conservative, near-lossless fallback, and `INT16` a point dominated on every hardware axis, retained only where a particular workload's dynamic range demands it.

Two limitations frame these conclusions. The accuracy of each precision is cited rather than measured, so the cost–benefit argument rests on the hardware results paired with published accuracy; and, as discussed in Section 6.2, the power and energy figures capture the logic and registers but not the SRAM-array power (dynamic and leakage), so absolute energy is understated even though the precision-differentiating component is captured. Both are revisited in Chapter 7.

7 CONCLUSION

This work presented a design space exploration of integer quantization on the Gemmini DNN accelerator (GENC et al., 2021), characterizing how datapath precision and accumulator width affect area, power, frequency, throughput, and energy through a real ASIC logic-synthesis flow with Hammer and Cadence Genus targeting the ASAP7 predictive 7 nm kit paired with cycle-accurate simulation of ResNet-50 in Verilator. To support the precisions under study, the Gemmini template was extended with `INT4` and `INT16` configurations derived from a common `INT8` baseline, and the Mvin and accumulator-scale stages were modified to saturate to the output element range with clip-before-cast semantics, applied consistently in the Chisel hardware and in the C reference so that the simulated and synthesized behavior agree.

The measurements draw a coherent picture across the two axes. The precision axis is the dominant lever, and it acts on the compute: the spatial array grows to $1.93\times$ the baseline area and draws roughly twice its power at `INT16`, and shrinks to about half on both counts at `INT4`, while the critical path and the tiling — and thus the cycle count — follow the operand width in the same direction. Because the scratchpad SRAM is sized by a fixed byte capacity, it stays essentially constant for equal storage width (identical for `INT4` and `INT8`, and only 2.3% larger for `INT16`) and, accounting for about half of the cell area, it damps the total-area swings to $+17\%$ for `INT16` and -8% to -10% for the `INT4` variants.

The accumulator-width axis is a smaller and more localized lever: halving `accType` from `acc32` to `acc16` saves 2.6% of total area and 12% of power and gains 43 MHz of frequency, entirely from the accumulator *logic* — the scale unit and the read–modify–write adders — while the byte-capacity-sized accumulator SRAM is left unchanged. At the system level the effects compound: `INT4/acc16` delivers $1.25\times$ the baseline throughput at $0.60\times$ its energy per inference, whereas `INT16` delivers only $0.42\times$ the throughput at $2.5\times$ the energy.

Paired with the accuracy behavior reported in the quantization literature (JACOB et al., 2018; NAGEL et al., 2021; ESSER et al., 2020), where 8-bit inference is essentially lossless and carefully applied 4-bit quantization remains competitive for convolutional networks, these results position the low-precision end of the explored space as the efficient operating point for ResNet-50 inference: `INT4`. With a 32-bit accumulator its near-lossless accuracy is supported directly by the literature, while the `acc16` variant is the

smallest, fastest, and most energy-efficient configuration measured, at an accumulator-saturation risk this work does not quantify and leaves to future verification. `INT8` is the conservative, near-lossless fallback, and `INT16` a configuration dominated on every hardware axis, justified only where a particular workload's dynamic range demands it.

Furthermore, this thesis contributes a reproducible, open-source methodology for attributing area, power, and energy to individual quantization choices on Gemmini, of use to the open-architecture community.

7.1 Limitations

The conclusions above are bounded by the scope of the methodology, and three limitations should be kept in view.

First, classification accuracy is cited rather than measured. As stated in Chapter 5, the cycle counts are data-independent, so the throughput results do not depend on any particular set of quantized weights; but this also means the accuracy of each precision is taken from the published literature rather than evaluated end-to-end on the same networks. The cost–benefit argument therefore rests on the hardware results measured here paired with external accuracy, not on a single joint measurement.

Second, the power and energy figures characterize the logic and registers, not the storage arrays. The predictive ASAP7 SRAM macros carry no power model — neither dynamic nor leakage — so their power is absent from the report (Section 6.2). The reported totals capture the datapath and control — precisely the part the precision axis changes — but understate full-chip power, since the dominant scratchpad access energy is omitted. The estimates are additionally leakage-dominated, a property of the ASAP7 predictive node, and use default switching activities rather than workload-derived ones.

Third, the results are pre-layout and single-point. Area, frequency, and power come from logic synthesis without place-and-route, so they do not include real wire parasitics, routing congestion, or a built clock tree, and the ASAP7 absolute magnitudes carry the kit's $4\times$ drawn-cell scaling. The exploration also fixes a single 16×16 weight-stationary mesh and a single workload (ResNet-50); other network topologies, mesh dimensions, and dataflows may shift the balance between the compute and storage costs that this study isolates.

7.2 Future Work

Several directions extend naturally from this work. The most direct is to widen the precision axis beyond integers to the floating-point element types Gemmini already supports — FP16 and BF16 — and to mixed, per-layer precision, so that dynamic-range-sensitive workloads can be placed on the same cost–benefit map built here.

A second direction is to carry the configurations through place-and-route, turning the pre-layout synthesis estimates into layout-accurate area, timing, and power, and to complement the vectorless power with workload-driven (activity-based) estimation so that switching power reflects the actual ResNet-50 traffic.

A third is to close the storage-power gap left by the SRAM-macro caveat through gate-level power on the mapped macros. Finally, measuring end-to-end accuracy for each precision, including per-channel scaling and quantization-aware training for the `INT4` configurations, would let the accuracy and the hardware cost be reported jointly on the same networks, replacing the cited accuracy with a measured one and completing the trade-off the exploration sets out to inform.

7.3 Code and Data Availability

The following information was supplied regarding data and code availability:

- **Synthesis and simulation infrastructure:** The Gemmini configurations, the modified scaling stages, the Hammer/Genus configurations files and reporting setup, and the collected area, power, and timing reports are hosted at <https://github.com/matheusalsi/qgemm>.
- **ONNX-to-Gemmini weight converter:** Used to generate the quantized ResNet-50 weights for the `INT16` configuration, this tool is hosted at <https://github.com/matheusalsi/onnx2gemm>.

Together with the original Chipyard weights, these repositories reproduce the design points of Table 5.1 and the measurements reported in Chapter 6.

REFERENCES

AMID, A. et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. **IEEE Micro**, v. 40, n. 4, p. 10–21, jul. 2020. ISSN 1937-4143. Available from Internet: <https://ieeexplore.ieee.org/document/9099108>.

ASHAR, N. et al. QuantMAC: Enhancing Hardware Performance in DNNs With Quantize Enabled Multiply-Accumulate Unit. **IEEE Access**, v. 12, p. 43600–43614, 2024. ISSN 2169-3536. Available from Internet: <https://ieeexplore.ieee.org/document/10476602>.

Berkeley Architecture Research. **Gemmini: Berkeley's Systolic Array Generator**. 2024. <https://github.com/ucb-bar/gemmini>. Available from Internet: <https://github.com/ucb-bar/gemmini>. Accessed in: 4 jun. 2025.

Cadence Design Systems. **Genus Synthesis Solution**. <https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html>. RTL logic synthesis tool. Accessed: 2025-06-08.

CHEN, A. et al. Communication bounds for convolutional neural networks. In: **Proceedings of the Platform for Advanced Scientific Computing Conference**. New York, NY, USA: Association for Computing Machinery, 2022. (PASC '22), p. 1–10. ISBN 978-1-4503-9410-9. Available from Internet: <https://dl.acm.org/doi/10.1145/3539781.3539784>.

CLARK, L. T. et al. ASAP7: A 7-nm finFET predictive process design kit. **Microelectronics Journal**, v. 53, p. 105–115, jul. 2016. ISSN 1879-2391. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S002626921630026X>.

CONG, J.; WANG, J. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In: **2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [s.n.], 2018. p. 1–8. ISSN 1558-2434. Available from Internet: <https://ieeexplore.ieee.org/document/8587678>.

DAI, L. et al. An Energy-Efficient Bit-Split-and-Combination Systolic Accelerator for NAS-Based Multi-Precision Convolution Neural Networks. In: **2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)**. [s.n.], 2022. p. 448–453. ISSN 2153-697X. Available from Internet: <https://ieeexplore.ieee.org/document/9712509>.

ESSER, S. K. et al. **Learned Step Size Quantization**. arXiv, 2020. ArXiv:1902.08153 [cs.LG]. Available from Internet: <http://arxiv.org/abs/1902.08153>.

FINKELSTEIN, A.; ALMOG, U.; GROBMAN, M. **Fighting Quantization Bias With Bias**. arXiv, 2019. ArXiv:1906.03193 [cs]. Available from Internet: <http://arxiv.org/abs/1906.03193>.

GENC, H. et al. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In: **2021 58th ACM/IEEE Design Automation Conference (DAC)**. [s.n.], 2021. p. 769–774. ISSN 0738-100X. Available from Internet: <https://ieeexplore.ieee.org/document/9586216>.

GOOKYI, D. A. N. et al. Deep Learning Accelerators' Configuration Space Exploration Effect on Performance and Resource Utilization: A Gemmini Case Study. **Sensors**, v. 23, n. 5, p. 2380, feb. 2023. ISSN 1424-8220. Available from Internet: <https://www.mdpi.com/1424-8220/23/5/2380>.

GROOTENDORST, M. **A Visual Guide to Quantization**. 2024. Exploring Language Models (newsletter). Available from Internet: <https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization>.

HENNESSY, J. L.; PATTERSON, D. A. A new golden age for computer architecture. **Commun. ACM**, v. 62, n. 2, p. 48–60, jan. 2019. ISSN 0001-0782. Available from Internet: <https://dl.acm.org/doi/10.1145/3282307>.

HOROWITZ, M. 1.1 Computing's energy problem (and what we can do about it). In: **2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)**. [s.n.], 2014. p. 10–14. ISSN 2376-8606. ISSN: 2376-8606. Available from Internet: <https://ieeexplore.ieee.org/document/6757323>.

HUBARA, I. et al. Binarized neural networks. In: **Proceedings of the 30th International Conference on Neural Information Processing Systems**. Red Hook, NY, USA: Curran Associates Inc., 2016. (NIPS'16), p. 4114–4122. ISBN 978-1-5108-3881-9.

INCI, A. et al. QUIDAM: A Framework for Quantization-aware DNN Accelerator and Model Co-Exploration. **ACM Trans. Embed. Comput. Syst.**, v. 22, n. 2, p. 33:1–33:21, jan. 2023. ISSN 1539-9087. Available from Internet: <https://dl.acm.org/doi/10.1145/3555807>.

JACOB, B. et al. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In: **2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition**. [s.n.], 2018. p. 2704–2713. ISSN 2575-7075. Available from Internet: <https://ieeexplore.ieee.org/document/8578384>.

JONES, N. How to stop data centres from gobbling up the world's electricity. **Nature**, Nature Publishing Group, v. 561, n. 7722, p. 163–166, sep. 2018. Bandiera_abtest: a Cg_type: News Feature Subject_term: Energy, Engineering, Research data, Computer science. Available from Internet: <https://www.nature.com/articles/d41586-018-06610-y>.

JOUPPI, N. P. et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In: **Proceedings of the 44th Annual International Symposium on Computer Architecture**. New York, NY, USA: Association for Computing Machinery, 2017. (ISCA '17), p. 1–12. ISBN 978-1-4503-4892-8. Available from Internet: <https://dl.acm.org/doi/10.1145/3079856.3080246>.

KIM, N. et al. Leakage current: Moore's law meets static power. **Computer**, v. 36, n. 12, p. 68–75, dec. 2003. ISSN 1558-0814. Available from Internet: <https://ieeexplore.ieee.org/document/1250885>.

Kung. Why systolic architectures? **Computer**, v. 15, n. 1, p. 37–46, jan. 1982. ISSN 1558-0814. Available from Internet: <https://ieeexplore.ieee.org/document/1653825>.

LI, G. et al. LUT-DLA: Lookup Table as Efficient Extreme Low-Bit Deep Learning Accelerator. In: **2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [s.n.], 2025. p. 671–684. ISSN 2378-203X. Available from Internet: <https://ieeexplore.ieee.org/document/10946705>.

LIEW, H. et al. Hammer: a modular and reusable physical design flow tool: invited. In: **Proceedings of the 59th ACM/IEEE Design Automation Conference**. New York, NY, USA: Association for Computing Machinery, 2022. (DAC '22), p. 1335–1338. ISBN 978-1-4503-9142-9. Available from Internet: <https://dl.acm.org/doi/10.1145/3489517.3530672>.

LIU, W. et al. A survey of deep neural network architectures and their applications. **Neurocomputing**, v. 234, p. 11–26, abr. 2017. ISSN 0925-2312. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0925231216315533>.

MASADEH, M.; HASAN, O.; TAHAR, S. Input-Conscious Approximate Multiply-Accumulate (MAC) Unit for Energy-Efficiency. **IEEE Access**, v. 7, p. 147129–147142, 2019. ISSN 2169-3536. Available from Internet: <https://ieeexplore.ieee.org/document/8863375>.

MELLER, E. et al. Same, Same But Different: Recovering Neural Network Quantization Error Through Weight Factorization. In: **Proceedings of the 36th International Conference on Machine Learning**. PMLR, 2019. p. 4486–4495. ISSN 2640-3498. Available from Internet: <https://proceedings.mlr.press/v97/meller19a.html>.

MOREAU, T. et al. A Hardware–Software Blueprint for Flexible Deep Learning Specialization. **IEEE Micro**, v. 39, n. 5, p. 8–16, sep. 2019. ISSN 1937-4143. Available from Internet: <https://ieeexplore.ieee.org/abstract/document/8764458>.

NAGEL, M. et al. **A White Paper on Neural Network Quantization**. arXiv, 2021. ArXiv:2106.08295 [cs]. Available from Internet: <http://arxiv.org/abs/2106.08295>.

PECCIA, F. N.; BRINGMANN, O. Integration of a systolic array based hardware accelerator into a DNN operator auto-tuning framework. In: **Proceedings of the 2023 Workshop on Compilers, Deployment, and Tooling for Edge AI**. New York, NY, USA: Association for Computing Machinery, 2024. (CODAI '23), p. 21–26. ISBN 979-8-4007-0337-9. Available from Internet: <https://doi.org/10.1145/3615338.3618130>.

PECCIA, F. N. et al. Efficient Edge AI: Deploying Convolutional Neural Networks on FPGA with the Gemmini Accelerator. In: **2024 27th Euromicro Conference on Digital System Design (DSD)**. [s.n.], 2024. p. 418–426. ISSN 2771-2508. Available from Internet: <https://ieeexplore.ieee.org/document/10741661>.

ROY, K.; MUKHOPADHYAY, S.; MAHMOODI-MEIMAND, H. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits. **Proceedings of the IEEE**, v. 91, n. 2, p. 305–327, feb. 2003. ISSN 1558-2256. Available from Internet: <https://ieeexplore.ieee.org/document/1182065>.

SHARMA, H. et al. From high-level deep neural models to FPGAs. In: **2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [s.n.], 2016. p. 1–12. Available from Internet: <https://ieeexplore.ieee.org/document/7783720>.

SHARMA, H. et al. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In: **2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)**. [s.n.], 2018. p. 764–775. ISSN 2575-713X. Available from Internet: <https://ieeexplore.ieee.org/document/8416871>.

SIJSTERMANS, F. The nvidia deep learning accelerator. In: **Hot Chips**. [S.l.: s.n.], 2018. v. 30, p. 19–21.

SkyWater Technology Foundry; Google. **SkyWater Open Source PDK**. 2022. <https://github.com/google/skywater-pdk>. Open-source 130 nm process design kit. Accessed: 2025-06-08.

SNYDER, W. **Verilator**. <https://www.veripool.org/verilator/>. Open-source Verilog/SystemVerilog cycle-accurate simulator. Accessed: 2025-06-08.

STATIC Timing Analysis for Nanometer Designs: A Practical Approach. Boston, MA: Springer US, 2009. ISBN 978-0-387-93819-6 978-0-387-93820-2. Available from Internet: <https://link.springer.com/10.1007/978-0-387-93820-2>.

STRUBELL, E.; GANESH, A.; MCCALLUM, A. Energy and Policy Considerations for Deep Learning in NLP. In: KORHONEN, A.; TRAUM, D.; MàRQUEZ, L. (Ed.). **Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics**. Florence, Italy: Association for Computational Linguistics, 2019. p. 3645–3650. Available from Internet: <https://aclanthology.org/P19-1355/>.

UFRGS. **Centro de Infraestrutura Computacional para Microeletrônica (CADMicro)**. [S.l.], 2025. Acesso em: 1 out. 2025. Available from Internet: <https://www.ufrgs.br/cadmicro>.

VENKATESAN, R. et al. MAGNet: A Modular Accelerator Generator for Neural Networks. In: **2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [s.n.], 2019. p. 1–8. ISSN 1558-2434. Available from Internet: <https://ieeexplore.ieee.org/document/8942127>.

WESTE, N. H. E.; HARRIS, D. M. **CMOS VLSI Design: A Circuits and Systems Perspective**. 4th. ed. Boston, MA, USA: Addison-Wesley Publishing Company, 2011. ISBN 9780321547743.

XU, R. et al. A Survey of Design and Optimization for Systolic Array-based DNN Accelerators. **ACM Computing Surveys**, v. 56, n. 1, p. 20:1–20:37, aug. 2023. ISSN 0360-0300. Available from Internet: <https://dl.acm.org/doi/10.1145/3604802>.

ZHANG, X. et al. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. In: **2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [s.n.], 2018. p. 1–8. ISSN 1558-2434. Available from Internet: <https://ieeexplore.ieee.org/document/8587697>.

APPENDIX A — SYNTHESIS ENVIRONMENT AND CONFIGURATION

This appendix records the setup used to reproduce the logic-synthesis results of Chapter 6. The synthesis was carried out on the CADMicro infrastructure at UFRGS (UFRGS, 2025); access to the EDA tools and to the ASAP7 process design kit is granted by CADMicro. The PDK install paths shown below are reached through that authorization.

Because the flow is driven by Hammer (LIEW et al., 2022), there is no hand-written Tcl script: the configuration is expressed as a set of YAML files that Hammer consumes to generate and invoke the underlying Cadence Genus (Cadence Design Systems,) commands.

The listings that follow reproduce the configuration relevant to the front-end (synthesis) flow used in this work. The complete files, together with the step-by-step build instructions, are available in the project repository (<https://github.com/matheusalsi/qgemm>).

A.1 Tool and License Configuration

Listing A.1 configures the build system and the synthesis tool (Section 5.3).

```
# Build system and synthesis tool
vlsi.core.build_system: make
vlsi.core.synthesis_tool: "hammer.synthesis.genus"
synthesis.genus.genus_bin: "/tools/cadence/DDIEXPORT23/GENUS231/tools.
    lnx86/bin/genus"
synthesis.genus.version: "231"
```

Listing A.1: CADMicro Cadence Tools configuration file.

A.2 ASAP7 Technology Configuration

Listing A.2 is the specific Hammer configuration file that selects the ASAP7 predictive 7 nm technology plugin (CLARK et al., 2016) and its SRAM compiler, points to the local PDK paths, and sets the common 5 ns clock used for the iso-frequency sweep; the 0.5 ns uncertainty is the 10% margin of Section 5.3.3.

```
# Technology: ASAP7
vlsi.core.technology: "hammer.technology.asap7"
technology.asap7.tarball_dir: "/pdk/asu/asap7"
technology.asap7.pdk_install_dir: "/pdk/asu/asap7/asap7_pdk_r1p7"
technology.asap7.stdcell_install_dir: "/pdk/asu/asap7/asap7sc7p5t_27"
vlsi.core.max_threads: 12

# Power spec (Hammer auto-generates a CPF)
vlsi.inputs.power_spec_mode: "auto"
vlsi.inputs.power_spec_type: "cpf"

# Clock: 5 ns period with a 10% (0.5 ns) uncertainty margin
vlsi.inputs.clocks: [
  {name: "clock", period: "5ns", uncertainty: "0.5ns"}
]

# Build system and SRAM compiler
vlsi.core.build_system: make
vlsi.core.sram_generator_tool: "hammer.technology.asap7.sram_compiler"
```

Listing A.2: ASAP7 configuration file (PDK paths require CADMicro authorization).