

Article

Hardware-Validated HLS Engines and Design-Time HBM Partitioning for AI Inference on AMD Alveo V80

Andrei-Alexandru Ulmămei *  and Vlad-Gabriel Șerbu

Department of Electronic Devices, Circuits and Architectures, National University of Science and Technology Politehnica Bucharest, 060042 Bucharest, Romania; vlad_gabriel.serbu@upb.ro

* Correspondence: alexandru.ulmamei@upb.ro

Abstract

Field-programmable gate arrays (FPGAs) with on-package high-bandwidth memory (HBM) are an attractive substrate for low-latency, precision-customizable AI inference, yet high-level synthesis (HLS) flows expose little control over how tensors are distributed across many independent memory channels. The AMD Alveo V80 spreads its nominal 820 GB/s across 64 pseudo-channels, each capped at 12.8 GB/s, so delivered bandwidth is governed by an explicit tensor-to-channel partitioning decision that existing framework-based flows do not surface. This paper presents a methodology for HLS-based inference on the V80 built on two coupled contributions: a design-time partitioning framework that assigns each tensor to on-chip (BRAM/URAM) storage, a single HBM channel, or a stripe across several channels according to its reuse and access pattern, and a three-stage HLS flow that separates functional baseline, unroll-and-partition throughput extraction, and precision-aware DSP packing. The methodology is developed through five model kernels—a parameterized GEMM, ResNet-18, ViT-Small, a BERT attention block, and GPT-2 Small—and realized as nine engines spanning GEMM (FP32, INT8, INT4), convolution, attention, layer normalization, SoftMax, pooling, and embedding lookup. All nine engines are synthesized in Vitis HLS 2024.2 and placed, routed, and executed on the physical Alveo V80 at 400 MHz (2.5 ns period), and every engine closes timing with positive worst-case slack. We report per-engine cycle counts, post-route utilization, and post-route dynamic-power estimates, and compare engine latency and energy against NVIDIA Titan RTX (GPU) and Intel Xeon W-3223 (CPU) baselines on identical kernels. A central result, confirmed on silicon, is that INT8 roughly halves the GEMM DSP58 footprint relative to FP32 (27 to 14 slices), whereas INT4 yields no further compute reduction and acts purely as a memory-placement lever. The full-model compositions, the roofline classification, and the tensor-to-channel placement framework are analytical, design-time results rather than end-to-end measured performance. The bandwidth-scaling premise underlying the placement framework is confirmed directly on the V80: a 1-to-64 channel sweep shows aggregate HBM bandwidth scaling near-linearly to within 5–13% of the device peak, and placing the GPT-2 LM-head operand on a single channel versus the eight the framework assigns it yields a measured $6.81 \times$ speedup—a direct on-board test of the striping decision. The resulting design guidelines target HLS practitioners working with HBM-equipped FPGAs.



Academic Editor: Tee Hui Teo

Received: 17 June 2026

Revised: 4 July 2026

Accepted: 5 July 2026

Published: 14 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article

distributed under the terms and

conditions of the [Creative Commons](#)

[Attribution \(CC BY\)](#) license.

Keywords: FPGA; HLS; AI inference; HBM; quantization; Versal; deep learning acceleration; reconfigurable computing; hardware validation; power efficiency

1. Introduction

Deep learning inference has become one of the dominant workloads in modern computing infrastructure, spanning image classification, object detection, large-scale language modeling, and personalized recommendation across both data-center and edge deployments. FPGAs occupy a distinctive position among the hardware platforms serving these workloads: they offer post-fabrication reconfigurability, fine-grained datapath customization, support for arbitrary numerical precision, and low-latency deterministic execution. Historically, however, FPGAs were constrained for AI inference by limited off-chip memory bandwidth, which throttled the execution of models whose weights could not fit on-chip.

The integration of HBM directly on the FPGA package has changed this constraint fundamentally. The AMD Alveo V80, built around a Versal HBM adaptive system-on-chip (SoC) in a 7 nm process, integrates 2.6 million lookup tables (LUTs), 10,848 DSP58 slices, and 64 HBM2e pseudo-channels delivering an aggregate 820 GB/s of memory bandwidth across 32 GB of on-package capacity [1,2]. The on-package memory is HBM2e, and every subsequent reference to HBM in this paper denotes this HBM2e subsystem. Yet this aggregate figure is misleading as a design target: it is the sum of 64 independent 12.8 GB/s channels, and a kernel realizes only the bandwidth of the channels it actually binds and accesses efficiently. A tensor mapped to a single channel is capped at 12.8 GB/s regardless of the 820 GB/s nominal peak. A tensor that must be read with poor burst locality realizes far less. Translating the platform's nominal bandwidth into delivered performance is therefore not an implementation detail but a central design problem—one of deciding which tensors reside on-chip, which are mapped to HBM, and how the latter are distributed across the 64 channels.

HLS tools, in particular AMD Vitis HLS, allow FPGA kernels to be expressed in C++ with pragma-directed control over pipelining, memory binding, and parallelism, substantially reducing design effort relative to register-transfer level (RTL) development. A substantial body of work has used HLS to deploy neural networks on FPGAs through automated, framework-based flows such as hls4ml [3], FINN [4], and fpgaConvNet [5]. These frameworks prioritize automation and typically target a single model family with a fixed accelerator template. They do not expose, or systematically reason about, the tensor-to-channel partitioning decisions that govern bandwidth utilization on a many-channel HBM device. To our knowledge, the 64-channel topology of the V80 has not previously been treated as a workload-level tensor-partitioning problem for HLS-based AI inference, although we do not claim an exhaustive survey of the rapidly growing FPGA LLM-accelerator literature. Prior HBM-equipped Alveo platforms such as the U50, U55C, and U280 [6–8] expose 32 pseudo-channels, and the partitioning strategies developed for them do not transfer directly to a device with twice the channel count and a different fabric generation.

This paper addresses that gap with a methodology for HLS-based AI inference on the Alveo V80 whose engine library is validated on hardware and whose model-level composition and tensor-partitioning layers are design-time analytical, organized around two coupled contributions. First, we develop a framework for *HBM partitioning*: a principled basis for deciding, per tensor, between on-chip (BRAM/URAM) and off-chip (HBM) placement, for assigning HBM-resident tensors to channels, and for striping bandwidth-critical tensors across multiple channels. Second, we present a three-stage *HLS methodology*: a correctness-first baseline, a throughput stage based on loop unrolling and local-memory partitioning, and a precision stage based on verified DSP58 packing. HBM channel binding is treated as a placement lever rather than as a separate compute-latency optimization. The two contributions are coupled because partitioning decisions are expressed and realized through the same HLS interfaces, storage pragmas, and local-buffer choices used by the kernels.

The proposed methodology is developed through five model kernels, chosen so that each exposes a distinct partitioning challenge rather than to maximize benchmark coverage: a parameterized quantized general matrix multiplication (GEMM) engine, which establishes the compute-bound ceiling and the DSP-packing baseline; ResNet-18, a streaming convolutional network with high weight reuse; ViT-Small, a vision transformer whose attention matrices raise an on-chip-versus-off-chip buffering decision; a BERT attention block, whose parallel query/key/value projections raise a channel-allocation decision; and GPT-2 Small (prefill), whose tied vocabulary matrix exposes both token-indexed lookup and bandwidth-dominant LM-head streaming access. These kernels decompose into a library of nine reusable engines—GEMM in FP32, INT8, and INT4; convolution; attention; layer normalization; SoftMax; pooling; and embedding lookup—which are the unit of implementation and measurement throughout the paper. All nine engines are written in Vitis HLS 2024.2, placed and routed in Vivado 2024.2, and executed on the physical Alveo V80 at 400 MHz. Per-engine post-route timing, utilization, and power are reported as the primary results and interpreted through a roofline-based analytical model. Engine latency and energy are then compared against GPU and CPU baselines on identical kernels, which delineates the workloads for which the HLS-based FPGA approach is energy-competitive rather than asserting a uniform advantage.

- We formulate the HBM partitioning problem for AI inference on a 64-channel FPGA and present a design-time partitioning framework spanning on-chip/off-chip placement, channel assignment, and striping, grounded in a per-tensor reuse and bandwidth analysis.
- We present a three-stage HLS methodology for the Versal HBM fabric and characterize, through synthesis and on-silicon measurement, which optimization levers reduce compute latency, which affect only placement, and which collapse into the same scheduler outcome on the V80.
- We report per-engine post-route timing, resource utilization, and power for nine HLS engines placed, routed, and executed on the physical Alveo V80 at 400 MHz, providing a hardware-measured dataset for this platform; all nine engines close timing with positive worst-case slack.
- We classify each kernel and precision into compute- and memory-bound regimes using a roofline model anchored by the measured resource and power figures, and compare per-engine latency and energy against NVIDIA Titan RTX (GPU) and Intel Xeon W-3223 (CPU) baselines on identical kernels, identifying the engines for which the FPGA is energy-competitive and those for which it is not.
- We distill the results into a set of partitioning and optimization guidelines intended to transfer to other HBM-equipped FPGAs with appropriate scaling of channel count and bandwidth.

The remainder of this paper is organized as follows. Section 2 reviews related work in FPGA-based inference acceleration, HBM characterization, and FPGA quantization. Section 3 describes the V80 platform, the HLS toolchain, and the analytical performance model. Section 4 defines the mathematical operations implemented by the engine library. Section 5 presents the three-stage HLS methodology. Section 6 presents the HBM partitioning framework and the per-kernel channel maps. Section 7 reports synthesis results and the roofline analysis. Section 8 distills design guidelines, and Section 9 concludes.

2. Related Work

This section reviews four threads of prior work that frame the contributions of this paper: HLS-based frameworks for FPGA neural-network inference, HBM characterization and exploitation on data-center FPGAs, transformer and large-language-model acceleration

on FPGAs, and quantization techniques targeting FPGA datapath efficiency. We position this work at their intersection: a methodology with a hardware-validated engine library that treats tensor-to-channel partitioning on a 64-channel HBM device as a first-class design-time problem across structurally distinct model families, rather than as an implementation detail of a single-family accelerator.

2.1. HLS Frameworks for FPGA Neural-Network Inference

The use of high-level synthesis to deploy neural networks on FPGAs has been explored extensively over the past decade. The hls4ml framework, originally developed for ultra-low-latency inference in particle physics trigger systems [3], automatically translates Keras 3.5.0 and PyTorch 2.4.0 models into Vitis HLS 2024.2 C++, achieving sub-microsecond inference latency for compact networks; subsequent extensions added convolutional support [9] and a broader FPGA/ASIC codesign workflow [10]. The FINN framework [4] pioneered binarized neural-network inference on FPGAs, and its successor FINN-R [11] generalized this to arbitrary low-precision quantization with explicit design space exploration. The fpgaConvNet toolflow [5] targets convolutional networks through a synchronous dataflow formalism that navigates the compute–memory–throughput trade-off. AMD’s commercial Vitis AI flow, built around a fixed deep processing unit (DPU) overlay, provides a turnkey deployment path for quantized CNNs on UltraScale+ and Versal devices [12].

These frameworks prioritize automation over microarchitectural specialization, and—critically for the present work—they abstract away the memory system: tensor placement and off-chip bandwidth management are handled implicitly by a fixed template or overlay rather than exposed as design decisions. On a device whose nominal bandwidth is distributed across 64 independent channels, those implicit decisions determine the bulk of achievable performance. This paper takes a complementary approach, using manual HLS to make tensor-to-channel partitioning explicit and to reason about it systematically.

2.2. HBM Characterization and Exploitation on FPGAs

The introduction of HBM-equipped FPGAs created an immediate need for empirical characterization, since theoretical peak bandwidth rarely translates directly into application throughput. Wang et al. [6] introduced Shuhai, a benchmarking framework on the Alveo U280 that characterized achievable bandwidth across all 32 pseudo-channels and documented the substantial gap between nominal and realized bandwidth. Choi et al. [7] extended this analysis to HLS-generated designs, showing that HLS imposes overhead beyond the raw HBM controller and proposing access-pattern optimizations that materially improve effective bandwidth. The same group’s HBM Connect [13] addressed the built-in crossbar’s limitations when many processing elements share many channels, through a high-throughput HLS-friendly interconnect. Huang et al. [14] demonstrated a fully pipelined VGG-16 accelerator on the U280 that exploited HBM for a large bandwidth advantage over DDR4-based equivalents.

This body of work establishes that realized HBM bandwidth depends heavily on how data are mapped to channels and accessed, but it approaches the problem from the perspective of microbenchmarks [6,7] or general-purpose interconnect [13], not from that of a workload-specific partitioning strategy for complete neural networks. Moreover, all of these studies target 32-channel U50/U280 devices with a nominal peak near 460 GB/s. The V80 doubles the channel count to 64 and raises the aggregate peak to 820 GB/s with HBM2e, which both enlarges the partitioning search space and changes its structure: a single tensor’s bandwidth ceiling is still one channel, so the additional channels are useful only if tensors are deliberately distributed across them. The framework developed here

addresses this workload-level mapping problem, which the prior microbenchmark and interconnect studies do not.

2.3. Transformer and Large-Language-Model Acceleration on FPGAs

The scaling of transformer-based models has motivated substantial FPGA acceleration work. FTRANS [15] introduced an energy-efficient transformer accelerator using block-circulant matrix compression, evaluated on BERT-class models. DFX [8] built a multi-FPGA GPT-2 inference appliance across four U280 boards with fully utilized HBM channels, reporting significant speedup and energy-efficiency gains over a multi-GPU baseline. FlightLLM [16] demonstrated complete LLaMA2-7B and OPT-6.7B inference on a single U280 using a configurable sparse DSP chain and an on-chip decode scheme, and mapped the same architecture to a Versal VHK158 device. Sun et al. [17] accelerated Vision Transformers through an FPGA-aware mixed-scheme quantization framework with per-layer precision allocation.

These systems represent the state of the art in single-family FPGA inference, and several—DFX and FlightLLM in particular—make sophisticated use of HBM channels. They differ from the present work in intent. Each is engineered to maximize performance for one model class, with a memory mapping hand-tuned for that class and reported as part of a finished design. None abstracts the underlying placement decisions into a reusable framework, nor examines how those decisions differ across structurally distinct workloads. Our contribution is orthogonal and complementary: rather than a faster accelerator for one family, we provide a design-time partitioning framework, illustrated across five kernels spanning convolutional, vision-transformer, attention, and embedding-dominated workloads, that exposes *why* a given tensor belongs on-chip, on one channel, or striped across many.

2.4. Quantization for FPGA Datapath Efficiency

FPGA inference work has consistently exploited reduced precision to extract throughput from a fixed silicon budget. Beyond FINN's binarization, mixed-precision schemes tailored to FPGA datapaths have received sustained attention. The MSP framework [18] combined fixed-point and powers-of-two representations to exploit both DSP and LUT resources. FILM-QNN [19] assigned different bitwidths to different filters within a layer to balance accuracy and parallelism. DeepBurning-MixQ [20] co-optimized DSP packing with quantization-aware design space exploration, recognizing that bitwidth and DSP allocation must be chosen jointly. The present work uses the same broad idea—precision changes both arithmetic and memory cost—but verifies the actual V80 behavior directly against the DSP58 architecture and HLS synthesis. The Versal DSP58 provides an INT8 vector dot-product mode using three 9-by-8 two's-complement products per DSP58 [21]; we do not assume a native INT4 arithmetic mode.

We adopt FP32, INT8, and INT4 as precision modes, treating precision not as an accuracy-optimization target—the prior work above already addresses mixed-precision accuracy in depth—but as a lever that changes a kernel's arithmetic intensity and therefore its position relative to the memory and compute roofs. Reducing precision can shrink the DSP cost per operation and always shrinks the bytes transferred per inference. A central result of this paper is that these effects must be separated: INT8 reduces the measured GEMM DSP footprint, whereas INT4 provides no additional measured DSP reduction on the V80 and acts primarily as a memory-placement lever.

2.5. Summary and Positioning

Three observations recur across these threads. First, HLS substantially lowers FPGA development effort but abstracts away the memory-mapping decisions that, on a many-

channel HBM device, dominate achievable bandwidth. Second, realized HBM bandwidth has been characterized at the microbenchmark and interconnect level, but not as a per-tensor partitioning problem for complete networks across model families. Third, reduced precision is established as a throughput lever, but its interaction with memory mapping—specifically, its effect on which side of the roofline a workload lands on—has received little explicit treatment. This paper addresses these gaps for the 64-channel V80 by coupling an HBM partitioning framework with a three-stage HLS methodology and a roofline analysis spanning five structurally distinct kernels.

Stated directly, the novelty of this work is neither a faster accelerator nor a new bandwidth benchmark, but the treatment of tensor-to-channel placement as a first-class, reusable design-time decision on a 64-channel device. Prior HBM characterization work [6,7,13] measures or improves raw channel bandwidth at the microbenchmark or interconnect level; it establishes that placement matters but does not prescribe, per tensor, where a given weight or activation should reside. Prior single-family accelerators [8,15,16] make sophisticated use of HBM channels, but each hand-tunes one memory mapping for one model class and reports it as a finished artifact, without isolating the placement decision or testing how it changes across structurally different workloads. The contribution here is the decision rule itself—classify each tensor by residency, access pattern, reuse, and service bandwidth, then place it on chip, on a single channel, or striped across several—together with the evidence that makes it credible: a nine-engine library validated on silicon, a per-tensor application of the rule across five model families spanning convolutional, vision-transformer, attention, and embedding-dominated access, an on-board channel-scaling sweep, and a direct on-board placement experiment that measures a $6.81 \times$ difference between the prescribed and the single-channel placement of the same tensor. To our knowledge, this per-tensor, cross-family treatment—on the doubled 64-channel V80 fabric rather than the 32-channel devices of prior work—has not been presented before. The precision-as-placement result, that INT4 is a memory lever rather than a further compute-packing step on the DSP58, is a second and smaller novelty in the same spirit: it reclassifies a precision knob as a placement knob.

3. Platform and Methodology

This section describes the target FPGA platform, the HLS toolchain and implementation flow used to obtain all reported figures, the nine engine kernels through which the methodology is developed and validated, and the analytical performance model used to interpret the results. Primary performance figures—per-engine cycle counts, post-route timing, resource utilization, and power—are obtained from physical execution and post-route analysis on the Alveo V80. Composition bounds and roofline regime classifications are derived analytically from these measured engine characterizations.

3.1. Target Platform: AMD Alveo V80

The target platform is the AMD Alveo V80 compute accelerator card [1], built around the AMD Versal HBM XCV80 adaptive system-on-chip (SoC) in a 7 nm process. The XCV80 provides 2.6 million lookup tables (LUTs), 10,848 DSP58 slices, 132 Mb of block RAM (BRAM), and 541 Mb of UltraRAM (URAM) in its programmable logic fabric. Taken together, the reported BRAM and URAM capacities provide approximately 84.1 MiB of on-chip storage available to HLS kernels before implementation overheads. It integrates 32 GB of HBM2e in a dual-stack configuration accessed through 64 pseudo-channels, each 256 bits wide. At the 400 MHz target frequency used throughout this paper, each channel sustains $256 \text{ bit} \times 400 \text{ MHz} = 12.8 \text{ GB/s}$, for an aggregate peak of $64 \times 12.8 = 819.2 \text{ GB/s}$, which we round to 820 GB/s. Host connectivity is via PCIe Gen5; the card carries a maximum thermal design power of 190 W. We use AMD's

AVED V80 documentation only to cross-check the shell and channel context, not as a source of performance measurements for the results in this paper [2].

The 64-channel structure, rather than the aggregate bandwidth figure, is the salient platform characteristic for this work. Because each channel is an independent 12.8 GB/s port, a tensor mapped to a single channel cannot exceed that rate, regardless of the 820 GB/s aggregate, and the aggregate is approached only when many tensors, or many slices of one tensor, are accessed concurrently across distinct channels. The partitioning framework of Section 6 is concerned precisely with making that distribution explicit.

3.2. HLS Toolchain and Synthesis Flow

All kernels are written in C++ and synthesized with AMD Vitis HLS 2024.2, targeting the XCV80 device at a 2.5 ns clock period (400 MHz). The flow for each engine proceeds in four stages. First, C-simulation verifies functional correctness against a reference output generated by the corresponding PyTorch 2.4.0 model, and correctness is checked under a per-kernel maximum-absolute-error tolerance, ensuring that each optimization level preserves output fidelity. Second, C-synthesis produces latency estimates, and a preliminary resource report, and synthesis logs are inspected for warnings indicating that a requested pragma was not honored, and only designs free of such warnings for the relevant construct proceed. Third, each engine is placed and routed in Vivado 2024.2 at the 2.5 ns target, and the post-route timing summary provides the achieved worst-case slack (WNS) per engine. Fourth, the placed-and-routed design is executed on the physical Alveo V80: cycle counts are measured from an on-chip free-running counter latched on the `ap_start/ap_done` handshakes in the engine's 400 MHz clock domain (detailed in Section 7.1), and per-engine power is extracted from Vivado's post-route power analysis (`report_power`).

From each post-route implementation, we record: the WNS from the timing summary, the kernel latency in cycles from on-board measurement, the achieved clock frequency (all engines target 400 MHz), the utilization of LUTs, flip-flops, DSP58 slices, BRAM, and URAM, and the total on-chip power estimate from `report_power`. These measured figures are the primary reported results. Loop initiation intervals and the DSP binding assignments are additionally inspected in the C-synthesis log to validate the multiplier-to-DSP mapping assumed by the analytical model.

3.3. Channel Binding, Compiler Configuration, and Clocking

Unlike earlier Alveo accelerators, the V80 is not programmed through the Vitis/XRT `.xclbin` acceleration flow, which the card does not support. It follows the AVED-based flow [2], in which each Vitis HLS engine is exported as RTL and integrated into a Vivado block design on the AVED shell. Two consequences follow for the design flow. First, the association between a kernel AXI4-MM interface and a physical HBM pseudo-channel is user-managed rather than XRT-managed: each `m_axi` master is routed through the Versal programmable network-on-chip (NoC) to a specific HBM controller pseudo-channel by an explicit, design-time NoC configuration in the block design, not by an `--connectivity.sp` directive. Channel binding is therefore fixed at implementation time and fully under designer control, which is what makes the per-tensor channel maps of Section 6 realizable as concrete interface assignments. Second, because there is no `v++` link step, no associated linker optimization level (such as `-O2` or `-O3`) applies. Optimization is instead expressed at the HLS level through the 2.5 ns clock target and the pipelining and array-partitioning pragmas of Section 5, and at the Vivado level through the place-and-route strategy. Vitis HLS is run with its default scheduling at the stated clock target, with no non-default `config_compile` or `config_interface` options beyond the per-engine interface and pipeline pragmas described in Section 5.

The 400 MHz operating point is likewise specified manually rather than through an XRT clock directive: the 2.5 ns period is given to Vitis HLS during synthesis and to the Vivado clocking resources during implementation. Each standalone engine is characterized in a single user clock domain at 400 MHz that drives both the compute datapath and its AXI4-MM front end. The hardened HBM controller manages its memory-side (PHY) timing internally, so the clock-domain crossing to the HBM2e stacks lies inside the controller and not in the synthesized user logic. Running the BRAM/URAM-backed compute logic in a slower clock domain than the HBM AXI front end is a recognized lever for reducing dynamic power, but it requires a multi-domain integrated design. The single-domain characterizations reported here do not exploit it, and multi-frequency power optimization is identified as part of the integrated-accelerator work in Section 8.

3.4. Kernels

The methodology is developed through five kernels, listed in Table 1, chosen so that each exposes a distinct partitioning challenge rather than to maximize benchmark coverage. The parameterized quantized GEMM engine is a pure compute primitive that isolates DSP-packing behavior across precisions and establishes the compute-bound ceiling. ResNet-18 is a streaming convolutional network in which weights exhibit high reuse and are candidates for on-chip residence. ViT-Small introduces attention matrices whose size relative to the on-chip memory budget governs an on-chip-versus-off-chip buffering decision. The BERT attention block poses a channel-allocation decision among its parallel query, key, and value projections and its feed-forward weights. GPT-2 Small (prefill) is dominated by a large tied vocabulary matrix whose lookup and LM-head uses require distinct placement treatment. Together, these span convolutional, vision-transformer, attention, and embedding-dominated access patterns.

Table 1. [Specification.] Kernels through which the methodology is developed. Parameter counts are reported at batch size one; transformer sequence lengths are specified in the input column.

Kernel	Family	Partitioning Challenge	Params (M)	Input
GEMM (packing baseline)	Compute	DSP packing baseline	—	fixed tile
ResNet-18	CNN	On-chip weight reuse	11.7	$3 \times 224 \times 224$
ViT-Small	ViT	On-chip attention buffer	22.1	$3 \times 224 \times 224$
BERT-attn	Transformer	Parallel projection channels	7.0	128 tokens
GPT-2 Small	Transformer	Embedding-table striping	117.0	128 tokens

Pre-trained weights for ResNet-18, ViT-Small, the BERT attention block, and GPT-2 Small are obtained from torchvision 0.19.0, timm 1.0.11, and HuggingFace Transformers 4.45.0, respectively, and exported to disk for use as C-simulation references; the GEMM kernel uses parameterized random inputs. We do not re-evaluate model accuracy: published task metrics are cited from the original works where relevant. Quantization to INT8 and INT4 is applied to the exported weights offline, so that the kernels operate on pre-quantized data and the synthesis figures reflect the corresponding datapaths.

3.5. Analytical Performance Model

We interpret the synthesis results through a roofline model of the XCV80 [22]. The nominal compute ceiling for a given precision p can be written as

$$F_{\text{peak}}(p) = N_{\text{DSP}} \times m(p) \times f, \quad (1)$$

where $N_{\text{DSP}} = 10,848$, f is the relevant engine clock, and $m(p)$ is the arithmetic work a DSP58 sustains under the chosen datapath. For this paper, however, we avoid substituting

a theoretical INT4 packing factor. We use the post-route hardware-measured GEMM result in Table 2: the equal-schedule packed GEMM uses 27 DSP58 slices in FP32, 14 in INT8, and 14 in INT4, confirmed by on-device execution on the physical V80. Thus, the effective compute-packing factor relative to the FP32 GEMM is 1.0 for FP32 and $27/14 = 1.93$ for both INT8 and INT4. INT4 is therefore modeled as compute-equivalent to INT8 and memory-lighter than INT8, not as an additional DSP-packing mode. For reference, the AM004 dot-product specification implies a nominal INT8 factor of 3.0 (three INT8 products per DSP58); the roofline analysis of Section 7 draws both this nominal ceiling and the $1.93 \times$ ceiling actually achieved, and the gap between them is the HLS packing overhead. Both horizontal lines are *resource-normalized device ceilings*: they scale the hardware-measured per-DSP58 packing factor of the implemented GEMM schedule to the full $N_{\text{DSP}} = 10,848$ budget. They are therefore neither datasheet theoretical peaks nor measured all-DSP throughputs; no engine in this paper instantiates more than a few tens of DSP58 slices. They instead report the throughput the device would sustain if the measured per-slice packing efficiency were replicated across the entire DSP array. This distinction matters when reading the roofline of Section 7. The per-engine operating points lie far below the ceilings mainly because each engine occupies only 14–27 of the 10,848 slices, so the vertical gap is dominated by DSP under-occupancy (replication headroom, quantified in Section 8) rather than by arithmetic-intensity placement. The device-level HBM ceiling is 820 GB/s, but the memory ceiling relevant to a kernel or tensor is its effective assigned-channel bandwidth,

$$B_{\text{eff}}(k) = N_{\text{ch}}(k) BW_{\text{ch}} \eta(k), \quad (2)$$

where $N_{\text{ch}}(k)$ is the number of HBM channels assigned to the tensor or kernel, $BW_{\text{ch}} = 12.8\text{GB/s}$ is the per-channel peak bandwidth, and $\eta(k)$ denotes the access-efficiency factor. Unless otherwise stated, the reported roofline classifications use ideal access efficiency and should therefore be interpreted as upper-bound classifications.

Table 2. [Measured: post-route.] DSP-packing microbenchmark (post-route, Alveo V80): GEMM DSP58, LUT, and FF per precision, used to establish the packing rule (FP32 to INT8/INT4) in the analytical model. This is a standalone packing microbenchmark and is *distinct* from the lean engine-library GEMM characterized in the Results (Section 7.1): the two share the same DSP58 packing (27 to 14) by construction, but the microbenchmark carries additional harness logic and therefore reports a larger LUT/FF footprint. All rows use the same tiled schedule and achieved inner-loop $\Pi = 1$.

Precision	DSP58	LUT	FF	Compute Factor vs. FP32
FP32	27	10,750	11,793	1.00
INT8	14	10,664	8554	1.93
INT4	14	10,242	8203	1.93

For each kernel, we compute an arithmetic intensity

$$I = \frac{\text{operations per inference}}{\text{bytes transferred to/from HBM per inference}}, \quad (3)$$

where the numerator follows from the kernel's dimensions and the denominator is determined by the partitioning strategy of Section 6: only tensors resident in HBM contribute to the byte count, and the relevant precision sets the bytes per element. The roofline then classifies the kernel as compute-bound when $I > F_{\text{peak}}(p)/B_{\text{eff}}(k)$ (the ridge point) and memory-bound otherwise.

Estimated throughput is obtained from the synthesis-reported kernel latency L in cycles as operations/ (L/f) . We report this synthesis-based estimate rather than the roofline ceiling itself; the roofline is used to classify each kernel's regime and to explain the gap

between the estimate and the relevant ceiling, not as a source of performance numbers. This distinction is important: the roofline ceilings are analytical upper bounds, whereas the throughput figures we tabulate derive from the HLS scheduler’s latency estimate for the actual synthesized design. Table 3 summarizes the evidence type—measured, estimated, or modeled—behind each principal quantity reported in this paper, and is the key to the evidence tags carried by every subsequent table and figure caption.

Table 3. Evidence type for the principal quantities reported in this paper. Quantities set in bold are obtained by direct hardware measurement, as distinct from estimated (post-route or C-synthesis) and modeled values.

Quantity	Source	Status
Kernel latency (cycle count)	On-board start/done counter	Measured
Post-route timing (WNS)	Vivado report_timing_summary	Measured
Resource utilization (LUT/FF/DSP/BRAM/URAM)	Vivado post-route utilization report	Measured
Per-engine power	Vivado report_power (post-route)	Estimated (post-route)
GEMM projection latency (transformer linear layers)	Vitis HLS C-synthesis report (128 × 768 × 768)	Estimated (C-synthesis)
Composition bounds	Analytical model over measured engine latencies and the C-synthesis GEMM projection	Modeled
HBM bandwidth	Platform specification and channel model	Theoretical peak/assigned-channel bound
DSP multiplier binding (attention)	C-synthesis and post-route Vivado audit	Measured (confirmed; not provisional)

4. Engine Mathematical Definitions

The engine library is constructed from a small set of mathematical operators that cover the dominant compute and memory-access patterns in the five target models. This section defines those operators before describing their HLS realization. The purpose is not to introduce new neural-network algorithms, but to make explicit which reductions, elementwise operations, and tensor accesses each HLS engine must implement.

4.1. Dense Matrix Multiplication and Linear Projections

The GEMM engine implements

$$C_{m,n} = \sum_{k=0}^{K-1} A_{m,k} B_{k,n}, \quad 0 \leq m < M, \quad 0 \leq n < N. \quad (4)$$

This operator is used directly for the parameterized GEMM study and as the mathematical form of transformer linear layers. For a sequence tensor $X \in \mathbb{R}^{T \times D_{\text{in}}}$ and weight matrix $W \in \mathbb{R}^{D_{\text{in}} \times D_{\text{out}}}$, the projection is

$$Y_{t,j} = \sum_{i=0}^{D_{\text{in}}-1} X_{t,i} W_{i,j} + b_j. \quad (5)$$

Query, key, value, output, feed-forward network (FFN), and language-model head (LM-head) projections all reduce to Equation (5); only T , D_{in} , and D_{out} change. The HLS consequence is a three-deep reduction nest over rows, columns, and the shared dimension. The packing experiments isolate this operator because it gives a clean equal-schedule comparison across FP32, INT8, and INT4.

4.2. Two-Dimensional Convolution

The convolution engine implements the standard spatial convolution

$$Y_{n,k,o_h,o_w} = b_k + \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} W_{k,c,r,s} X_{n,c,o_h u+r-p_h, o_w u+s-p_w}, \quad (6)$$

where u is the stride and (p_h, p_w) is the padding. Out-of-range input coordinates contribute zero. The output dimensions are

$$O_h = \left\lfloor \frac{H + 2p_h - R}{u} \right\rfloor + 1, \quad O_w = \left\lfloor \frac{W_{in} + 2p_w - S}{u} \right\rfloor + 1. \quad (7)$$

The arithmetic reduction is over (c, r, s) for every output pixel and output channel. In the INT8 convolution engine, this channel reduction is grouped in triples to match the DSP58 dot-product datapath, while any $C \bmod 3$ remainder is accumulated explicitly. This grouping changes the HLS datapath but not Equation (6).

4.3. Attention Core

A transformer block first forms query, key, and value tensors using the projection operator in Equation (5). For one attention head of width d_h , the attention core then computes

$$S_{i,j} = \frac{1}{\sqrt{d_h}} \sum_{\ell=0}^{d_h-1} Q_{i,\ell} K_{j,\ell} + M_{i,j}, \quad (8)$$

$$m_i = \max_j S_{i,j}, \quad (9)$$

$$P_{i,j} = \frac{\exp(S_{i,j} - m_i)}{\sum_{r=0}^{T-1} \exp(S_{i,r} - m_i)}, \quad (10)$$

$$O_{i,\ell} = \sum_{j=0}^{T-1} P_{i,j} V_{j,\ell}. \quad (11)$$

The mask $M_{i,j}$ is zero for visible tokens and a large negative value for disallowed tokens; in the GPT-2 prefill case, this implements the causal constraint $j \leq i$. The HLS attention engine implements Equations (8)–(11) as separate stages: query-key (QK) score generation, row-wise SoftMax, and attention-value (AV) reduction. This decomposition is the mathematical reason the final attention engine avoids the monolithic SoftMax-between-reductions loop structure that violated the perfect-nest structure required for efficient HLS scheduling.

4.4. SoftMax and Layer Normalization

The standalone SoftMax engine computes the same row operator as Equation (10) for a vector x :

$$\text{softmax}(x)_i = \frac{\exp(x_i - x_{\max})}{\sum_j \exp(x_j - x_{\max})}, \quad x_{\max} = \max_j x_j. \quad (12)$$

The subtraction of $x_{\max}$ is mathematically neutral and improves numerical range. The HLS implementation, therefore, needs a maximum reduction, an exponential-and-sum pass, and a normalization pass. Layer normalization is applied independently to each row vector $x \in \mathbb{R}^D$:

$$\mu = \frac{1}{D} \sum_{i=0}^{D-1} x_i, \quad (13)$$

$$\sigma^2 = \frac{1}{D} \sum_{i=0}^{D-1} (x_i - \mu)^2, \quad (14)$$

$$y_i = \gamma_i \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta_i. \quad (15)$$

Thus the engine requires two reductions followed by an elementwise affine transform. The gamma and beta vectors are small, repeatedly reused static tensors; staging them into separate local memories follows directly from Equation (15), where both are consumed in the same output pass.

4.5. Pooling

The pooling engine covers the two pooling operators used by the ResNet path. Max pooling over a $R \times S$ window is

$$Y_{n,c,o_h,o_w} = \max_{0 \leq r < R, 0 \leq s < S} X_{n,c,o_h u + r - p_h, o_w u + s - p_w}. \quad (16)$$

Global average pooling reduces each feature map to one value:

$$y_{n,c} = \frac{1}{HW_{in}} \sum_{h=0}^{H-1} \sum_{w=0}^{W_{in}-1} X_{n,c,h,w}. \quad (17)$$

Max pooling is a local read-window problem rather than a multiply-reduction problem, which explains why its HLS bottleneck is memory-port contention. Global average pooling is a pure reduction over spatial positions.

4.6. Embedding Lookup and Tied Output Projection

GPT-2 uses a token-embedding table $E \in \mathbb{R}^{V \times D}$ and a position table $P \in \mathbb{R}^{T_{max} \times D}$. For token index z_t , the embedding engine computes

$$X_{t,d} = E_{z_t,d} + P_{t,d}. \quad (18)$$

The position term is sequential in t , but the token term is indexed by data-dependent vocabulary IDs. This distinction creates the random-access versus streaming placement split used later by the HBM framework.

When the language-model head is tied to the embedding table, the output logits are

$$\ell_{t,v} = \sum_{d=0}^{D-1} H_{t,d} E_{v,d}. \quad (19)$$

Equation (19) is again a GEMM, but over the full vocabulary dimension. The same physical tensor E therefore appears as a random lookup operand in Equation (18) and as a large streaming GEMM operand in Equation (19); the framework treats these as distinct accesses even though they refer to the same weight table.

4.7. Quantized Arithmetic

Quantized tensors are produced offline and presented to the HLS kernels as integer arrays. Conceptually, a real value x is represented as

$$q = \text{clip}_{[q_{\min}, q_{\max}]} \left(\text{round} \left(\frac{x}{s} \right) + z \right), \quad \hat{x} = s(q - z), \quad (20)$$

where s is the scale and z is the zero point. The synthesis experiments in this paper do not evaluate accuracy under different quantizers; they use pre-quantized operands to characterize datapaths, resource use, and byte pressure. In the analytical composition, INT8 reduces the synthesis-reported GEMM DSP footprint, whereas INT4 provides no additional synthesis-reported DSP reduction. The verified DSP58 result in Section 5 then determines whether those byte reductions also correspond to a compute-packing reduction.

4.8. Quantization Scheme

The quantizer instantiating Equation (20) in this work is symmetric, so every zero point is identically zero ($z = 0$) and no zero-point arithmetic appears in any datapath. Weight scales are computed offline from the trained weights: per output channel (reduction over all non-output axes) for the convolutional ResNet-18 weights, and per tensor for the GEMM operands. Activations are quantized per tensor with a dynamic scale taken from the running tensor maximum. INT8 uses signed 8-bit operands, and INT4 signed 4-bit operands for both weights and activations in the ResNet and GEMM integer paths; the LayerNorm, SoftMax, attention, pooling, and embedding engines are characterized in FP32. Each integer-GEMM and convolution datapath accumulates in a 32-bit signed accumulator (ap_int<32>). The characterized standalone engines compute the integer multiply–accumulate and emit the raw 32-bit accumulator: the affine rescale returning the result to the operand range, and any bias addition, are applied outside the characterized core—offline for the exported weights, or in a separate requantization stage in the integrated layer kernels. The per-engine resource, latency, and power figures in Section 7.1 are therefore the cost of the integer multiply–accumulate core, not of a fused quantize–accumulate–requantize layer. Rounding in the offline quantizer is round-to-nearest with saturation to the signed range; the integer accumulators themselves are not saturated, the 32-bit width being sufficient for the tensor shapes used. INT4 is applied to both operands in the ResNet and GEMM paths, whereas in the integrated BERT block, INT4 is restricted to the Q/K/V/O projections with the feed-forward layers kept at INT8; we report these configurations as characterized and do not claim a single uniform INT4 datapath across all kernels. Consistent with the datapath-and-resource scope of this paper, model task accuracy is not re-evaluated; the quantization configuration is specified here to characterize the synthesized integer datapaths, not to assert deployable inference quality.

5. HLS Methodology

The HLS flow maps the operators in Section 4 into synthesizable loop nests and memory interfaces. In this paper, that flow is best understood as three stages: a correctness-first baseline; a throughput stage based on loop pipelining, unrolling, and local array partitioning; and a precision stage based on verified DSP packing. Two cross-cutting observations frame these stages. HBM channel binding creates distinct memory bundles but is latency-neutral for the compute-bound engine reports characterized here, and double buffering is useful only when external memory service time, not the scheduled compute loop, is on the critical path. The methodology below, therefore, separates the decisions that changed the schedule from those that changed only placement.

5.1. Stage 1: Baseline Engine Characterization

Each kernel is first reduced to a standalone engine with a C-simulation testbench and an HLS report. The engine library includes GEMM, convolution, attention, layer normalization, SoftMax, pooling, and embedding lookup. A result is used only if the C-simulation passes and the synthesis report has finite latency, nonzero resources where arithmetic is expected, and no unresolved structural warning for the construct under discussion. This excludes auxiliary exploratory outputs and the earlier monolithic whole-model kernels, which are retained only as motivation for why the engine decomposition was needed.

The attention engine is the clearest example. A direct monolithic SoftMax-between-reductions structure violated the perfect-nest structure required for efficient HLS scheduling. The final attention engine separates the computation into QK score generation, row-wise SoftMax, AV reduction, and output storage, with the attention matrix materialized in local BRAM. The corresponding synthesis reports contain no remaining non-perfect-nest warning for the QK and AV reductions, and C-simulation passes for sequence lengths 128 and 197, including the causal GPT-2 case.

5.2. Stage 2: Unroll and Partition for Throughput

The second stage exposes parallelism only where the memory system and local banking can feed it. In GEMM, the tiled schedule uses cyclic partitioning of the local A and B tiles and complete partitioning of the C tile lanes, which allows the inner loop to reach $II = 1$. In convolution, the INT8 engine groups input channels by three to match the DSP58 dot-product datapath, partitions the local activation and weight buffers accordingly, and handles non-multiple-of-three channel remainders explicitly. Across the seven convolution sweep points, the same source structure is used, and the packed dot-product binding remains active.

LayerNorm shows the complementary storage lesson. Staging gamma and beta into separate local BRAM arrays removes the previous external-port contention in the normalize loop and restores $II = 1$. Pooling shows the limit of the rule: maxpool remains at $II = 9$ because the 3-by-3 window requests contend for the same input memory port, and that cost is reported rather than hidden. These examples make the throughput stage a co-optimization of unroll factor, local banking, and access schedule, rather than the independent application of pragmas.

HBM channel binding belongs in this stage only as an interface-placement mechanism. The fixed-GEMM projection binds A, B, and C to distinct HBM bundles, and HLS generates separate AXI master interfaces for those bundles. The synthesis-estimated latency, however, remains governed by the scheduled tiled compute loop. Thus, HBM binding is necessary for exposing bandwidth to a full accelerator composition, but in these compute-bound engine reports, it does not by itself reduce compute latency.

5.3. Stage 3: Precision and DSP Packing

The final stage changes the datatype and packing. AM004 specifies the relevant DSP58 integer mode: the INT8 vector dot-product unit computes a three-element dot product using 9-bit by 8-bit two's-complement products per DSP58 [21]. There is no native INT4 dot-product mode to exploit in the same way. The verified GEMM packing experiment therefore provides the compute rule used by the rest of the paper: FP32 uses 27 DSP58 slices, INT8 uses 14, and INT4 also uses 14 under the same schedule (Table 2). INT8 gives a real compute-resource benefit. INT4 does not reduce compute resources further; its role is to halve tensor bytes relative to INT8 and thereby alter placement pressure.

The convolution sweep generalizes that the DSP58 dot-product primitive can be inferred inside a nested convolution reduction, but it does not provide a clean equal-schedule unpacked baseline. We therefore use GEMM for the quantitative packing number and use convolution only as evidence that the packed form is robust in another dense engine.

6. HBM Partitioning Framework

This section presents the *design-time partitioning framework* whose central bandwidth-scaling premise is confirmed on board. Its predictions are placement *classes*—on-chip, single-channel, or striped across several channels—checked for consistency against the channel maps the kernels actually use. The premise underlying striping, that the V80’s pseudo-channels deliver independent, additive bandwidth up to the $64 \times 256\text{-bit} \times f$ aggregate, is validated directly by the on-board channel sweep of Section 6.1; the per-tensor class assignments themselves are checked at the class level rather than against per-tensor on-board placement. As Section 7.1 shows, the batch-one schedules characterized here do not by themselves stress HBM bandwidth: the required bandwidth stays well below a single channel for every modeled case. The framework is therefore a prescriptive tool for the bandwidth-bound regimes that a replicated, higher-throughput accelerator would enter, rather than a model fitted to measured bandwidth pressure. We state this scope up front so that the leave-one-out analysis later in this section is read as a class-level consistency check; the channel-scaling behavior it assumes is the part established empirically. To avoid ambiguity about the nature of the contribution, we state it explicitly. The framework is a *design-time partitioning framework*: a per-tensor decision rule that classifies each weight or activation by residency, access pattern, reuse, and required service bandwidth, and assigns it to one of three placements—resident on chip, bound to a single HBM channel, or striped across several. It is not a runtime allocation or scheduling algorithm; the placement is fixed before synthesis and does not adapt during execution. Nor do we claim it as a theoretical advance over the memory mappings used in prior single-family accelerators, which make placement decisions of the same kind but make them implicitly and per design. The contribution is to make that decision explicit, reusable, and applicable per tensor across structurally distinct workloads, and to validate both the premise it rests on (additive per-channel bandwidth) and one of its placement decisions directly on silicon (Sections 6.1 and 6.2). Throughout, we reserve the term *design-time partitioning framework* for this placement artifact and *three-stage HLS methodology* for the separate engine-construction process of Section 3; the two are distinct, and we do not use *methodology*, *guidance*, and *framework* interchangeably between them.

The main design problem on the V80 is not whether the device has enough aggregate bandwidth, but whether each tensor is assigned to a storage class that matches its lifetime and access pattern. The composition results show zero modeled activation spill for all reported batch-one cases. Intermediate activations for ResNet-18, ViT-Small, BERT, and GPT-2 are smaller than the approximately 84.1 MiB on-chip BRAM+URAM budget, so producer-consumer activations should remain on chip. Once activations are resident, HBM pressure moves to static tensors: convolution weights, transformer projection and FFN weights, embeddings, and output logits. The decisive property is the access pattern. A tensor with regular streaming access can exploit wide bursts and channel striping, whereas a token-indexed embedding lookup cannot, even when each selected row is contiguous. The embedding engine exhibits this distinction directly in synthesis: HLS widened the word token-embedding (WTE) row reads but could not analyze the outer token-indexed WTE access, while the word position embedding (WPE) and output traffic were widened as sequential streams. This synthesis evidence independently supports the streaming-versus-gather split and motivates the three placement classes in Table 4.

Table 4. [Framework definition.] Tensor placement classes used by the HBM partitioning framework.

Placement Class	Use
On-chip	Live intermediate activations and small repeated parameters when the live set fits in BRAM/URAM. LayerNorm gamma/beta are staged here to remove port contention.
Single HBM channel	Regular externally stored tensors whose required bandwidth fits one channel, block-local weight shards, or random/gather accesses without a deterministic banking function.
Striped across HBM channels	Large streaming or statically strided tensors whose required service bandwidth, size, or parallel consumers require more than one channel while preserving useful burst length.

For a tensor or tensor shard, define its size S , reuse count R , arithmetic intensity AI , access pattern P , live activation footprint A_{live} , target service time T_{target} , average burst length $B_{\text{burst,obs}}$, and externally chosen parallel shard count N_{parallel} . The platform parameters are the HBM channel count C , per-channel bandwidth BW_{ch} , on-chip capacity B_{on} , reserved on-chip capacity B_{reserve} , and minimum useful burst length B_{burst} . Let

$$B_{\text{avail}} = B_{\text{on}} - B_{\text{reserve}}, \quad BW_{\text{req}} = \frac{S}{T_{\text{target}}}. \quad (21)$$

The decision procedure is:

1. If P is an intermediate activation and $A_{\text{live}} \leq B_{\text{avail}}$, place the tensor on chip. If it does not fit, spill it to HBM and stripe only if the spill access is burstable.
2. If P is a small parameter and $S + A_{\text{live}} \leq B_{\text{avail}}$, stage it on chip. Otherwise, place it on one HBM channel.
3. If P is random or row-gather and no deterministic banking function is available, classify the access as a single-channel or random-bucket access. If the same physical tensor also has a dominant streaming use, classify that physical storage by the dominant streaming use and the gather sub-use separately.
4. If P is streaming or statically strided, choose

$$N = \min\left(C, \max\left(1, \left\lceil \frac{BW_{\text{req}}}{BW_{\text{ch}}} \right\rceil, N_{\text{size}}, N_{\text{parallel}}\right)\right), \quad (22)$$

where N_{size} is greater than one, only for a dominant high-reuse tensor too large to keep resident. If $N > 1$ and $B_{\text{burst,obs}} \geq B_{\text{burst}}$, stripe across N channels; otherwise keep a single-channel shard.

The framework predicts placement class. Exact stripe count is not determined from tensor size alone; it also depends on the parallel schedule, host-side layout, and DMA packing plan. GPT-2 WTE illustrates the caveat: the framework predicts that the physical WTE/tied-LM-head matrix should be striped because the LM-head use is a dominant streaming GEMM operand, while the token-embedding sub-use remains a random row-gather access.

Table 5 applies the decision procedure to each tensor and checks the predicted placement class against the choice encoded in the corresponding per-kernel channel map. This is a consistency check: because the rule and the channel maps are derived from the same per-tensor analysis, agreement confirms that the framework is well defined and applied uniformly across structurally distinct kernels, rather than constituting independent validation. One row is stronger than a consistency check—the GPT-2 WTE/WPE pair, whose predicted access classes are corroborated by independent synthesis-log evidence (widened streaming WPE reads versus an unanalyzed token-indexed WTE loop).

Table 5. [Analytical: consistency check.] Consistency check of the partitioning framework: the decision procedure of Section 6 applied to each tensor, compared against the placement encoded in the per-kernel channel maps.

Item	Features Exposed	Predicted Class	Reference Decision
Intermediate activations	Activation live sets below 84.1 MiB	On chip	Composition reports zero activation spill.
LayerNorm gamma/beta	Tiny repeated parameters	On chip/staged	LayerNorm engine stages both arrays in local BRAM and achieves normalize II = 1.
ResNet layer4 weights	Regular convolution weights, channel-sized shard	Single-channel shard	Channel map assigns layer4 weights to one HBM group.
ViT encoder block pair	Regular transformer block weights, about 14.2 MiB FP32 per two-block shard	Single-channel shard	Channel map spreads block pairs across HBM0–HBM5.
BERT FC1/FC2	Regular dense FFN weights	Single-channel shard	Channel map separates FC1 and FC2 channels.
GPT-2 decoder block pair	Regular block weights, about 56.7 MiB FP32 per two-block shard	Single-channel shard	Channel map assigns two-block shards to HBM8–HBM13.
GPT-2 WTE lookup	Token-dependent row bases with contiguous selected rows	Random-bucket/single-channel access	Embedding report shows widened rows but unanalyzed token loop (independent evidence).
GPT-2 WTE/tied LM-head physical tensor	Large dominant streaming GEMM operand	Striped	Channel map stripes WTE over HBM0–HBM7.
GPT-2 WPE	Sequential position table, small size	Single channel	Embedding report shows widened streaming WPE reads (independent evidence).

6.1. On-Board HBM Bandwidth Scaling

The framework’s central mechanism—that distributing a streaming tensor across more HBM channels raises delivered bandwidth proportionally—rests on the assumption that the V80’s pseudo-channels deliver independent, additive bandwidth up to the $64 \times 256\text{-bit} \times f$ aggregate. We tested that assumption directly on the device with a synthetic read-bandwidth kernel: 64 independent 256-bit AXI masters, each bound to a distinct HBM pseudo-channel, with a runtime control selecting how many channels are active. Sweeping the active-channel count from 1 to 64 and timing steady-state device execution gives the delivered aggregate bandwidth in Figure 1.

Delivered bandwidth scales near-linearly with active-channel count across the full range, tracking the per-channel $256\text{-bit} \times 400\text{MHz} = 12.8\text{GB/s}$ reference with no saturation knee before 64 channels. At 64 active channels the two builds deliver 11.14 and 12.16GB/s per channel—87% and 95% of the per-channel ceiling—for aggregates of approximately 713 and 778GB/s against the 819GB/s design peak. This confirms two assumptions the rest of the paper relies on: that the $64 \times 256\text{-bit} \times f$ figure used as the HBM ceiling (Section 6 and the memory roofs of Figure 2) is delivered in practice to within 5–13%, and that striping a streaming operand across N channels yields close to $N \times$ the single-channel bandwidth—the premise behind the GPT-2 tied LM-head striping decision in Table 5.

The near-linear scaling in Figure 1 is a direct consequence of the access pattern, and it is worth making explicit why the curve does not bend. Each AXI master in the sweep is bound to its own dedicated pseudo-channel and issues long sequential bursts, so the channels

operate independently and at high row-buffer efficiency: no two masters share a channel, and no master crosses the NoC to a non-local channel. Delivered bandwidth therefore adds across channels, and N active channels reach close to N times the single-channel rate. The sub-linear behavior one might otherwise expect—where effective HBM throughput can fall by roughly a factor of three relative to the sequential-burst peak—arises under the opposite conditions: when several masters contend for one shared channel, when a master crosses the NoC crossbar to a non-local channel, or when accesses are issued as short or randomly addressed bursts that defeat row-buffer locality. Those are precisely the conditions of the random-gather access that the framework excludes from striping (Section 6), which is why the streaming-versus-gather distinction is central to the placement rule rather than an incidental caveat.

The experiment validates the streaming, channel-scaling premise but not the full placement framework. It uses a synthetic sequential-read pattern rather than the model tensors in situ, so it establishes the bandwidth model and the striping benefit for streaming access, but it does not measure the random-gather case, where the framework predicts that striping does *not* help. The streaming-versus-gather distinction therefore remains analytical, and the per-tensor class assignments of Table 5 remain consistency checks rather than measured placements.

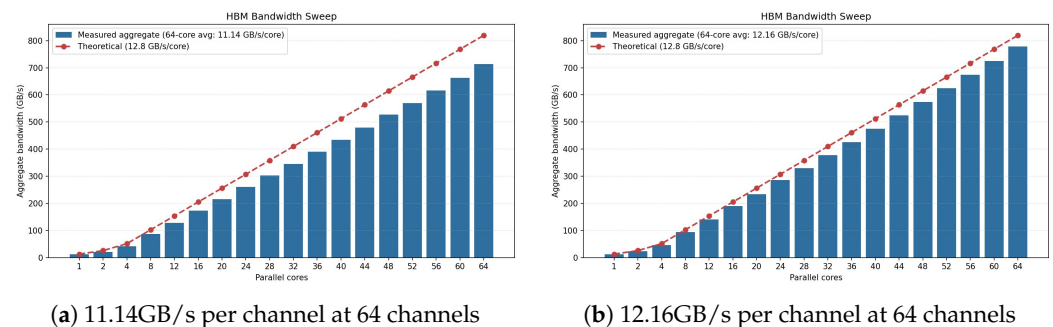


Figure 1. [Measured: on-board.] Measured aggregate HBM read bandwidth on the Alveo V80 versus the number of active 256-bit pseudo-channels, for two kernel-clock builds. Bars are measured steady-state aggregate bandwidth; the dashed line is the per-channel 256-bit $\times$ 400MHz = 12.8GB/s reference (819GB/s aggregate at 64 channels). Both builds scale near-linearly with no saturation before 64 channels, delivering (a) 11.14 and (b) 12.16 GB/s per channel at 64 channels (87% and 95% of the per-channel ceiling).

6.2. Direct Validation of a Placement Decision

The channel sweep of Section 6.1 establishes that striping a streaming tensor across N channels delivers close to $N \times$ the single-channel bandwidth, but it does so on a synthetic kernel rather than on a tensor that the framework actually places. To test a placement *decision* directly—not as a consistency check against a chosen channel map, but as a measured comparison between two placements of the same tensor—we took the one tensor the framework prescribes striping, the GPT-2 tied LM-head weight operand, and ran the batch-one LM-head GEMM engine on the physical V80 under two placements of that operand: bound to a single HBM channel, and striped across the eight channels (HBM0–HBM7) that the channel map of Table 5 assigns to it. The two runs differ only in the channel binding of the weight operand; the kernel, schedule, and clock are identical.

The single-channel engine completes in 698,202,689 cycles (1.746 s at 400 MHz); the eight-channel engine completes the same work in 102,459,635 cycles (0.256 s), a $6.81 \times$ speedup. That is 85% of the ideal $8 \times$, in close agreement with the 87–95% per-channel efficiency measured independently in Section 6.1, and corresponds to an effective delivered bandwidth of approximately 12.8 GB/s for the single channel and 87 GB/s across the eight. The placement lever predicted by the framework is therefore confirmed by on-board measurement on the

exact tensor the framework places, with the measured speedup matching the independently measured channel-scaling efficiency to within a few percent. The single-channel on-board latency is itself within 0.18% of the C-synthesis estimate for this shape (699,432,187 cycles; from the GEMM projection latency calibration of Section 7.2), which confirms that the synthesized schedule corresponds to single-channel HBM service; the eight-channel placement falls $6.81 \times$ below that synthesis estimate. The three figures, therefore, triangulate the kernel across evidence types—an analytical schedule estimate, a single-channel on-board measurement that matches it, and an eight-channel on-board measurement that exposes the bandwidth headroom the idealized-memory synthesis could not.

The result also makes a property of the engine explicit. The $6.81 \times$ gain shows the as-built LM-head engine to be bandwidth-sensitive in practice: at a single channel, it is throttled by HBM service time rather than by the compute schedule. The reason is that the engine, in its present form, does not buffer the 36.8 MiB INT8 weight operand on chip but re-streams it from HBM across the tile schedule, so its delivered HBM traffic is far larger than the single-pass logical footprint. An on-chip weight-reuse buffer would cut the repeated traffic and move the engine toward the schedule-bound regime; adding that buffer, and with it a fused multi-engine LM-head stage, is left to the integrated-accelerator work of Section 8.

7. Results

This section reports results in two tiers. Primary results—per-engine timing, utilization, power, and on-board latency—are obtained from physical execution on the Alveo V80 and from Vivado post-route analysis (Tables 6 and 7; per-engine power is reported in Section 7.6). Derived results—composition bounds, roofline regime classifications, and byte-pressure figures—are obtained from the analytical model applied to the measured engine characterizations. Auxiliary exploratory outputs and earlier calibration artifacts are excluded. To keep this distinction visible at the point of use, every table and figure caption in the paper carries a leading tag stating its evidence type—*measured* (on-board or post-route), *synthesis* (Vitis HLS C-synthesis estimate), *estimated* (post-route report_power), or *analytical* (modeled from measured inputs); Table 3 is the corresponding key for the principal reported quantities.

Table 6. [Measured: on-board cycles; post-route timing.] Per-engine measured timing. Cycle counts are obtained from physical device runs; achieved clock period and worst negative slack (WNS) are taken from the post-route timing summary at the 2.5 ns (400 MHz) target. Latency is computed as cycles $\times$ achieved period.

Engine	Cycles	Target (ns)	WNS (ns)	Achieved f (MHz)	Latency
GEMM FP32	7711	2.5	0.242	400	19.277 μ s
GEMM INT8	7591	2.5	0.251	400	18.977 μ s
GEMM INT4	7579	2.5	0.343	400	18.947 μ s
Convolution (INT8)	22,836,562	2.5	0.199	400	57.091 ms
Attention core	4,268,833	2.5	0.043	400	10.672 ms
LayerNorm	613,316	2.5	0.161	400	1.533 ms
Softmax (seq = 128)	708	2.5	0.409	400	1.77 μ s
Pooling	3,459,966	2.5	0.381	400	8.649 ms
Embedding	98,600	2.5	0.043	400	246.50 μ s

Table 7. [Measured: post-route.] Per-engine post-route resource utilization on the XCV80 at the 2.5 ns target. BRAM is reported in 18-Kb-block (RAMB18) equivalents and URAM in URAM288 blocks. Parenthetical values are the utilization as a percentage of the device total (2,600,000 LUT; 5,148,416 FF; 10,848 DSP58; 7482 RAMB18 blocks $\approx$ 132 Mb BRAM; 541 Mb URAM).

Engine	LUT	FF	DSP58	BRAM	URAM
GEMM FP32	5969 (0.23)	10,068 (0.20)	27 (0.25)	38 (0.51)	0
GEMM INT8	6172 (0.24)	7657 (0.15)	14 (0.13)	38 (0.51)	0
GEMM INT4	5869 (0.23)	7583 (0.15)	14 (0.13)	38 (0.51)	0
Convolution	10,458 (0.40)	12,097 (0.23)	17 (0.16)	27.5 (0.37)	9 (0.47)
Attention core	10,086 (0.39)	9814 (0.19)	6 (0.06)	278 (3.72)	0
LayerNorm	6508 (0.25)	9475 (0.18)	4 (0.04)	29.5 (0.39)	0
Softmax (seq = 128)	5133 (0.20)	5935 (0.12)	5 (0.05)	12.5 (0.17)	0
Pooling	3979 (0.15)	4371 (0.08)	1 (0.01)	5 (0.07)	0
Embedding	7844 (0.30)	7246 (0.14)	1 (0.01)	132 (1.76)	0

7.1. Per-Engine Timing and Resource Utilization

Table 6 reports the hardware-measured timing for the nine HLS engines after place-and-route at the 2.5 ns (400 MHz) target. Cycle counts are obtained from on-board start/done handshake counters during physical execution on the Alveo V80, while the worst negative slack (WNS) and achieved clock period are taken from the post-route timing summary (report_timing_summary).

We describe the latency measurement explicitly to remove any residual ambiguity about what it captures. Each engine uses the Vitis HLS ap_ctrl_hs block-level protocol, and a free-running counter in the engine's own 400 MHz user clock domain is latched on the ap_start handshake and again on the ap_done handshake; the reported cycle count is the difference between the two latched values, read back through the engine's control interface after the engine signals done. The counter, therefore, counts engine clock cycles directly, and the interval it brackets is the full on-device kernel execution: it begins when the engine is released and ends at its done handshake, so it includes the engine's own HBM read traffic and the initial HBM fill to first data. It excludes host-side activity entirely—host-to-card and card-to-host DMA transfers, host synchronization, and AXI control setup all occur outside the latched interval—so the reported figures are device-side kernel execution latency, not end-to-end host-observed latency. Because the engine schedules are data-independent (fixed loop trip counts for the given tensor shapes), the cycle count is exact and repeatable run-to-run rather than a sampled statistic; the latency in Table 6 is that cycle count multiplied by the post-route achieved clock period. This differs from the CPU and GPU latencies of Section 7.7, which are statistical summaries (a median over many timed software iterations) rather than an exact hardware cycle count. All nine engines close timing with positive WNS, ranging from 0.043 ns for the attention core and embedding engines to 0.409 ns for SoftMax, which confirms the 400 MHz frequency target assumed by the analytical model. End-to-end latencies span more than four orders of magnitude, from 1.77 μ s for SoftMax at sequence length 128 to 57.091 ms for the INT8 convolution engine, reflecting the wide range of arithmetic workloads covered by the engine library.

The corresponding post-route resource utilization is reported in Table 7. Each engine occupies a small fraction of the XCV80 budget: peak DSP58 usage is 27 slices for GEMM FP32 (0.25% of 10,848), and BRAM and URAM consumption remain well within the device totals. The most distinctive entries are the attention core, whose 278 BRAM blocks reflect the locally materialized attention matrix, and the embedding engine, whose 132 BRAM blocks back the position-table and output-staging buffers. Comparing the three GEMM rows isolates the DSP-packing effect: the FP32 variant consumes 27 DSP58 slices, while both INT8 and INT4 collapse to 14, confirming on the physical device the packing rule used by the analytical model and detailed in Section 5.

These measured tables anchor the remainder of the section: the composition bounds reported next use the per-engine latencies of Table 6, the roofline classification builds on the DSP and memory footprints in Table 7, and the per-engine power figures in Section 7.6 are extracted from the same post-route implementations.

7.2. GEMM Projection Latency Calibration

Transformer linear layers dominate the model composition bounds, so those bounds are only as sound as the latency assigned to each GEMM projection. Rather than scale a single calibration point by logical MAC count, we directly synthesize the fixed-tile INT8 projection kernel at every distinct shape the composition uses—the $128 \times 768 \times 768$ query, key, value, or output projection; the ViT-Small and BERT/GPT-2 FFN up- and down-projections; the ViT projection; and the tied GPT-2 LM-head ($768 \rightarrow 50, 257$)—and fit an overhead-inclusive scaling law to the resulting C-synthesis cycle counts. All shapes use the same loop body, $16 \times 16 \times 16$ tiling, HBM interface, and array-partition pragmas, target part, and 2.5 ns clock as the anchor kernel; only the compile-time dimensions change. C-simulation passes against the PyTorch 2.4.0 reference. The synthesized points are listed in Table 8.

The cycle counts follow the kernel's tile-loop trip counts rather than its logical MAC count. Writing $I = \lceil M/16 \rceil$, $J = \lceil N/16 \rceil$, and $K_t = \lceil K/16 \rceil$, the C-synthesis cycles for the 16-aligned shapes are reproduced within rounding by

$$\text{cycles} = 1 + I(J(546 K_t + 295) + 2),$$

which separates a per-output-tile overhead of 295 cycles and a per-row-tile overhead of 2 cycles from the MAC-throughput term of 546 cycles per $16 \times 16 \times 16$ tile, i.e., $4096/546 \approx 7.50$ padded INT8 MAC/cycle. The one exception is the non-16-aligned LM-head ($N = 50, 257$), whose output dimension is not a multiple of the tile width and whose C-synthesis cycle count does not reduce to this closed form; for that stage, the directly synthesized value of Table 8 is used. This shape-resolved law, together with the directly synthesized LM-head figure, is the latency primitive entering the composition bounds for every GEMM stage.

The law matters because naive logical-MAC scaling from the anchor is not uniformly safe. For the $M = 197$ ViT shapes, the fixed-tile kernel executes $\lceil 197/16 \rceil = 13$ row tiles, i.e., 208 padded rows for 197 logical rows, so anchor-only MAC scaling underpredicts the FFN up-projection and the projection latency by about 6.3%. The non-16-aligned LM-head is underpredicted by anchor-only scaling by about 4.8%, and because its output dimension breaks the tile-loop closed form, its latency is taken directly from synthesis (Table 8). The $M = 128$, 16-aligned shapes scale almost exactly (within 0.0001%). The tile-loop law absorbs the aligned edge-tile effects, which a single-point extrapolation would have missed.

Table 8. [Synthesis: Vitis HLS C-synthesis.] Multi-point C-synthesis calibration of the fixed-tile INT8 GEMM projection kernel. All rows share the same loop body, $16 \times 16 \times 16$ tiling, pragmas, target part (xcv80-lsva4737-2MHP-e-S), and 2.5 ns clock; only the compile-time M, K, N differ. Cycles are Vitis HLS C-synthesis figures, not on-board measurements. Per-engine resource footprints are in Table 7.

Shape ($M \times K \times N$)	Composition Role	Logical MACs	Synth. Cycles
$128 \times 768 \times 768$	QKVO projection (anchor)	75,497,472	10,177,169
$197 \times 384 \times 384$	ViT projection	29,048,832	4,180,515
$197 \times 384 \times 1536$	ViT FFN up	116,195,328	16,721,979
$197 \times 1536 \times 384$	ViT FFN down	116,195,328	16,445,859
$128 \times 768 \times 3072$	BERT/GPT-2 FFN up	301,989,888	40,708,625
$128 \times 3072 \times 768$	BERT/GPT-2 FFN down	301,989,888	40,368,785
$128 \times 768 \times 50, 257$	GPT-2 tied LM-head	4,940,464,128	699,432,187

7.3. Composition Bounds

Table 9 reports the regenerated sequential-sum upper bound and perfect-overlap lower bound. The gap remains the main scheduling result, but its interpretation is model-specific rather than a blanket statement. ResNet INT8/INT4 is limited by a late 3-by-3 layer4 convolution. ViT-Small and BERT are limited by FFN up-projection GEMMs. GPT-2 Small is limited by the tied LM-head GEMM. The regenerated data show ratios from $4.5\times$ to $98.5\times$ across the characterized rows, with the interpretation depending on the critical stage of each model.

Table 9. [Analytical.] Analytical composition bounds after fixed-GEMM calibration. Precisions that share the same calibrated schedule are listed together; precision sets the modeled byte footprint and placement pressure but is not used to infer a distinct end-to-end schedule (see text).

Model	Precision	Sequential	Lower Bound	Gap	Critical Invocation
ResNet-18	INT8/INT4	833.048 ms	68.924 ms	$12.1\times$	layer4.0.conv2 $3 \times 3s1$ C512 to K512 7×7
ViT-Small	FP32/INT8	3005.626 ms	30.518 ms	$98.5\times$	block0.ffn_up GEMM 384 to 1536 over 197
BERT-attn	FP32/INT8	357.706 ms	74.293 ms	$4.8\times$	bert_block.ffn_up GEMM 768 to 3072 over 128
GPT-2 Small	FP32/INT8	5522.424 ms	1222.521 ms	$4.5\times$	LM-head GEMM 768 to 50,257 tied

Precisions that share a calibrated schedule are reported as one row: precision sets the modeled byte footprint and placement pressure, but the present composition model does not infer a distinct end-to-end schedule from it. A precision-specific row appears only where an independently synthesized precision-specific engine exists, which is why the ResNet FP32 row is omitted—the convolution sweep is INT8-only—and we report such entries as missing rather than filling them by analytical conversion.

For all reported rows, peak one-engine-active resource usage is far below the V80 budget: the largest reported DSP footprint is 27 of 10,848 DSP58 slices, and BRAM/URAM usage is also small relative to device capacity. This does not imply a complete placed design would be trivial; it means the characterized sequential engine schedule is not resource-saturating the device. The resource-headroom observation includes the attention engine: an explicit multiplier-to-DSP audit, cross-checked between the C-synthesis and post-route Vivado reports, confirms its 6-DSP58 footprint with per-instance attribution. The figure is low not because multipliers spill into LUT fabric—every FP32 multiply is DSP-bound—but because the Versal DSPFP32 primitive implements a complete IEEE-754 FP32 multiply or fused-multiply-add in a single slice (versus the two-to-three DSP48 tiles the same operation costs on UltraScale+), and because HLS time-multiplexes the QK^T and AV fused-multiply-add onto a single shared instance across the sequentially executed stages. The six slices are then two DSPFP32 for the shared multiply and fused-multiply-add and four in the SoftMax row loop (subtract, exponential, and accumulation), with the reciprocal deliberately mapped to LUT fabric.

7.4. Roofline Regime Characterization

Figure 2 places the V80's compute and memory ceilings on a single roofline. The two horizontal INT8 lines quantify the central precision result: HLS realizes the dot-product at $1.93\times$ FP32 (solid), well below the $3.00\times$ the DSP58 specification would suggest (dashed). The three diagonals are the assigned-channel memory roofs; striping a tensor across more channels lifts its roof and slides the ridge point left, the partitioning lever expressed quantitatively. Against this backdrop, Table 10 places each INT8 network quantitatively: its arithmetic intensity, the channel count, the dominant tensor it is assigned in the existing header maps, the resulting effective bandwidth and ridge point, and the side of that ridge on which the network falls. Two readings must be kept distinct. In the *ideal* limit—a kernel driven to a roof—the single-channel networks (ResNet-18, ViT-Small, BERT-attn) sit to the left of their

ridge and would be memory-bound on one channel, whereas GPT-2, whose tied LM-head operand is striped across eight channels, sits to the right and would be compute-bound. In the *realized* schedule, none of the four approaches either roof. The attained throughput of the last column is 50 to 670 times below the nearest ceiling, and the required bandwidth never exceeds 0.09 GB/s; the operative limit at batch one is therefore the per-engine schedule, not the assigned channel. The ridge analysis, therefore, predicts which roof each network would meet first *if* its schedule were optimized, and is the design-time argument for striping the LM-head while leaving the single-channel weight shards alone.

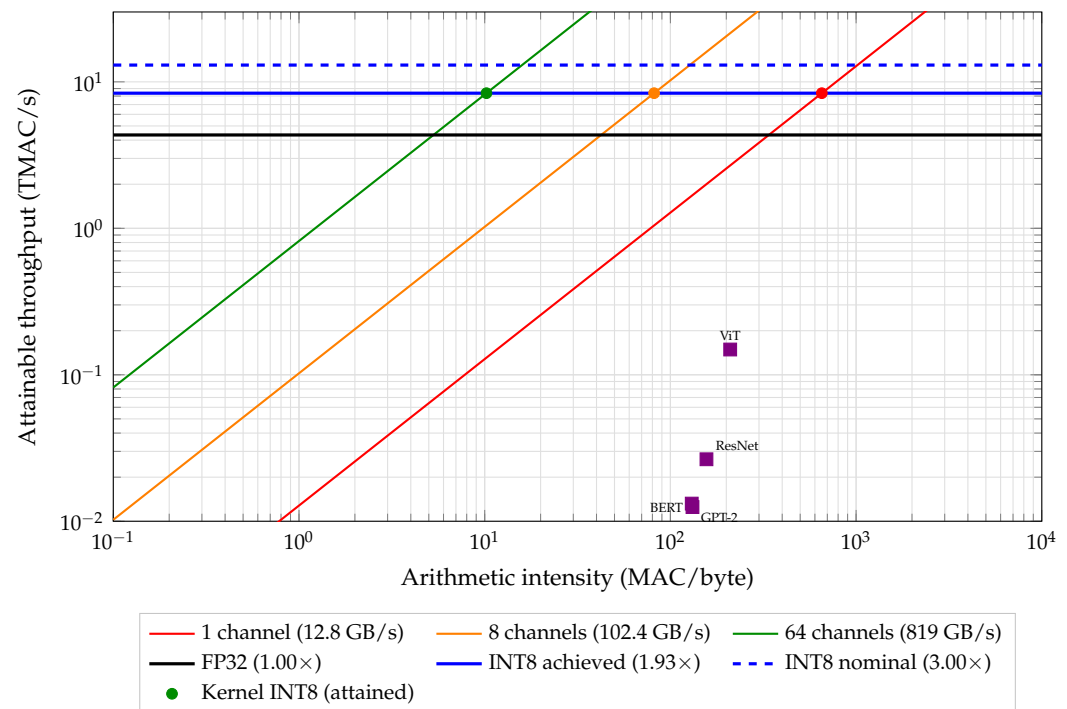


Figure 2. [Analytical: measured anchors, modeled ceilings.] Roofline for the XCV80 at 400 MHz. Horizontal lines are compute ceilings under the FP32-normalized DSP58 packing factor $m(p)$; the INT8 ceiling is shown at both the AM004 nominal factor (3.00×, dashed) and the hardware-achieved factor ($27/14 = 1.93\times$, solid, Table 2). The gap between the two blue lines is the headline precision result: HLS realizes the INT8 dot-product at $1.93\times$ FP32, not the $3.00\times$ the dot-product specification would suggest. Diagonal lines are the assigned-channel memory roofs $B_{\text{eff}} = N_{\text{ch}} BW_{\text{ch}}$ for 1, 8, and 64 channels. The solid circular markers are the ridge points of the INT8-achieved compute ceiling against each of these roofs, colour-matched to the roof they belong to: green at 10.2 MAC/byte for the 64-channel roof, orange at 81.7 MAC/byte for the 8-channel roof, and red at 654 MAC/byte for the single-channel roof. Striping a tensor from one channel to 64 therefore moves its ridge point from 654 to 10.2 MAC/byte, the quantitative form of the partitioning lever. Square markers are the four INT8 networks at their arithmetic intensity and their attained throughput, computed from the pipelined lower-bound latency of Table 9. Every network sits one-and-a-half to two-and-a-half orders of magnitude below the nearest roof: at batch one, the per-engine schedules, not bandwidth or peak compute, set the realized rate, and the vertical distance to the roof is the headroom a fused or replicated schedule could recover.

The dominant stage behind each row is the late layer4 convolution for ResNet-18, the FFN up-projection GEMM for ViT-Small and BERT-attn, and the tied LM-head GEMM for GPT-2; the channel counts follow the placement rule of Section 6, which assigns the single-channel weight shards and stripes the LM-head operand across eight channels. The arithmetic intensities cluster tightly between 130 and 210 MAC/byte because all four are weight-streaming networks at batch one, yet GPT-2 alone is compute-side: not because its intensity is higher but because striping its dominant operand lowers its ridge below the cluster.

This is the placement lever stated as a table—the same network can be moved across the ridge by changing how its largest tensor is mapped to channels, without altering its arithmetic.

Table 10. [Analytical.] Quantitative roofline placement of each INT8 network under the assigned-channel bandwidth model. Arithmetic intensity is total MACs/inference divided by HBM weight traffic; N_{ch} is the channel count assigned to the dominant tensor in the existing header maps (Section 6); the ridge is the arithmetic intensity at which the INT8-achieved compute ceiling (8.37 TMAC/s) meets the effective memory roof. The ideal-limit regime is the side of the ridge; the attained throughput is computed from the pipelined lower-bound latency and reported as a multiple below the nearer roof.

Network	MACs/Inf ($\times 10^9$)	AI (MAC/B)	N_{ch}	B_{eff} (GB/s)	Ridge (MAC/B)	Ideal-Limit Regime	Below Roof
ResNet-18	1.83	156	1	12.8	654	memory-side	316×
ViT-Small	4.54	210	1	12.8	654	memory-side	56×
BERT-attn	0.93	132	1	12.8	654	memory-side	668×
GPT-2 Small	16.11	130	8	102.4	81.7	compute-side	635×

7.5. Precision as Memory Placement

The byte results in Table 11 support the precision-as-placement conclusion. INT8 reduces weight traffic and storage by a factor of four relative to FP32, and INT4 halves bytes again where it is modeled. Because the verified GEMM result gives the same DSP footprint for INT4 and INT8, the INT4 benefit is a placement and HBM traffic benefit, not an additional compute-packing benefit.

The GPT-2 row is the strongest example. The full FP32 weight footprint is 471.759 MiB, and the tied LM-head alone accounts for a 147.237 MiB streaming GEMM operand. The WTE lookup reads only 0.375 MiB of random rows at sequence length 128 in FP32, but the same physical matrix is the dominant LM-head weight tensor. The placement framework, therefore, classifies the WTE lookup access and the WTE/LM-head physical storage differently.

Table 11. [Analytical: modeled byte counts.] HBM byte pressure by model and precision. Activations have zero modeled spill in all rows.

Model	Precision	Weight Bytes	Activation Spill	Embedding Random
ResNet-18	FP32	44.567 MiB	0.000 MiB	0.000 MiB
ResNet-18	INT8	11.142 MiB	0.000 MiB	0.000 MiB
ResNet-18	INT4	5.571 MiB	0.000 MiB	0.000 MiB
ViT-Small	FP32	82.538 MiB	0.000 MiB	0.000 MiB
ViT-Small	INT8	20.635 MiB	0.000 MiB	0.000 MiB
BERT-attn	FP32	27.012 MiB	0.000 MiB	0.000 MiB
BERT-attn	INT8	6.753 MiB	0.000 MiB	0.000 MiB
GPT-2 Small	FP32	471.759 MiB	0.000 MiB	0.375 MiB
GPT-2 Small	INT8	117.940 MiB	0.000 MiB	0.094 MiB

7.6. Per-Engine Power

Table 12 reports per-engine dynamic power from Vivado report_power run on the post-route netlist at the 400 MHz operating point. Static power is excluded to isolate the contribution of each engine to total chip power. The dominant dynamic-power consumers reflect the resource mix characterized in Table 7: GEMM FP32 draws the highest DSP switching power, the convolution and pooling engines dominate due to their long run-times and memory-access patterns, and the SoftMax engine—despite its sub-microsecond latency—has minimal power contribution.

Table 12. [Estimated: post-route report_power.] Per-engine power from Vivado report_power (post-route, 400 MHz). Dynamic power only; static power excluded.

Engine	Dynamic (W)	Latency	Energy/Inference (mJ)
GEMM FP32	0.397	19.277 μ s	0.007653
GEMM INT8	0.350	18.977 μ s	0.006642
GEMM INT4	0.327	18.947 μ s	0.006196
Convolution (INT8)	0.358	57.091 ms	20.439
Attention core	0.897	10.672 ms	9.573
LayerNorm	0.375	1.533 ms	0.574875
Softmax (seq = 128)	0.260	1.77 μ s	0.000460
Pooling	0.126	8.649 ms	1.090
Embedding	0.482	246.50 μ s	0.118813

Because report_power is a switching-activity estimate from the post-route netlist rather than a physical measurement, we validated it against the device. Table 13 compares the post-route estimate with dynamic power measured on the V80 itself through the AMD AVED platform power-measurement tool (via its ami_tool sensors), as the in-action draw minus the idle draw, at the same 400 MHz operating point, together with the resulting device-measured energy per inference. The estimate tracks the device measurement in the same rank order and to within 6–27% across all nine engines, and—importantly—it is a consistent *lower* bound: the device draws more dynamic power than report_power predicts on every engine, with the largest gap on the long-running convolution and attention cores. The post-route estimate is therefore a faithful but slightly optimistic proxy for on-device dynamic power, and we use the device-measured figures as the primary FPGA energy numbers in the cross-platform comparison of Section 7.8, retaining the estimate only as the cross-check shown here.

Table 13. [Measured: device telemetry vs post-route estimate.] Validation of the report_power dynamic-power estimate against on-device measurement. “Estimate” is the post-route Vivado report_power dynamic power of Table 12; “device” is dynamic power measured on the V80 through the AMD AVED platform power-measurement tool (ami_tool sensors, in-action minus idle) at 400 MHz; “device energy” is that device power times the measured latency. The estimate is a consistent lower bound, within 6–27% of the device measurement on every engine.

Engine	Est. (W)	Device (W)	Device Energy (mJ)	Device/est.
GEMM FP32	0.397	0.421	0.008116	+6.0%
GEMM INT8	0.350	0.376	0.007135	+7.4%
GEMM INT4	0.327	0.359	0.006802	+9.8%
Convolution (INT8)	0.358	0.453	25.862	+26.5%
Attention core	0.897	1.097	11.707	+22.3%
LayerNorm	0.375	0.427	0.654591	+13.9%
Softmax (seq = 128)	0.260	0.307	0.000543	+18.1%
Pooling	0.126	0.152	1.315	+20.6%
Embedding	0.482	0.543	0.133850	+12.7%

7.7. Baseline Methodology

The CPU and GPU baselines are per-engine reference implementations in PyTorch 2.4.0, evaluated at the same shapes as the synthesized engines, so each FPGA engine is compared against the corresponding host kernel rather than an end-to-end model. The GPU is an NVIDIA Titan RTX, and the CPU is an Intel Xeon W-3223; both were the hardware available in our laboratory and are representative of a Turing-generation workstation GPU and a contemporary server/workstation CPU, respectively, rather than current data-center accelerators. They are intended as accessible reference points on commodity hardware, and the sensitivity of the comparison to this choice is discussed in Section 7.8. On the GPU, no graph-level optimization path (torch.compile, TensorRT,

ONNX Runtime) is used, so the comparison is against stock eager PyTorch. Each engine runs under `torch.inference_mode`; latency is the mean over the measured iterations after warm-up, each iteration timed with `perf_counter` bracketed by a device synchronization, with median, minimum, and maximum also recorded. FP32 GEMM and convolution use `torch.matmul` and `F.conv2d` (cuBLAS/cuDNN, as distributed with the CUDA toolkit and NVIDIA driver used on the GPU); LayerNorm, SoftMax, pooling, embedding, and attention are evaluated in FP32, matching the FP32 characterization of those engines on the FPGA. Two precision caveats are stated plainly. First, the GPU integer-GEMM paths (INT8, INT4) do not use the DP4A/IMMA tensor-core datapath: because on-device integer-matmul support is inconsistent across backends, the kernel computes the integer product through a portable elementwise multiply and int64 reduction over 64-element K blocks. This preserves exact integer arithmetic but is substantially slower and higher-power than an optimized INT8 GEMM, so the GPU integer-GEMM figures are a functional reference, not the GPU's attainable INT8 throughput, and FPGA-versus-GPU integer-GEMM comparisons are scoped as such. Second, the INT8 convolution baseline is computed in FP32 (clamped inputs, `F.conv2d`, then rounded), as no integer convolution datapath is instantiated on the host; it is a numerical reference at INT8 input range rather than an integer convolution. Power and energy are measured per platform: GPU board power is sampled from `nvidia-smi` (`power.draw`, as distributed with the NVIDIA driver) at 50 ms, and energy is mean board power times measured wall time; CPU energy is the Intel RAPL package-domain energy delta, read through the Linux kernel powercap interface, over the measured window, with power that energy divided by wall time. The FPGA dynamic power is measured on the V80 through the AMD AVED platform power-measurement tool (`ami_tool` sensors) as the in-action-minus-idle draw (Table 13), placing all three platforms on a common protocol: device- or package-level dynamic power (active minus idle) times the measured per-engine latency. The earlier `report_power` estimate is retained only as a validated cross-check, so the cross-platform comparison no longer rests on a synthesis estimate; the residual heterogeneity is in the software stacks and numerical paths, not in the energy methodology, and is discussed in Section 7.9.

7.8. Comparison with GPU and CPU Baselines

Table 14 compares measured V80 engine latency and energy per inference against the GPU and CPU baselines of Section 7.7, on the same per-engine kernels. All three platforms report *dynamic* energy on a common protocol: device- or package-level active-minus-idle power times the measured per-engine latency. The FPGA figure is the device-measured dynamic energy of Table 13; for the GPU and CPU, we additionally report the total measured energy (which includes static and idle draw) so that the static fraction is visible. The integer-GEMM and INT8-convolution rows carry the baseline-precision caveats of Section 7.7.

The difference between the total and dynamic columns is the static (idle) draw, and it dominates the measured baseline energy: idle power accounts for roughly 77 to over 99% of the total GPU energy and 58 to 93% of the total CPU energy across the eight engines. Reporting dynamic energy on all three platforms removes this asymmetry: the FPGA, GPU, and CPU figures are now each an active-minus-idle measurement times the measured latency, so the comparison no longer mixes a dynamic FPGA number against static-inclusive baselines. The residual caveat is smaller and shared by all three: `nvidia-smi` reports whole-board power, RAPL reports whole-package power, and the V80 AVED telemetry reports whole-board power, so each dynamic figure still includes other on-board or on-die activity and is not a perfectly isolated per-kernel number; the protocol is, however, the same on every platform.

The energy comparison is shown in Figure 3. On this consistent dynamic-energy basis, the FPGA is most efficient on the small-GEMM, and SoftMax engines: GEMM (FP32 and INT8) and SoftMax draw the lowest dynamic energy of the three platforms. LayerNorm is close to the GPU—marginally behind on the device-measured figure—and far below the

CPU. On pooling and embedding, the FPGA sits between the baselines: more dynamic-energy-efficient than the CPU but beaten by the GPU, a change from the total-energy reading, in which the FPGA appeared ahead of both. On the two long-running engines, INT8 convolution and the attention core, the FPGA is least efficient on dynamic energy as well: its multi-millisecond runtimes outweigh its sub-watt power, and both the GPU and CPU finish the same work in well under a millisecond. This split—narrower than a total-energy comparison suggests, measured the same way on all three platforms, and explicitly dynamic-versus-dynamic—is the honest characterization supported by the data.

Table 14. [Measured: device/package dynamic energy on all three platforms.] Latency and energy-per-inference comparison: Alveo V80 vs. GPU and CPU baselines, on the same per-engine kernels. GPU: NVIDIA Titan RTX; CPU: Intel Xeon W-3223. Energy is in mJ. The FPGA energy is device-measured dynamic energy (V80 AVED platform telemetry, in-action minus idle; Table 13). For the GPU and CPU, each energy cell gives the total measured energy with the *dynamic* energy (active minus idle) in parentheses, both from the RAPL package and `nvidia-smi` board power. The dynamic figures place all three platforms on a common active-only basis; the total-minus-dynamic gap is the static fraction quantified in the text. Latency is the median runtime. The GPU integer-GEMM and INT8-convolution rows use the host reference paths of Section 7.7, not optimized integer kernels.

Engine	FPGA (V80)		GPU		CPU	
	Lat.	Energy (Dyn.)	Lat.	Energy Tot. (Dyn.)	Lat.	Energy Tot. (Dyn.)
GEMM FP32	19.277 μ s	0.00812	32.0 μ s	2.158 (0.0298)	8.6 μ s	0.652 (0.048)
GEMM INT8	18.977 μ s	0.00714	164.2 μ s	10.66 (0.259)	21.5 μ s	1.760 (0.173)
Convolution (INT8)	57.091 ms	25.86	0.128 ms	9.959 (1.652)	0.628 ms	63.22 (16.62)
Attention core	10.672 ms	11.71	94.3 μ s	6.351 (0.104)	68.0 μ s	6.319 (1.316)
LayerNorm	1.533 ms	0.655	0.140 ms	9.770 (0.591)	0.157 ms	17.75 (4.842)
Softmax (seq = 128)	1.77 μ s	0.00054	20.7 μ s	1.419 (0.00603)	5.6 μ s	0.470 (0.033)
Pooling	8.649 ms	1.315	31.5 μ s	2.785 (0.633)	0.746 ms	93.18 (38.87)
Embedding	246.50 μ s	0.134	49.5 μ s	3.396 (0.0425)	48.0 μ s	5.628 (1.855)

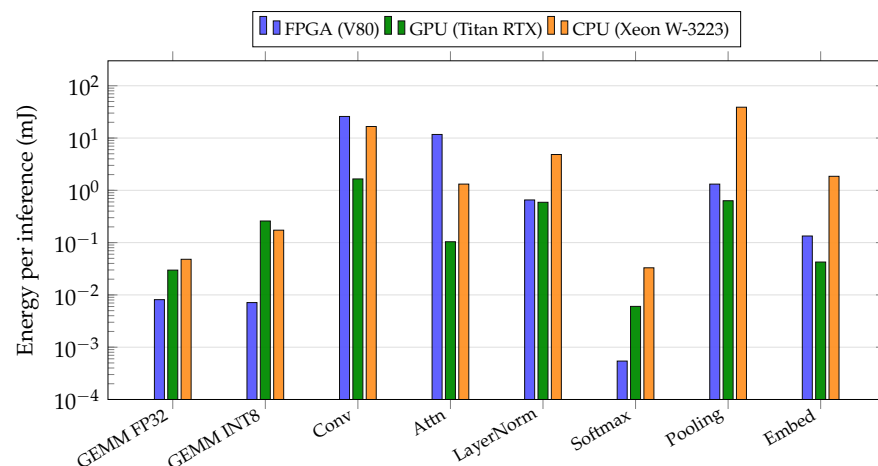


Figure 3. [Measured: device/package dynamic energy on all three platforms.] Dynamic energy per inference per engine on the Alveo V80, NVIDIA Titan RTX, and Intel Xeon W-3223 (log scale). All three bars are *dynamic* energy measured by the same protocol: device- or package-level active-minus-idle power times the measured latency. The FPGA bar is the V80 device-measured dynamic energy (AVED platform telemetry; the post-route `report_power` estimate that it validates is in Table 13), and the GPU and CPU bars are measured active energy minus an idle baseline (their total measured energy, including static draw, is in Table 14). The GPU integer rows use the host reference paths of Section 7.7 rather than optimized integer kernels. On this basis, the FPGA is most efficient on the small-GEMM and SoftMax engines, is close to the GPU on LayerNorm, sits between the baselines on pooling and embedding, and is least efficient on the long-running convolution and attention engines.

7.9. Sensitivity to the Baseline Choice

Because the GPU and CPU were fixed by laboratory availability, it is worth asking how much the comparison depends on that choice. A newer baseline would shift the magnitudes but not the qualitative split, for reasons that follow from where each engine's cost comes from. On the compute-heavy engines—INT8 convolution and the attention core, and the integer GEMMs—a current GPU would widen its lead substantially: it would be faster on raw throughput, and, more importantly, it would expose a real INT8 tensor-core datapath (DP4A/IMMA) that the portable elementwise reference path used here deliberately avoids (Section 7.7). The GPU integer figures reported here are therefore already conservative against the GPU, and a newer device would only enlarge the FPGA's disadvantage on those engines. On the latency-bound small kernels—SoftMax and the small GEMMs—the comparison is dominated by host kernel-launch overhead in eager execution, which is largely generation-independent, so a newer GPU would not materially erode the FPGA's latency advantage there. On dynamic energy, a newer process node lowers energy per operation on both the CPU and the GPU, which would narrow the FPGA's remaining margins on the small-GEMM and SoftMax engines; conversely, a higher-TDP data-center part would have a larger idle draw, widening the static fraction documented above and the total-versus-dynamic gap. The direction of the split—the FPGA is favorable on memory-light and latency-bound small kernels and unfavorable on the long-running convolution and attention engines—is set by the FPGA's own sub-watt-but-slow batch-one schedules rather than by the specific baseline, and would persist. The comparison should therefore be read as a workload-shape characterization on accessible commodity hardware, not as a fixed ranking against the latest accelerators.

8. Discussion

The results connect directly to the HBM framework. Since intermediate activations fit on chip, the memory problem is not activation spill. It is a static tensor placement. The critical lower-bound stages in the regenerated composition are the largest dense streaming stages for ViT, BERT, and GPT-2, and a high-reuse late convolution for ResNet. This explains why the framework starts from residency and access pattern: the tensors that matter most are not just large, but large and repeatedly consumed by a scheduled compute stage.

The measured results also clarify precision. INT8 is both a compute resource and a memory lever, confirmed by the post-route GEMM utilization: 27 DSP58 for FP32 and 14 DSP58 for both INT8 and INT4 on the physical device. INT4 is only a memory lever on this device under the tested HLS schedule, because it maps to the same INT8-oriented DSP58 dot-product structure and produces an identical 14-DSP post-route footprint. This makes INT4 useful for fitting or striping tensors but not for claiming another compute-packing step—a conclusion that is now supported by hardware-measured utilization rather than synthesis estimation alone.

We are careful not to overstate this. The result is an experimental observation about the configuration we analyzed—the Vitis HLS INT4 path mapped onto the DSP58 in our fixed-tile schedules—and not an absolute property of INT4 arithmetic on Versal devices. It has a clear and stable cause: in our builds, Vitis HLS does not pack INT4 multiplies the way it packs INT8 and FP32, and the DSP58 exposes an INT8 vector dot-product mode but no native INT4 mode [21], so the INT4 operands are computed on the same INT8-width datapath and do not reduce the slice count further. Because that cause is the tool-plus-architecture mapping rather than a specific kernel, we expect the same behavior across the DSP58-bound integer-GEMM and convolution schedules in this paper; we do not extrapolate beyond them. Two qualifications make the boundary explicit. First, it concerns the DSP-mapped datapath only: an implementation that packs INT4 multiplies

into LUT fabric instead of DSP58 slices is a different design point we did not characterize, and could in principle extract additional parallelism per LUT—which is precisely why we treat the finding as configuration-specific rather than as a statement about INT4 in general. Second, the equality is in compute resources, not memory: INT4 always halves the bytes of INT8, so its value as a placement and bandwidth lever holds regardless. We therefore report the no-further-compute-reduction result as an observed property of the Vitis HLS DSP58 mapping in our configuration, consistent with the DSP58 lacking a native INT4 mode, while restricting the exact resource counts to the characterized schedules.

8.1. Device Utilization and Replication Headroom

The engines characterized here are single instances that occupy a small fraction of the device—the largest reported footprint is 27 of 10,848 DSP58 slices—so the per-engine results should not be read as a high-throughput V80 design. A first-order replication analysis bounds how far the methodology would have to scale to fill the device. Take the INT8 GEMM engine of Table 7 as representative, with a post-route footprint of 14 DSP58, 6172 LUTs, 7657 flip-flops, and 38 BRAM (18-Kb-block) equivalents. That BRAM is consumed entirely by the engine’s per-instance HBM read adapters; the intrinsic GEMM tile buffers map to distributed LUTRAM and use no hard BRAM, so a multi-engine design with a shared memory subsystem amortizes the read buffering rather than multiplying it, and logic—not BRAM—governs replication. Against the V80 budget the instance count is bounded by $10,848/14 \approx 775$ on DSP, $5,148,416/7657 \approx 672$ on flip-flops, and $2,600,000/6172 \approx 421$ on LUT, so LUT capacity binds at roughly 420 instances. (Naively replicating the full engine, including its private read adapters, would instead be BRAM-bound at $7482/38 \approx 196$ instances.) At the measured single-engine rate of about 4 GMAC/s, roughly 420 LUT-bound instances deliver on the order of 1.7 TMAC/s of device-level INT8 GEMM throughput, about 20% of the resource-normalized INT8 ceiling of Section 3.5. The remaining gap is the optimization space a production accelerator would need to close—higher per-DSP58 multiplier utilization and fewer LUTs per MAC—and the estimate is deliberately optimistic, since it ignores routing congestion, the static shell, and HBM channel contention, all of which lower the achievable figure. This analysis is analytical and is offered to size the headroom rather than to report a measured multi-engine design; an implemented replicated accelerator remains future work.

8.2. From Analytical Composition to an Integrated Accelerator

The composition bounds of Section 7 are analytical: they combine measured per-engine latencies under two scheduling assumptions—no overlap (the sequential sum) and perfect overlap (the maximum stage)—and a built accelerator would land between them. It is worth being explicit about what this model does and does not provide for a reader whose goal is a complete V80 design rather than an engine library. What it provides is a design-time planning instrument: the lower bound identifies the critical stage that sets the floor on end-to-end latency—the late `layer4` convolution for ResNet-18, the FFN up-projection for ViT and BERT, and the tied LM-head for GPT-2—and the gap to the sequential sum quantifies the headroom that inter-engine pipelining could recover, which is where integration effort is best spent. What it does not provide is a measured end-to-end number. An integrated design adds scheduler, DMA, synchronization, clock-domain-crossing, and shell-routing overheads, and—as the LM-head placement experiment of Section 6.2 demonstrates—behaviors the per-engine model omits, such as repeated weight fetches and HBM channel contention, can dominate the realized rate until they are addressed with on-chip reuse buffers and deliberate channel striping. The composition model should therefore be read as bounding and prioritizing the integration work, not as a substitute for building and

measuring it; closing the gap between these bounds and a placed, fused design is the principal remaining step toward a full accelerator.

8.3. Design Guidelines

The study suggests five design guidelines for HLS inference designs on HBM-equipped FPGAs. Intermediate activations should remain on chip whenever the live set fits in BRAM/URAM. HBM bandwidth should be treated as per-channel bandwidth rather than as a single aggregate resource. Striping should be reserved for tensors with regular streaming or statically strided access. Random embedding lookup and tied LM-head streaming should be treated as distinct access modes, even when they refer to the same physical tensor. Finally, INT4 should be used as a storage and bandwidth lever unless a distinct INT4 compute datapath has been verified.

The scope of the study is bounded by several methodological limitations. First, the composition bounds are analytical. The sequential sum assumes no inter-engine overlap, whereas the lower bound assumes perfect overlap around the maximum stage. An end-to-end pipelined implementation integrating all engines into a complete V80 shell would lie between these bounds and would additionally incur scheduler, DMA, synchronization, and shell-routing overheads not captured by the per-engine characterization. The attention engine's resource attribution, flagged as provisional in earlier drafts, has since been resolved: a per-instance multiplier-to-DSP audit cross-checked between the C-synthesis and post-route Vivado reports confirms the 6-DSP58 footprint, attributes every FP32 multiply to a DSP slice, and explains the low count through the Versal DSPFP32 primitive and cross-stage time-multiplexing (Section 7.1); attention figures are therefore reported as final rather than provisional. Second, all residency conclusions are established at batch size one and the studied prefill sequence lengths (128 and 197 tokens). Longer sequences, autoregressive decode with a growing key/value cache, or batched inference enlarge the live activation footprint and may reintroduce activation spill, which would shift placement pressure back toward intermediate tensors and change the framework's emphasis. Finally, the framework predicts placement class rather than exact stripe width; stripe width depends on host layout, DMA packing, and the selected parallel schedule.

Three further limitations bound the claims. First, the engines are characterized as a library of independent kernels, not as an integrated accelerator, and the performance of a complete inference system depends on factors this study does not measure: inter-engine interaction, concurrent contention for HBM channels, dataflow efficiency between stages, and global scheduling. The composition bounds and the replication-headroom analysis (Sections 8.1 and 8.2) bound and prioritize that integration work, but do not substitute for building and measuring it; the placement experiment of Section 6.2 is the only inter-placement effect we measure directly. Second, the HLS methodology and the measured packing factors are specific to AMD Vitis HLS and the Versal DSP58 architecture; we target AMD devices only and neither claim nor demonstrate transferability to Intel FPGAs or other HLS toolchains, on which both the engine-construction steps and the precision-packing behavior would have to be re-established. Third, the comparison against the CPU and GPU baselines should be read as indicative rather than as a controlled benchmark: although the energy methodology is now uniform across the three platforms (device- or package-level active-minus-idle power times measured latency), the software stacks, optimization levels, and numerical paths differ, so the cross-platform figures characterize workload-shape tendencies rather than a precise ranking (Section 7.9).

9. Conclusions

This paper presented a methodology for HLS-based AI inference on the AMD Alveo V80 whose engine library is validated on hardware and whose model-level composition and partitioning layers are design-time analytical. The main contribution is an HBM partitioning framework that classifies tensors by residency, access pattern, reuse, and service bandwidth, then maps them to on-chip storage, single-channel HBM shards, or striped HBM placement. This design-time partitioning framework is anchored by engine-level hardware measurements, and its central bandwidth-scaling premise—that striping a streaming tensor across N HBM channels delivers close to N times the single-channel bandwidth—is confirmed by an on-board channel sweep that scales near-linearly to the full 64 channels at 87–95% of the per-channel ceiling (Section 6.1). The per-tensor placement classes are checked for class-level consistency against the existing channel maps, and the striping prescription is validated directly on hardware: placing the GPT-2 LM-head operand on a single channel versus the eight channels the framework assigns it yields a measured $6.81 \times$ speedup (Section 6.2). Validating the remaining per-tensor classes as in situ placements, and measuring the random-gather case that the framework treats as the striping exception, remains the natural next step.

Nine HLS engines—covering GEMM (FP32, INT8, INT4), convolution, attention, layer normalization, SoftMax, pooling, and embedding—were placed, routed, and executed on the physical Alveo V80 at 400 MHz. All nine engines close timing with positive worst-case slack (WNS range: 0.043 ns to 0.409 ns). Post-route timing, utilization, and power are reported per engine and compared against NVIDIA Titan RTX and Intel Xeon W-3223 baselines on the same kernels, on a consistent dynamic-energy basis measured the same way on all three platforms (device- or package-level power, active minus idle). On that basis, the FPGA is most efficient on the small-GEMM and SoftMax engines, is close to the GPU on layer normalization, sits between the two baselines on pooling and embedding, and is least efficient on the long-running convolution and attention engines; idle draw accounts for most of the baseline platforms' total energy. The post-route power estimate was validated against on-device measurement, which it tracks within 6–27% as a consistent lower bound.

The HLS results support a three-stage optimization process: establish a correct standalone engine, expose throughput through unrolling and local banking, and then apply verified precision-specific DSP packing. The hardware-confirmed DSP58 result is central: INT8 reduces GEMM DSP use from 27 to 14, while INT4 uses the same 14 DSPs and acts as a memory-placement optimization rather than an additional compute-packing mode. The composition model, anchored by measured engine latencies, indicates potential scheduling headroom under inter-engine overlap, with lower bounds set by the actual critical stages. Realizing this headroom in a fully integrated shell implementation is identified as the subject of subsequent work.

Author Contributions: Conceptualization, A.-A.U. and V.-G.Ş.; methodology, A.-A.U.; software, A.-A.U.; validation, A.-A.U.; formal analysis, A.-A.U.; investigation, A.-A.U.; resources, A.-A.U. and V.-G.Ş.; data curation, A.-A.U.; writing—original draft preparation, A.-A.U.; writing—review and editing, A.-A.U. and V.-G.Ş.; supervision, V.-G.Ş. All authors have read and agreed to the published version of the manuscript.

Funding: This study was supported by the Romanian Ministry of Research, Innovation, and Digitization through the ATLAS CERN-RO grants.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The HLS source files, build and measurement scripts, generated Vitis HLS and Vivado synthesis/implementation reports, and benchmark scripts are available from the corresponding author on reasonable request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial intelligence
AXI	Advanced eXtensible Interface
BRAM	Block random-access memory
DSP	Digital signal processing
FF	Flip-flop
GEMM	General matrix multiplication
HBM	High-bandwidth memory
HLS	High-level synthesis
II	Initiation interval
LUT	Lookup table
URAM	UltraRAM
WPE	Word position embedding
WTE	Word token embedding

References

1. AMD. AMD Alveo V80 Compute Accelerator Card Product Brief. 2024. Available online: <https://www.amd.com/content/dam/amd/en/documents/products/accelerators/alveo/v80/alveo-v80-product-brief.pdf> (accessed on 22 May 2026).
2. AMD. AMD Alveo Versal Example Design (AVED) Documentation for Alveo V80. 2024. Available online: https://xilinx.github.io/AVED/amd_v80_gen5x8_exdes_1_20231204/index.html (accessed on 22 May 2026).
3. Duarte, J.; Han, S.; Harris, P.; Jindariani, S.; Kreinar, E.; Kreis, B.; Ngadiuba, J.; Pierini, M.; Rivera, R.; Tran, N.; et al. Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrum.* **2018**, *13*, P07027. <https://doi.org/10.1088/1748-0221/13/07/P07027>.
4. Umuroglu, Y.; Fraser, N.J.; Gambardella, G.; Blott, M.; Leong, P.; Jahre, M.; Vissers, K. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*; ACM: New York, NY, USA, 2017; pp. 65–74. <https://doi.org/10.1145/3020078.3021744>.
5. Venieris, S.I.; Bouganis, C.S. fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *30*, 326–342. <https://doi.org/10.1109/TNNLS.2018.2844093>.
6. Wang, Z.; Huang, H.; Zhang, J.; Alonso, G. Shuhai: Benchmarking high bandwidth memory on FPGAs. In *Proceedings of the IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*; IEEE: Piscataway, NJ, USA, 2020; pp. 111–119. <https://doi.org/10.1109/FCCM48280.2020.00024>.
7. Choi, Y.; Chi, Y.; Wang, J.; Guo, L.; Cong, J. When HLS meets FPGA HBM: Benchmarking and bandwidth optimization. *arXiv* **2020**, arXiv:2010.06075. <https://doi.org/10.48550/arXiv.2010.06075>.
8. Hong, S.; Moon, S.; Kim, J.; Lee, S.; Kim, M.; Lee, D.; Kim, J.Y. DFX: A low-latency multi-FPGA appliance for accelerating transformer-based text generation. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*; IEEE: Piscataway, NJ, USA, 2022; pp. 616–630. <https://doi.org/10.1109/MICRO56248.2022.00051>.
9. Aarrestad, T.; Loncar, V.; Ghielmetti, N.; Pierini, M.; Summers, S.; Ngadiuba, J.; Petersson, C.; Linander, H.; Iiyama, Y.; Di Guglielmo, G.; et al. Fast convolutional neural networks on FPGAs with hls4ml. *Mach. Learn. Sci. Technol.* **2021**, *2*, 045015. <https://doi.org/10.1088/2632-2153/ac0ea1>.
10. Fahim, F.; Hawks, B.; Herwig, C.; Hirschauer, J.; Jindariani, S.; Tran, N.; Carloni, L.P.; Di Guglielmo, G.; Harris, P.; Krupa, J.; et al. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices. *arXiv* **2021**, arXiv:2103.05579. <https://doi.org/10.48550/arXiv.2103.05579>.
11. Blott, M.; Preusser, T.B.; Fraser, N.J.; Gambardella, G.; O'Brien, K.; Umuroglu, Y.; Leeser, M.; Vissers, K. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. Reconfigurable Technol. Syst.* **2018**, *11*, 16. <https://doi.org/10.1145/3242897>.
12. Tousi, N.M.; Lujan, M. Deep learning inferencing with high-performance hardware accelerators. *ACM Trans. Intell. Syst. Technol.* **2023**, *14*, 73. <https://doi.org/10.1145/3594221>.

13. Choi, Y.; Chi, Y.; Qiao, W.; Samardzic, N.; Cong, J. HBM connect: High-performance HLS interconnect for FPGA HBM. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '21)*; ACM: New York, NY, USA, 2021; pp. 116–126. <https://doi.org/10.1145/3431920.3439301>.
14. Huang, Y.C.; Wang, T.H.; Lee, C.H.; Chen, Y.S.; Chang, Y.H. Implementation of fully-pipelined CNN inference accelerator on FPGA and HBM2 platform. *IEEE Access* **2022**, *10*, 39181–39196. <https://doi.org/10.1587/transinf.2022EDP7155>.
15. Li, B.; Pandey, S.; Fang, H.; Lyv, Y.; Li, J.; Chen, J.; Xie, M.; Wan, L.; Liu, H.; Ding, C. FTRANS: Energy-efficient acceleration of transformers using FPGA. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '20)*; ACM: New York, NY, USA, 2020; pp. 175–180. <https://doi.org/10.1145/3370748.3406567>.
16. Zeng, S.; Liu, J.; Dai, G.; Yang, X.; Fu, T.; Wang, H.; Ma, W.; Sun, H.; Li, S.; Huang, Z.; et al. FlightLLM: Efficient large language model inference with a complete mapping flow on FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24)*; ACM: New York, NY, USA, 2024; pp. 223–234. <https://doi.org/10.1145/3626202.3637562>.
17. Sun, M.; Li, Z.; Lu, A.; Ma, H.; Yuan, G.; Xie, Y.; Tang, H.; Li, Y.; Leeser, M.; Wang, Z.; et al. FPGA-aware automatic acceleration framework for vision transformer with mixed-scheme quantization. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22)*; ACM: New York, NY, USA, 2022; pp. 895–900. <https://doi.org/10.1145/3489517.3530618>.
18. Chang, S.E.; Li, Y.; Sun, M.; Shi, R.; So, H.K.H.; Qian, X.; Wang, Y.; Lin, X. Mix and match: A novel FPGA-centric deep neural network quantization framework. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '21)*; IEEE: Piscataway, NJ, USA, 2021; pp. 208–220. <https://doi.org/10.1109/HPCA51647.2021.00027>.
19. Sun, M.; Ma, H.; Kang, G.; Jiang, Y.; Chen, T.; Ma, X.; Wang, Z.; Wang, Y. FILM-QNN: Efficient FPGA acceleration of deep neural networks with intra-layer, mixed-precision quantization. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '22)*; ACM: New York, NY, USA, 2022; pp. 134–145. <https://doi.org/10.1145/3490422.3502364>.
20. Wei, E.; Yu, Z.; Yang, M.; Fan, D.; Yu, T.; Cheng, K.T.; Wang, Y. DeepBurning-MixQ: An open source mixed-precision neural network accelerator design framework for FPGAs. In *Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD '23)*; IEEE: Piscataway, NJ, USA, 2023; pp. 1–9. <https://doi.org/10.1109/ICCAD57390.2023.10323831>.
21. AMD. Versal ACAP DSP Engine Architecture Manual (AM004): Dot-Product Unit and INT8 Integer Multiplication. 2022. Available online: <https://docs.amd.com/r/en-US/am004-versal-dsp-engine/Dot-Product-Unit> (accessed on 22 May 2026).
22. Williams, S.; Waterman, A.; Patterson, D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* **2009**, *52*, 65–76. <https://doi.org/10.1145/1498765.1498785>.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.