Technical Architecture and Systematic Benchmarking of the OpenXLA Ecosystem: A Cross-Platform Analysis of Modern ML Compilers

Anonymous Author(s)

Abstract

Status: This paper presents a *proposed research protocol*. The architectural analysis and benchmarking methodology are complete; experimental results are forthcoming pending hardware access. We welcome feedback on the methodology, hypotheses, and experimental design.

The fragmentation of machine learning (ML) frameworks and hardware backends presents a critical barrier to portable, cost-effective model deployment. OpenXLA addresses this challenge through a unified compiler ecosystem built on StableHLO as a portability layer and PJRT as a pluggable hardware interface. In this paper, we provide an architectural analysis of the OpenXLA compiler pipeline—from its intermediate representation hierarchy (CHLO, StableHLO, VHLO) through target-independent optimization passes to hardware-specific code generation via LLVM—informed by direct contributions to the OpenXLA codebase. We characterize three contemporary accelerator families: Google Cloud TPU v6e (Trillium), NVIDIA H200 (Hopper), and AMD Instinct MI300X, examining how each interacts with the XLA compiler's fusion, buffer analysis, and partitioning strategies. We propose a systematic 3×3 benchmarking methodology—evaluating three representative workloads (LLM inference, dense model training, and sparse embedding training) across all three hardware backends—and formalize testable hypotheses regarding abstraction overhead, optimization transferability, and SPMD scaling efficiency. We release the benchmarking protocol to enable reproducible, community-driven extension of this work.

1 Introduction

The consolidation of machine learning infrastructure has reached a pivotal juncture. The proliferation of frontend frameworks—JAX Bradbury et al. [2018], PyTorch Paszke et al. [2019], and TensorFlow Abadi et al. [2016]—alongside an increasingly heterogeneous hardware landscape comprising GPUs, TPUs, and custom ASICs, has created a fragmentation problem that impedes portable and efficient model deployment.

OpenXLA Google [2024a] represents an industry-collaborative response to this challenge. Developed jointly by Google, AMD, Intel, NVIDIA, and Amazon Web Services, OpenXLA provides a performant, portable, and extensible set of compiler infrastructure components. At its core, the Accelerated Linear Algebra (XLA) compiler optimizes linear algebra operations to enhance execution speed and minimize memory overhead, enabling developers to decouple the mathematical representation of models from the physical constraints of diverse hardware architectures.

Despite the promise of "write once, run anywhere" compilation, significant questions remain regarding the performance overhead introduced by hardware abstraction layers, the effectiveness of compiler optimizations across fundamentally different microarchitectures, and the economic implications of

Preprint.

platform selection for production workloads. As inference workloads grow to dominate AI compute budgets, these questions become increasingly consequential for infrastructure planning.

This paper makes the following contributions:

1. An architectural analysis of the OpenXLA compiler pipeline, from its intermediate representation hierarchy through hardware-specific code generation, informed by direct contributions to the OpenXLA codebase.
2. A detailed characterization of three contemporary accelerator families—TPU v6e, NVIDIA H200, and AMD MI300X—and their interaction with the XLA compiler.
3. A comparative economic analysis of cross-platform deployment costs and energy efficiency.
4. A formal 3×3 benchmarking methodology for systematic, reproducible evaluation of ML compiler performance across heterogeneous hardware.

Critically, this work differs from existing benchmarking efforts in several ways:

- **We benchmark the compiler, not just the hardware.** Existing suites (MLPerf, vendor benchmarks) measure end-to-end throughput without attributing performance to specific compiler passes. We use `hlo-opt` ablation to isolate the contribution of individual optimizations (fusion, CSE, algebraic simplification) on each backend.
- **We measure the portability tax.** For each hardware platform, we compare the OpenXLA compilation path (JAX $\rightarrow$ StableHLO $\rightarrow$ XLA $\rightarrow$ hardware) against the native path (vLLM/CUDA on NVIDIA, PyTorch/ROCm on AMD, direct HLO on TPU) to quantify the overhead introduced by the portability abstraction layer.
- **We quantify optimization transferability.** The same target-independent passes (fusion, CSE) are applied identically to systolic arrays (TPU), SIMT (NVIDIA), and CDNA (AMD). We measure whether each pass helps equally or whether the benefit is architecture-dependent—e.g., fusion may yield larger gains on TPU where systolic dataflow benefits most from reduced memory round-trips, versus AMD where high HBM bandwidth partially masks the memory wall.
- **We expose compiler version sensitivity.** Through direct contributions to XLA, we have observed that individual commits can materially shift benchmark results—AMD shuffle promotion to DPP instructions, collective permute barrier placement, and cost model corrections for integer GEMMs all affect measured performance. We document this sensitivity and pin all experiments to a specific XLA commit hash.
- **We cover three accelerator paradigms.** Unlike two-platform studies, we span systolic array (TPU), SIMT (NVIDIA), and CDNA (AMD), enabling cross-paradigm comparison of how one compiler adapts to fundamentally different hardware.

2 Related Work

Cross-platform benchmarking of ML accelerators has received growing attention as the hardware landscape diversifies. Mattson et al. Mattson et al. [2020] established the MLPerf benchmark suite as a standardized framework for comparing training and inference performance, though MLPerf submissions are vendor-controlled and do not expose compiler-level behavior. Sada et al. Sada et al. [2025] recently benchmarked 12 LLMs (124M–70B parameters) across Qualcomm Cloud AI 100 Ultra and NVIDIA A100 configurations, measuring inference throughput, power consumption, and energy efficiency (tokens/s/W). Their work demonstrated that purpose-built inference accelerators can achieve 10–35$\times$ power reduction over data-center GPUs on 70B-parameter models, establishing energy efficiency as a first-class benchmarking metric.

Our work differs from and extends these efforts in several key dimensions. First, *we benchmark the compiler, not just the hardware*: while Sada et al. use vLLM as a black-box serving layer, we instrument the XLA compiler pipeline to attribute performance differences to specific optimization passes (fusion, CSE, algebraic simplification) via `hlo-opt` ablation. Second, we cover three distinct accelerator paradigms—systolic array (TPU), SIMT (NVIDIA), and CDNA (AMD)—compared to the two-platform designs in prior work. Third, we introduce the concept of *portability tax*: quantifying

the overhead introduced by the StableHLO/PJRT abstraction layer relative to native, hardware-specific compilation paths. No existing benchmark suite measures this compiler-level abstraction cost.

We adopt several methodological best practices from Sada et al., including steady-state power sampling at 1-second intervals, coefficient of variation thresholds ($< 5\%$) for measurement stability, and energy efficiency (tokens/s/W) as a primary reporting metric alongside throughput and cost.

3 Intermediate Representation Hierarchy

The efficacy of OpenXLA as a portability layer is predicated on its high-level intermediate representations, organized in a multi-level dialect hierarchy within the MLIR framework Lattner et al. [2021].

3.1 StableHLO: The Portability Contract

StableHLO functions as a bridge between frontend frameworks and backend compilers such as XLA and IREE IREE Authors [2023]. Any framework capable of producing StableHLO programs is inherently compatible with any compiler that can consume them.

The StableHLO specification defines approximately 100 operations, encompassing verifiers, type inference, and comprehensive support for dynamism and quantization. A critical design feature is its commitment to long-term stability: five years of backward compatibility and two years of forward compatibility. This stability is managed through the VHLO (Versioned StableHLO) dialect, which provides snapshots of the StableHLO dialect at specific points in time. VHLO is inherently "add-only," tracking the evolution of every operation to ensure that consumers can deserialize and upgrade payloads even when produced by older versions of the stack.

3.2 CHLO: Client-Level Abstractions

The CHLO (Client HLO) dialect bridges the gap between high-level client APIs and the lower-level StableHLO dialect. CHLO contains operations that align with the API surface area of `XlaBuilder` in C++, modeling "syntactic sugar" and high-level mathematical compositions before they are materialized into lower-level dialects for backend processing. This hierarchy enables hardware-independent program simplification and the refining of dynamically shaped programs using concrete input arguments.

3.3 Data Representation and Quantization Semantics

StableHLO types emphasize domain-specific requirements for tensor manipulation. Tensors are defined by a shape and an element type, where the shape represents non-negative or unknown dimension sizes in ascending order.

Quantized element types are governed by rigorous constraints to ensure mathematical accuracy across hardware implementations. The specification includes parameters such as `storage_type` (integer), `expressed_type` (floating-point), and `scales` (floating-point constants). For per-axis quantization, the `quantization_dimension` must satisfy:

$$\texttt{quantization\_dimension} < \text{rank}(\mathbf{T}) \tag{1}$$

ensuring that scales and zero points align with the corresponding tensor axes. These constraints (C1–C13 in the specification) enforce that `storage_min` and `storage_max` remain within the limits of the storage type and that scales are strictly positive and finite. By enforcing these semantics at the compiler level, OpenXLA prevents numerical issues commonly arising from lower-precision data formats.

3.4 Shardy: Unified Tensor Partitioning

Shardy Google [2024b] is an MLIR-based tensor partitioning system developed through collaboration between the GSPMD and PartIR teams. It provides a unified partitioning dialect integrated into the StableHLO layer, enabling consistent SPMD partitioning semantics across all OpenXLA backends. Shardy is particularly relevant to our proposed scaling experiments (Section 9, H3), as it determines how computation graphs are sharded across multi-chip and multi-node configurations.

Table 1: XLA compilation pipeline stages, from framework-level Python to native hardware execution.

Stage	Dialect / Format	Optimization Level	Core Tools
Frontend	Python / NumPy (JAX)	Framework-level	JAX / TF / PyTorch
Intermediate	StableHLO / CHLO	Target-independent	MLIR / XLA Passes
Lowering	HLO Dialect	Target-specific	XLA Backend
Backend	LLVM IR / SPIR-V	Microarchitectural	LLVM / NVCC / ROCm
Native	PTX / Machine Code	Hardware-specific	LLVM / Driver

4 The XLA Compiler Pipeline

The XLA compiler architecture is divided into target-independent analysis passes and target-specific code generation, enabling high-level optimizations that benefit all architectures while exploiting unique microarchitectural features of specific hardware.

4.1 Target-Independent Optimization

During the initial compilation stage, XLA performs optimizations on the StableHLO graph that are hardware-agnostic. These include:

Common Subexpression Elimination (CSE). Redundant computations producing identical results are identified and consolidated.

Algebraic Simplification. Mathematically equivalent but computationally cheaper operation sequences are substituted.

Operation Fusion. Perhaps XLA's most critical optimization, fusion combines multiple subgraphs into a single computation kernel, eliminating the overhead of short-lived operations and intermediate storage buffers. For example, a computation such as:

$$Y = \texttt{reduce\_sum}(X + Y \times Z) \tag{2}$$

would normally launch separate kernels for multiplication, addition, and reduction. XLA fuses these into a single kernel, streaming results directly through hardware registers rather than writing intermediate values back to main memory. This process significantly reduces the "memory wall" bottleneck common in modern accelerators.

Additionally, the compiler performs *buffer analysis* to allocate runtime memory for the computation and *shape specialization* to enable more aggressive constant propagation.

Note: The internal MHLO (Meta-HLO) dialect is currently being deprecated, with useful passes (canonicalization, folder patterns) migrating to StableHLO. This consolidation ensures that all hardware-independent graph simplifications are available to every PJRT plugin regardless of target compiler IR.

4.2 Target-Specific Code Generation and LLVM Integration

After target-independent passes, the compiler converts the StableHLO dialect into an internal HLO dialect and dispatches it to a hardware-specific backend. Backends for CPUs and GPUs utilize the LLVM framework Lattner & Adve [2004] for low-level IR generation and native code emission. At this stage, the backend performs pattern-matching to replace operation combinations with optimized library calls (e.g., cuDNN Chetlur et al. [2014] or MKL) and determines the optimal partitioning of computations into parallel streams.

Table 1 summarizes the complete compilation pipeline.

5 PJRT: The Pluggable Hardware Interface

To facilitate the "run anywhere" objective, OpenXLA incorporates PJRT (Pluggable Just-in-Time Runtime), a hardware- and framework-independent interface for ML compilers and runtimes. PJRT

simplifies new hardware integration by providing a stable C API that abstracts device management, memory allocation, and executable execution.

5.1 Architectural Components

The PJRT infrastructure comprises several key abstractions:

- **PjRtClient**: Manages all communication and owns the devices and memory spaces for a given plugin.
- **PjRtDevice**: Describes a single device, including its unique hash identifier and location within a local or global grid.
- **PjRtMemorySpace**: Distinguishes between unpinned and pinned memory associated with specific devices.
- **PjRtBuffer**: Holds data on the device in a format optimized for the plugin (e.g., proprietary tensor formats or MLIR element attributes).
- **PjRtCompiler**: Takes an input module (such as StableHLO) and returns a `PjRtLoadedExecutable`.

This plugin architecture allows hardware vendors to develop support independently from the main OpenXLA or framework repositories. For example, the Intel Extension for TensorFlow and the AMD ROCm plugin utilize the PJRT C API to integrate seamlessly with JAX and PyTorch without modifying core framework code.

6 Hardware-Specific Implementations

The OpenXLA ecosystem supports diverse hardware, each with specialized units designed to accelerate matrix-heavy workloads. We characterize three contemporary accelerator families.

6.1 Google Cloud TPU v6e (Trillium)

Google's Tensor Processing Units are specialized matrix processors designed around systolic arrays—grids of multiply-accumulators (MACs) that process matrix operations in a weight-stationary dataflow, where weights are fixed in the array while activations stream across the grid.

TPU v6e (Trillium) features 256×256 systolic arrays, enabling 65,536 MAC operations per cycle—a significant increase from the 128×128 arrays in prior generations. Trillium is optimized for Transformer-based models, delivering $2.8\times$ better performance and $2.1\times$ improvement in performance per watt over its predecessors.

A distinguishing feature is the *SparseCore*, a dataflow processor designed to accelerate models with sparse operations, such as recommendation systems relying on massive embedding tables. SparseCores operate in parallel with TensorCores, enabling pipelined dense and sparse computation.

6.2 NVIDIA H200 (Hopper)

NVIDIA's integration into OpenXLA leverages the maturity of the CUDA ecosystem. Recent developments focus on scaling LLM training for ultra-long contexts. By integrating NVSHMEM—a communication library based on Partitioned Global Address Space (PGAS) semantics—with XLA, researchers have achieved up to 36% speedups over standard NCCL for sequence lengths up to 256K tokens NVIDIA [2024].

XLA supports CUDA Graphs on NVIDIA GPUs, amortizing kernel launch overhead by capturing operation sequences into a single graph executable. This is particularly beneficial for models with many small, short-lived operations where launch latency would otherwise dominate execution time.

6.3 AMD Instinct MI300X

AMD, a founding member of OpenXLA, integrates its Instinct MI300 series accelerators via the ROCm stack. The MI300X features CDNA 3 architecture with Matrix Core Technology, supporting

Table 2: Hardware specification comparison across the three target accelerator platforms.

Specification	TPU v6e (Trillium)	NVIDIA H200	AMD MI300X
Peak BF16 Compute	918 TFLOPS/chip	989 TFLOPS	1,307 TFLOPS
HBM Capacity	32 GB	141 GB	192 GB
HBM Bandwidth	1.6 TB/s	4.8 TB/s	5.3 TB/s
Interconnect Topology	2D Torus (ICI)	NVLink	Infinity Fabric
Pod / Node Scale	256 chips	8 GPUs (DGX)	8 GPUs (MI300X)
Sparse Acceleration	SparseCore (2/chip)	Structured Sparsity	Sparse Matrix Ops

precisions including FP8 and MXFP4. The CDNA 3 compute units operate on 64-wide wavefronts (compared to NVIDIA's 32-wide warps), with each compute unit containing dedicated Matrix Cores for accelerated GEMM operations.

A key distinction from NVIDIA's CUDA ecosystem is the use of RCCL (ROCm Communication Collectives Library), AMD's fork of NCCL. While the XLA compiler shares GPU scheduling code between NVIDIA and AMD backends (`gpu_hlo_schedule.cc`), runtime behavior differs: RCCL operates over AMD's Infinity Fabric interconnect rather than NVLink, and intra-wavefront communication primitives (DPP, ds_swizzle) differ from NVIDIA's warp-shuffle instructions. Recent XLA contributions have begun closing this gap—for example, promoting generic `gpu.shuffle` operations to AMD-specific DPP register-to-register instructions for reduction kernels, eliminating the shared-memory overhead of the generic `ds_bpermute` path.

In production environments, Meta has deployed MI300X for Llama 3 and Llama 4 inference Meta [2024], demonstrating competitive performance per total cost of ownership (TCO). AMD's rack-scale AI infrastructure designs utilize ROCm to provide a unified software stack for cross-platform model portability.

Table 2 presents a comparative summary of key hardware specifications.

7 Analytical Tools for Performance Diagnostics

Optimizing models within the OpenXLA stack requires understanding the interactions between the HLO graph and the hardware. OpenXLA provides a suite of diagnostic tools.

7.1 XProf

XProf is a profiling suite for JAX, TensorFlow, and PyTorch/XLA that helps developers understand and optimize TPU and GPU application performance. Key components include:

- **Trace Viewer**: Displays an execution timeline for both host and device, identifying communication gaps or idle periods.
- **HLO Op Stats**: Highlights time-consuming operations, providing metrics such as GFLOPS/s and rematerialization overhead.
- **Memory Profile Viewer**: Monitors HBM usage to identify peak heap consumption and potential stack exhaustion.

7.2 hlo-opt: Isolated Pass Measurement

The `hlo-opt` tool executes individual compiler passes independently of the full compilation pipeline. This isolation is invaluable for pinpointing performance regressions. Developers can measure execution time of specific passes—such as `AlgebraicSimplifier` or `HloRematerialization`—on a given input module.

Table 3 summarizes the diagnostic toolkit.

Table 3: OpenXLA diagnostic tools and their primary use cases.

Tool	Primary Use Case	Target Platform
`hlo-opt`	Pass development and IR conversion	CPU, GPU, TPU
`run_hlo_module`	Microbenchmarking HLO snippets	CPU, GPU, TPU
`xprof`	End-to-end execution profiling	GPU, TPU
`multihost_hlo_runner`	SPMD and multi-node benchmarking	Distributed

Table 4: Vendor-reported cost and efficiency estimates (pre-experimental reference). Values sourced from published vendor benchmarks. Independent validation is a goal of the proposed 3×3 study.

Metric	TPU v6e	NVIDIA H200	Advantage
Cost per Hour	~$1.38	~$2.50+	TPU (45% cheaper)
Inference Perf. / $	$4\times$ baseline	Baseline	TPU
Power Efficiency	60–65% less	Baseline	TPU
Framework Maturity	JAX (native)	CUDA (universal)	NVIDIA

8 Economic Motivation for Cross-Platform Benchmarking

The transition from training-dominated to inference-dominated workloads is reshaping the AI infrastructure market. SemiAnalysis SemiAnalysis [2025] predicts that by 2030, inference will consume 75% of all AI compute. In this landscape, platform economics become a decisive factor in infrastructure planning.

Published vendor benchmarks suggest substantial cost differentials between platforms. Google reports that TPU v6e offers up to $4\times$ better performance per dollar compared to NVIDIA H100 for large-scale LLM inference and recommendation workloads Google [2024a]. However, these figures are derived from vendor-controlled configurations and may not generalize to arbitrary workloads or deployment scenarios. Independent validation through standardized benchmarking—such as the methodology proposed in Section 9—is necessary to substantiate or qualify these claims.

9 Systematic Benchmarking Methodology: The 3×3 Experimental Design

We propose a formal 3×3 benchmarking methodology incorporating best practices from MLPerf Mattson et al. [2020] and academic benchmarking methodologies.

9.1 Objectives

The primary objective is to establish a high-fidelity performance baseline for the OpenXLA compiler across heterogeneous architectures. This need is underscored by the current state of OpenXLA's own benchmark infrastructure, which covers only CPU backends with limited model coverage (Gemma 2), leaving GPU and TPU end-to-end benchmarks as a significant gap. Specifically, we seek to:

1. Quantify the abstraction overhead introduced by the StableHLO/PJRT layer compared to native, hardware-specific stacks. For NVIDIA, the native baseline is vLLM with direct CUDA/cuDNN for inference and Megatron-LM for training. For AMD, the baseline is PyTorch with native ROCm (without XLA). For TPU, where JAX is the native frontend and XLA is the only compiler, we measure the overhead of the StableHLO portability layer by comparing direct HLO compilation against StableHLO-to-HLO roundtrip compilation.
2. Evaluate the transferability of target-independent optimizations (e.g., fusion) across systolic array (TPU) and SIMT (GPU) architectures.
3. Assess the scalability of OpenXLA's SPMD partitioning model in multi-node configurations.

9.2 The 3×3 Matrix Configuration

The study evaluates three representative ML workloads across three hardware backends, as shown in Table 5.

Table 5: The 3×3 experimental matrix: three workloads evaluated across three hardware backends. Each cell represents a distinct experiment with defined KPIs.

	TPU v6e	**NVIDIA H200**	**AMD MI300X**
Task A: LLM Inference	TTFT, TPS, $/tok	TTFT, TPS, $/tok	TTFT, TPS, $/tok
Task B: Dense Training	TFLOPS, TTC	TFLOPS, TTC	TFLOPS, TTC
Task C: Sparse Training	SC util, BW	BW, throughput	BW, throughput

Hardware backends.

- **Backend 1**: Google Cloud TPU v6e (Trillium)—specialized rack-scale ASIC with 2D torus interconnects.
- **Backend 2**: NVIDIA H200 (Hopper)—industry-standard general-purpose accelerator with NVLink and CUDA Graphs.
- **Backend 3**: AMD Instinct MI300X—open-software-stack alternative with high HBM capacity and ROCm integration.

Computational tasks.

- **Task A**: LLM Inference (Llama 3.1 70B)—measuring time-to-first-token (TTFT) and throughput (tokens/s) at various concurrency levels.
- **Task B**: Dense Model Training (ViT / ResNet-50)—focusing on peak TFLOPS utilization and algebraic simplification efficacy. FP8 precision is included as an experimental variable on supported backends (H200 and MI300X), following active community interest in FP8 integration within XLA.
- **Task C**: Sparse Embedding Training (DLRM v2)—evaluating SparseCore efficiency vs. GPU memory bandwidth for high-cardinality lookups. Structured 2:4 sparsity patterns are evaluated where hardware support is available.

9.3 Experimental Methodology

To ensure reproducibility and fair comparison, we follow a structured four-step process.

Step 1: Environment and baseline standardization. We utilize a unified software stack based on JAX and the respective PJRT plugins for each hardware type. Input dataset sizes are standardized to exceed host memory, forcing benchmarks to measure actual data movement patterns and memory hierarchies. Model accuracy is verified against reference metrics to ensure compiler optimizations do not compromise numerical integrity. Notably, XLA's lowering of certain operations (e.g., `erf`) can produce shape-dependent numerical results—the same input may yield outputs differing by several ULPs depending on whether the compiler selects a scalar or vectorized code path. We document any such divergences observed across backends and report per-operation numerical tolerance thresholds used for cross-platform validation.

Step 2: HLO capture and isolated micro-benchmarking. HLO dumping is enabled via `XLA_FLAGS="-xla_dump_to=/tmp/experiment"` to capture pre- and post-optimization graphs. The `hlo-opt` tool isolates specific hardware-independent passes (e.g., `ReshapeMover`) to measure their impact across backends. Deviceless compilation for GPU targets identifies optimization opportunities without physical device constraints.

Step 3: End-to-end performance measurement. The 3×3 matrix tasks are executed with profiles captured via XProf. Primary metrics include TTFT and tokens/s for inference, and time-to-convergence for training.

Power consumption is sampled at 1-second intervals using hardware-specific telemetry: `nvidia-smi -query-gpu=power.draw` for NVIDIA H200, `rocm-smi -showpower` for AMD MI300X, and the Google Cloud Monitoring API

Table 6: Anticipated reporting format for Task A: LLM Inference (Llama 3.1 70B).

Metric	TPU v6e	NVIDIA H200	AMD MI300X
TTFT (ms)	—	—	—
TPS (tokens/sec)	—	—	—
Tokens / $	—	—	—
Power (W)	—	—	—
Energy Eff. (tok/s/W)	—	—	—

Table 7: Anticipated reporting format for Task C: Sparse Embedding Training (DLRM v2).

Metric	TPU v6e (SC)	NVIDIA H200	AMD MI300X
Embedding Lookup Latency	—	—	—
HBM Bandwidth Util. (%)	—	—	—
SparseCore Idle Time (%)	—	N/A	N/A
Throughput (samples/s)	—	—	—

(`compute.googleapis.com/instance/accelerator/power_usage`) for TPU v6e. All power measurements are synchronized with throughput metrics and represent steady-state inference after model loading, excluding compilation and initialization overhead. Following Sada et al. [2025], measurement stability is validated by requiring coefficient of variation $< 5\%$ across repeated trials. Energy efficiency (tokens/s/W) is reported alongside throughput and latency for all 9 cells of the 3×3 matrix, enabling direct comparison with published hardware benchmarks and providing actionable guidance for power-constrained deployments.

Step 4: Bottleneck attribution and analysis. XProf's Trace Viewer correlates gaps in accelerator activity with host-side preprocessing bottlenecks. GFLOPS/s for critical kernels (e.g., `matmul`) are compared across architectures to identify hardware-specific stalls. Results are synthesized into a comparative TCO analysis incorporating cloud instance costs versus measured throughput.

Compiler version sensitivity. A critical consideration for reproducibility is that XLA's performance characteristics are sensitive to the specific compiler commit used. Through direct contributions to the OpenXLA codebase, we have observed that individual commits can materially affect benchmark results. For example, the promotion of generic `gpu.shuffle` to AMD-specific DPP instructions affects every reduction kernel on AMD GPUs; the correction of barrier placement in collective permute thunks directly alters measured communication latency; and the GPU performance model's treatment of integer-typed GEMMs can lead to suboptimal fusion decisions for INT8 workloads. All experiments therefore pin a specific XLA commit hash, and we recommend that any reproduction effort use the same commit or document deviations.

9.4 Proposed Reporting Schema and Statistical Protocol

Results will be presented as KPI tables for each task–hardware combination. Tables 6 and 7 show the proposed reporting format for Tasks A and C.

9.4.1 Statistical Protocol

All experiments follow a structured warm-up and measurement protocol:

- **Warm-up phase.** Each experiment executes 10 warm-up iterations (discarded) to ensure JIT compilation, memory allocation, and device initialization are complete before measurement begins.
- **Measurement phase.** A minimum of 30 timed iterations per configuration, with additional iterations added until the 95% confidence interval for the primary metric (TTFT, TFLOPS, or throughput) is within $\pm 3\%$ of the sample mean.

- **Reporting.** All results report median, mean, standard deviation, and 95% confidence intervals. We report both wall-clock time and device-only time (excluding host preprocessing) to separate data pipeline bottlenecks from accelerator performance.
- **Environment control.** Each experiment records: XLA commit hash, PJRT plugin version, driver version, cloud instance type, HBM temperature at start of run, and any co-tenancy warnings. Experiments are repeated across at least 2 independent cloud instances to control for hardware lottery effects.
- **Outlier handling.** Runs exceeding 3σ from the sample mean are flagged and reported separately but excluded from primary statistics.

10 Expected Contributions and Hypotheses

The 3×3 benchmarking framework proposed in this paper is designed to answer three specific questions about the OpenXLA compiler ecosystem. We formalize each as a testable hypothesis.

H1 (Abstraction Overhead). The StableHLO/PJRT portability layer introduces measurable but bounded latency overhead compared to native, hardware-specific compilation paths (e.g., direct CUDA/cuDNN on NVIDIA, native TPU compilation via JAX). We hypothesize this overhead is $< 5\%$ for compute-bound workloads (dense matmul) but potentially $> 15\%$ for memory-bound workloads (sparse embedding lookups) where data movement patterns are less amenable to fusion.

H2 (Optimization Transferability). Target-independent optimizations—particularly operation fusion—transfer unevenly across architectures. We expect fusion to yield the largest gains on TPU v6e, where the systolic array dataflow benefits most from reduced memory round-trips, and comparatively smaller gains on MI300X, where high HBM bandwidth (5.3 TB/s) partially masks the memory wall that fusion addresses. A confounding factor is that XLA's internal cost model—which drives fusion decisions—has known inaccuracies for certain data types: the roofline model for integer GEMMs assumes peak utilization without accounting for kernel launch or quantization overhead, leading to up to 576% prediction error at medium tensor shapes. We control for this by reporting both XLA's default fusion behavior and the effect of manually overriding fusion decisions via `xla_gpu_auto_spmd_partitioning` flags where applicable.

H3 (SPMD Scaling Efficiency). OpenXLA's SPMD partitioning via Shardy scales sub-linearly across multi-node configurations, with the efficiency gap widest on NVIDIA hardware due to the transition from intra-node NVLink to inter-node InfiniBand, and narrowest on TPU pods where the ICI torus provides uniform bisection bandwidth.

10.1 Limitations

This study has several deliberate scope constraints that should be acknowledged:

- **Hardware coverage.** The 3×3 matrix excludes emerging accelerator architectures (Groq LPU, Cerebras WSE, Intel Gaudi 3) that may interact with the XLA compiler differently than the systolic array and SIMT paradigms studied here.
- **Workload coverage.** We do not evaluate fine-tuning, reinforcement learning, or mixture-of-experts workloads, all of which present distinct compilation challenges (irregular compute graphs, dynamic routing).
- **Framework scope.** Our experiments use JAX as the sole frontend. While OpenXLA supports TensorFlow and PyTorch/XLA, cross-framework comparison would introduce confounds from framework-level overhead unrelated to the compiler.
- **Temporal validity.** Hardware specifications and compiler optimizations evolve rapidly. Results from the proposed experiments will reflect a point-in-time snapshot of the XLA compiler at a specific commit hash, which we will report for reproducibility.

10.2 Future Work

Beyond the initial 3×3 evaluation, several extensions are natural:

1. **Native baseline comparison.** Pair each OpenXLA experiment with a native-stack equivalent (e.g., vLLM on CUDA for inference, Megatron-LM for training) to directly measure the portability tax.
2. **Cost-performance Pareto analysis.** Combine measured throughput with real-time cloud pricing to produce Pareto frontiers for each task, enabling practitioners to make informed platform selection decisions.
3. **Compiler ablation study.** Use `hlo-opt` to selectively disable individual optimization passes (fusion, CSE, algebraic simplification) and measure their isolated contribution on each backend.
4. **Community benchmark registry.** Release HLO snapshots and profiling scripts as a public benchmark suite, enabling third-party reproduction and extension to new hardware.
5. **Specialized inference accelerator comparison.** Extend the benchmark matrix to include purpose-built inference chips (Qualcomm Cloud AI 100 Ultra, Groq LPU) to evaluate whether OpenXLA's portability layer can target non-GPU/non-TPU accelerators without sacrificing energy efficiency, building on the cross-architecture methodology of Sada et al. Sada et al. [2025].

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 265–283, 2016.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: Composable transformations of Python+NumPy programs. `http://github.com/jax-ml/jax`, 2018.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Google. OpenXLA: An open ecosystem of portable and composable ML infrastructure. `https://github.com/openxla`, 2023–2024.

Google. Shardy: MLIR-based tensor partitioning. `https://github.com/openxla/shardy`, 2024.

IREE Authors. IREE: Intermediate Representation Execution Environment. `https://iree.dev`, 2023.

Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 75–86, 2004.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: Scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021.

Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., Tang, H., Wei, G.-Y., Bailis, P., Bittorf, V., et al. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020.

Meta. Deploying Llama at scale with AMD MI300X. Meta Engineering Blog, 2024.

NVIDIA. NVSHMEM: NVIDIA OpenSHMEM Library. `https://developer.nvidia.com/nvshmem`, 2024.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 8024–8035, 2019.

Sada, R., Papadimitriou, G., Nandagiri, S., Zhuang, Z., Majumdar, S., and Würthwein, F. Serving LLMs in HPC clusters: A comparative study of Qualcomm Cloud AI 100 Ultra and NVIDIA data center GPUs. *arXiv preprint arXiv:2507.00418v3*, 2025.

SemiAnalysis. The inference economy: Why 75% of AI compute will be inference by 2030. Technical report, 2025.

A Extended TPU Specification Comparison

Table 8: Extended TPU specification comparison between v5p and v6e (Trillium).

Specification	**TPU v5p**	**TPU v6e (Trillium)**
Peak BF16 Compute	459 TFLOPS/chip	918 TFLOPS/chip
Systolic Array Size	128×128	256×256
MACs per Cycle	16,384	65,536
HBM Capacity	96 GB	32 GB
HBM Bandwidth	2.8 TB/s	1.6 TB/s
ICI Topology	3D Torus	2D Torus
Pod Size (Chips)	8,960	256
SparseCores	4 per chip	2 per chip