VKML: Inference Without Lock-In

Demonstrating Performant, Practical, and Heterogeneous Inference on Vulkan

Lucas Hofmockel-Spanakis Greg Baker

Abstract

The dominance of proprietary compute stacks such as CUDA has created a bifurcation in machine learning: developers must choose between performance with vendor lock-in or portability with suboptimal efficiency. We present VKML, a greenfield Vulkan-based ONNX inference runtime implemented in Rust. VKML leverages the explicit control of Vulkan 1.4 to provide a unified, vendor-agnostic compute backend that runs natively on hardware ranging from legacy NVIDIA GPUs to modern mobile SoCs.

We evaluate VKML against ONNX Runtime (ONNX-RT) using the CUDA Execution Provider on identical NVIDIA hardware. VKML achieves a 1.66× speedup (97 µs vs 161 µs median) on the mnist-12 micro-model and a 1.46× speedup on a FP32 convolution-heavy network, driven by minimized driver dispatch overhead that significantly lowers the cost per operation. Beyond raw latency, we demonstrate unique capabilities enabled by Vulkan's heterogeneity: the automatic, graph-based partitioning of a single model across hardware from different vendors (NVIDIA dGPU and Intel iGPU) using a single unified compute framework, along with an order-of-magnitude reduction in host memory usage.

Crucially, we address the historical performance gap in matrix multiplication by architecting support for the `VK_NV_cooperative_matrix2` extension. While our current evaluation utilizes portable tiled kernels, VKML is verified to enable this extension on compatible hardware, providing the infrastructure for mixed-precision Tensor Core acceleration. This approach is supported by recent industry findings demonstrating that Vulkan compute kernels are highly competitive with native CUDA, in some cases even exceeding its performance [1], suggesting that the performance floor for Vulkan-based ML is now near or at parity with proprietary stacks.

1 Introduction

The landscape of high-performance computing is defined by a trilemma: one can typically choose only two of *no vendor lock-in*, *high performance*, and *ease of use*. CUDA prioritizes performance and ease of use at the cost of total vendor lock-in. OpenCL and WebGPU offer portability but often trail in absolute performance or feature set. Vulkan, primarily a graphics API, offers a path to portability and high performance, but is notoriously difficult to use.

The maturation of the Vulkan ecosystem, coupled with its native support for advanced compute features, prompts a re-evaluation of its role in the machine learning stack: Can a vendor-agnostic API serve as the primary backend for general-purpose, high-performance inference?

Inspired by the success of projects like VkFFT [2], which demonstrated that Vulkan Fast Fourier Transforms can match or exceed CUDA performance, we developed VKML to test this hypothesis for deep learning. The timing is critical. Recent presentations at Vulkanised 2025 [1] have shown that Vulkan extensions for hardware-accelerated matrix multiplication can reach performance levels highly competitive with, and in certain workloads exceeding native CUDA. Furthermore, the ecosystem has matured: Vulkan 1.4 is supported on hardware as old as the NVIDIA GTX 750 (2014) [3], on Apple Silicon via MoltenVK [4], Imagination chips [5], Android phones with Qualcomm and MediaTek chips, AMD integrated and discrete GPUs, Intel integrated and discrete GPUs, and many more [6].

This work is driven by a primary research question: Can a widely supported open standard serve as a competitive foundation for high-performance inference, challenging the dominance of vendor-locked solutions? We decompose this into two distinct investigations. First, we examine whether the explicit control provided by Vulkan enables a third-party runtime to match, or even exceed, native CUDA performance on identical NVIDIA hardware. To isolate the contribution of Vulkan itself from VKML's specific architecture, we additionally benchmark against `ncnn` [7], a mature Vulkan-based inference framework. Second, we explore whether a framework architected from scratch around these primitives can inherently unlock advanced capabilities, such as running a segmented compute graph across multi-GPU vendor systems without introducing the complexities of mixing different proprietary compute libraries.

Contributions

1. **VKML Architecture:** A runtime library featuring a custom lock-free event-based graph scheduler built on a custom thread pool, `zero-pool` [8], and a new zero-copy ONNX model file and data loader, `onnx-extractor` [9].
2. **Native Heterogeneity:** A demonstration of automatic graph partitioning across different-vendor GPUs (e.g., NVIDIA + Intel) within a single process, managed by a greedy allocator that injects transfer operations transparently.
3. **Closing the GEMM Gap:** Infrastructure for `VK_NV_cooperative_matrix2`, verified to enable hardware-specific paths for Tensor Core utilization on compatible devices.
4. **Empirical Evaluation:** Competitive results on small-to-medium models (up to 1.46× faster than ONNX-RT on a 266 MiB CNN), with an analysis of the remaining bottlenecks in pure FP32 workloads.

2 Background and Terminology

The current landscape of GPU-accelerated machine learning is dominated by proprietary stacks, most notably NVIDIA's CUDA. To understand the architectural decisions behind VKML, it is necessary to contrast the design philosophies of these established proprietary APIs with the open standard of Vulkan.

CUDA benefits from tight vertical integration where NVIDIA controls the hardware, the driver, and the API specification. This enables rapid deployment of bleeding-edge features and a dual-layer abstraction model. Developers can utilize high-level libraries like cuDNN and cuBLAS, which abstract away hardware details, or drop down to the CUDA Runtime and Driver APIs for finer control. In contrast, Vulkan is a specification maintained by the Khronos Group. It defines a contract that hardware vendors (such as NVIDIA, AMD, Intel, and Qualcomm) must implement via their own drivers. While this enables the "write once, run anywhere" promise of cross-vendor portability, it fundamentally alters the innovation cycle. New features must first be ratified by a working group and then implemented by individual vendors, often leaving the standard spec trailing behind proprietary innovations.

The most significant divergence lies in the responsibility for state management. CUDA's runtime API often manages synchronization and memory residency implicitly. For instance, a sequence of kernel launches on a CUDA stream is guaranteed to execute in order. Vulkan, conversely, adheres to an explicit low-level philosophy where the specification provides no safety net against race conditions; the application is solely responsible for defining execution dependencies. This manifests primarily in two areas: synchronization and memory management. In Vulkan, the runtime must manually inject *Pipeline Barriers* to manage memory availability and visibility between operations. Failure to do so results in undefined behavior rather than a driver-managed stall. Furthermore, while CUDA exposes flat memory pointers, Vulkan exposes

the raw GPU memory hierarchy, requiring applications to query heaps and bind memory to resources manually. Rather than enforcing a rigid driver-level policy, this open design encourages the creation of tailored abstractions, allowing developers to either leverage established solutions like the Vulkan Memory Allocator (VMA) [10] or implement custom allocation strategies optimized for their specific constraints.

This explicit control comes at the cost of significant engineering complexity compared to managed runtimes. A simple inference operation in Vulkan requires a verbose setup that involves managing *Physical* and *Logical Devices*, recording *Command Buffers*, and handling *Fences* for host-device signaling. However, this verbosity eliminates the "black box" driver overheads inherent in high-level runtimes. By removing the driver's need to guess the application's intent, Vulkan offers a theoretical ceiling for latency minimization that exceeds that of managed runtimes.

The central question this paper addresses is whether this theoretical advantage holds in practice. Is the Vulkan ecosystem sufficiently mature to serve as a competitive backend for machine learning, matching the throughput of vendor-locked libraries while offering the latency benefits of a thin driver layer?

3 Design of VKML

3.1 Goals

The primary goal of VKML is to demonstrate that a unified compute API can be:

- **Vendor Agnostic:** Running the same compiled binary on NVIDIA, Intel, and AMD graphics hardware.
- **Performant:** Minimizing driver overhead to compete with or beat proprietary runtimes on forward pass inference latency.
- **Heterogeneous:** Treating all available compute devices (dGPU, iGPU, CPU) as a single compute pool.

3.2 Architecture

VKML is architected as a graph-based execution engine that strictly separates resource allocation from execution planning. At its core, the runtime is built upon `zero-pool` [8], a custom lock-free thread-pool library designed to minimize latency and ensure high consistency. This foundation allows VKML to employ zero-copy data structures even across multi-threaded contexts, granting the runtime precise control over data residency and flow.

The execution pipeline begins with the `ComputeManager`, which employs a greedy allocation strategy to partition the compute graph. It iterates through operations in topological order, assigning each to the first device that satisfies its memory requirements. This allocation process is fully automatic; when an operation is assigned to a device distinct from its inputs, the allocator transparently injects explicit `TransferToDevice` instructions and manages the allocation of intermediate tensors on the target device. This design enables seamless heterogeneous execution, allowing a single graph model to be partitioned across different vendor physical devices automatically.

To minimize submission overhead, the scheduler groups contiguous operations assigned to the same device into *Execution Chunks*, each recorded as a single Vulkan command buffer. Inter-chunk dependencies are resolved through a self-propagating, event-driven mechanism. As each chunk completes, it decrements the dependency counters of its successors, enabling autonomous graph execution without centralized coordination.

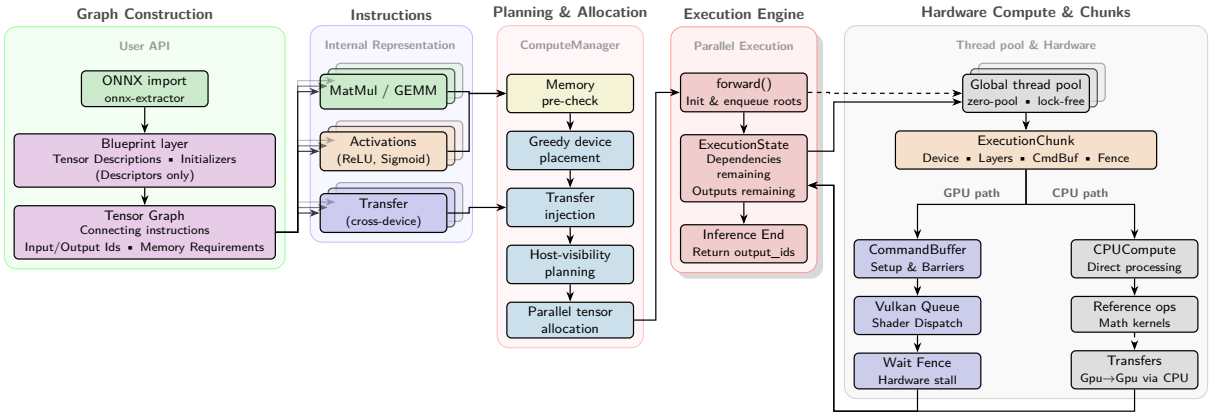

Figure 1: The modular architecture of VKML, illustrating the flow from ONNX model ingestion to hardware-specific execution.

4 Implementation Overview

VKML is implemented using the `vulkanalia` [11] library for Vulkan bindings and `libloading` [12] for dynamic loading. The runtime architecture is designed to map high-level deep learning operations directly to low-level Vulkan primitives with minimal abstraction overhead.

At the user level, the barrier to running an ONNX model in VKML is intentionally simple. A model can be loaded from disk, executed, and its outputs read back with a small number of library calls.

```
use vkml::{ComputeManager, DataType, Tensor, TensorDesc, VKMLError};

fn main() -> Result<(), VKMLError> {
  let mut manager = ComputeManager::new_from_onnx_path("mnist-12.onnx")?;

  let desc = TensorDesc::new(vec![1, 1, 28, 28], DataType::Float);
  let input = Tensor::new_cpu(desc.clone(), vec![0u8; desc.size_in_bytes()].into());

  let out_ids = manager.forward(vec![input])?;
  let outputs = manager.tensor_read_vec(&out_ids);
  Ok(())
}
```

Listing 1: VKML usage for loading and executing an ONNX model with a zeroed input.

The execution lifecycle begins with the loading of an ONNX model. VKML utilizes a loader, `onnx-extractor`, which parses the model schema and maps tensor data into memory. Upon initialization, the runtime queries the host system to identify all Vulkan-capable devices. Rather than abstracting these into a single virtual device, VKML maintains a pool of distinct physical devices, allowing the upper layers to orchestrate work while strictly respecting the individual memory limits and compute capabilities of each hardware component.

A core component of the initialization phase is the `ComputeManager`, which performs a static analysis of the ONNX graph to generate a graph partitioning map from model operations to device memory. This process employs a greedy algorithm that partitions the graph across available devices. The allocator prioritizes placing operations on discrete GPUs, falling back to integrated GPUs, and finally the CPU only when graphics memory limits are exceeded. Crucially, this allocator is hardware-aware: on systems supporting Above 4G Decoding (Resizable BAR; ReBAR), it maps model weights and inputs directly into the GPU's host-visible memory region,

bypassing staging buffers entirely. On legacy hardware, it automatically falls back to a staged streaming approach.

For execution, the linear sequence of graph operations is compiled into a directed acyclic graph of *Execution Chunks*. A chunk represents a subgraph of operations that can be executed on a single device without interruption. This grouping strategy is key to performance: rather than submitting a separate command buffer for every layer, a common source of overhead in naive implementations, VKML records entire subgraphs into a single command buffer. Within these chunks, the graph compiler inserts fine-grained pipeline barriers using `vkCmdPipelineBarrier2` to manage data hazards between operations, thereby maximizing the freedom of the vendor-specific Vulkan driver to optimize workgroup scheduling, parallel execution, and cache residency for its underlying hardware architecture.

Runtime execution is driven by an event-based graph scheduler. When a forward pass is initiated, input tensors are copied to their assigned devices, utilizing parallel transfer queues for multi-tensor inputs, and the scheduler begins dispatching chunks whose dependencies are satisfied. As each chunk completes, it signals a set of atomic counters on its dependent chunks; once a counter reaches zero, the dependent chunk is submitted for execution immediately. This self-propagating mechanism enables autonomous, data-driven graph execution, synchronizing with the host only when data must cross a PCIe bus between different devices or when the final output tensors are ready to be read back to the application.

While the current implementation utilizes portable tiled SPIR-V kernels to ensure broad compatibility and verify heterogeneity, the architecture is designed to support vendor-specific extensions. The infrastructure for `VK_NV_cooperative_matrix2` is already present, allowing future versions to seamlessly swap generic kernels for hardware-accelerated implementations on supported hardware without altering the high-level execution flow.

5 Experimental Methodology

All latency-sensitive experiments were conducted on a single desktop workstation. Table 1 summarizes the hardware and software configuration.

Table 1: Hardware and software configuration for all benchmarks.

Hardware	
CPU	AMD Ryzen 9 5900X @ 4.8 GHz
RAM	32 GB DDR4 @ 3600 MHz
GPU	NVIDIA GeForce RTX 3080
OS	Linux (Kernel 6.18.5)
Software	
VKML	Rust 1.92 (release mode)
ONNX-RT	1.23.2 (`CUDAExecutionProvider`)
Python	3.14.2
CUDA	13.1
cuDNN	9.17.1.4
NVIDIA Driver	590.48.1
`ncnn`	1.0.20260114

5.1 Workloads

We evaluate VKML across a range of models representing different compute characteristics. Table 2 details the specific models used, including their size, operation count, and types.

Table 2: Characteristics of the evaluated models. All models use FP32 precision.

Model Filename	Size (MiB)	# Ops	Operation Types
`mnist-12.onnx`	0.15	12	Conv, Add, ReLU, MaxPool, Reshape, MatMul
`mnist_mlp_png_probs.onnx`	0.91	7	Reshape, Gemm, ReLU, Softmax
`mnist_conv-few-15MB.onnx`	15.14	11	Conv, ReLU, MaxPool, Reshape, MatMul, Add, Softmax
`mnist_conv-many-255MB.onnx`	266.37	15	Conv, ReLU, MaxPool, Reshape, MatMul, Add, Softmax
`mnist_matmul-1024MB-16layers.onnx`	972.66	19	Reshape, MatMul, Softmax
`mnist_matmul-1024MB-1layers.onnx`	1024.81	4	Reshape, MatMul, Softmax
`multi_chain_add-N.onnx`	<1	N	Add (Chained)

Our primary demonstration of VKML's low-overhead advantage is `mnist-12.onnx`, a publicly available test model from the ONNX repository[1]. This micro model is dominated by driver submission and runtime overheads rather than raw compute, making it an ideal stress test for the efficiency of the Vulkan backend.

To demonstrate scaling, we include `mnist_conv-many-255MB.onnx`, a ∼266 MiB model. This workload shows that VKML's performance advantages extend to larger, convolution-heavy graphs, where it outperforms ONNX-RT despite the increased computational load.

5.2 Verification & Precision

With the exception of the standard `mnist-12` benchmark, all models were created and trained specifically for this evaluation using PyTorch, then exported to `.pt` and `.onnx`. For comparison against `ncnn` [7], a mature, mobile-first Vulkan framework, we use the `pnnx` [13] package to convert the `.onnx` file to FP32 `.param` and `.bin` formats. Converted model files had their operations validated to ensure they were as identical as each framework's internal representation allows, using both the internal inspection tools of each library and a third-party visualizer `Netron` [14]. The model set includes a mix of networks trained to varying degrees of convergence. Correctness was verified by comparing VKML's outputs against PyTorch, ONNX-RT, and `ncnn`. All four frameworks demonstrated consistency to six decimal places. Divergence beyond the sixth decimal place is expected due to the non-associative nature of floating-point arithmetic ($(a + b) + c \neq a + (b + c)$). In IEEE 754 single-precision floating-point format (float32) [15], the 23-bit significand guarantees approximately 7.22 decimal digits of precision ($\log_{10}(2^{24}) \approx 7.22$). Since every library employs different optimization strategies for operation order, minor deviations in the 7th decimal place are an inherent property of 32-bit floating-point math rather than a calculation error.

5.3 Protocol

We report the *second-pass* inference latency with a batch size of 1. The first pass is excluded to avoid measuring compilation and initialization overheads, ensuring a fair comparison of steady-state runtime performance. To ensure a rigorous evaluation of general-purpose inference capabilities, VKML is compiled as a model-agnostic runtime. No model-specific optimizations or static graphs are embedded within the executable; all architecture definitions, weights, and operation dependencies are parsed and processed dynamically at runtime. We compare against ONNX-RT running on the same hardware with the `CUDAExecutionProvider`. ONNX-RT serves as our representative CUDA-backed baseline, excluding PyTorch from the final results as it was consistently outperformed by ONNX-RT across our tested benchmarks.

Unless otherwise noted, ONNX-RT was configured with default session and provider options. We additionally include `ncnn` as a second Vulkan-based baseline to separate VKML's architectural contributions from advantages inherent to the Vulkan API. To ensure a fair comparison, `ncnn` was

[1] `https://github.com/onnx/models/blob/main/validated/vision/classification/mnist/model/mnist-12.onnx`

configured to strictly utilize FP32 compute paths by disabling its default Tensor Core acceleration and FP16 downsampling using the `use_sgemm_convolution=False`, `use_fp16_packed=False`, and `use_fp16_storage=False` flags. This ensures that both Vulkan-based runtimes operate on a strictly FP32 foundation, whereas the ONNX-RT baseline maintains opaque Tensor Core and TF32 utilization in its CUDA backend; these features could not be fully bypassed without compromising the runtime's internal optimization engine. As demonstrated in our results (Table 3), this acceleration advantage is isolated to large matrix multiplication workloads, while other operation types remain directly comparable.

Where appropriate, we additionally report percentile statistics (e.g., p05..p95 and p99) to characterize tail latency on dispatch-dominated workloads.

6 Results

We evaluate VKML against two baselines: ONNX-RT with the CUDA Execution Provider, representing the proprietary state of the art, and `ncnn`, a mature Vulkan framework that serves as a control for isolating VKML's architectural contributions from the inherent characteristics of Vulkan. Our analysis centers on the VKML vs. ONNX-RT comparison, with `ncnn` results used to contextualize the source of observed performance differences.

Table 3: Performance comparison between VKML, ONNX-RT and `ncnn`. We report the second-pass inference latency (steady state), the maximum resident set size (host memory usage), and the total execution time for the benchmark process (including startup/teardown). Memory and total runtime data captured using the GNU `time` utility.

	2nd Pass Latency (ms)			Host Memory (MiB)			Total Runtime (s)		
Model	**VKML**	**ONNX-RT**	**ncnn**	**VKML**	**ONNX-RT**	**ncnn**	**VKML**	**ONNX-RT**	**ncnn**
mnist-12.onnx	**0.10**	0.16	0.59	**117**	1351	212	**0.22**	0.65	0.56
mnist_mlp_png_probs.onnx	**0.08**	0.13	0.61	**117**	1227	212	**0.26**	0.58	0.53
mnist_conv-few-15MB.onnx	**1.85**	3.53	2.54	**127**	1241	236	**0.25**	0.60	0.57
mnist_conv-many-255MB.onnx	**13.62**	19.94	13.67	**378**	1446	673	**0.39**	0.84	0.71
mnist_matmul-1024MB-16layers.onnx	4.31	**1.61**	7.80	**1087**	1462	1246	**0.62**	1.30	1.08
mnist_matmul-1024MB-1layers.onnx	19.09	**1.63**	20.24	**1138**	2445	2244	**0.74**	1.34	1.02

6.1 Performance Analysis

The results highlight VKML's competitive advantage in latency-sensitive scenarios. For small, operation-dense graphs and medium-sized convolutional networks, VKML consistently outperforms ONNX-RT, driven by its minimized dispatch overhead and graph-aware synchronization.

Beyond raw latency, VKML demonstrates a fundamental efficiency advantage in system resource utilization. As shown in Table 3, VKML consistently requires an order of magnitude less host memory than ONNX-RT; for the `mnist-12` workload, VKML consumes just 117 MiB compared to ONNX-RT's 1351 MiB. This efficiency extends to startup time, where VKML initializes and completes two forward passes $\sim 3\times$ faster (0.22 s vs 0.65 s). These gains validate the premise that by removing the heavy abstraction layer of CUDA, Vulkan allows for a significantly lighter base application footprint.

The `ncnn` results clarify the source of these gains. On dispatch-dominated workloads (e.g., `mnist-12`), `ncnn`'s latency (0.59 ms) substantially exceeds both VKML (0.10 ms) and ONNX-RT (0.16 ms), indicating that Vulkan alone does not guarantee low-overhead dispatch; VKML's scheduler and cache systems are the primary drivers. However, on the compute-bound 266 MiB convolution model, `ncnn` (13.67 ms) converges with VKML (13.62 ms), and both outperform ONNX-RT (19.94 ms), suggesting a structural advantage in Vulkan's explicit pipeline model for these workloads. Both Vulkan runtimes also exhibit the same performance deficit on large

MatMul workloads relative to ONNX-RT, confirming that this gap is a consequence of portable FP32 kernels rather than a VKML-specific limitation.

6.2 Memory Scaling Analysis

A distinct difference is observed in the memory scaling of ONNX-RT on the large 1 GiB models (`mnist_matmul-1024MB`). Despite both models having the same total weight size, the 1-layer variant consumes 2445 MiB of host memory, while the 16-layer variant consumes only 1462 MiB. This discrepancy illustrates the impact of differing allocation strategies and initialization costs between the runtimes.

First, we observe a significant baseline memory footprint associated with the CUDA context. Even for the 0.15 MiB `mnist-12` model, ONNX-RT reports a maximum resident set size of approximately 1.32 GiB when using the `CUDAExecutionProvider`. We isolated this cost by running the same model with the `CPUExecutionProvider`, which reduced the footprint to ~60 MiB. This indicates that the initial gigabyte of usage is the fixed cost of initializing the CUDA and cuDNN driver runtime libraries. VKML, by targeting the lower-level Vulkan API, maintains a significantly smaller graphics-accelerated runtime footprint (~117 MiB).

Second, we observe a difference in model loading strategies. ONNX-RT employs a file-streaming approach to minimize peak memory usage. This is evident in the 16-layer model (total 1 GiB, each layer ~62 MiB), where memory usage remains low (~1.43 GiB) because individual layers are loaded, transferred, and freed one by one. However, the 1-layer model forces ONNX-RT to allocate a single contiguous 1 GiB buffer, causing a massive spike to 2.39 GiB. In contrast, the current implementation of VKML eagerly loads the entire model into host memory before GPU allocation. This means that VKML's host memory usage scales linearly with total model size.

6.3 Driver Overhead Analysis

To isolate the impact of driver submission overhead, we evaluated VKML against ONNX-RT and `ncnn` on a synthetic benchmark consisting of a chain of N dependent vector additions (`multi_chain_add-N.onnx`), scaling N from 1 to 10,000. Each instruction in the chain performs an addition where a vector of size 1 with value 0.0 is added to the result of the previous operation. For this specific benchmark, we configured ONNX-RT with `ort.GraphOptimizationLevel.ORT_DISABLE_ALL` to prevent the runtime from fusing the chain into a single kernel or instruction. This ensures that the test measures the dispatch cost per operation rather than the execution of a single optimized kernel, providing a fair comparison where the runtimes must submit and execute N distinct operations. For each N, we recorded performance over 120 iterations, discarding the first 20 runs as warmup. This workload minimizes arithmetic intensity, making the dispatch efficiency of the runtime the dominant factor. Table 4 details the breakdown of these results.

Table 4: Driver overhead scaling results. We report the median latency and 5th-95th percentile spread for each chain length N, excluding the first 20 runs of 120 as warmup. Speedup columns show performance relative to VKML.

N	VKML Median (ms) [p05..p95]	ONNX-RT Median (ms) [p05..p95]	ncnn Median (ms) [p05..p95]	ONNX-RT/VKML	ncnn/VKML
1	0.0356 [0.0344..0.0389]	0.0722 [0.0703..0.0804]	0.5185 [0.5045..0.5840]	2.03×	14.57×
10	0.0508 [0.0480..0.0553]	0.0882 [0.0854..0.0966]	0.5446 [0.5320..0.5872]	1.74×	10.72×
100	0.1806 [0.1788..0.1959]	0.2786 [0.2749..0.2922]	0.8230 [0.8054..2.7050]	1.54×	4.56×
1000	1.4755 [1.4635..1.4891]	2.1310 [2.1169..2.1956]	3.8445 [3.6154..4.5219]	1.44×	2.61×
10000	16.2286 [16.2213..16.3555]	20.6214 [20.4482..21.1759]	35.5254 [35.2215..36.4755]	1.27×	2.19×

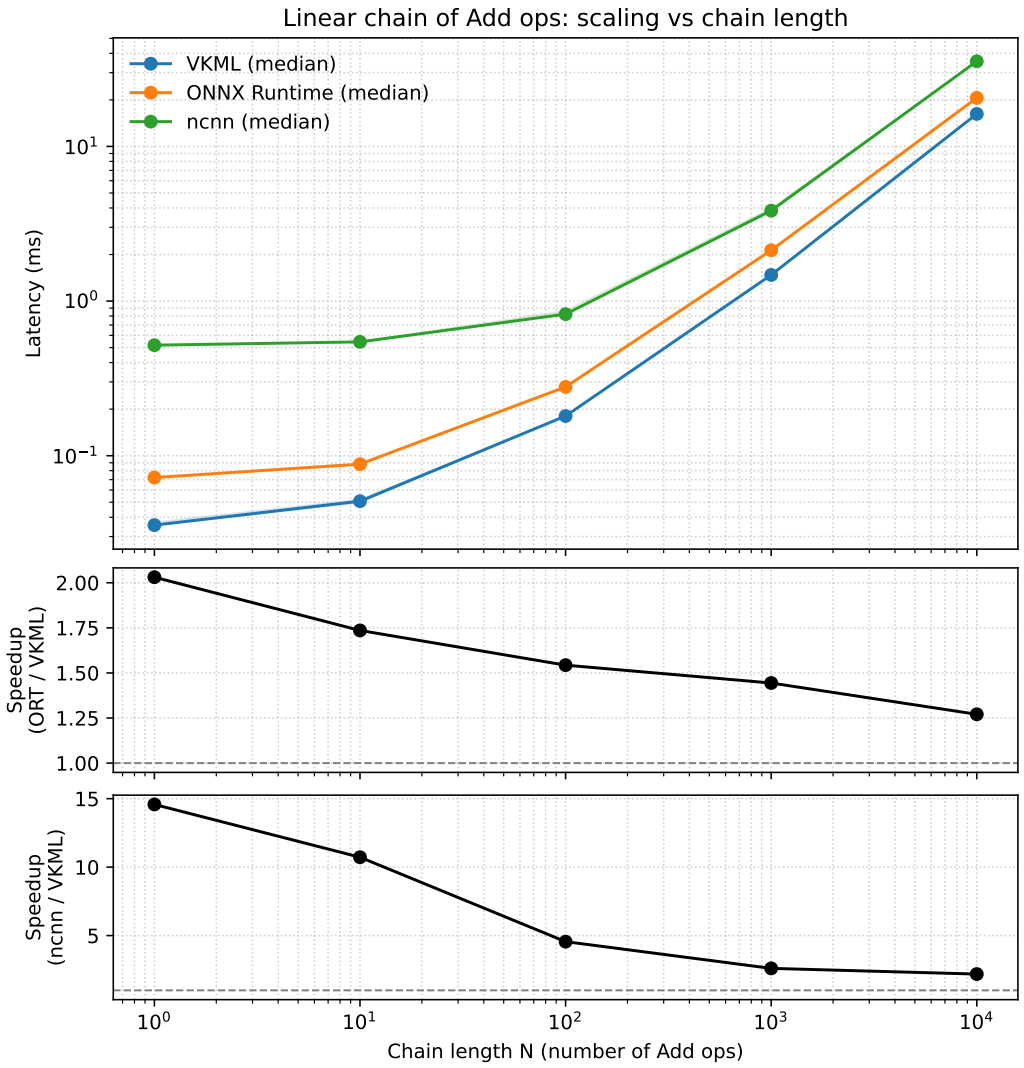

Figure 2: Speedup of VKML over ONNX-RT and `ncnn` as a function of chain length N. VKML maintains a consistent lead, with the highest relative performance at low N due to lower constant overhead.

As shown in Figure 2, VKML consistently outperforms ONNX-RT. For a single operation ($N = 1$), VKML achieves a median latency of 0.036 ms versus 0.072 ms for ONNX-RT, a **2.03**× speedup. This advantage persists as the chain grows:

- $N = 10$: 1.74× speedup (0.051 ms vs 0.088 ms)
- $N = 1000$: 1.44× speedup (1.48 ms vs 2.13 ms)

Linear regression analysis of the latency ($L \approx C + S \cdot N$) reveals that VKML has both a lower constant overhead (C) and a lower per-operation dispatch cost (S). The per-operation slope for VKML is 1.62 µs/op compared to 2.06 µs/op for ONNX-RT. This confirms that VKML's command recording and submission path is leaner. `ncnn` exhibits a markedly higher constant overhead (0.46 ms) and a per-operation cost of 3.51 µs/op.

6.4 Latency Distribution

While average latency provides a high-level view, consistency is critical for real-time applications. We executed 1,100 runs of the `mnist-12.onnx` model, discarding the first 100 as warmup, to analyze the latency distribution of the three runtimes.

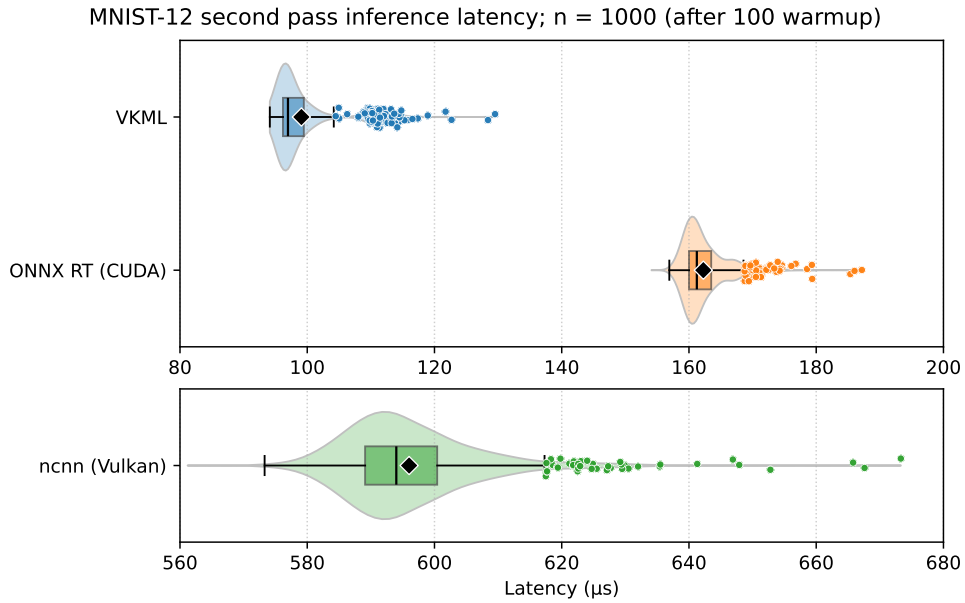

Figure 3: Latency distribution for VKML vs. ONNX-RT and `ncnn` (N=1000). Results are presented in two panels with shifted x-axes and a consistent horizontal scale across all frameworks. While ONNX-RT shows tighter variance, its fastest runs are slower than VKML's slowest runs.

As illustrated in Figure 3 and detailed in our statistical analysis:

- **Median Latency:** VKML achieved a median of 97.0 µs, compared to 161.2 µs for ONNX-RT.
- **Consistency:** VKML exhibited a tighter interquartile range (IQR 3.3 µs vs. 3.5 µs for ONNX-RT), but a higher standard deviation (5.2 µs vs 3.5 µs) due to outliers. We attribute these outliers to the current VKML implementation allocating redundant host memory during a forward pass over a cached model.
- **Tail Latency:** Crucially, VKML's 99th percentile latency (114.8 µs) remains over 46 µs faster than ONNX-RT's median.

This suggests that while the current design of VKML exhibits increased variability, the minimized dispatch overhead of the Vulkan-native approach establishes a latency floor significantly lower than that of the managed baseline.

6.5 Mixed Precision (Extension-Ready)

VKML is designed to incorporate Vulkan cooperative-matrix extensions for mixed precision on supported hardware. Since this capability is still evolving and not yet representative of VKML's general-purpose FP32 performance, we present preliminary mixed-precision results in Section A.

7 Cross-Vendor Heterogeneous Execution (Capability Demonstration)

To validate VKML's ability to orchestrate execution across hardware from different vendors, we deployed the runtime on a laptop equipped with an Intel Core i7-10750H, an NVIDIA GTX 1660 Ti, and an Intel UHD Integrated GPU. We artificially constrained the memory budget of the discrete GPU to force the scheduler to partition the 266 MiB CNN model (`mnist_conv-many-255MB.onnx`).

7.1 Execution Flow & Performance

VKML's greedy allocator successfully partitioned the graph into five execution chunks, automatically injecting transfer operations where dependencies crossed device boundaries. Listing 2 illustrates the generated execution plan, while Table 5 details the execution time for each chunk during the steady-state second forward pass.

```
Chunk 0: device=GPU 0 ops=8 layers=8 preds=[] deps=[1]
  initial_dep_count=0 is_output=false needs_host_wait_fence=true

Chunk 1: device=CPU ops=1 layers=1 preds=[0] deps=[2]
  Instruction: TransferToDevice(src=6, dst=27, from=Gpu(0), to=Gpu(1))

Chunk 2: device=GPU 1 ops=1 layers=1 preds=[1] deps=[3]
  initial_dep_count=1 is_output=false needs_host_wait_fence=true

Chunk 3: device=CPU ops=1 layers=1 preds=[2] deps=[4]
  Instruction: TransferToDevice(src=14, dst=28, from=Gpu(1), to=Gpu(0))

Chunk 4: device=GPU 0 ops=6 layers=6 preds=[3] deps=[]
  initial_dep_count=1 is_output=true needs_host_wait_fence=true
```

Listing 2: Runtime execution plan showing automatic heterogeneous partitioning.

Table 5: Execution breakdown of the heterogeneous forward pass. Times reported are from the second pass. Total includes other overheads.

Chunk	**Device**	**Operation Type**	**Time (ms)**
Chunk 0	NVIDIA GTX 1660 Ti	Compute (Operations 0-7)	17.29
Chunk 1	CPU (PCIe Bridge)	Transfer (NVIDIA $\rightarrow$ Intel)	7.43
Chunk 2	Intel UHD Graphics	Compute (Operation 8)	534.15
Chunk 3	CPU (PCIe Bridge)	Transfer (Intel $\rightarrow$ NVIDIA)	0.64
Chunk 4	NVIDIA GTX 1660 Ti	Compute (Operations 9-14)	0.69
Total			**560.37**

The total execution time for the split workload was 560.37 ms. For context, running the entire model purely on the NVIDIA GTX 1660 Ti took 53.41 ms, while running purely on the Intel UHD graphics took 785.91 ms. This result demonstrates a capability: VKML can correctly execute a single ONNX graph across multiple vendor devices without code changes when the model size exceeds the memory capacity of any single GPU.

7.2 Resource Usage & Precision

A key concern in multi-device inference is the overhead of maintaining contexts for multiple backends. We observed that VKML's host memory usage remained remarkably consistent across device configurations. The runtime consumed 341 MiB for the NVIDIA-only configuration, 334 MiB for the Intel-only configuration, and 373 MiB for the heterogeneous split. Algebraic analysis of these footprints allows us to isolate the per-vendor driver overhead: approximately 32 MiB for the Intel context and 39 MiB for the NVIDIA context, with the base VKML runtime (excluding model data) accounting for 36 MiB. The cost of enabling a second heterogeneous device is therefore lightweight, adding only the specific driver overhead of the new hardware without compounding global state.

Furthermore, we verified that there was no loss of precision during the cross-vendor transfers. The final output tensors for the NVIDIA-only, Intel-only, and Heterogeneous configurations

were identical, confirming that VKML correctly manages memory layouts and data types across different hardware vendors.

7.3 Limitations of Multi-Device Execution

While the architecture successfully functionalizes cross-vendor execution, two primary limitations remain for high-performance deployment.

First is the distinction between logical memory sharing and physical data transport. Vulkan's `VK_KHR_external_memory` extension allows devices to share resource ownership by exporting allocation handles (via `VkExportMemoryAllocateInfoKHR`) that can be imported by other devices (via `VkImportMemoryFdInfoKHR`). However, this handshake only establishes visibility; it does not guarantee a physical peer-to-peer (P2P) DMA path. In cross-vendor scenarios, drivers generally lack a shared peer memory protocol to negotiate direct PCIe transactions, forcing the implementation to fall back to a system-memory staging path to ensure data consistency. Consequently, VKML explicitly manages this transfer via the host CPU (Source Device $\rightarrow$ Host $\rightarrow$ Destination Device) to conform to these architectural constraints. This round-trip introduces latency, as seen in Chunk 1 (7.43 ms). For sequential models, this transfer cost is additive; theoretically, splitting this model across two identical GTX 1660 Ti cards would result in a total latency of 62.11 ms (54.04 ms compute + 8.07 ms transfer), making it ~15% slower than a single-card execution. Thus, without hardware-supported P2P, multi-device scaling is currently viable primarily for expanding memory capacity or for parallelizable graph branches.

Second, while Vulkan can leverage proprietary interconnects (e.g., NVLink) for identical GPUs via the `VK_KHR_device_group` extension, VKML currently treats all GPUs as distinct logical devices to maximize cross-vendor compatibility.

8 Discussion

For small graphs and op-heavy workloads, VKML outperforms the baseline. This is primarily attributed to two key factors in our low-overhead architecture.

First, the graph execution model is designed to be lock-free, avoiding the overhead of a global staging system. This is enabled by the underlying `zero-pool` thread pool library [8], which provides a low-latency foundation for the event-based scheduler. Furthermore, GPU execution utilizes reusable `vkFence` objects and `vkWaitForFences` to manage host-device synchronization efficiently.

Second, the runtime achieves a high degree of control through vertical integration. By developing purpose-built libraries for critical components, including file loading, model parsing, and the thread pool, VKML precisely orchestrates data flow across all execution stages. This implementation strategy demonstrates that a greenfield, vertically integrated execution stack architected specifically for Vulkan offers distinct benefits, including effective minimization of runtime overheads and a unified heterogeneous architecture.

Our primary baseline is ONNX-RT, which provides a vendor-optimized CUDA reference. The inclusion of `ncnn` as a secondary Vulkan baseline allows us to decompose VKML's observed advantages: the dispatch-efficiency gains are attributable to VKML's specific runtime architecture, while the consistent competitiveness of both Vulkan runtimes on compute-bound workloads suggests a structural benefit of the Vulkan execution model itself.

9 Current Limitations and Future Work

The performance gap in the large GEMM workload reflects the current implementation status of specialized kernels within VKML. While the runtime is verified to enable the `VK_NV_cooperative_matrix2` extension on supported hardware, it does not yet provide kernels

that utilize this extension. Consequently, these workloads rely on the portable tiled FP32 baseline, which lacks Tensor Core acceleration and the implicit TF32 downsampling typically used by CUDA-based libraries such as cuBLAS.

To address this and further mature VKML from a proof-of-concept into a usable library, we propose the following roadmap:

1. **Kernel Optimization & Mixed Precision:** Priority will be placed on optimizing matrix multiplication primitives (GEMM/GEMV). This includes expanding `VK_NV_cooperative_matrix2` support to accelerate standard FP32 ONNX models automatically, and integrating emerging data types such as FP8. Initial integration of the Slang [16] shader runtime compiler is showing positive results for shader performance and datatype-generic shader development.
2. **Advanced Memory Management:** To further reduce peak host memory usage for large models, we plan to implement a streaming loader similar to ONNX-RT, utilizing memory mapping to load layers on-demand. Additionally, we aim to implement true PCIe Peer-to-Peer transfers via `VK_KHR_external_memory` to increase cross-device buffer transfer performance.
3. **Platform & Hardware Expansion:** We intend to validate VKML on Apple Silicon via MoltenVK (following its recent Vulkan 1.4 support [17]) and add explicit support for `VK_KHR_device_group` to leverage proprietary interconnects like NVLink.
4. **Hybrid Execution & Interoperability:** Future work includes extending VKML to support multi-threaded CPU model operations for better hybrid inference performance and adding direct support for more model representations such as PyTorch `.pt` files.

10 Conclusion

VKML challenges the assumption that cross-vendor portability in machine learning must necessarily compromise performance. By building a runtime from first principles on Vulkan 1.4, we show that a Vulkan-native design can achieve competitive latency for ONNX inference while offering a unified execution model across heterogeneous devices.

Our results show that for latency-sensitive, small-to-medium workloads, VKML can outperform ONNX-RT (CUDA Execution Provider) on identical hardware, achieving a 1.66× reduction in median latency on an operation-heavy micro-model and consistent wins in a dispatch-dominated add-chain benchmark. We also demonstrate that Vulkan's explicit model enables capabilities such as cross-vendor multi-device execution within a single process (as a capability demonstration rather than a scaling result).

Throughput for large matrix multiplications remains an area for growth; however, the integration path for cooperative-matrix extensions, such as `VK_NV_cooperative_matrix2`, provides a clear route to future acceleration. Overall, VKML serves as a proof of existence that a unified, non-vendor-locked compute stack can be practical and performant for latency-focused inference.

Appendix

A Mixed Precision with Cooperative Matrix Extensions

To demonstrate that VKML is not limited to standard FP32 datatypes and portable kernels, we integrated support for cooperative matrix extensions. These extensions allow the runtime to target vendor-specific hardware accelerators (such as NVIDIA Tensor Cores) through a standardized Vulkan interface. We evaluated an FP16 variant of the convolutional network (`mnist_conv_fp16.onnx`) to test this capability. The benchmark consisted of 120 runs, with the first 20 discarded as warmup. Table 6 details the network architecture and highlights the specific matrix multiplication operation offloaded to the cooperative matrix accelerator.

Table 6: Layer breakdown of `mnist_conv_fp16.onnx`. The MatMul operation is accelerated via a Vulkan cooperative-matrix extension.

Operation	**Input Shape(s)**	**Output Shape**	**Notes**
Conv2D	[1, 1, 28, 28], [8, 1, 5, 5], [8]	[1, 8, 28, 28]	
ReLU	[1, 8, 28, 28]	[1, 8, 28, 28]	
MaxPool	[1, 8, 28, 28]	[1, 8, 14, 14]	
Conv2D	[1, 8, 14, 14], [16, 8, 5, 5], [16]	[1, 16, 14, 14]	
ReLU	[1, 16, 14, 14]	[1, 16, 14, 14]	
MaxPool	[1, 16, 14, 14]	[1, 16, 4, 4]	
Reshape	[1, 16, 4, 4]	[1, 256]	
MatMul	[1, 256], [256, 10]	[1, 10]	Cooperative matrix
Add	[1, 10], [10]	[1, 10]	
Softmax	[1, 10]	[1, 10]	

Table 7: FP16 convolutional model performance (N=100). Comparison of latency distribution between VKML (cooperative matrix) and ONNX-RT.

Runtime	**Median (μs)**	**p05..p95 (μs)**	**Std Dev (μs)**
VKML	99.48	94.73 .. 113.43	6.73
ONNX-RT	257.82	244.92 .. 276.41	9.07
Speedup	**2.59**×		

As detailed in Table 7, VKML achieved a median latency of 99.48 µs, representing a **2.59**× speedup over ONNX-RT (257.82 µs) on this benchmark. We verified numerical correctness by comparing outputs across VKML, ONNX-RT, and PyTorch. All three frameworks produced results consistent to three decimal places. This level of precision aligns with the theoretical limits of IEEE 754-2008 half-precision floating-point format (binary16) [18], where the 11-bit significand provides approximately 3.31 decimal digits of precision ($\log_{10}(2^{11}) \approx 3.31$).

Code Availability

The source code for the runtime and its core components is available under the MIT license at:

- **VKML:** `https://github.com/void-research/vkml`
- **zero-pool:** `https://github.com/void-research/zero-pool`
- **onnx-extractor:** `https://github.com/void-research/onnx-extractor`

References

[1] J. Bolz, "Nvidia vulkan update," Presented at Vulkanised 2025, 2025. [Online]. Available: https://www.vulkan.org/user/pages/09.events/vulkanised-2025/T47-Jeff-Bolz-NVIDIA.pdf

[2] D. Tolmachev, "Vkfft - a performant, cross-platform and open-source gpu fft library," *IEEE Access*, vol. 11, pp. 12 039–12 058, 2023. [Online]. Available: https://doi.org/10.1109/ACCESS.2023.3242240

[3] NVIDIA, "Vulkan driver support," 2024. [Online]. Available: https://developer.nvidia.com/vulkan-driver

[4] The Khronos Group, "Moltenvk." [Online]. Available: https://github.com/KhronosGroup/MoltenVK

[5] Imagination Technologies, "Imagination gpus now support vulkan 1.4," 2024. [Online]. Available: https://blog.imaginationtech.com/imagination-gpus-now-support-vulkan-1.4-and-android-16

[6] The Khronos Group, "Vulkan conformant products," 2025. [Online]. Available: https://www.khronos.org/conformance/adopters/conformant-products/vulkan

[7] Tencent, "ncnn: High-performance neural network inference framework optimized for the mobile platform," GitHub repository, 2025. [Online]. Available: https://github.com/tencent/NCNN

[8] L. Hofmockel-Spanakis, "zero-pool," GitHub repository. [Online]. Available: https://github.com/void-research/zero-pool

[9] L. Hofmockel-Spanakis, "onnx-extractor," GitHub repository. [Online]. Available: https://github.com/void-research/onnx-extractor

[10] Advanced Micro Devices, Inc., "Vulkan memory allocator (vma)," 2025. [Online]. Available: https://gpuopen.com/vulkan-memory-allocator/

[11] K. Mayes, "vulkanalia," GitHub repository. [Online]. Available: https://github.com/KyleMayes/vulkanalia

[12] nagisa, "rust_libloading," GitHub repository. [Online]. Available: https://github.com/nagisa/rust_libloading/

[13] PNNX Authors, "Pnnx: Pytorch neural network exchange," GitHub repository, 2025. [Online]. Available: https://github.com/pnnx/pnnx

[14] L. Roeder, "Netron: Visualizer for neural network, deep learning, and machine learning models," GitHub repository, 2025. [Online]. Available: https://github.com/lutzroeder/netron

[15] IEEE, "Ieee standard for binary floating-point arithmetic," ANSI/IEEE Std 754-1985, 1985. [Online]. Available: https://ieeexplore.ieee.org/document/30711

[16] Slang Contributors, "Slang." [Online]. Available: https://shader-slang.org/

[17] The Khronos Group, "Moltenvk v1.4.0 release," 2025. [Online]. Available: https://github.com/KhronosGroup/MoltenVK/releases/tag/v1.4.0

[18] IEEE, "Ieee standard for floating-point arithmetic," IEEE Std 754-2008, 2008. [Online]. Available: https://ieeexplore.ieee.org/document/4610935