Optimizing Vulkan Dispatch Schedules for Real-Time U-Net Denoising

KARL SASSIE, JOHANNES HANIKA, LUCAS ALBER, REINER DOLP, and CARSTEN DACHSBACHER, Karlsruhe Institute of Technology, Germany

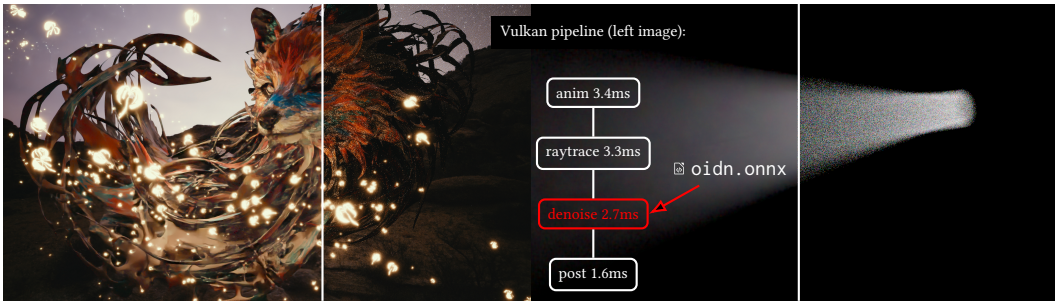


Fig. 1. Interactive applications with Open Image Denoise (OIDN) (left of the splits). Efficient in-renderer deployment of denoising is crucial for the viability of real-time path tracing. It also enables faster iterations in look development and lighting where the final appearance including expensive global illumination needs to be judged. We propose a method to generate efficient shader dispatches from an ONNX description (shown in red), using autotuning and profile guided optimization. We demonstrate our approach by generating Vulkan dispatches for OIDN. Timings in the graph are for the left image (7M animated triangles).

Image denoising is fundamental to Monte Carlo rendering. Recently, real-time path-traced applications have become viable, relying heavily on efficient denoising. Modern denoisers are often based on neural networks, most commonly variants of the U-Net architecture. While tooling for development and training of custom neural networks is well established in Python, existing deployment and interop strategies typically rely on external inference frameworks, incur host synchronization, data movement, or heavyweight dependencies, making them impractical for tightly integrated, real-time rendering pipelines. We present a light-weight and portable ONNX to Vulkan conversion framework that is designed for efficient, fully GPU-resident deployment of U-Net-based networks. We employ graph-based and profile-guided optimizations to determine optimal Vulkan dispatch schedules, including block sizes, data layouts, and kernel fusion decisions. Our method includes effective pruning of the vast search space, significantly reducing compile times. Using the Open Image Denoise (OIDN) network, we demonstrate runtime improvements over established inference frameworks such as TensorRT.

CCS Concepts: • **Computing methodologies** → *Graphics processors*; **Image processing**; *Ray tracing*.

ACM Reference Format:
Karl Sassie, Johannes Hanika, Lucas Alber, Reiner Dolp, and Carsten Dachsbacher. 2026. Optimizing Vulkan Dispatch Schedules for Real-Time U-Net Denoising. *Proc. ACM Comput. Graph. Interact. Tech.* 9, 4, Article 53 (July 2026), 20 pages. https://doi.org/10.1145/3820016

Authors' Contact Information: Karl Sassie, karl.sassie@student.kit.edu; Johannes Hanika, johannes.hanika@kit.edu; Lucas Alber, lucas.alber@kit.edu; Reiner Dolp, reiner.dolp@kit.edu; Carsten Dachsbacher, dachsbacher@kit.edu, Karlsruhe Institute of Technology, Karlsruhe, Germany.

ACM 2577-6193/2026/7-ART53
https://doi.org/10.1145/3820016

1 Introduction

Physically-based rendering primarily relies on Monte Carlo methods to solve the light transport problem. While general, it introduces stochastic noise that converges slowly and yields diminishing returns after a modest number of samples. To truncate the long-tail error distribution, a denoising stage as an image-space post process is employed, often guided by sidecar information such as per-pixel depth, albedo, and normals [Dammertz et al. 2010; Schied et al. 2017].

Today, the highest quality denoisers employ neural networks [Chaitanya et al. 2017; Marshall 2021; Thomas et al. 2022; Vogels et al. 2018] and, paired with upsampling to a higher resolution, are frequently found in modern games[1]. Implementing Monte Carlo path tracing efficiently in graphics engines has been made easier by introducing specialized hardware units for ray tracing in GPUs [Nvidia 2018]. Similarly, new hardware units for efficient neural network inference has been introduced, for example NVIDIA's Tensor Cores. Vulkan is a good choice to implement these techniques in graphics engines, as it is platform and OS independent, allows access to the graphics pipeline as well as to both specialized hardware units using the Vulkan ray tracing or cooperative matrix extension, respectively. Unfortunately, while access to specialized hardware is available, it still incurs substantial implementation effort to optimize network inference for speed [Bolz 2025]. This is mainly due to a wide range of possible implementation choices and parameter settings which grossly affect performance, but also heavily depend on the device and network architecture, for example, the supported matrix shapes or the number of channels per layer in the network. The implementation choices might be the data layout (`HWC`, `CHW`, `CHWC8`, ...), and type (`f32` vs. `f16`), and parameters might be workgroup sizes, block sizes of data to be prefetched into shared memory, the dimensions of the cooperative matrix operations, etc. This poses a challenge for deploying a high-performance implementation that runs efficiently across diverse network and device architectures. Existing deployment solutions [Chen et al. 2018; Google 2026; Microsoft 2026; NVIDIA 2026] often require heavy dependencies that can increase the size of a shipped product by several gigabytes. These solutions are rarely native to Vulkan and require interoperability between the APIs, which must be handled carefully to avoid additional overhead, such as synchronization costs. Also, they are designed for generality and broad coverage of potential operations, rather than for peak performance.

We present a lightweight approach for deploying neural network inference kernels directly within Vulkan. Our method is designed to minimize integration overhead while enabling rapid iteration when experimenting with different network architectures and configurations. In contrast to existing deployment frameworks, our approach allows developers to quickly obtain realistic end-to-end performance measurements within their target rendering environment, allowing practical optimization and informed design decisions. We demonstrate our approach on U-Net–based denoising networks and show that it achieves competitive and portable inference performance across GPUs supporting the cooperative matrix extension. The source code of the compiler is publically available at https://github.com/kistenklaus/denox.git.

In summary, our contributions are

- An ONNX-to-Vulkan compiler tailored to U-Net denoising, enabling lightweight and efficient deployment of inference pipelines without external runtime dependencies.
- Vulkan-native execution plans that integrate directly into the native resource and scheduling systems of existing renderers while delivering competitive real-time performance.
- A profile-guided optimization strategy that jointly explores operator fusion, tensor layouts, and kernel parameters to automatically identify high-performance configurations across diverse devices.

[1] https://gpuopen.com/amd-fsr-sdk/, https://developer.nvidia.com/rtx/dlss, https://game.intel.com/de/xess-gaming/

2 Related Work

2.1 Denoising Monte Carlo Renders

Classic Approaches. Denoising path traced images has a long history. Approaches include diffusion [McCool 1999], wavelets [Dammertz et al. 2010], regression [Moon et al. 2014], non-local means [Rousselle et al. 2012], and statistical tests [Sakai et al. 2024]. This broad family of methods is highly regarded for their understandable algorithms, and the fastest denoisers fall into this category.

Neural Methods. Among the neural approaches are the highest quality results, yet they are more resource-hungry and slower. These methods are often based on the U-Net architecture [Ronneberger et al. 2015]. Similar to decimated wavelets, this architecture reduces the resolution of the input image successively by *pooling* operations, and then increases it again by *upsampling* operations (see fig. 3). Much like detail coefficients in wavelet methods, there are *skip connections*, which are appended to the channels after upsampling by means of a *concatenation* operation. Unlike wavelets, a U-Net typically increases the number of channels before pooling.

Some specific artifacts such as gradient reversals and color shifts can appear in the output of direct-prediction U-Nets. Kernel-prediction [Bako et al. 2017; Vogels et al. 2018] addresses this by learning filter kernel weights instead of directly predicting an output image. This also enables more aggressive quantization of the inner network weights.

Transformer/attention operations are popular elements today, and can be implemented using a 1×1 convolution and element-wise multiplication [Chen et al. 2022]. Many neural denoisers target offline rendering, with a smaller subset targeting interactive [Chaitanya et al. 2017] or real-time usage. Often, light transport is separated into effects before denoising, e.g. into direct and indirect illumination [Schied et al. 2017], or diffuse, specular, and deep [Vicini et al. 2018]. This aggravates the problem of slow execution since it requires the denoiser to be run multiple times per frame. Recent work amortizes cost by joint denoising and upsampling [Kazmierczyk et al. 2025; Thomas et al. 2022].

2.2 GPU Compilers and Runtimes for Neural Networks

Modern neural network toolchains are in practice compilers: they accept a high-level graph description of a model and produce optimized executable code for a target backend and device. Some systems compile ahead of time (AOT), emitting a self-contained binary or library before deployment, while others operate just in time (JIT) ingesting the network graph as data at execution time. Regardless of when compilation occurs, the resulting optimization strategies can be broadly categorized along the *level of abstraction* at which transformations are expressed.

High-Level, Graph-Based Intermediate Representation. At one end of the spectrum, high-level operators are treated as atoms. From a formal perspective these systems use a graph-based intermediate representation (IR) [Braun et al. 2011; Click and Paleczny 1995; Leißa et al. 2015], where operators go through a pipeline of operator fusion, bufferization of tensors, linearization into command lists, etc. These have many parallels to the traditional compilation steps instruction selection, register allocation, instruction scheduling and so forth. These systems often do not support cyclic data dependencies and authors rely on ad-hoc solutions. Transformations performed on these IRs are often limited to *legalization* passes, i.e. transformations required for correct execution by the backend. In many cases this implies transforming operators from the data exchange format (e.g. ONNX), to operators of vendor libraries (e.g. cuBLAS). TensorRT [NVIDIA 2026] is a proprietary inference compiler and runtime with ONNX import. ONNX Runtime [Microsoft 2026] is a general-purpose inference runtime for ONNX models. Both perform graph-based optimizations, including operator

fusion and layout optimization. Compared to our approach, they emphasize broad applicability rather than a Vulkan-Native path specialized for tightly integrated real-time denoising.

Low- and Multi-Level IRs. At the other end, high-level operators are opened down to the machine instruction-level for more generalized optimization. These systems [Baghdadi et al. 2019; Vasilache et al. 2018] are often grounded in polyhedral analysis, where (affine) loopnests are transformed. Emerging systems rewrite at both high and low-level [Rotem et al. 2018], or at multiple levels as many problems are better modeled at different abstraction levels. Especially MLIR [Lattner et al. 2021], LLVM's multi-level intermediate representation provides this concept as an extensible infrastructure and has found wide-spread adoption in industry projects: Triton [Tillet et al. 2019] is a compiler for custom deep learning operators. OpenXLA [OpenXLA Project 2026] is able to compile many frontends to many backends using StableHLO as an IR optimized for interoperability and exchange. TVM [Chen et al. 2018] combines graph-level rewrites with lower-level code generation. And IREE [Google 2026] is an MLIR-based compiler offering both JIT and AOT compilation targeting CPU and GPU through multiple APIs, including the workflow of our proposed work: execution of ONNX with Vulkan. In many cases, these projects constitute a whole ecosystem of technologies, making them a heavyweight dependency.

Graph-Based Optimizations. Many problems in neural-network compilers are NP-hard graph optimization problems. For example, fusion and graph-substitution decisions induce a large space of equivalent computation graphs. The problem of optimizing computation graphs using graph substitutions has been formalized as OCGGS and shown to be NP-hard [Fang et al. 2020]. MetaFlow explores this space using cost-based backtracking search over graph substitutions [Jia et al. 2019b], while TASO extends this idea by automatically generating and verifying substitution rules [Jia et al. 2019a]. DNNFusion focuses specifically on fusion-plan generation, constructing fusion blocks greedily, using operator classifications and lightweight profiling [Niu et al. 2021].

Among the zoo of neural network compilers and runtimes, we focus on a lean and performance portable design, supporting a single highly portable compilation path: ONNX to Vulkan. We demonstrate that competitive performance is achievable through the use of compute shaders by focusing on aggressive operator fusion (to minimize memory traffic) and autotuning of operators.

3 System Overview

Figure 2 illustrates the intended workflow of our system: U-Net models are designed and trained using machine learning frameworks, such as PyTorch or TensorFlow. Once training is complete, the model is exported to the ONNX format, which serves as a stable representation of the network topology and parameters. The exported ONNX is then processed by our offline compiler. During this stage, the compiler parses the network structure, determines data layouts, identifies fusion opportunities, and selects concrete Vulkan native kernel implementations. This is resolved entirely offline, yielding a lightweight artifact without the need to embed a general neural network runtime. The result of this process is a fully materialized execution plan that captures the exact sequence of compute dispatches required for inference. This is then serialized into a *flatbuffer* that can be integrated into the renderer's asset pipeline together with generated code for the dispatches, or loaded dynamically at runtime and translated into compute dispatch commands.

The rest of this document is structured as follows: The remainder of section 3 details the contents of the flatbuffer artifact and gives an integration example with generated code. Section 4 recapitulates the operators in a U-Net to provide the context for the schedule optimization. Section 5 details how the schedule is found by first building a *supergraph* including all options and then selecting a minimum latency subgraph. Finally, section 6 presents results.

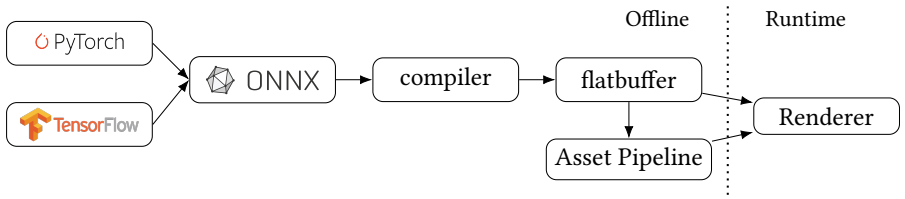


Fig. 2. Models are designed and trained using for instance PyTorch or TensorFlow. The finalized model is exported to ONNX and then processed by our offline compiler. The compiler creates a concrete Vulkan native dispatch schedule, including auto-tuned configuration parameters, fused operators, and selected data layouts. Finally, the execution plan is serialized into a flatbuffer which can be integrated into the renderer without dependencies on external libraries.

3.1 Structure of the Flatbuffer Artifacts

Dispatch Schedule. The flatbuffer representation encodes an ordered list of compute dispatches. Each entry contains the associated SPIR-V binary together with its descriptor bindings and resource access semantics. When executed sequentially, this dispatch sequence performs a complete model inference without requiring additional graph interpretation or runtime decision making. We deliberately do not encode explicit synchronization primitives or dependency graphs within the artifact. Instead, each resource binding specifies its access semantics (e.g., read, write, read-write), which allows the host application to derive the necessary barriers. Most modern rendering engines already employ custom render graph systems or barrier management strategies. Therefore, providing explicit synchronization would unnecessarily restrict integration flexibility.

Device Resources. Memory requirements are expressed in terms of buffers that store tensor data in contiguous regions of device memory. A buffer represents a contiguous allocation, while tensors define how this memory is interpreted during computation. In the common case, each buffer stores exactly one tensor. Consistent with our overall design philosophy, the memory interface is kept minimal. The compiler does not perform aggressive memory optimizations such as buffer aliasing or lifetime-based reuse. Most modern rendering engines already implement their own memory management and resource aliasing strategies, and duplicating such mechanisms within the artifact would only make integration more complicated.

Parameters. Model parameters are represented as *initializers* within the artifact. Each initializer references exactly one tensor and stores constant data as a binary blob. Since different kernel implementations may require weights and biases in different memory layouts and data types, the binary blob is stored exactly in the format expected by the selected implementation, such that uploading an initializer to device memory is a direct memory copy without further transformation. The artifact does not prescribe how parameters are managed on the device. Before a dispatch accesses a tensor backed by an initializer, the corresponding data must be available in device memory. Whether parameters are uploaded once and reused across frames or streamed per inference is left entirely to the renderer implementation.

Dynamic Shapes. Since our system targets real-time denoising, support for dynamic input resolutions is required. All size-dependent quantities, including buffer sizes, tensor offsets, dispatch dimensions, and push constant values may depend on the input resolution. At runtime, these values must be available before buffers are allocated, descriptors are updated, or dispatches are recorded. Since these values depend on the input resolution, we must be able to efficiently recompute them whenever the resolution changes. We therefore store, such dependent values as a linear sequence

of binary integer operations arranged in evaluation order. At runtime, given a concrete input resolution, this sequence is executed once from beginning to end. Each operation reads previously computed values and writes its result into a temporary array. Dynamic fields, such as dispatch sizes or push constants, reference indices in this array, so retrieving their values reduces to a simple index lookup.

3.2 Integration Example

As a proof of concept, we integrated the proposed workflow into two different systems: an existing real-time renderer based on a render graph architecture and a lightweight internal testing and benchmarking framework. In the first case, the renderer provides infrastructure for render graphs, resource management, and dispatch scheduling, where each node represents a single dispatch. During build time, a small code generation tool processes the model's flatbuffer artifact. The tool extracts and converts the stored parameters into an engine-specific asset format and generates C code that constructs a render subgraph corresponding to the model's dispatch schedule. Listing 1 shows snippets of the generated C code. At runtime, this generated subgraph is inserted into the renderer's render graph, allowing the model to execute using the engine's native resource and scheduling mechanisms. In the second case, such advanced infrastructure is not available. Instead, synchronization barriers and memory placement are derived in a single pass while loading the model, introducing only negligible host-side overhead. These two integration paths demonstrate that the artifacts can be embedded into a wide range of rendering architectures with minimal effort while achieving integration quality comparable to custom, engine-specific implementations.

```
void create_rendergraph(graph_t *graph, module_t *module, uint64_t W, uint64_t H) {
  const int64_t r2 = W + 127; // Evaluate dynamic values
  const int64_t r3 = r2 / 128;
  // ...
  const int64_t r122 = r121 * 6;
  // Adding nodes to render graph
  const uint32_t memory_pad_pc[6] = {(uint32_t)(r22), (uint32_t)(r8), 0, (uint32_t)(r21), 0, (uint32_t)(r7)};
  const int memory_pad_id = node_add(graph, module, "oidn", "dispatch0.spv",
      1, r120, r118, // dispatch size
      sizeof(memory_pad_pc), memory_pad_pc, // push constant
      "in", "read", "ssbo", "f16", &resource_desc0, // descriptors
      "out", "write", "ssbo", "f16", &resource_desc1);
  // ...
  // Adding connections to render graph.
  node_connect_named(graph, memory_pad_id, "out", conv0_id, "in");
  // ...
}
```

Listing 1. Generated C code, which constructs a render graph inside an engine.

4 U-Nets: It's all Convolutions?

U-Net denoisers follow a regular encoder decoder structure with skip connections (fig. 3). Besides convolutions, U-Net computational graphs commonly contain operations such as activations, pooling, upsampling, and channel-wise concatenation. Many of these operations are poor standalone dispatches: they perform little arithmetic work and instead mainly transform how data is read or written around a subsequent convolution. Many of these operations are therefore worth fusing into neighboring convolutions. The benefit is often larger than simply removing the dispatch overhead of the unfused operation. For some operators, fusion also reduces the amount of intermediate data that must be materialized and can improve the efficiency of the surrounding convolution. For example, when max pooling is fused into a convolution, the kernel can write the downsampled result directly instead of first materializing the full resolution output and then reducing it in a separate dispatch. These operators are therefore natural fusion targets. Once they are fused into

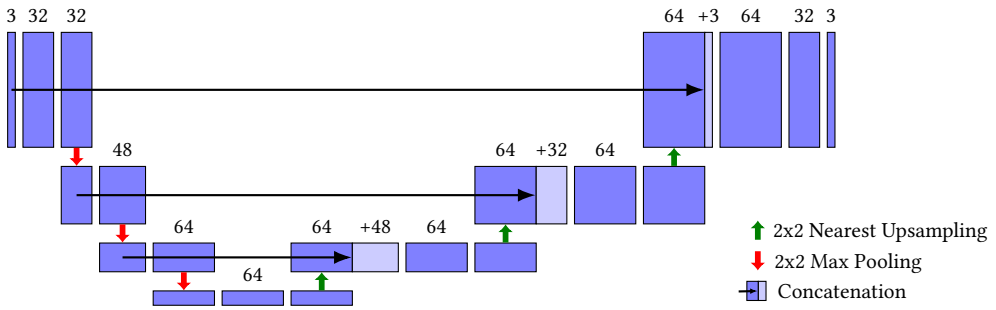


Fig. 3. Illustration of an example U-Net similar to the OIDN direct prediction denoising network. U-Nets are prominently made up of convolutions (not explicitly shown in the drawing, connecting the blocks from left to right on the same level). However, for fastest inferencing performance, optimizing the other operations (upsampling, pooling, and concatenation) is equally important. We address this by aggressive operator fusion.

neighboring convolution kernels, the remaining dispatches reduce to a small set of convolution based shaders.

Implicit GEMM. We implement convolutions as implicit GEMMs using cooperative matrix operations. A convolution can be lowered to a matrix multiplication over a lowered input feature map and a filter matrix. In efficient implementations, this lowered representation is generated on the fly rather than materialized explicitly. This retains the regular structure of GEMM while avoiding the memory cost of explicitly forming the full lowered input [Nvidia 2026; Zhou et al. 2021]. Because the resulting computation is a tiled matrix multiplication, it maps naturally to hardware cooperative matrix operations. Our implementation follows a software-pipelined tiled-GEMM structure, parameterized by macro definitions for tensor layouts, cooperative-matrix shapes, tile sizes, and workgroup dimensions. For each reduction tile, activation and filter values are prefetched from global memory into registers and then written to shared memory in the layout expected by cooperative-matrix loads. Cooperative-matrices are then loaded from shared memory and accumulated using matrix multiply-add operations. The reduction loop prefetches the next activation and filter tile while the current tile is multiplied, exposing independent memory and arithmetic work to the compiler and hardware scheduler. Finally the accumulator tile is then written back to global memory in the selected output layout. Although alternatives such as Winograd and FFT convolutions exist, we use implicit GEMM throughout. The shader does not implement spatial reuse explicitly, instead relying on the cache hierarchy to make repeated accesses cheap. We use f16 accumulation, as preliminary experiments showed lower latency with no visually perceivable impact on output quality. Section 5 describes how we choose the shader macro definitions to select performant configurations.

Down and Upsampling Fusion. For nearest-neighbor upsampling fusion is particularly simple. In the common 2×2 case, nearest-neighbor upsampling is fused into the following convolution by reading from $(\lfloor x/2 \rfloor, \lfloor y/2 \rfloor)$ in the original feature map. This only changes the input address calculation and requires minimal modification to the base convolution kernel. Max pooling can be fused into the writeback path of the preceding convolution. For 2×2 max pooling, this only requires performing the pooling in shared memory during the epilog before the final store to global memory. Compared to the complexity of the main implicit-GEMM loop, this additional swizzling logic is negligible. Both fusions avoid materializing intermediate tensors, which reduces memory traffic. This is especially beneficial for real-time applications, where layers are often partially memory bound and can become compute bound after fusion, significantly improving performance.

Concat Fusion. Concatenation followed by convolution can be fused into a single dispatch. The key observation is that a single convolution can be decomposed into two convolutions and an addition. Each convolution operates on a disjoint subset of the input channels with the corresponding subset of the filter weights.

$$\mathrm{conv}(\mathrm{concat}(x_1, x_2), w) = \mathrm{conv}(x_1, w_1) + \mathrm{conv}(x_2, w_2),$$

Given this observation, a concatenation followed by a convolution can be fused by evaluating the two convolutions back-to-back, where both use the same accumulator tile in registers. This makes the fusion particularly simple to implement, since it mainly amounts to duplicating the convolution loop within a single dispatch, with each convolution loading activations and filter weights from different input buffers. This does not increase register or shared memory usage significantly, as only the accumulator registers must remain live across both convolutions, while the other temporary resources can be reused. This fusion can improve performance beyond removing the intermediate tensor. For example, a convolution with 67 input channels cannot be vectorized cleanly in groups of 8, while after fusion it may instead be evaluated as two convolutions with 64 and 3 input channels, allowing the convolution with 64 input channels to use a more efficient vectorized path. This can also be combined with upsampling, such that upsampling, concatenation, and a convolution are performed together in a single dispatch.

Layout Transformations. Convolutions with different numbers of input and output channels may favor different tensor layouts, and the optimal choice can further depend on the tiling parameters used by the kernel. Layout transformations between convolutions are therefore often desirable. In the implicit-GEMM convolution, such transformations can be fused into the convolution itself. Activations are first loaded from global memory into shared memory and then into cooperative matrices, while writeback follows the reverse path from cooperative matrices through shared memory back to global memory. Once the data is stored in opaque cooperative-matrix registers, the computation no longer depends on the external tensor layout. This makes layout transformation particularly simple to implement, as it only requires selecting different read and write paths depending on the input and output layout. As a result, a convolution can read activations in one layout and write them back in another without requiring a separate layout-transformation dispatch.

5 Global Optimization

This section describes how a high level ONNX graph is transformed into a statically scheduled sequence of Vulkan compute dispatches. Our formulation follows established compiler principles: we model alternative layout choices and kernel implementations explicitly and select an optimal realization under a latency objective. While the individual techniques are not novel, their combination enables aggressive fusion and layout specialization tailored to real-time U-Net inference. A set of carefully chosen heuristics (cf. section 5.4) allows us to cut down the search space significantly, while still surpassing the performance of many frameworks and remaining comparable to hand-tuned implementations (see section 6).

Figure 4 provides an overview of how an ONNX graph is transformed across several compiler passes into an implementation search space in form of a *supergraph*, spanning all options of memory layouts, fused kernels, and configuration parameters. To select the final dispatch schedule, we associate each edge in this graph with a latency cost obtained via benchmarking. The optimization problem then reduces to selecting a minimum-latency subgraph that derives the network outputs from the network inputs (see section 5.5). We then linearize this subgraph into a topologically ordered sequence of compute dispatches, which forms the execution plan described in section 3.

5.1 Input: The ONNX Graph

Our compiler takes an ONNX model as input. ONNX stores a neural network as a graph of high-level logical operators together with associated constant parameters. This representation specifies the semantics of the network, but not its implementation. In particular, it does not encode explicit memory layouts of intermediate tensors, and operators remain abstract high-level operations such as convolutions, activations, pooling, upsampling, and concatenation. ONNX models may also contain shape computations, for example to determine padding or cropping from the input resolution. We evaluate these computations separately using a small unsigned symbolic-expression engine. After this separation, the remaining graph consists of tensor operators that describe the network layers executed on the GPU. We interpret those operators as a computational hypergraph (see fig. 4a), where logical tensors are represented as nodes, and logical operations are represented as edges connecting one or more input tensors to an output tensor. This formulation naturally captures operations such as concatenation, which depends on multiple inputs (see fig. 4a).

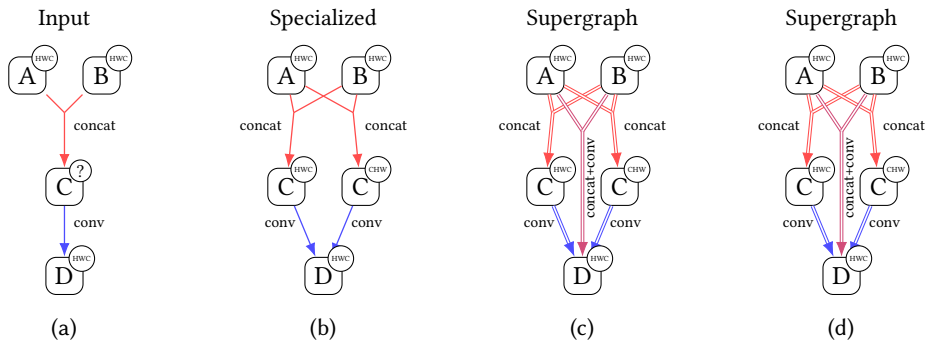


Fig. 4. Overview of the compiler pipeline illustrating an example graph with a `concat` operation, taking two tensors as input. A,B, and D are shown with fixed layouts, as they are the network inputs and outputs, whose layout is controlled via compilation flags.

5.2 Specialization: Memory Layouts

As discussed in section 4, different convolution implementations may favor different physical memory layouts. We therefore treat tensor layouts as an explicit optimization choice. In the initial representation, tensor nodes do not carry a fixed memory layout. The only exceptions are the network inputs and outputs, whose layout is controlled with compilation flags to allow for a well defined interface between our dispatch schedules and rendering engines (see A, B, and D in fig. 4a). During the specialization pass, each logical tensor is duplicated for each valid memory layout (HWC, CHW, etc., see fig. 4b and fig. 5b). After specialization, the hypergraph implicitly encodes all valid layout assignments. For example, in Figure 5b, the specialized graph for the `conv-relu` network duplicates each tensor in the *HWC* and *CHW* layouts. In total, there are 2^3 possible derivations of the logical value **C**, each corresponding to a different choice of tensor layouts. For larger graphs enumerating all such assignments explicitly quickly becomes unmanageable. The hypergraph representation avoids this by compactly representing multiple layout choices for the same computation within the topology of the graph itself. All subgraphs of the specialized graph connecting the input and output tensors correspond to realizations of the computation under specific layout choices.

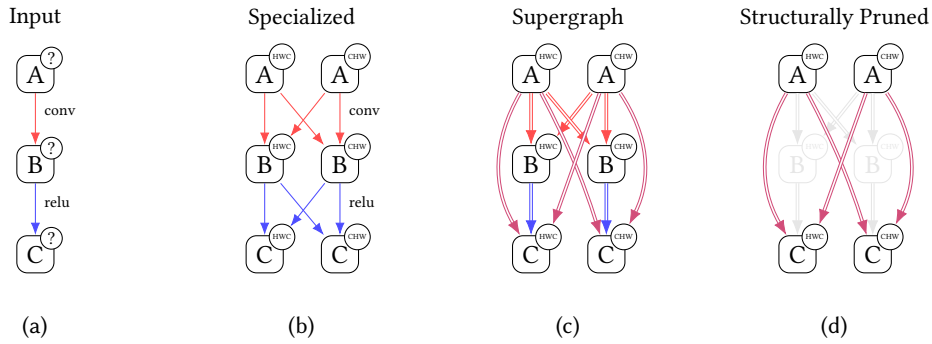


Fig. 5. Overview of the compiler pipeline from logical model representation to implementation search space.

5.3 The Supergraph: mapping Shaders

In the next pass, we map concrete shader implementations onto the specialized hypergraph. Each *shader* defines a structural pattern over tensor nodes and logical operations of the specialized hypergraph. A structural pattern is a graph pattern that describes a class of subgraphs for which the *shader* is applicable. In other words, it describes what subgraphs of the specialized hypergraph can be implemented by the shader. We search the specialized hypergraph for occurrences of these patterns, and for each match, we introduce corresponding implementation edges connecting the inputs of the matched subgraph to its output. For the simple `conv-relu` network, this is illustrated in Figure 5c. Notably, there is no implementation edge between $\mathbf{B}^{\mathrm{HWC}}$ and $\mathbf{C}^{\mathrm{CHW}}$, because no *shader* implements an activation function together with a layout transformation. Furthermore, both Figure 4c and Figure 5c contain implementation edges that bypass intermediate tensors, realizing a fusion of two operations into a single dispatch. Such edges arise from structural patterns that describe larger subgraphs; for example, a convolution followed by an activation. These patterns can be arbitrarily large, allowing us to model the capabilities of any shader within the same hypergraph abstraction as layout selection.

Shader Parameters. Most shaders expose additional parameters, such as tile sizes or workgroup sizes. A concrete assignment of these parameters is specified via macro definitions and called a *configuration*. We model different shader configurations in the same abstraction as layout choices and shader selection by introducing one implementation edge per valid configuration. As a result, a single matched subgraph can produce multiple implementation edges, where each edge represents a different configuration of the same shader. This is illustrated in Figure 5c and Figure 4c as multi edges.

After mapping all *shaders* onto the specialized hypergraph and introducing implementation edges per *configuration*, we obtain a new computational hypergraph that describes all possible realizations of the model's computation, including all possible shader configurations, layout choices, and fusion decisions. We refer to this graph as the *supergraph*, and it can be thought of as a compact representation of the global implementation search space. The key advantage of the *supergraph* representation is that it defers implementation decisions to later compilation stages rather than committing greedily to local choices such as layouts, fusion, and kernel implementations. This is particularly important because these choices are strongly coupled: the best layout may depend on whether neighboring operators are fused, and the best shader configuration depends on the chosen layouts. The disadvantage is that the resulting implementation search space can become very large.

5.4 Pruning: Search Space Reduction

A naive implementation would instantiate every valid configuration for every matched shader pattern, yielding millions of implementation edges and making exhaustive benchmarking impractical. We therefore reduce the search space in two stages. First, for each shader, we precompute a reduced configuration space from offline benchmarks spanning 8 GPUs from 3 different vendors. This is described in detail in the next paragraph. Second, after instantiating the reduced supergraph for a concrete model, we remove implementation edges that are structurally dominated by alternatives with fewer dispatches.

Configuration Space Reduction. As previously discussed, a *shader* denotes a kernel implementation that realizes structural patterns in the specialized graph and is parameterized by macro definitions. A concrete assignment of these macro definitions is a *configuration*. Let Σ denote the set of all configurations of a shader. An *operation* describes an abstract computation, such as a convolution or a fused computation. For a given shader and tuple $t = (\text{operation}, \text{device})$, only a subset of configurations $\Sigma_t \subseteq \Sigma$ are valid, e.g. due to layout compatibility, cooperative-matrix support, or shared-memory and register limits. Configuration-space reduction now amounts to choosing a subset $\Sigma' \subseteq \Sigma$ that contains configurations competitive across many relevant operations and devices, while keeping the time required to benchmark the remaining dispatches small. With a reduced configuration space Σ', we instantiate only configurations in $\Sigma_t \cap \Sigma'$ instead of creating one implementation edge for every configuration in Σ_t. This is motivated by the observation that a large fraction of valid configurations are extremely uncompetitive across many operations and devices, but would still have to be instantiated and benchmarked, leading to significantly slower compilation times.

To select Σ' we ran offline benchmarks. Let Φ denote the set of tuples (operation, device) included in these benchmarks. We choose Φ to cover common U-Net operations, layouts, channel counts, and fusions across multiple GPUs from different vendors. The reduction relies on the assumption that configurations that remain competitive across many benchmarked operations and devices are also likely to be good candidates for related operations and devices not included in Φ. For each $t \in \Phi$, we benchmark all valid configurations $c \in \Sigma_t$ and store the measured latency $L_{t,c}$. We define a loss function $\mathcal{L}_0$ as the average latency lost relative to the optimal measured configuration when restricting the search to the best k configurations $\Sigma_t^{(k)} \subseteq \Sigma' \cap \Sigma_t$.

$$\mathcal{L}_0(\Sigma') = \sum_{t \in \Phi} \sum_{c \in \Sigma_t^{(k)}} \frac{\omega_{d(t)}}{k} (L_{t,c} - L_t^*) \tag{1}$$

where $L_t^* = \min_{c \in \Sigma_t} L_{t,c}$ is the best measured latency for t, and $w_{d(t)}$ is a device-specific weight, which is chosen proportional to the performance of the device, such that slower devices do not dominate the objective. Using the best k configurations instead of only the best, makes the reduced set more robust to measurement noise and architectural differences, as each (operation,device) tuple requires at least k competitive configurations instead of only a single one.

We further define a second loss function $\mathcal{L}_1$ which models the cost of executing and measuring the latencies of the remaining configurations.

$$\mathcal{L}_1(\Sigma') = \sum_{t \in \Phi} \sum_{c \in \Sigma_t \cap \Sigma'} \omega_{d(t)} L_{t,c} \tag{2}$$

Because $\mathcal{L}_1$ is based on measured latencies instead of only the size of Σ', it rewards smaller configuration spaces stronger if t is a slow operation, such as a convolution with high channel counts. This

is exactly what we want, as the majority of the compilation time is spent measuring implementation edges, which implement those slow operations.

We combine those two loss functions, and choose Σ' by solving a minimization problem.

$$\underset{\Sigma' \subseteq \Sigma}{\operatorname{argmin}} \left\{ \alpha \mathcal{L}_0(\Sigma') + \beta \mathcal{L}_1(\Sigma') \right\} \tag{3}$$

where α and β are tuning parameters, which determine whether more emphasis is put on run time latency or the compile time. We solved this minimization problem once offline with a Mixed Integer Linear Programming (MILP) library. In the appendix A we describe how this minimization problem can be transformed into a MILP program. For shaders with especially large configuration spaces $\Sigma_t \cap \Sigma'$, we further reduce it using policies. In particular, we discard variants that tile across output channels when non-tiled alternatives exists, variants whose tile sizes are larger than lowered matrix multiplication itself, and variants that substantially under- or overallocate shared resources such as registers or shared memory.

Structural Pruning. After the configuration-space reduction, the instantiated supergraph is already significantly smaller, but it still contains many implementation edges that are unlikely to be competitive. In particular, unfused implementations remain even though fused alternatives produce the same values with fewer dispatches. We therefore prune the supergraph topologically and retain only implementation edges that belong to at least one minimum-dispatch subgraph. (–) This is a heuristic rather than an exact latency criterion and may discard an unfused implementation that would perform better in isolation. For our target U-Net workloads, however, it is highly effective, since fusion is usually beneficial in practice due to lower memory traffic. As a secondary benefit, this pruning also reduces the impact of measurement noise. The additional dispatches introduced by unfused implementations are often small, low-arithmetic kernels whose measurement latency depends strongly on whether their data is still in cache. Since these dispatches are short, even modest absolute latency variance constitutes a large fraction of their runtime and can make the overhead of the unfused schedule appear smaller than it is in end-to-end execution.

5.5 Minimum Latency Subgraph

After configuration-space reduction and structural pruning, the remaining implementation *supergraph* is small enough to measure the latencies of the remaining implementation edges. We then use the measured latencies as edge costs and select the dispatch schedule by searching for the minimum-latency subgraph that derives the output tensors from the input tensors. We implement this search by exploring candidate subgraphs, using branch-and-bound to prune partial solutions whose cost already exceeds the best schedule found so far, and memoization to avoid recomputing identical subproblems. We additionally use a heuristic ordering of the search to visit promising candidates first, which improves pruning efficiency in practice. Although the underlying problem is hard in general, the regular structure of U-Nets considered here allows for good heuristic search ordering, which, together with memoization, keeps the effective amount of searched subgraphs small. The selected subgraph is then linearized into a sequence of compute dispatches and serialized into the execution plan described in Section 3.

Overall, our compiler initially constructs a large global implementation search space, then reduces the considered configurations based on offline benchmarks, removes unfused implementations, and finally selects a minimum-latency schedule. Section 6 shows that this yields both practical compile times and high-quality dispatch schedules, resulting in competitive inference performance.

6 Evaluation

For the evaluation, we compile the OIDN U-Nets [Áfra 2026] in the *fast* and *balanced* quality settings and benchmark it on 9 GPUs spanning 3 vendors and 6 unique architectures. We compare against five baselines covering vendor-specific inference compilers, portable runtimes, a handwritten implementation, and general-purpose ML compiler stacks. For each framework, we ensure that the reported latencies reflect GPU execution time by cross-checking the measurements with NVIDIA Nsight. We do not lock GPU clocks to base frequencies, as this would introduce an unrealistic bias. All measurements are reported over at least 10 timed iterations, preceded by 10 warm-up iterations.

6.1 Kernel Performance

As a case study on kernel performance we take a deeper look at a convolution with 67 input and 64 output channels, which appears in the OIDN models and contributes around 24% of the inference latency. Table 1 shows average occupancy, SM throughput, peak register counts, and VRAM throughput measured on a RTX 4070. Interestingly on the RTX 4070, this operation lies just between memory and compute limits, with a arithmetic intensity of 295 FLOPS/Byte and a machine balance of 232 FLOPS/Byte.

	SM occ.	SM tput.	Regs.	VRAM tput.
TensorRT	16.5%	69.4%	194	31.6%
Ours	82.6%	49.6%	45	27.7%
Ours + Fusion	33.0%	74.5%	127	28.5%

Table 1. Profiling metrics for a convolution with 67 input and 64 output channels, collected with NVIDIA NSight on a RTX 4070. The last row shows fusion of 2x2 upsampling, concatenation and the convolution.

The implicit GEMM convolution implemented by TensorRT, outperforms our implementation, reaching higher SM and VRAM throughputs, with significantly lower occupancy. We theorize, that this is most likely due to optimized address generation, horizontal and vertical reuse, improved memory coalescing, as well as shared memory access patterns. However within the OIDN networks, this convolution appears after a upsampling and concatenation layer. Fusing those into the convolution increases the arithmetic intensity to 478 FLOPS/Byte, as only the downscaled input has to be loaded from memory. Further fusion of the concatenation layer as discussed in section 4 can allow for improved memory accesses through vectorization. Table 1 shows that although our convolution in isolation performs worse, fusion can increase arithmetic intensity, allowing us to hide memory access latencies with lower occupancies, which allows for larger tile sizes, which in turn improves SM throughput. In practice, fusing upsampling and concatenation reduces the latency to approximately 0.6 times the unfused baseline.

Although profiling metrics are only shown for NVIDIA, fusion has a similar impact on latency on AMD and Intel devices. However, our convolution implementation itself is less competitive on these platforms. For example on Intel GPUs, the native OIDN SYCL backend is often more than twice as fast. At this point we cannot isolate a single cause for this slowdown. A plausible explanation is that our current implicit-GEMM kernel is missing explicit spatial reuse, which makes performance more dependent on vendor-specific cache and local-memory behaviour. We therefore attribute the AMD and Intel gap primarily as a kernel-level implementation issue rather than a limitation of the fusion and autotuning strategies.

6.2 Dispatch Schedule

Further we compare dispatch schedules produced by our compiler against those generated by existing inference frameworks. Figure 6 shows that progressively enabling the fusion opportunities from section 4 leads to shorter schedules and lower latency. In particular, aggressive fusion eliminates several low-arithmetic, memory-dominated dispatches such as standalone pooling, upsampling and copy operations. Among the evaluated frameworks, we found no general-purpose compiler that automatically fuses pooling or upsampling into neighboring convolutions, nor one that absorbs channel-wise concatenation into the subsequent convolution without first materializing an intermediate tensor. Intel Open Image Denoise is the only notable exception: its hand-written SYCL backend [Áfra 2026] produces a schedule that is almost identical to ours. ONNX Runtime, IREE, and TVM often introduce even more dispatches by materializing padding or inserting explicit layout-transformation steps, which further increase memory traffic and overall latency. A more detailed example of a complete schedule can be found in appendix B.

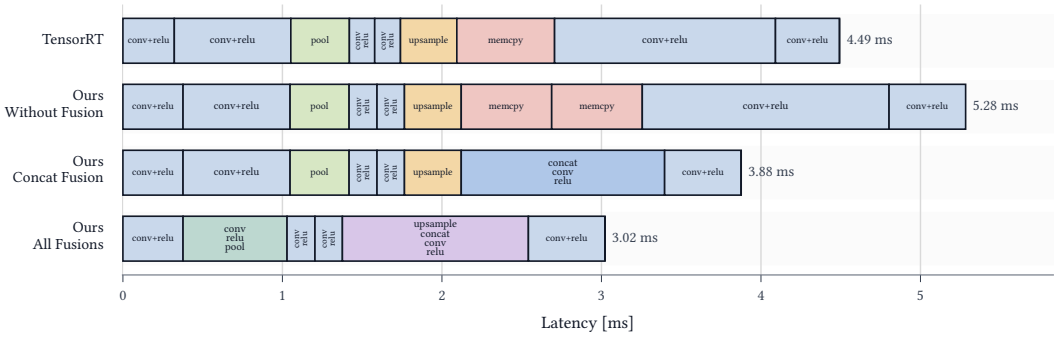


Fig. 6. Dispatch schedules for TensorRT and three variants of our compiler with progressively stronger fusion, shown for a simplified U-Net with a single pooling and upsampling stage. All convolutions are implemented as implicit GEMM convolutions, including those generated by TensorRT. The network is deliberately minimal and serves only to illustrate the impact of fusion on the generated dispatch schedule.

6.3 Inference Latency

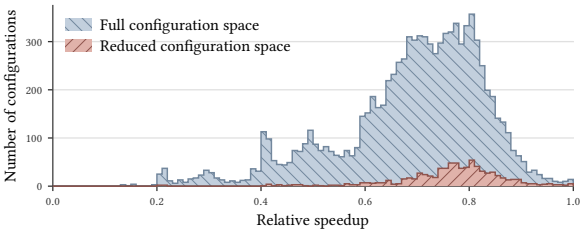
Table 2 shows that our compiler achieves the strongest results on NVIDIA GPUs, where it consistently outperforms both TensorRT and the handwritten OIDN backend across all reported resolutions and both quality settings. The advantage generally grows with resolution, which is consistent with the reduced memory traffic and shorter dispatch schedules enabled by our more aggressive fusion. By contrast, results on AMD and Intel are less favorable: on the Radeon RX 7900 XTX our implementation remains roughly competitive with OIDN and ONNX Runtime, but on the Radeon RX 6800 XT and Arc B580 it falls behind substantially, especially at higher resolutions. We attribute this gap primarily to the fact that our current convolution kernels are optimized for NVIDIA architectures and are not yet as well tuned for AMD or Intel GPUs. In addition, the Radeon RX 6800 XT does not support the Vulkan cooperative matrix extension. On this device, our compiler therefore falls back to simpler, less fused dispatch schedules, which further increases latency. The results, therefore, suggest that our global schedule optimization is effective across devices, but that end-to-end performance still depends critically on the quality of the underlying convolution implementations.

Table 2. End-to-end inference latencies for the OIDN U-Net in the *fast* and *balanced* quality settings across all evaluated GPUs and input resolutions. We compare our compiler against NVIDIA TensorRT, ONNX Runtime, TVM, IREE, and the native OIDN GPU backend. ONNX Runtime is evaluated using its CUDA, ROCm, and OpenVINO execution providers depending on the target device; TVM uses the CUDA code generation path; IREE uses its Vulkan/SPIR-V backend; and OIDN uses its native GPU backend. TensorRT is only available on NVIDIA GPUs and is therefore omitted on AMD and Intel hardware. Since IREE, TVM, and the ONNX Runtime failed to compile with dynamic shapes, we only evaluate it at a fixed resolution, without dynamic input padding and output cropping layers, which would be required for proper alignment of the input tensors. The ONNX runtime failed to compile the 4K models on some devices, we mark those entries with "error". TVM also supports AMD and Intel, we ommit those timings due to time pressure. Relative slowdown is reported with respect to the best result for the same model, resolution, and target GPU. Lower is better.

	RTX 2070	**RTX** 2080 Ti	**RTX** 3080 Ti	**RTX** 4060 Ti	**RTX** 4070	**RTX** 4080 Super	**RX** 7900 XTX	**RX** 6800 XT	**Arc** B580
OIDN balanced @1280x720									
TensorRT	10.79(135%)	6.76 (142%)	4.97 (114%)	8.43 (181%)	5.57 (157%)	3.53 (161%)	unsupported	unsupported	unsupported
ONNX runtime	26.02(325%)	16.02(337%)	11.73(268%)	25.02(538%)	15.06(424%)	8.91 (407%)	5.00 (109%)	**11.419**	26.81(735%)
OIDN	12.33(154%)	7.29 (153%)	4.93 (113%)	7.53 (162%)	5.18 (146%)	3.28 (150%)	**4.58**	22.59(198%)	**3.65**
TVM	28.09(351%)	14.61(307%)	10.21(234%)	13.70(295%)	9.09 (256%)	5.65 (258%)	-	-	-
IREE	406 (5075%)	231 (4853%)	201 (4600%)	260 (5591%)	188 (5296%)	120 (5480%)	43.9 (959%)	52.9 (463%)	991(27151%)
Ours	**8.00**	**4.76**	**4.37**	**4.65**	**3.55**	**2.19**	4.92 (107%)	39.07(342%)	8.61 (236%)
OIDN balanced @1920x1080									
TensorRT	24.72(141%)	15.14(144%)	11.29(161%)	19.45(183%)	12.74(163%)	8.30 (171%)	unsupported	unsupported	unsupported
ONNX runtime	57.85(330%)	37.11(354%)	26.04(371%)	59.55(560%)	35.52(453%)	21.47(443%)	11.39(107%)	**19.75**	59.19(738%)
OIDN	27.61(157%)	16.10(154%)	10.00(143%)	17.47(164%)	11.79(150%)	7.57 (156%)	**10.60**	50.98(258%)	**8.02**
TVM	63.49(368%)	37.11(334%)	23.34(333%)	31.58(297%)	22.52(287%)	13.04(269%)	-	-	-
IREE	920 (5245%)	524 (5000%)	460 (6562%)	577 (5428%)	426 (5434%)	281 (5794%)	107 (1009%)	144 (729%)	2062 (25711%)
Ours	**17.54**	**10.48**	**7.01**	**10.63**	**7.84**	**4.85**	10.96(103%)	87.55(443%)	18.58(232%)
OIDN balanced @3840x2160									
TensorRT	94.86(138%)	58.20(141%)	46.33(167%)	76.89(186%)	50.37(162%)	32.65(172%)	unsupported	unsupported	unsupported
ONNX runtime	error	error	103.1(373%)	242.0(584%)	143.6(315%)	97.92(517%)	46.99(109%)	**76.71**	234 (736%)
OIDN	114 (165%)	65.86(159%)	44.83(162%)	73.34(177%)	49.40(159%)	32.04(169%)	44.72(103%)	215.0(280%)	**31.79**
TVM	251.6(365%)	144.3(348%)	82.72(299%)	132.7(320%)	89.88(289%)	48.34(255%)	-	-	-
IREE	5620(8150%)	3109(7506%)	2088(7549%)	2645(6383%)	1966(6315%)	1265(6683%)	556 (1284%)	772 (1006%)	4932 (15514%)
Ours	**68.96**	**41.42**	**27.66**	**41.44**	**31.13**	**18.93**	**43.30**	336.9(439%)	72.07(227%)
OIDN fast @1280x720									
TensorRT	6.41 (133%)	4.02 (136%)	2.61 (135%)	5.12 (176%)	3.11 (151%)	2.00 (160%)	unsupported	unsupported	unsupported
ONNX runtime	-	-	7.50 (389%)	14.79(508%)	9.77 (474%)	5.29 (423%)	2.98 (120%)	**4.18**	16.37(668%)
OIDN	7.97 (166%)	4.72 (159%)	3.18 (165%)	5.24 (180%)	3.48 (169%)	2.15 (172%)	2.70 (108%)	16.98(406%)	**2.45**
TVM	17.80(370%)	9.56 (323%)	5.66 (293%)	9.37 (322%)	6.48 (315%)	3.32 (266%)	-	-	-
IREE	201 (4179%)	116 (3919%)	99.3(5145%)	130 (4467%)	92.2(4476%)	59.0(4720%)	19.0 (763%)	25.3 (605%)	466(19020%)
Ours	**4.81**	**2.96**	**1.93**	**2.91**	**2.06**	**1.25**	**2.49**	19.38(464%)	4.81 (196%)
OIDN fast @1920x1080									
TensorRT	14.29(133%)	8.82 (135%)	6.29 (134%)	11.84(179%)	7.19 (156%)	4.88 (176%)	unsupported	unsupported	unsupported
ONNX runtime	38.49(358%)	24.07(370%)	17.23(366%)	35.55(537%)	23.20(502%)	14.06(508%)	6.81 (117%)	**13.07**	34.61(643%)
OIDN	17.83(166%)	10.36(159%)	6.98 (148%)	11.89(180%)	7.95 (172%)	5.07 (183%)	6.03 (104%)	38.24(293%)	**5.38**
TVM	39.85(370%)	20.91(321%)	13.24(281%)	20.20(305%)	13.25(287%)	7.90 (285%)	-	-	-
IREE	447 (4154%)	261 (4009%)	255 (5414%)	290 (4381%)	210 (4545%)	138 (4982%)	46.2 (795%)	64.7 (495%)	1088 (20223%)
Ours	**10.76**	**6.51**	**4.71**	**6.62**	**4.62**	**2.77**	**5.81**	44.36(339%)	10.54(196%)
OIDN fast @3840x2160									
TensorRT	57.32(135%)	34.46(136%)	27.21(171%)	46.90(178%)	27.82(150%)	19.36(163%)	unsupported	unsupported	unsupported
ONNX runtime	156.3(368%)	93.30(368%)	67.89(428%)	147.5(559%)	95.81(517%)	error	27.64(114%)	**51.56**	133.7(611%)
OIDN	74.38(175%)	42.67(168%)	28.98(182%)	49.93(189%)	33.07(178%)	21.26(179%)	26.68(110%)	160 (310%)	**21.89**
TVM	168.0(395%)	87.19(344%)	49.61(312%)	78.32(297%)	51.19(276%)	33.44(282%)	-	-	-
IREE	2623(6167%)	1462(5774%)	972 (6121%)	1222(4627%)	908 (4900%)	582 (4907%)	239 (987%)	377 (731%)	4249 (19411%)
Ours	**42.53**	**25.32**	**15.88**	**26.41**	**18.53**	**11.86**	**24.21**	173.2(336%)	40.68(186%)

6.4 Compile Times

In principle, selecting optimal kernel parameters, such as tile sizes, amounts to benchmarking all valid configurations and choosing the best. In practice, this is infeasible, so we reduce the search space as described in section 5, making benchmarking practical without significantly increasing final inference latency. Figure 7a compares the full and reduced search spaces on an RTX 4070. To evaluate whether the approach generalizes to unseen architectures, we solved the MILP without including any measurements from the Ada Lovelace architecture. Although the reduced space still contains suboptimal configurations, it preserves competitive ones with relative speedups close to 1 while removing a large number of uncompetitive variants. This substantially reduces the time required to benchmark the remaining dispatches. Table 7b shows that search-space reduction and structural pruning drastically reduce compilation time, while only marginally affecting inference latency. This tradeoff is particularly valuable when developers must iterate quickly over U-Net architectures and assess the performance impact of architectural changes without incurring excessive compilation cost.



(a) Search-space distribution.

Search Space	Size	Compile-time	Latency
All valid configurations	413650	4 h 13 min	7.79 ms
Reduced (with Ada)	23034	7 min 31 s	7.84 ms
Reduced (without Ada)	21763	7 min 7 s	8.07 ms
Structurally pruned (with Ada)	6693	2 min 2 s	7.82 ms
Structurally pruned (without Ada)	5802	1 min 51 s	8.05 ms

(b) Search-space summary.

Fig. 7. (a) Full vs reduced configuration space; the reduced space was computed without the Ada Lovelace measurements. (b) Impact of search-space reduction on compile time and inference latency, with and without measurements of the Ada Lovelace architecture.

7 Conclusion and Future Work

We present an offline compiler for Vulkan-native deployment of U-Net denoisers in real-time rendering. Although we show integration within Vulkan renderers, we believe our approach to also be applicable to other graphics APIs. By optimizing operator fusion, tensor layouts, and kernel parameters, our method produces compact execution plans that integrate directly into the native resource and scheduling systems of existing renderers, without relying on external dependencies. For the OIDN denoising networks, our approach yields competitive inference performance, together with a small deployment footprint and practical compile times enabled by search-space reduction and profile-guided optimizations. Our results demonstrate that near-peak performance and optimal schedule generation are feasible for simple U-Net architectures, allowing exploratory workflows in which network architectures can be changed, recompiled, and evaluated directly in the target renderer. Our system is specialized and, at this point, only supports simple U-Net architectures; however, we expect our compilation process to also be applicable to more complex denoising architectures. In particular, it does not yet support kernel-prediction or transformer based denoisers due to the lack of a general GEMM implementation and specialized 1×1 convolution shaders. In the future, we aim to improve the performance portability of our convolution implementation and extend our compiler towards more complex denoising architectures.

Acknowledgments

8 Acknowledgements

The teaser image in fig. 1 uses the fox model by By Daniel Bystedt – CC-BY-SA 3.0.

References

Attila T. Áfra. 2026. Intel® Open Image Denoise. https://www.openimagedenoise.org.

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony Derose, and Fabrice Rousselle. 2017. Kernel-predicting convolutional networks for denoising Monte Carlo renderings. *ACM Trans. Graph.* 36, 4 (2017), 97–1.

Jeff Bolz. 2025. Machine Learning in Vulkan with Cooperative Matrix 2. In *Vulkanized*.

Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. 2011. *Firm-a graph-based intermediate representation*. KIT, Fakultät für Informatik.

Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–12.

Liangyu Chen, Xiaojie Chu, Xiangyu Zhang, and Jian Sun. 2022. Simple baselines for image restoration. In *European conference on computer vision*. Springer, 17–33.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 578–594.

Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. *ACM Sigplan Notices* 30, 3 (1995), 35–49.

Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik P. A. Lensch. 2010. Edge-Avoiding À-Trous Wavelet Transform for Fast Global Illumination Filtering. In *Proceedings of the Conference on High Performance Graphics* (Saarbrucken, Germany) *(HPG '10)*. Eurographics Association, Goslar, DEU, 67–75.

Jin Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2020. Optimizing DNN computation graph using graph substitutions. *Proceedings of the VLDB Endowment* 13 (2020), 2734 – 2746. https://api.semanticscholar.org/CorpusID:221539744

Google. 2026. *IREE*. https://iree.dev/ Accessed: 2026-03-13.

Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019a. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. New York, NY, USA, 47–62.

Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019b. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 27–39.

Pawel Kazmierczyk, Sungye Kim, Wojciech Uss, Wojciech Kalinski, Tomasz Galaj, Mateusz Maciejewski, and Rama Harihara. 2025. Joint denoising and upscaling via multi-branch and multi-scale feature network. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 8, 1 (2025), 1–18.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A graph-based higher-order intermediate representation. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 202–212.

Carl S. Marshall. 2021. Practical machine learning for rendering: from research to deployment. In *ACM SIGGRAPH 2021 Courses* (Virtual Event, USA) *(SIGGRAPH '21)*. Association for Computing Machinery, New York, NY, USA, Article 10, 239 pages. doi:10.1145/3450508.3464564

Michael D McCool. 1999. Anisotropic diffusion for Monte Carlo noise reduction. *ACM Transactions on Graphics (TOG)* 18, 2 (1999), 171–194.

Microsoft. 2026. *ONNX Runtime*. https://onnxruntime.ai/ Accessed: 2026-03-29.

Bochang Moon, Nathan Carr, and Sung-Eui Yoon. 2014. Adaptive rendering based on weighted local regression. *ACM Transactions on Graphics (TOG)* 33, 5 (2014), 1–14.

Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. New York, NY, USA, 883–898.

Nvidia. 2018. NVIDIA Turing GPU architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

Nvidia. 2026. CUTLASS: Implicit GEMM Algorithm. https://docs.nvidia.com/cutlass/latest/media/docs/cpp/implicit_gemm_convolution.html#cutlass-convolution

NVIDIA. 2026. *TensorRT*. developer.nvidia.com/tensorrt Accessed: 2026-03-13.

OpenXLA Project. 2026. *OpenXLA*. https://openxla.org/ Accessed: 2026-03-29.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*. Springer, 234–241.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. 2012. Adaptive rendering with non-local means filtering. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 1–11.

Hiroyuki Sakai, Christian Freude, Thomas Auzinger, David Hahn, and Michael Wimmer. 2024. A statistical approach to monte carlo denoising. In *SIGGRAPH Asia 2024 Conference Papers*. 1–11.

Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of high performance graphics*. 1–12.

Manu Mathew Thomas, Gabor Liktor, Christoph Peters, Sungye Kim, Karthik Vaidyanathan, and Angus G Forbes. 2022. Temporally stable real-time joint neural denoising and supersampling. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5, 3 (2022), 1–22.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018), 5–14.

Delio Vicini, David Adler, Jan Novák, Fabrice Rousselle, and Brent Burley. 2018. Denoising Deep Monte Carlo Renderings. *Computer Graphics Forum* 38, 1 (Aug. 2018), 316–327. doi:10.1111/cgf.13533

Thijs Vogels, Fabrice Rousselle, Brian McWilliams, Gerhard Röthlin, Alex Harvill, David Adler, Mark Meyer, and Jan Novák. 2018. Denoising with kernel prediction and asymmetric loss functions. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–15.

Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 2–3.

A MILP Program

Given the following minimization problem described in Section 5.4:

$$\operatorname*{argmin}_{\Sigma' \subseteq \Sigma} \left\{ \alpha \mathcal{L}_0(\Sigma') + \beta \mathcal{L}_1(\Sigma') \right\} \tag{4}$$

$$= \operatorname*{argmin}_{\Sigma' \subseteq \Sigma} \left\{ \alpha \sum_{t \in \Phi} \sum_{c \in \Sigma_t^{(k)}} \frac{\omega_{d(t)}}{k} (L_{t,c} - L_t^*) + \beta \sum_{t \in \Phi} \sum_{c \in \Sigma_t \cap \Sigma'} \omega_{d(t)} L_{t,c} \right\} \tag{5}$$

We introduce binary variables $x_c \in \{0,1\}$ and $z_{t,c} \in \{0,1\}$. The variable x_c indicates whether configuration c is retained in Σ' (i.e. $x_c = 1 \Leftrightarrow c \in \Sigma'$). The variable $z_{t,c}$ indicates whether configuration c is under the best k configurations for t (i.e. $z_{t,c} = 1 \Leftrightarrow c \in \Sigma_t^{(k)}$). Because $\Sigma_t^{(k)} \subseteq \Sigma' \cap \Sigma_t \subseteq \Sigma'$, it follows that $\forall t\colon z_{t,c} \leq x_c$. Further $|\Sigma_t^{(k)}| = k$ implies that $\forall t\colon \sum_c z_{t,c} = k$. With this, any sum over $\Sigma' \cap \Sigma_t$ can be expressed as a sum over Σ_t, where each term is multiplied by x_c and a similarly a sum over $\Sigma_t^{(k)}$ can be expressed by multiplying each term with $z_{t,c}$. With this substitution, the subset-selection problem becomes:

$$\operatorname*{argmin}_{x_c \in \{0,1\}} \left\{ \operatorname*{argmin}_{z_{t,c} \in \{0,1\}} \left\{ \alpha \sum_{t \in \Phi} \sum_{c \in \Sigma_t} \left(z_{t,c} \cdot \frac{\omega_{d(t)}}{k} (L_{t,c} - L_t^*) \right) + \beta \sum_{t \in \Phi} \sum_{c \in \Sigma_t} \left(x_c \cdot \omega_{d(t)} L_{t,c} \right) \right\} \right\} \tag{6}$$

This is a mixed integer linear program: all decision variables are binary, the objective is linear in these variables, and all constrains are linear. The measured latencies $L_{t,c}$, L_t^*, device weights $\omega_{d(t)}$, parameters α and β, and k are constants. For fixed x_c, the first objective term makes the optimal $z_{t,c}$ the k fastest configurations for each t. After solving, the minimization problem Σ' is constructed as $\Sigma' = \{c \in \Sigma \mid x_c = 1\}$.

B Detailed Example

For interested readers, we consider a complete example. Figure 8 shows the dispatch schedule for the OIDN model at balanced quality setting, showing, which fusions are implemented. Notable after fusion almost all dispatches are primarily convolutions. Figure 9 shows the corresponding profiling metrics recorded with NVIDIA Nsight Graphics. In U-Net architectures arithmetic intensity generally grows towards the bottleneck and is highest in the decoding layers (i.e., `upsample+concat+iconv-cm`). The profiling metrics show that the best configurations for memory bound operations, aim for higher occupancy and primarily stall on long scoreboards,

```
            HWC[3] ----------------memory-pad----------------> HWC[3]                :  0.056ms ( 445 GB/s)
            HWC[3] -----------------iconv-cm-----------------> CHWC8[32]             :  0.313ms ( 467 GB/s, 12 TFLOPS)
         CHWC8[32] -----------------iconv-cm-----------------> CHWC8[32]             :  0.654ms ( 255 GB/s, 59 TFLOPS)
         CHWC8[32] ------------iconv-cm+max-pool------------> CHWC8[48]             :  0.216ms ( 213 GB/s, 67 TFLOPS)
         CHWC8[48] ------------iconv-cm+max-pool------------> CHWC8[64]             :  0.104ms ( 161 GB/s, 69 TFLOPS)
         CHWC8[64] ------------iconv-cm+max-pool------------> CHWC8[80]             :  0.046ms ( 121 GB/s, 65 TFLOPS)
         CHWC8[80] ------------iconv-cm+max-pool------------> HWC[96]               :  0.023ms ( 134 GB/s, 50 TFLOPS)
           HWC[96] -----------------iconv-cm-----------------> CHWC8[96]             :  0.026ms ( 129 GB/s, 53 TFLOPS)
{CHWC8[96],CHWC8[64]} ---------upsample+concat+iconv-cm-------> CHWC8[112]            :  0.150ms (  89 GB/s, 70 TFLOPS)
        CHWC8[112] -----------------iconv-cm-----------------> CHWC8[112]            :  0.108ms ( 138 GB/s, 68 TFLOPS)
{CHWC8[112],CHWC8[48]} ---------upsample+concat+iconv-cm-------> HWC[96]              :  0.447ms ( 101 GB/s, 81 TFLOPS)
           HWC[96] -----------------iconv-cm-----------------> HWC[96]              :  0.298ms ( 169 GB/s, 73 TFLOPS)
  {HWC[96],CHWC8[32]} ---------upsample+concat+iconv-cm-------> CHWC8[64]            :  0.914ms ( 137 GB/s, 84 TFLOPS)
         CHWC8[64] -----------------iconv-cm-----------------> CHWC8[64]            :  0.499ms ( 268 GB/s, 77 TFLOPS)
   {CHWC8[64],HWC[3]} ---------upsample+concat+iconv-cm-------> CHWC8[64]            :  1.894ms ( 183 GB/s, 85 TFLOPS)
         CHWC8[64] -----------------iconv-cm-----------------> CHWC8[32]            :  1.569ms ( 256 GB/s, 49 TFLOPS)
         CHWC8[32] -----------------iconv-cm-----------------> HWC[3]               :  0.459ms ( 318 GB/s,  8 TFLOPS)
            HWC[3] --------------memory-slice--------------> HWC[3]                :  0.048ms ( 519 GB/s)
                                                                          Total time :  7.825ms ( 219 GB/s)
```

Fig. 8. Generated dispatch schedule for an RTX 4070 and the balanced OIDN model at 1920×1080.

which is expected. Those operations also often correspond to convolutions with low channel count, here spatial reuse within the lowered input feature map is small, which results in high temporal L1 cache locally. Compute bound operations, like convolutions with high channel counts, follow the opposite rules. Best configurations primarily rely on high tensor throughput, which often requires larger tile sizes, which implies lower occupancy. However, as our convolution implementation, does not implement explicit spatial reuse, we theorize that larger tile sizes, reduce L1 cache locality, which may result in cache-misses of values, previously loaded. Regardless of these inefficiency we believe that the metrics reflect a well balanced implementation, which manually auto-tuned implementations can use as a reference.

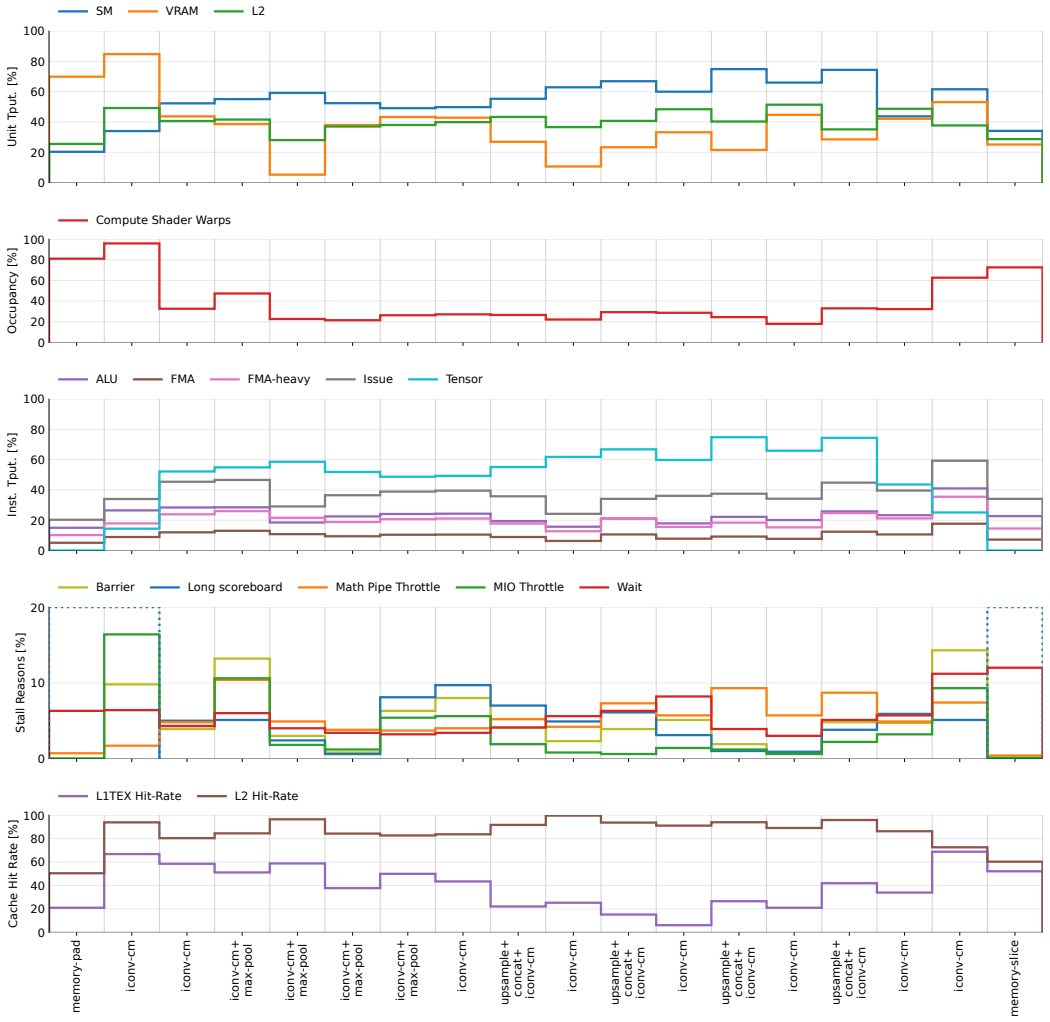


Fig. 9. Profiling metrics recorded with NVIDIA NSight Graphics, on a RTX 4070. The x-axis does not reflect execution time