Peak-Memory-aware Partitioning and Scheduling for Multi-tenant DNN Model Inference

Jaeho Lee

Department of Electrical and Electronic Engineering

Graduate School

Yonsei University

Peak-Memory-aware Partitioning and Scheduling for Multi-tenant DNN Model Inference

Advisor : Prof. Hanjun Kim

A Dissertation Submitted
to the Department of Electrical and Electronic Engineering
and the Committee of the Graduate School
of Yonsei University in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy in Electrical and Electronic Engineering

Jaeho Lee

August 2026

Peak-Memory-aware Partitioning and Scheduling for Multi-tenant DNN Model Inference

This Certifies that the Dissertation of JAEHO LEE is Approved

Committee Chair KIM HANJUN
2026-07-08 21:04:43

Committee Member Song William Jinho
2026-07-03 18:00:58

Committee Member Ro Won Woo
2026-07-07 13:17:15

Committee Member Kim Youngsok
2026-07-06 18:37:29

Committee Member Park Yongjun
2026-07-06 13:01:39

Department of ELECTRICAL AND ELECTRONIC ENGINEERING
Graduate School
Yonsei University
August 2026

감사의 글

이렇게 또 하나의 긴 여정이 마무리됩니다. 지난 시간을 돌아보면, 박사라는 이름은 결코 홀로 이룬 것이 아니라 수많은 분들의 믿음과 도움 위에 조심스레 쌓아 올린 것이었습니다. 그 무거운 책임과 큰 영광을 얻기까지, 스스로를 깎아내고 다시 세우는 지난한 과정을 함께 견뎌주신 모든 분들께 마음 깊이 감사드립니다.

먼저 지도교수님이신 김한준 교수님께 진심으로 감사드립니다. 학부생 시절부터 대학원 생활을 마무리하는 오늘에 이르기까지, 교수님께서는 단 한 번도 저를 향한 믿음을 거두지 않으시고 묵묵히 지지하며 이끌어주셨습니다. 돌이켜보면 저는 교수님과 함께한 시간 내내 참으로 서툴고 말썽 많은 제자였습니다. 대학원에 처음 들어왔을 때는 간단한 코딩 과제 하나조차 버거워했고, 지금 생각하면 얼굴이 붉어질 만큼 철없던 순간도 참 많았습니다. 그럼에도 교수님께서는 끝내 저를 포기하지 않으시고, 앞이 보이지 않을 때마다 나아갈 길을 밝혀주셨습니다. 오늘 제가 한 사람의 연구자로서 첫걸음을 내디딜 수 있게 된 것은 온전히 교수님의 가르침 덕분입니다. 그 한결같은 믿음과 가르침에 다시 한번 깊이 고개 숙여 감사드립니다.

바쁘신 중에도 부족한 학위 논문을 정성껏 심사해주신 심사위원 교수님들께 감사드립니다. 노원우 교수님, 송진호 교수님, 박영준 교수님, 김영석 교수님께 깊은 감사의 말씀을 전합니다. 학부생 시절 노원우 교수님과 송진호 교수님의 강의를 들으며 어렴풋이 컴퓨터 시스템 분야의 연구자를 꿈꾸었습니다. 그때 마음에 심어진 작은 씨앗이 이렇게 한 편의 논문으로 결실을 맺어, 어엿한 박사가 되었다는 사실에 가슴이 벅차오릅니다. 또한 박영준 교수님과 김영석 교수님께서는 대학원 시절 제 연구에 늘 함께해주시며 아낌없는 조언을 주셨습니다. 두 분과 나눈 연구 협업의 시간은 제가 한 사람의 연구자로 자리 잡는 데 더없이 소중한 밑거름이 되었습니다. 이렇듯 네 분 교수님께서 졸업 심사뿐만 아니라 학부 시절부터 대학원에 이르기까지 주신 날카로우면서도 따뜻했던 조언들 덕분에, 이 논문을 한층 더 단단하고 나은 모습으로 마무리할 수 있었습니다.

대학원 생활 동안 동고동락한 선후배님들께도 감사의 인사를 드립니다. 좋은 기억은 언제나 좋은 사람에게서 비롯된다는 말을, 저는 이곳에서 비로소 실감했습니다. 함께 밤을 지새우고 작은 성취에도 제 일처럼 기뻐해준 좋은 분들 덕분에, CoreLab은 제게 오래도록 따뜻하고 즐거운 기억으로 남을 것입니다. 이경민 박사님, 김봉준 박사님, 김창수 박사님, 허선영 교수님, 조성준 박사님, 이용우 교수님, 김동관, 천선영, 윤성우, 박현준, 이찬, 염호윤, 정건모, 김성호, 김재민에게 감사드립니다. 특별히 첫 연구에 큰 도움을 주셨던 정신녕 박사님, 송승빈 박사님, 최희림, 김근우에게, 그리고 여러 우여곡절 끝에 마

침내 결실을 맺은 두 번째 연구에 함께해준 이주민, 정해은, 권현호에게 더욱 깊은 감사를 전합니다. 또한 제가 연구에만 집중할 수 있도록 항상 도움을 주신 남주현 선생님께도 감사드립니다.

언제나 저의 가장 든든한 울타리가 되어준 가족에게 감사드립니다. 빛나던 순간에도, 지치고 흔들리던 순간에도 변함없이 제 곁을 지켜준 가족은 이 긴 여정을 끝까지 버티게 한 가장 큰 힘이었습니다. 세상의 어떤 성취도 나를 믿어주는 사람 없이는 이룰 수 없다는 것을, 저는 가족을 통해 배웠습니다. 지금의 저를 있게 해주신 아버지와 어머니, 그리고 동생에게 이루 다 말할 수 없는 사랑과 감사를 전합니다. 쫑이에게는 감사의 말 대신 좋아하는 간식으로 그 마음을 대신하려 합니다. 또한 지친 일상 속에서도 언제나 저의 즐거움을 책임져준 소중한 친구들에게도 감사의 말씀을 전합니다. 특별히 학부생 시절부터 지금까지, 학업과 연구에 매몰되기 쉬운 제가 일상의 여유와 웃음을 잃지 않도록 곁을 지켜준 석동훈, 윤형석에게 고마운 마음을 전합니다.

이 모든 분들과 함께한 순간순간은 무엇과도 바꿀 수 없는 소중한 시간이었습니다. 받은 마음을 잊지 않고, 앞으로 더 나은 연구자이자 더 나은 사람으로 살아가는 것으로 그 은혜에 보답하겠습니다. 다시 한번, 진심으로 감사드립니다.

서로 사랑하라

내가 너희를 사랑한 것 같이
너희도 서로 사랑하라

요한복음 13:34

저를 여기까지 인도하신 하나님께 감사드리며,
모든 영광을 하나님께 돌립니다.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT

Peak-Memory-aware Partitioning and Scheduling for Multi-tenant DNN Model Inference

Deep neural network (DNN) inference on GPU-based systems spends a large fraction of its time on memory management. Dynamic memory allocation through CUDA APIs consumes 33% to 50% of the total inference time, while the computation itself accounts for only 20% to 25%. The problem worsens in multi-tenant deployments, where the temporary workspace that backend libraries such as cuDNN and cuBLAS allocate internally stays invisible to graph-level analysis and triggers unexpected out-of-memory (OOM) errors.

This study presents a DNN inference compiler and scheduler that plans GPU memory before execution. Because an inference model has a fixed computation graph and deterministic tensor shapes, its memory demand can be analyzed before the model runs, which allows per-tensor dynamic allocation to be replaced with a small number of pool-level allocations. A tensor liveness analysis first unifies visible tensors and the hidden backend workspace into a single memory demand profile, and operation-level optimization such as layer fusion and tensor coalescing removes redundant memory operations. A constraint programming formulation then solves the underlying Dynamic Storage Allocation problem with the CP-SAT solver, producing a provably optimal layout for each memory pool. For concurrent execution, the Tailor-and-Stitch algorithm partitions a

model into memory-balanced schedulable units called Tasks according to its compile-time peak memory profile. A dual memory pool design keeps tensors shared across Task boundaries resident in GPU memory and avoids the backup and restore transfers they would otherwise require. A compiler-generated scheduler interface drives a yield-based scheduler that admits each new Task only after a worst-case memory check, so that concurrent tenants run free of OOM failures.

On NVIDIA RTX 3090 and Jetson AGX Xavier, single-tenant inference uses 34.6% less memory than PyTorch on average and runs 1.25 times faster in geometric mean. On NVIDIA RTX 4090 and Jetson Orin Nano, the system runs multiple models concurrently without any OOM failure at an average performance overhead of 3.20% on the desktop GPU, and it serves requests with up to 10.8% lower latency than the NVIDIA Triton Inference Server and 43.81% lower latency than SwapNet on embedded platforms.

Keywords: DNN inference, memory management, compile-time optimization, memory pool, model partitioning, multi-tenant scheduling, out-of-memory prevention, MLIR

1. Introduction

1.1 Compile-Time Memory Planning for DNN Inference

Deep neural networks (DNNs) are becoming deeper and wider to achieve higher accuracy across diverse application domains, including image classification [1, 2, 3], object detection [4, 5], super-resolution [6], pose estimation, semantic segmentation, and natural language processing [7]. However, the increasing depth and width of these models demand proportionally larger GPU memory capacity. PyTorch [8] requires 24 GB of memory to execute an SRGAN model for image super-resolution, a capacity that commodity GPU devices can barely support. The NVIDIA GeForce RTX 4090, a high-end desktop GPU, provides exactly 24 GB of GDDR6X memory, leaving no headroom for concurrent model execution or other GPU workloads.

This memory constraint becomes severe on embedded devices, which have far more limited memory budgets. The NVIDIA Jetson Orin Nano [9] has only 8 GB of unified CPU-GPU memory that must be shared among the operating system, the application framework, and all DNN tensors. The NVIDIA Jetson AGX Xavier provides 16 GB of unified memory, but even this capacity is insufficient for multiple concurrent DNN models. As DNN models continue to grow in scale, memory management has become a first-class performance concern that can no longer be deferred to the runtime system.

This study is motivated by the observation that DNN inference models expose memory characteristics that differ from those of general-purpose programs. Inference models operate on fixed computation graphs with statically deterministic tensor shapes, and they do not perform backward passes or maintain gradient buffers. The size and lifetime of every tensor in the computation graph can be determined at compile time without executing the model, and the hidden backend workspace is measured by a one-time profiling pass. This static analyzability enables a compile-time approach to memory management that is unavailable for general-purpose programs, where memory usage is typically data-dependent and unpredictable. Model compression techniques such as pruning and quantization [10] are orthogonal to this approach. Compression reduces memory by modifying the model itself, whereas the compile-time approach optimizes memory consumption without altering the model's computation or accuracy.

1.1.1 GPU Memory Allocation and Its Hidden Costs

GPU memory management operations have a substantial impact on DNN inference latency. On an NVIDIA Titan RTX, memory allocation and deallocation operations consume 33% to 50% of the total inference time for ResNet50 and YOLOv1, while the actual computation accounts for only 20% to 25% of the total latency (Figure 1.1). This overhead arises because GPU memory allocation through CUDA APIs such as `cudaMalloc` and `cudaMallocManaged` involves synchronization primitives, page table updates, and system calls that cannot be overlapped with GPU computation. On the NVIDIA Jetson Orin Nano, `cudaMallocManaged` incurs an average delay of 271 microseconds per 1 MB allocation [11], and this delay accumulates across dozens or hundreds of tensor alloca-

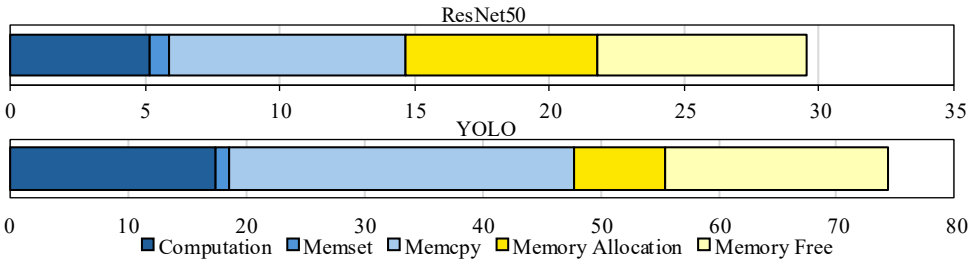
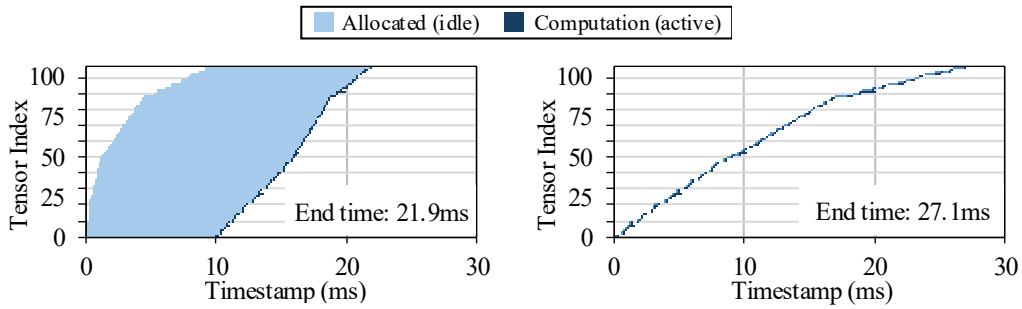


Figure 1.1: Inference time breakdown for ResNet50 and YOLOv1 on NVIDIA Titan RTX. Memory allocation and deallocation operations consume 33–50% of the total inference time, while computation accounts for only 20–25%.

tions in a single inference pass.

Beyond the visible tensor allocations managed by the DNN framework, backend GPU libraries allocate additional temporary workspace buffers at runtime. Libraries such as cuDNN and cuBLAS internally allocate workspace memory for kernel execution, particularly for convolution operations. For a single convolution operator with a 3×3 filter over a 224×224 feature map, the required workspace size varies dramatically depending on the algorithm selected, ranging from 0.39 MB for the Winograd algorithm to 2,096.39 MB for FFT-based convolution [12]. These hidden allocations are invisible in the computation graph and cannot be accounted for by graph-level analysis alone. A single unaccounted hidden allocation can immediately trigger an out-of-memory (OOM) error, even when the visible tensor analysis indicates sufficient available memory. The timing of memory allocation also significantly affects both performance and memory consumption. Two fundamental allocation strategies exist, each with distinct trade-offs (Figure 1.2).

Eager allocation loads all tensors into GPU memory at the beginning of the program. This eliminates allocation overhead during computation, reducing the ResNet50 infer-



(a) Eager allocation

(b) Lazy allocation

Figure 1.2: Eager allocation (a) loads all tensors at program start, achieving fast execution (21.9 ms) but consuming excessive memory. Lazy allocation (b) allocates each tensor immediately before use, reducing peak memory but incurring 24% longer execution time (27.1 ms) due to frequent allocation overhead.

ence time from 27.1 ms to 21.9 ms on an NVIDIA Titan RTX (Figure 1.2a). However, all tensors coexist in memory simultaneously, regardless of whether they are currently in use, resulting in excessive peak memory consumption. Lazy allocation, in contrast, allocates each tensor immediately before its first use and deallocates it after its last use (Figure 1.2b). This strategy reduces peak memory consumption by ensuring that only the tensors required for the current computation reside in GPU memory. However, the frequent allocation and deallocation operations incur substantial overhead, increasing the inference time by 24% (from 21.9 ms to 27.1 ms for ResNet50).

A memory pool strategy combines the advantages of both approaches. By allocating a contiguous memory pool and assigning each tensor a fixed offset within the pool, the compiled binary replaces per-tensor dynamic allocation with a small number of pool-level allocations, removing per-tensor allocation overhead while maintaining low peak memory consumption. However, determining the optimal tensor-to-offset map-

ping within the pool is an instance of the Dynamic Storage Allocation (DSA) problem, which is NP-hard in the general case. Existing frameworks such as PyTorch [8] and TensorFlow Lite Micro [13] employ heuristic allocation policies (e.g., first-fit, best-fit) that reduce fragmentation but cannot guarantee optimality. The compile-time approach adopted by PAGRUS solves this problem exactly using constraint programming, as described in Section 4.4.

1.1.2 Static Liveness Analysis and Optimal Pool Layout

These static properties enable memory optimizations that runtime approaches cannot generally provide. Because the computation graph is fixed and all tensor shapes are deterministic, the compiler can determine the exact liveness interval of every tensor, the precise point at which each tensor is first produced and last consumed, before execution begins. This global visibility over all tensor lifetimes enables the compiler to identify memory reuse opportunities that are invisible to local, per-operator allocation strategies.

Several runtime memory management systems have been proposed for DNN workloads. vDNN [14] offloads tensors to CPU memory during training, Capuchin [15] profiles memory access patterns to guide offloading decisions, and SuperNeurons [16] distinguishes between computation-intensive and memory-intensive layers. However, these systems are designed for DNN training, where the cost of runtime profiling can be amortized over hundreds of training iterations. DNN inference consists of a single forward pass with no repeated iterations, making runtime profiling approaches unsuitable because the profiling overhead cannot be amortized.

Runtime memory pool systems, such as those in TensorFlow Lite Micro [13] and TASO [17], pre-allocate memory pools to avoid per-tensor allocation overhead. However, these systems assign tensors to pool regions using greedy heuristics that make local decisions without considering the global allocation pattern, resulting in suboptimal packing and wasted memory. PyTorch's caching allocator rounds allocation sizes to reduce fragmentation and reuses freed blocks, but it does not minimize the total pool size. TensorFlow XLA [18] performs some compile-time memory planning but does not account for hidden backend allocations and does not support multi-tenant execution.

The compile-time approach adopted by PAGRUS addresses these limitations through three key capabilities. First, the liveness analyzer captures both visible tensors and hidden backend allocations, producing a unified memory demand profile that accounts for both visible and hidden GPU memory use (Section 4.1). Second, the optimal pool planner formulates the tensor layout problem as a constraint satisfaction problem and solves it using the CP-SAT exact solver, producing a provably optimal memory pool layout rather than a heuristic approximation (Section 4.4). Third, the peak memory graph enables memory-aware model partitioning for multi-tenant deployment, dividing the model into schedulable units whose memory demands are precisely known at compile time (Section 4.3).

1.2 OOM-Free Multi-Tenant Scheduling via Compile-Time Metadata

Edge computing applications increasingly require multiple DNN models to execute concurrently on a single GPU [19, 20, 21]. Autonomous vehicles run perception, planning, and prediction models simultaneously [22, 23]. Smart camera systems concurrently per-

form object detection, traffic monitoring, and anomaly detection, each relying on a distinct DNN [24, 25]. Cloud inference servers process requests from multiple clients, each potentially using a different model [26]. This multi-tenant deployment pattern is the dominant use case for both edge and cloud GPU systems.

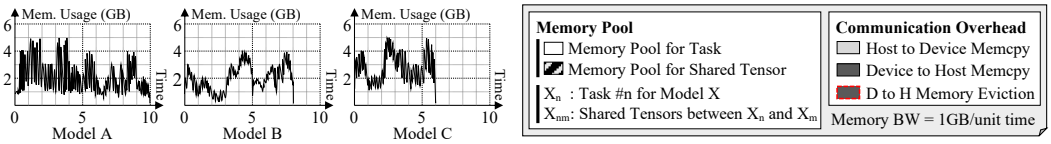
However, efficient multi-tenant DNN inference presents significant challenges due to constrained GPU memory resources. Each DNN model requires substantial memory for storing weights, intermediate activations, and backend workspace buffers. Running multiple models concurrently can quickly exceed the available GPU memory, causing OOM errors that terminate the entire process. On embedded devices such as the NVIDIA Jetson Orin Nano with 8 GB of unified memory, even two medium-sized models can exhaust the available capacity. To avoid these failures, developers commonly fall back on executing DNNs sequentially or severely limiting concurrent execution [27, 28], resulting in GPU underutilization and degraded throughput.

1.2.1 Memory Contention in Concurrent DNN Execution

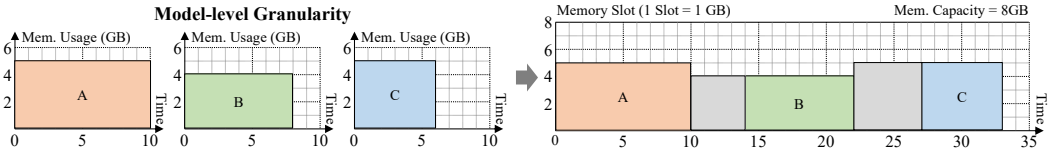
CUDA kernels, once launched, run to completion on the GPU. The GPU hardware does not support preemption of running kernels to reclaim memory, and on the explicit device allocation path that PAGRUS targets, a failed allocation cannot rely on an OS-level reclaim or paging mechanism for safe recovery. When a memory allocation request exceeds the available GPU capacity, the allocation fails immediately with an OOM error, terminating the application. This non-preemptive execution model means that memory contention must be resolved before kernel launch, not during execution.

Scheduling granularity. Figure 1.3 compares the execution timelines produced by different scheduling granularities for three DNN models (A, B, C) on an 8 GB GPU. The per-model memory footprint shown in Figure 1.3 is the liveness-aware peak memory derived by the proposed system, the moment-to-moment maximum of simultaneously live tensors, including weights, activations, and hidden backend workspace. This liveness-aware footprint is itself one contribution of this work, and it is already substantially smaller than the eager allocation of frameworks such as PyTorch that load all tensors at once, so every scheduling strategy in the figure operates on this reduced footprint. Model-level scheduling (Figure 1.3b) treats each entire model as an indivisible unit, reserving GPU memory equal to the model's peak demand before launch [29, 30]. This approach is simple and prevents OOM errors, but wastes memory because peak demand occurs only briefly and the reserved capacity remains idle during the rest of execution. Since the three models' aggregate peak memory exceeds the 8 GB capacity, models must execute sequentially, resulting in long end-to-end latency. Layer-level scheduling (Figure 1.3c) schedules individual layers and enables fine-grained interleaving of multiple models [31, 32, 33, 34]. Although per-unit memory demand is reduced, the large number of scheduling boundaries introduces frequent eviction-and-restore transfers between host and device memory, severely degrading throughput. Neither granularity provides compile-time information about per-unit memory demand to the scheduler, forcing the scheduler to rely on runtime estimation.

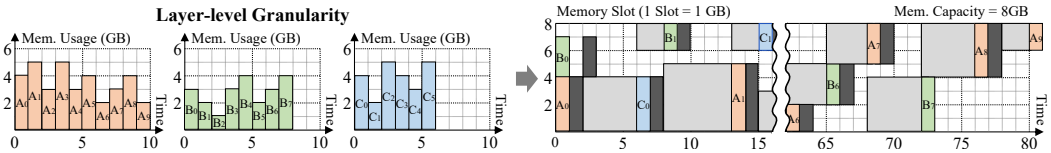
Task-level scheduling and shared tensor pooling. `Task`-level scheduling partitions each model into a small number of balanced `Tasks` (Figure 1.3d), offering a middle



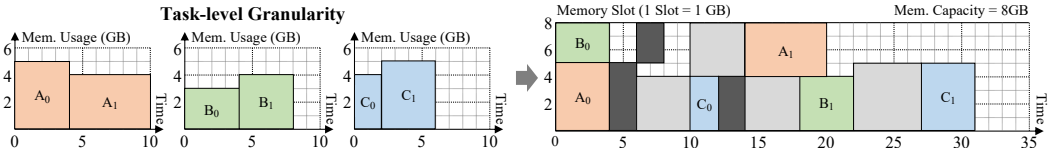
(a) Target models and notation



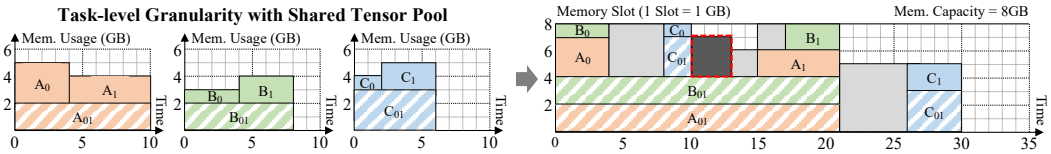
(b) Model-level scheduling



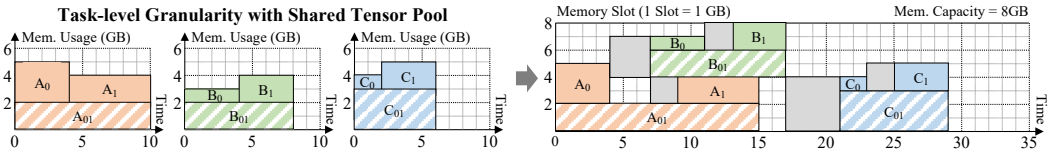
(c) Layer-level scheduling



(d) Task-level scheduling



(e) Task-level scheduling with shared tensor pool



(f) Task-level scheduling with shared tensor pool and yield execution

Figure 1.3: Execution timelines for three DNN models on an 8 GB GPU under six scheduling strategies, progressing from model-level to yield-based Task-level scheduling.

ground between model-level and layer-level granularity [35, 36, 37]. However, intermediate tensors shared between consecutive `Tasks` must remain accessible across `Task` boundaries. The conventional approach backs up these shared tensors to host memory after each `Task` completes and restores them before the next `Task` begins, incurring substantial host-device transfer overhead as illustrated by the dark regions in Figure 1.3d. A dedicated GPU-resident memory pool can retain shared tensors across `Task` boundaries (Figure 1.3e), eliminating these backup-restore transfers entirely.

However, retaining shared tensors in GPU memory causes stale tensor pools to accumulate, eventually preventing new `Tasks` from being deployed and forcing eviction to host memory (Figure 1.3e). Instead of reactive eviction, the proposed system adopts yield scheduling (Figure 1.3f), which defers `Task` deployment when worst-case memory analysis predicts future memory exhaustion. This approach eliminates all eviction overhead, producing the shortest end-to-end latency among all scheduling strategies shown in Figure 1.3. The detailed mechanics of eviction-based and yield-based scheduling, including a step-by-step comparison of their ready queue behavior, are presented in Section 5.2.2.

Runtime polling imprecision. Traditional schedulers monitor available GPU memory at runtime by polling `cudaMemGetInfo` before launching new kernels. However, this polling-based approach is imprecise due to the latency between a memory allocation request and its actual physical page allocation. As discussed in Section 1.1.1, the latency of `cudaMallocManaged` on embedded platforms can reach hundreds of microseconds per allocation. During this delay, the reported available memory does not

reflect the pending allocation, creating an inconsistency window in which the scheduler may overcommit GPU memory and trigger an OOM error. The yield strategy described above requires accurate memory tracking, which motivates the compile-time metadata approach adopted by the proposed system (discussed in Section 1.2.2).

1.2.2 Yield-Based Scheduling with Peak Memory Guarantees

The compile-time approach adopted by PAGRUS addresses the limitations of runtime-based scheduling through three mechanisms.

First, the PAGRUS compiler constructs a peak memory graph for each compiled model, recording the exact worst-case memory demand at every operator index (Section 4.3). This compile-time metadata provides the scheduler with precise, per-`Task` memory requirements without any runtime measurement. The scheduler maintains an internal estimate of available GPU memory based on this metadata, eliminating the need for imprecise runtime polling.

Second, the PAGRUS scheduler employs a yield-based scheduling policy (Section 5.2.2). Before deploying a new `Task`, the scheduler performs worst-case memory analysis using the compile-time peak memory graphs of all active and pending `Tasks`. If deploying the new `Task` would eventually cause GPU memory exhaustion, the scheduler defers (yields) the new `Task`'s execution rather than launching it and later evicting tensors. This policy avoids eviction overhead because yielding does not initiate backup, restore, or eviction transfers. The yielded `Task` remains queued until the completion of other `Tasks` releases sufficient memory.

Third, the compiler-generated scheduler interface (Section 4.5) enables `Tenants` to

communicate their exact memory requirements to the scheduler through signal operations inserted at compile time. Four signal event types, corresponding to `Task` start, `Task` completion, shared tensor pool allocation, and shared tensor pool release, allow the scheduler to maintain an accurate, up-to-date view of GPU memory state without runtime polling of device memory.

Thesis statement. No existing system performs end-to-end compile-time memory planning spanning from per-operator liveness analysis through hidden allocation tracking, optimal memory pool generation, memory-aware model partitioning, to multi-tenant yield-based scheduling. This study presents such a unified system. By shifting memory management decisions from runtime to compile time, a DNN compiler can (1) reduce dynamic allocation overhead by replacing per-tensor allocation with pool-level allocation, (2) reduce memory footprint through global analysis, and (3) enable safe, efficient multi-tenant execution, all without modifying the DNN model or sacrificing inference accuracy.

1.3 Contributions

This study makes the following contributions.

1. **Comprehensive tensor liveness analysis.** A compile-time analysis that tracks both visible tensors in the computation graph and hidden temporary workspace buffers allocated internally by backend GPU libraries (cuDNN, cuBLAS). The analysis produces a unified liveness table covering both visible tensors and hidden

backend workspace across the execution timeline, enabling accurate memory demand estimation that prevents unexpected OOM errors.

2. **Operation-level graph optimization.** Two domain-specific optimizations, layer fusion and tensor coalescing, that reduce the number of live tensors and eliminate redundant memory operations. Layer fusion merges consecutive operators into single fused operators, and tensor coalescing enables element-wise operations to reuse input memory for their output. Both optimizations are verified against the liveness table to ensure correctness.

3. **Provably optimal memory pool layout via CP-SAT.** A constraint programming formulation that solves the Dynamic Storage Allocation (DSA) problem exactly for each intra-`Task` memory pool. The CP-SAT solver assigns each tensor an integer offset within the pool such that temporally overlapping tensors do not share memory regions, while minimizing the total pool size. This approach produces provably optimal layouts, replacing heuristic algorithms (first-fit, best-fit) with an optimality guarantee.

4. **Peak-memory-aware model partitioning.** The Tailor-and-Stitch algorithm that divides a DNN model into balanced, schedulable `Tasks` based on memory usage patterns and estimated execution times. The Tailor phase identifies critical memory transition points and partitions the model into fine-grained `Tasklets`. The Stitch phase merges short `Tasklets` based on execution time estimates to prevent excessively fine-grained scheduling, producing `Tasks` that balance memory utilization against scheduling overhead.

5. **Dual memory pool generation.** A two-pool architecture consisting of an intra-`Task` pool for tensors used exclusively within a single `Task`, and a shared tensor pool for tensors that cross `Task` boundaries. The shared pool remains GPU-resident across `Task` transitions, enabling zero-move tensor sharing that eliminates the host-device transfer overhead incurred by conventional backup-and-restore approaches.

6. **Compiler-generated scheduler interface and yield-based scheduling.** A compile-time metadata structure and signal operations that connect compiled models to the runtime scheduler. The scheduler uses compile-time peak memory graphs to perform worst-case memory analysis and prevent OOM errors through yield-based scheduling, deferring `Task` deployment when memory exhaustion is anticipated rather than evicting tensors after the fact.

1.4 Thesis Organization

The remainder of this thesis is organized as follows.

Section 2 surveys prior work on DNN memory management, DNN compilation, and multi-tenant GPU scheduling, positioning this study relative to existing approaches.

Section 3 presents the overall architecture and design principles of PAGRUS, including the compilation pipeline overview, the DNN dialect, and the implementation platform.

Section 4 describes the PAGRUS compiler in detail. The section covers tensor memory profiling (visible liveness analysis and hidden backend allocation tracking), operation-

level optimization (layer fusion and tensor coalescing), peak-memory-aware model partitioning (the Tailor-and-Stitch algorithm), memory pool generation (CP-SAT exact solver and shared tensor pool), and scheduler interface code generation.

Section 5 presents the PAGRUS runtime scheduler, including the scheduler architecture, the OOM-aware availability checker, and the yield-based scheduling policy that prevents memory exhaustion through worst-case analysis.

Section 6 evaluates PAGRUS across three complementary scenarios. Single-tenant evaluation measures memory usage reduction and inference latency improvement for individual DNN models. Model partitioning evaluation analyzes the Tailor-and-Stitch algorithm and compares scheduling granularities. Multi-tenant evaluation demonstrates OOM-free concurrent execution with yield-based scheduling. Runtime overhead analysis quantifies the cost of scheduling and partitioning.

Section 7 summarizes the contributions and discusses directions for future work.

2. Related Work

This section surveys prior work in two areas that are directly relevant to PAGRUS. Section 2.1 reviews memory management techniques, tensor lifetime planning, and DNN compilers for inference optimization. Section 2.2 reviews multi-tenant scheduling strategies, task partitioning, and capacity-extension mechanisms for out-of-memory prevention. The discussion draws two distinctions throughout. The first separates compile-time planning of the resident footprint from capacity extension through runtime data movement. The second separates the data movement that compiled execution inherently requires from the backup, restore, and eviction transfers that a scheduling policy adds.

2.1 Memory Management and DNN Compilation

Inference-oriented memory management. Several inference frameworks implement memory reuse or pooling strategies to reduce runtime allocation overhead. TensorFlow Lite Micro [13] implements a memory pool with first-fit allocation for embedded CPUs, but the approach targets microcontrollers and does not address GPU-based platforms. PyTorch [8] preallocates tensors at program start and uses a caching allocator to reassign freed GPU memory to later tensors, reducing the cost of individual allocation calls. However, PyTorch's eager allocation tends to raise peak memory usage, because

tensors are allocated as the program requests them without a global lifetime plan. TensorFlow XLA [18] applies buffer aliasing during compilation, enabling input and output tensors to share memory when the computation permits. MXNet [38] employs similar in-place optimizations along with reference counting for automatic memory reclamation. Memory pooling techniques that preallocate a contiguous region and sub-allocate from it at runtime have also been explored in general-purpose GPU applications [39, 40]. Ji et al. [41] propose incremental weight loading for convolution layers, reducing memory occupancy by loading weight data on demand. TASO [17] applies integer linear programming to find an optimal trade-off between execution time and memory usage by varying operator implementations. These approaches reduce memory usage or allocation overhead in isolation, but they address only individual aspects of the memory management problem rather than optimizing the full pipeline from liveness analysis through pool layout to model partitioning.

Training-oriented memory offloading. A distinct body of work targets memory optimization for DNN training, where iterative execution allows profiling-based amortization of overhead. vDNN [14] virtualizes CPU and GPU memory by offloading intermediate tensors to CPU memory during the forward pass and prefetching them back to GPU memory during the backward pass. SuperNeurons [16] characterizes layers as computation-intensive or memory-intensive, and selectively offloads tensors in computation-intensive layers to overlap CPU-GPU communication with computation. Capuchin [15] profiles tensor access patterns at runtime and makes fine-grained prefetch and eviction decisions at tensor granularity. TSplit [42] splits tensors into micro-tensors at a finer

granularity, reducing memory fragmentation for models with large intermediate tensors. Yang and Cheng [43] propose grouping tensors by mobility and allocating each group in separate memory regions to reduce fragmentation during training. These systems establish important mechanisms for reducing training memory, including offloading, prefetching, recomputation, and tensor subdivision. Their assumptions differ from the static-shape inference that PAGRUS targets. Training systems amortize profiling and migration planning across repeated iterations, whereas PAGRUS receives a single compiled forward path and must expose a predictable scheduler interface before execution. PAGRUS therefore uses compile-time liveness, backend workspace accounting, and fixed pool offsets rather than runtime access-profile feedback.

Dynamic GPU memory allocation optimization. Another line of research optimizes the GPU memory allocation mechanism itself rather than the DNN memory usage pattern. Winter et al. [44] identify that dynamic memory allocation becomes a performance bottleneck in SIMT applications, where lock contention during allocation degrades GPU parallelism. Gelado and Garland [45] propose a two-stage concurrent GPU memory allocator that minimizes lock steps during allocation and deallocation, enabling concurrent allocations across multiple threads without significant performance loss. Shin et al. [46] optimize GPU page table walk scheduling to reduce the latency of virtual memory address translation for dynamically allocated memory. Unified memory and asynchronous copy mechanisms [47, 48, 49, 50] can hide data transfer latency, but they do not address the synchronization overhead of dynamic allocation and deallocation calls. GMLake [51] uses GPU virtual memory stitching to reduce allocator-level fragmentation

by mapping non-contiguous physical memory blocks into a contiguous virtual allocation. This allocator-level approach is complementary to liveness-aware layout. GMLake improves the behavior of allocation requests that the framework has already issued, whereas PAGRUS removes those requests from the compiled inference path. By determining tensor placements at compile time and mapping them to fixed offsets in a pre-allocated pool, it eliminates per-tensor runtime allocation and deallocation and the associated synchronization overhead.

Lifetime and placement optimization. MODeL [52] formulates neural network memory optimization as an integer linear program over tensor lifetime and memory location for training. This work is closest to PAGRUS in treating tensor lifetime as a planning object rather than an allocator side effect. PAGRUS specializes this direction for static-shape inference. It combines visible tensor liveness, hidden backend workspace tracking, CP-SAT pool layout, Tailor-and-Stitch partitioning, shared tensor pools, and yield scheduling metadata in one deployment pipeline.

DNN compilers. Modern DNN compilers focus primarily on computation optimization. TVM [53] generates optimized operator implementations through an automated search over hardware-specific schedules, targeting loop tiling, vectorization, and operator fusion. TASO [54] automatically generates graph substitutions and uses a cost-based search to find combinations that reduce execution time. XLA [18] performs whole-program optimization within TensorFlow, fusing operators and managing buffer assignments for TPU and GPU targets. Glow [55] applies graph-level and instruction-level op-

timizations for neural network deployment. MLIR [56] provides a multi-level intermediate representation infrastructure that enables progressive lowering from high-level graph representations to hardware-specific code. ONNX-MLIR [57] compiles ONNX models through MLIR's dialect infrastructure, producing standalone executables with CPU and accelerator support. These compilers optimize the computational aspects of DNN execution, such as operator fusion, scheduling, and code generation. However, these compilers do not combine liveness analysis with hidden backend allocation tracking, exact memory pool layout via constraint satisfaction (CP-SAT), memory-aware model partitioning, and scheduler interface code generation for multi-tenant coordination. The PAGRUS compiler extends the ONNX-MLIR infrastructure with these memory-focused capabilities, complementing rather than replacing the computation optimizations provided by existing compilers. The single-tenant memory optimization pipeline builds on preliminary work by Lee et al. [58], which demonstrated compile-time memory pool generation for individual DNN models.

2.2 Multi-Tenant Scheduling and OOM Prevention

Model-level scheduling. The simplest approach to multi-tenant GPU scheduling treats each DNN model as an indivisible unit. NVIDIA Multi-Process Service (MPS) enables multiple processes to share a single GPU context, reducing context-switching overhead [59, 60]. Spatial partitioning approaches allocate fixed fractions of GPU resources (streaming multiprocessors, memory bandwidth) to each tenant [61, 62, 63, 64, 65, 66]. NVIDIA CUDA streams allow overlapping of kernel execution and memory transfers within a single process [30, 31]. GPU kernel preemption has been studied as a mechanism for

priority-based scheduling and deadline enforcement [32, 33, 67, 68, 69, 70, 71, 72]. Co-optimization of scheduling and resource allocation has been explored to improve throughput on GPU clusters [23, 29]. These model-level approaches are simple to implement and prevent interference between tenants, but they reserve GPU memory equal to each model's peak demand for the entire execution duration, leading to significant memory underutilization as analyzed in Section 1.2.1.

Layer-level scheduling. Fine-grained layer-level scheduling enables interleaving of individual layers from different models, reducing the memory reserved by each tenant at any given time. Layer-level scheduling is widely used to reduce inference latency and enable pipeline-parallel execution [73, 74, 75, 76, 77]. PREMA [33] implements preemptive scheduling at the layer boundary, allowing high-priority models to interrupt lower-priority ones. Pipelined scheduling [32] overlaps the execution of layers from different models in a pipeline fashion. LayerWeaver [34] applies layer-aware scheduling that considers the computational characteristics of each layer when making scheduling decisions. Layer-level scheduling is also utilized to offload partial workloads to the host [78, 79] or to edge devices [80], reducing the compute burden on a single GPU. Layer-level scheduling provides fine-grained control over memory usage, but the large number of scheduling boundaries introduces frequent synchronization points, eviction-and-restore transfers, and context-switching overhead that degrades throughput, as demonstrated in the evaluation (Section 6.3.2).

Task-level scheduling. An intermediate granularity groups consecutive layers into tasks or subgraphs, balancing scheduling flexibility against overhead. Melon [81] introduces tensor-lifetime-aware partitioning that tracks memory usage across the model graph, but targets only single-model optimization and does not support multi-tenant deployment. SwapNet [35] partitions models into uniform segments and uses host-device tensor swapping to handle memory constraints on embedded GPUs. However, SwapNet's uniform partitioning does not account for per-segment memory demand, and its reliance on swapping introduces data transfer overhead. Demand Layering [82] and MAGIS [83] reduce runtime memory pressure by coordinating layer demand, graph transformation, and scheduling under memory constraints. FlexNN [84] plans slicing, loading, and computing for memory-constrained edge inference, targeting a different setting from `Task`-level sharing across concurrent `Tenants`. Heimdall [36] and Co-Edge [37] schedule task-level units across heterogeneous edge devices, optimizing for resource utilization in distributed settings. BAND [85] and HerTI [86] partition models into subgraphs for heterogeneous processing on mobile platforms. MOSAIC [87] and related work [88] explore task-level scheduling for memory-constrained environments. PAGRUS differs from these approaches in two respects. First, the Tailor-and-Stitch partitioning algorithm produces memory-balanced `Tasks` using compile-time peak memory analysis, unlike the memory-unaware uniform partitioning used by SwapNet or the single-model focus of Melon. Second, the yield-based scheduler prevents OOM by applying worst-case memory analysis before deployment rather than relying on reactive swapping or eviction. Because shared tensors remain in GPU-resident pools, scheduling does not introduce backup, restore, or eviction transfers for static-shape intermedi-

ate tensors. The partitioning and multi-tenant scheduling framework extends the work of Lee et al. [11], which introduced peak-memory-aware model partitioning and yield-based scheduling for multi-tenant DNN inference.

Runtime capacity extension. NVIDIA Unified Virtual Memory (UVM) extends addressable GPU memory by allowing transparent access to host memory, triggering page migration on demand. DeepUM [89] uses UVM migration and prefetching to run DNN workloads whose tensors exceed physical GPU memory. G10 [90] extends this capacity view with a unified GPU memory and storage architecture that schedules tensor movement across GPU memory, host memory, and storage. SwapAdvisor [91] applies DNN graph-level dependency analysis to guide tensor placement and swapping decisions. RT-Swap [92] addresses real-time multi-DNN inference by incorporating swap overhead into schedulability analysis. DeepWear [93] targets wearable devices, partitioning DNN inference between embedded processors and host devices to manage memory constraints. Not all of these systems are purely reactive, because G10 and SwapAdvisor schedule movement ahead of time using compiler or graph analysis. Their primary capacity mechanism is nonetheless the movement of tensors, pages, or memory objects between GPU memory and a backing store. In contrast, PAGRUS first reduces the GPU-resident footprint of static-shape inference through compile-time metadata and fixed-offset pool layout, then applies runtime admission control. The compiler constructs peak memory graphs, assigns optimal tensor offsets, and generates worst-case memory metadata for each compiled `Task`. The runtime scheduler uses this metadata to determine whether a `Task` can be deployed under the current memory state. Input,

constant, parameter, and output copies still occur during model execution, including H2D copies for inputs and constants and D2H copies for outputs, and progressive tensors can require tier promotion when the compiled model exceeds a tier capacity. These transfers are generated by the compiled program rather than by the scheduling policy. When the admission test predicts that deployment would cause memory exhaustion, the scheduler leaves the `Task` in the ready queue until other `Tasks` complete and release sufficient memory. Therefore, OOM prevention does not require the scheduler to initiate backup, restore, or eviction transfers.

Orthogonal memory reductions. Several techniques reduce memory pressure at a different level from PAGRUS. PagedAttention [94] targets autoregressive LLM serving by managing KV-cache blocks with a paging-inspired allocator. FlashAttention [95] reduces attention operator memory traffic by avoiding materialization of the full attention matrix and by improving movement between HBM and SRAM. AWQ [96], SmoothQuant [97], and QLoRA [98] reduce tensor byte size through quantization. These methods are complementary to PAGRUS rather than direct replacements. When they change tensor sizes, workspaces, or operator materialization patterns before compilation, PAGRUS can use the resulting lifetimes and sizes as inputs to pool layout and scheduling metadata.

3. PAGRUS System Overview

PAGRUS is a compile-time memory management framework for DNN inference on GPUs, targeting both single-tenant and multi-tenant deployment scenarios. The system consists of two major components. The PAGRUS compiler (Section 4) performs static analysis of the input DNN model and generates memory-optimized executable binaries with pre-planned memory pool layouts. The PAGRUS runtime scheduler (Section 5) coordinates the concurrent execution of multiple compiled models on a shared GPU, preventing out-of-memory errors through compile-time metadata.

The following sections describe the design principles (Section 3.1), the compilation pipeline (Section 3.2), the DNN dialect (Section 3.3), the implementation platform (Section 3.4), and a motivating example that illustrates the end-to-end workflow (Section 3.5).

3.1 Design Principles

The design of PAGRUS is guided by four principles that distinguish it from existing DNN inference frameworks.

Compile-time over runtime. All memory management decisions in PAGRUS are made statically during compilation. The compiler determines the size, offset, and lifetime of every tensor before execution begins, replacing per-tensor dynamic allocation with

constant-offset accesses into a small number of pre-allocated memory pools. This eliminates the overhead of runtime memory allocation and deallocation, which can consume 33–50% of inference time on discrete GPUs (Section 1.1.1). By shifting these decisions to compile time, PAGRUS avoids the performance penalties associated with both eager allocation (excessive peak memory) and lazy allocation (frequent allocation overhead).

Global analysis. PAGRUS analyzes the entire DNN model as a single unit rather than optimizing individual operators or layers in isolation. End-to-end visibility over all tensor lifetimes enables the compiler to identify memory reuse opportunities that are invisible to local, per-operator allocation strategies. For example, two tensors with non-overlapping lifetimes can share the same memory region, and the compiler can determine the globally optimal layout through exact constraint solving. This whole-program perspective also enables the peak memory graph construction that drives the model partitioning algorithm.

Unified pipeline. The PAGRUS compilation pipeline handles both single-tenant and multi-tenant deployment through the same analytical foundation. Single-tenant inference is the degenerate case of the multi-tenant pipeline. The model is treated as a single `Task` spanning the entire computation graph, one intra-`Task` memory pool is generated, and no scheduling signals or shared tensor pools are needed. For multi-tenant deployment, the same pipeline additionally partitions the model into multiple `Tasks`, generates shared tensor pools for cross-`Task` tensors, and inserts signal operations that interface with the runtime scheduler. This unified design keeps the multi-tenant compilation

stages off the single-tenant path, as the additional stages are bypassed for single-tenant deployment.

Accuracy-preserving. The PAGRUS compiler never modifies the DNN model's weights, architecture, or computation semantics. All optimizations operate exclusively on the memory management layer, rearranging how and when tensor memory is allocated and accessed. The inference output produced by a PAGRUS-compiled binary is numerically equivalent to that of the original model executed through standard frameworks such as PyTorch. This property is critical for deployment in safety-sensitive applications where model certification requires exact reproducibility.

3.2 Compilation Pipeline Overview

Figure 3.1 illustrates the overall architecture of PAGRUS. The compilation pipeline transforms an input DNN model described in the ONNX format into an executable binary with optimized memory management. The compiler pipeline consists of five stages, each corresponding to a section of Section 4, followed by a final back-end code generation step. The runtime scheduler (Section 5) operates on the compiled output.

Stage 1, Tensor Memory Profiling (Section 4.1). The front-end converts the ONNX model into the PAGRUS DNN dialect, which explicitly represents memory management operations alongside DNN computations. The visible tensor liveness analyzer determines the definition and last-use points of every tensor in the computation graph, inserting explicit `DNN.Malloc`, `DNN.Dealloc`, and `DNN.Memcpy` operations at the appropri-

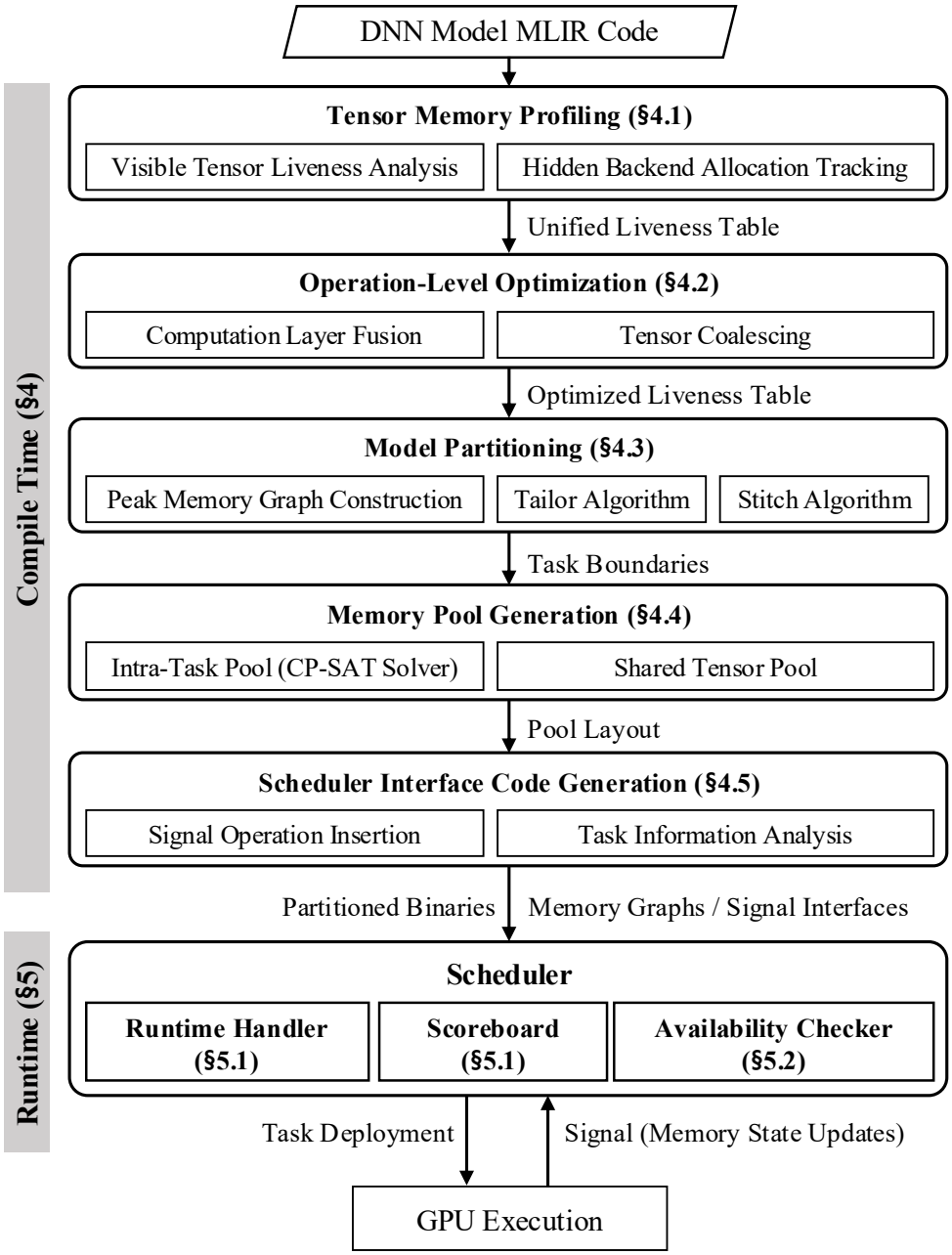


Figure 3.1: Overview of the PAGRUS system. The compiler transforms an ONNX model into memory-optimized executable binaries through a multi-stage pipeline. For multi-tenant deployment, the runtime scheduler coordinates concurrent model execution using compile-time metadata.

ate program points. The hidden backend allocation tracker then identifies temporary workspace buffers allocated internally by GPU libraries such as cuDNN and cuBLAS, modeling them as ephemeral tensors in the liveness table. The output is a unified liveness table that captures all GPU memory consumption at every point in the execution timeline.

Stage 2, Operation-Level Optimization (Section 4.2). Two domain-specific optimizations reduce the number of tensors and their memory footprint. Layer fusion merges consecutive operators (e.g., convolution followed by ReLU) into a single fused operator, eliminating intermediate tensor allocations. Tensor coalescing allows element-wise operations to reuse input memory for their output, removing redundant tensor copies. Both optimizations are applied only when the affected tensors are provably dead after the optimization point, as verified by the liveness table.

Stage 3, Peak-Memory-Aware Model Partitioning (Section 4.3). The peak memory analyzer simulates memory allocation over the execution timeline using a best-fit algorithm, producing a peak memory graph that records the maximum concurrent memory demand at each operator index. For multi-tenant deployment, the model partitioner applies the Tailor-and-Stitch algorithm to divide the model into `Tasks` based on the peak memory graph, balancing memory utilization within each `Task` against scheduling overhead. For single-tenant deployment, the entire model constitutes a single `Task`, and this stage is bypassed.

Stage 4, Memory Pool Generation (Section 4.4). The optimal pool planner uses a CP-SAT exact solver to compute provably optimal memory pool layouts for each intra-`Task` pool, determining the offset of every tensor such that the total pool size is minimized. For multi-tenant deployments, a shared tensor pool is additionally generated for tensors that cross `Task` boundaries. All `DNN.Malloc` operations are replaced with `DNN.Pool-init` and `DNN.Mem-offset` operations, replacing per-tensor allocation and deallocation with pool-based addressing in the compiled binary.

Stage 5, Scheduler Interface Code Generation (Section 4.5). For multi-tenant deployment, the scheduler interface code generation pass inserts signal operations (`DNN.Signal-send / recv`) that communicate memory events to the scheduler, and generates a data-sharing interface containing compile-time metadata (number of `Tasks`, peak memory graph, per-`Task` pool sizes). For single-tenant deployment, this stage is bypassed entirely, as no scheduler communication is needed. The back-end then lowers the DNN dialect to LLVM IR and produces an executable binary.

The key insight of this pipeline is that all memory management decisions, including pool sizes, tensor offsets, and partitioning boundaries, are determined before execution begins. At runtime, the compiled binary issues no per-tensor allocation or deallocation calls. Instead, it allocates its memory pools at startup and accesses all tensors through constant-offset arithmetic from each pool base address.

3.3 The DNN Dialect

The PAGRUS compiler introduces a custom MLIR dialect, the DNN dialect. This dialect extends the standard ONNX dialect with explicit memory management and scheduling operations, enabling the compiler to analyze and optimize GPU memory usage at the IR level.

Why a custom dialect is necessary. The standard ONNX dialect represents DNN models as pure computation graphs without any notion of memory allocation, host-device data transfer, or memory pool management. All tensors are implicitly treated as host-side values, and memory management is deferred entirely to the runtime system. This abstraction is insufficient for compile-time memory optimization because the compiler cannot reason about when and where tensors are allocated, how long they reside in GPU memory, or whether two tensors can share the same memory region. The DNN dialect addresses this gap by making memory management operations first-class citizens in the dialect, allowing the compiler to manipulate them through standard MLIR transformation passes.

Operation categories. The DNN dialect organizes operations into three categories, summarized in Table 3.1. Computation operations provide DNN dialect counterparts for standard ONNX operators (e.g., convolution, pooling, activation), with each operation explicitly declaring memory space attributes for its input and output tensors to distinguish between host and device memory. Memory management operations control GPU memory allocation, deallocation, host-device data transfer, memory pool initialization,

and offset-based tensor addressing within pools. Signal and interface operations, used only in multi-tenant compilation, connect the compiled binary to the runtime scheduler by notifying it of `Task` lifecycle events and receiving yield decisions. Their detailed behavior is described in Section 5. The complete list of DNN dialect operations across all categories is provided in Appendix A.

Table 3.1: The three operation categories of the DNN dialect, with representative operations. The complete operation list is provided in Table A.1 of Appendix A.

Category	**Representative operations**	**Purpose**
Computation	`dnn.convfwd`, `dnn.add`, `dnn.matmul2d`, `dnn.reshape`	DNN dialect counterparts of ONNX operators, with host and device memory space attributes
Memory management	`dnn.malloc`, `dnn.dealloc`, `dnn.memcpy`, `dnn.mempool-init`, `dnn.mem-offset`	GPU memory allocation, transfer, and pool-based offset addressing
Signal and interface	`dnn.schedule_queue`	Scheduler communication for `Task` lifecycle (multi-tenant only, Section 5)

IR transformation stages. The DNN dialect undergoes six transformation stages during compilation. IR 0 is the ONNX representation of the input model. IR 1 is the DNN dialect with explicit memory management operations inserted at each tensor's definition and last-use points based on liveness analysis. IR 2 extends IR 1 with hidden backend allocations such as cuDNN workspace tensors. IR 3 applies operation-level optimizations (layer fusion, tensor coalescing) to reduce the tensor count and memory footprint. IR 4 inserts model partitioning boundaries that divide the program into independently schedulable `Tasks`. IR 5 replaces all individual memory allocation operations with memory pool operations, producing the final memory-optimized code where every

tensor access is a constant-offset computation from a pre-allocated pool base. Section 4 presents each transformation stage in detail with worked examples.

3.4 Implementation Platform

The PAGRUS compiler is implemented as an extension of ONNX-MLIR (version 0.4.0), an open-source compiler that translates ONNX models to LLVM IR through the MLIR infrastructure. The DNN dialect is defined using MLIR TableGen, which generates the C++ boilerplate for operation definitions, type constraints, and verification routines. All analysis and optimization passes described in Section 4 are implemented as standard MLIR transformation passes operating on the DNN dialect.

The GPU computation backend uses the LibTorch C++ API (ATen functions) as wrappers around cuDNN and cuBLAS library calls. This design allows the compiled binary to leverage the same highly optimized GPU kernels used by PyTorch, while replacing PyTorch's dynamic memory management with the compile-time memory pool strategy. The compiled binary is a standalone executable that requires only the CUDA runtime and the LibTorch shared library at deployment time.

Two hardware platform configurations are used for evaluation, reflecting different deployment scenarios. Platform Set A consists of an NVIDIA GeForce RTX 4090 GPU (24 GB GDDR6X) with an AMD Ryzen 9 3950X processor on the desktop side, paired with an NVIDIA Jetson Orin Nano (8 GB unified memory) on the embedded side. This platform set targets multi-tenant and model partitioning evaluation (Sections 6.3 and 6.4). Platform Set B consists of an NVIDIA GeForce RTX 3090 GPU (24 GB GDDR6X) with an Intel Core i7-8700 processor, paired with an NVIDIA Jetson AGX Xavier (16 GB unified mem-

ory). An NVIDIA Titan RTX (24 GB GDDR6) is additionally used for allocation overhead microbenchmarks. This platform set is used for single-tenant evaluation (Section 6.2). The two platform sets are not mixed within a single experiment, and each evaluation section in Section 6 explicitly states which platform set was used.

3.5 Motivating Example

This section illustrates the end-to-end PAGRUS workflow using a simplified two-layer convolutional network with a residual connection. The network consists of an input tensor x, two convolution layers with weights (w_1, b_1) and (w_2, b_2), and a residual addition that sums the second convolution's output with the first activation's output via a skip connection. This example is intentionally small to demonstrate the key concepts. The same network serves as the running example throughout Section 4, where each stage is explained in full detail.

Step 1, Visible liveness analysis. The compiler analyzes the ONNX computation graph and determines the liveness interval of every tensor. For example, the input tensor x is defined at the beginning of the program and last used by the first convolution, while the weight tensor w_1 is defined when it is loaded from host memory and last used by the first convolution. The result is a liveness table mapping each tensor to its definition point, last-use point, and byte size (see Table 4.1 in Section 4). Based on this analysis, the front-end converts the ONNX dialect into the DNN dialect by inserting explicit `DNN.Malloc`, `DNN.Dealloc`, and `DNN.Memcpy` operations at the appropriate program points.

Step 2, Hidden allocation tracking. The convolution operators internally invoke cuDNN, which allocates temporary workspace buffers invisible in the computation graph. The compiler queries the cuDNN workspace API for each convolution configuration and inserts ephemeral tensor entries into the liveness table. The result is a unified liveness table that captures all GPU memory consumption, both user-visible tensors and library-internal workspace buffers (see Table 4.3 in Section 4).

Step 3, Operation-level optimization. The compiler identifies that the first convolution is followed by a ReLU activation, and fuses them into a single `DNN.Conv-relu` operator, eliminating one intermediate tensor allocation. The compiler also identifies that the residual addition is an element-wise operation whose output can reuse the input memory, coalescing the input and output tensors. The liveness table is updated to reflect the reduced tensor count.

Step 4, Peak memory graph construction. Using the unified liveness table, the compiler simulates memory allocation over the execution timeline with a best-fit algorithm. At each operator index, the simulator records the total amount of GPU memory occupied by all simultaneously live tensors. The result is the peak memory graph (Figure 4.5 in Section 4), a time-series that reveals the memory demand pattern of the model.

Step 5, Memory pool generation. For single-tenant deployment, the compiler formulates the memory layout problem as a constraint satisfaction problem and solves it using the CP-SAT exact solver. Each tensor is assigned an integer offset variable, and disjunctive constraints ensure that tensors with overlapping liveness intervals do not share

memory regions. The solver minimizes the total pool size, producing a provably optimal layout. All `DNN.Malloc` operations are then replaced with a single `DNN.Pool-init` followed by `DNN.Mem-offset` operations that compute each tensor's address as a constant offset from the pool base.

Step 6, Multi-tenant extensions. For multi-tenant deployment, two additional stages are applied before pool generation. First, the Tailor-and-Stitch algorithm partitions the model into multiple `Tasks` based on the peak memory graph, grouping operators with similar memory demand into the same `Task`. Second, the scheduler interface code generation pass inserts signal operations and produces the data-sharing interface that allows the scheduler to track per-`Task` memory usage. Tensors shared across `Task` boundaries are placed in a dedicated GPU-resident shared pool, avoiding costly host-device transfers at every `Task` transition. The scheduler uses the compile-time metadata to perform worst-case memory analysis and prevent out-of-memory errors through yield-based scheduling (Section 5).

Single-tenant as the degenerate case. When the target deployment is single-tenant, Steps 5 and 6 collapse. The entire model is a single `Task` with one memory pool, and the multi-tenant stages (partitioning, shared pool generation, signal insertion) are bypassed as described above. The compiled binary runs independently without any runtime scheduler involvement. In this degenerate case, single-tenant execution bypasses the multi-tenant stages and runs without scheduler involvement.

4. PAGRUS Compiler

The PAGRUS compiler plans memory before execution, through a compile-time analysis and optimization pipeline that includes a profiling step for backend workspace sizes. The pipeline proceeds in five stages. Tensor memory profiling (Section 4.1) constructs the unified liveness table, operation-level optimization (Section 4.2) reduces redundant tensors, memory-aware model partitioning (Section 4.3) divides the model into `Tasks`, memory pool generation (Section 4.4) produces the final pool layouts, and scheduler interface code generation (Section 4.5) emits the scheduler-facing signal operations for multi-tenant deployment. For single-tenant inference, Section 4.3 produces a single `Task` spanning the entire model, and Section 4.4 generates one whole-model pool. For multi-tenant inference, the model is partitioned into multiple `Tasks`, each with its own intra-`Task` pool, and a shared tensor pool is generated for cross-`Task` tensors.

4.1 Tensor Memory Profiling

The first stage of the PAGRUS compiler constructs a comprehensive memory demand profile for the input DNN model. This stage consists of two analysis activities. First, visible tensor liveness analysis (Section 4.1.1) determines the definition and last-use points of every tensor in the computation graph, converting the ONNX dialect into the DNN dialect with explicit memory management operations. Second, hidden backend allocation

tracking (Section 4.1.2) identifies temporary workspace buffers allocated internally by backend libraries such as cuDNN and cuBLAS, which are invisible in the computation graph. The output of this stage is a unified liveness table covering both visible tensors and hidden backend workspace buffers across the execution timeline.

4.1.1 Visible Tensor Liveness Analysis

Every operator in a DNN inference model consumes one or more input tensors and produces one or more output tensors. Each tensor has a definition point, the operator index at which it is first produced, and a last-use point, the operator index at which it is last consumed. The interval between these two points defines the tensor's liveness. The tensor must reside in GPU memory throughout this interval and can be freed afterward. Because DNN inference models have fixed computation graphs with statically determined tensor shapes, all liveness intervals can be computed at compile time without executing the model. Determining these intervals for all tensors is the task of the visible tensor liveness analysis.

Algorithm 4.1 presents the liveness analysis algorithm. The algorithm receives the ordered operator list from the input model and produces a liveness table LA that maps each tensor identifier to a tuple $(\mathit{begin}, \mathit{end}, \mathit{size})$. For each operator in the list, the analyzer first updates the last-use index of every operand consumed by that operator (Line 3–Line 5). Then, for each result tensor produced by the operator, a new table entry is created with both the first-use and last-use set to the current operator index, and the byte size is computed from the tensor's dimensions and data type (Line 6–Line 8). As subsequent operators consume this tensor, its last-use index is updated accordingly.

Algorithm 4.1: Tensor Liveness Analyzer

Input: *opList*: Operator lists of DNN inference model
Output: *LA*: Liveness table recording begin, end, and size of variables

```
1  Function LivenessAnalysis (LA, opList) :
2      for op ∈ opList do
3          for operand ∈ op.operands do
4              LA[operand].end ← op
5          end
6          LA[op].begin ← op                    // op returns a new tensor
7          LA[op].end ← op
8          LA[op].size ← op.size                // computed from dimensions
9      end
10     return LA
11 end
```

Running Example. To illustrate the compilation pipeline concretely, this section uses a simplified residual block as a running example. The model consists of two convolution layers with a skip connection, processing a $1 \times 3 \times 4 \times 4$ input image with 32-bit floating-point precision. Figure 4.1 shows the ONNX dialect representation of this network. Tensor `x` holds the input image, `w1` and `b1` are the weight and bias parameters for the first convolution, and `r1` is the first ReLU activation output. The second convolution is followed by an elementwise addition `onnx.Add` that combines the convolution output `c2` with the skip connection from `r1`, and a final ReLU activation produces the output `r2`. Because the skip connection feeds `r1` into both `onnx.Conv` (line 7) and `onnx.Add` (line 8), tensor `r1` must remain in GPU memory across a wider interval than it would without the skip connection.

Applying Algorithm 4.1 to this operator list produces an initial liveness table. Tensor `x` is defined at line 0 and last consumed by `onnx.Conv` at line 3, yielding the interval

```
0 x  = INPUT                  // input image (1×3×4×4)
1 w1 = onnx.Constant()        // conv1 weights (8×3×3×3)
2 b1 = onnx.Constant()        // conv1 bias (8)
3 c1 = onnx.Conv(x, w1, b1)   // conv1 output (1×8×4×4)
4 r1 = onnx.Relu(c1)          // ReLU activation 1
5 w2 = onnx.Constant()        // conv2 weights (8×8×3×3)
6 b2 = onnx.Constant()        // conv2 bias (8)
7 c2 = onnx.Conv(r1, w2, b2)  // conv2 output (1×8×4×4)
8 a1 = onnx.Add(r1, c2)       // skip connection
9 r2 = onnx.Relu(a1)          // ReLU activation 2 (output)
```

Figure 4.1: ONNX dialect of the running example, a simplified residual block with a skip connection. Tensor `r1` is consumed by both `onnx.Conv` at line 7 and `onnx.Add` at line 8, extending its liveness interval.

[0, 3]. Tensor `c1` has a short lifetime ([3, 4]), existing only between the convolution that produces it and the ReLU that consumes it. The key observation is that tensor `r1` has an extended liveness interval of [4, 8] because the skip connection causes it to be consumed by both `onnx.Conv` at line 7 and `onnx.Add` at line 8. Without the skip connection, `r1` would be freed at line 7. This extended liveness means that `r1` must coexist in GPU memory with the weight, bias, and output tensors of the second convolution layer, increasing the peak memory demand.

ONNX dialect to DNN dialect Conversion. Based on the liveness analysis result, the PAGRUS compiler inserts explicit GPU memory management operations and converts each ONNX operation to its DNN dialect counterpart. Four types of memory management operations are inserted. First, a `DNN.Malloc` is inserted at each tensor's definition point to allocate a device-memory buffer. Second, a `DNN.Dealloc` is inserted at each tensor's last-use point to release the allocated memory. Third, a `DNN.Memcpy` with direc-

```
1  x      = INPUT
2  d_x    = DNN.Malloc(192)
3  t0     = DNN.Memcpy(d_x, x) {H2D}
4  w1     = onnx.Constant()
5  d_w1   = DNN.Malloc(864)
6  t1     = DNN.Memcpy(d_w1, w1) {H2D}
7  b1     = onnx.Constant()
8  d_b1   = DNN.Malloc(32)
9  t2     = DNN.Memcpy(d_b1, b1) {H2D}
10 d_c1   = DNN.Malloc(512)
11 t3     = DNN.Conv(d_x, d_w1, d_b1, d_c1)
12 t4     = DNN.Dealloc(d_x)
13 t5     = DNN.Dealloc(d_w1)
14 t6     = DNN.Dealloc(d_b1)
15 d_r1   = DNN.Malloc(512)
16 t7     = DNN.Relu(d_c1, d_r1)
17 t8     = DNN.Dealloc(d_c1)
18 w2     = onnx.Constant()
19 d_w2   = DNN.Malloc(2304)
20 t9     = DNN.Memcpy(d_w2, w2) {H2D}
21 b2     = onnx.Constant()
22 d_b2   = DNN.Malloc(32)
23 t10    = DNN.Memcpy(d_b2, b2) {H2D}
24 d_c2   = DNN.Malloc(512)
25 t11    = DNN.Conv(d_r1, d_w2, d_b2, d_c2)
26 t12    = DNN.Dealloc(d_w2)
27 t13    = DNN.Dealloc(d_b2)
28 d_a1   = DNN.Malloc(512)
29 t14    = DNN.Add(d_r1, d_c2, d_a1)      // skip
30 t15    = DNN.Dealloc(d_r1)              // freed after Add
31 t16    = DNN.Dealloc(d_c2)
32 d_r2   = DNN.Malloc(512)
33 t17    = DNN.Relu(d_a1, d_r2)
34 t18    = DNN.Dealloc(d_a1)
35 h_out  = DNN.Host-malloc(512)
36 t19    = DNN.Memcpy(h_out, d_r2) {D2H}
37 t20    = DNN.Dealloc(d_r2)
```

Figure 4.2: DNN dialect (IR 1) for the running example after memory operation insertion. Lines 1 to 17 handle the first Conv-ReLU, lines 18 to 28 the second Conv, lines 29 to 34 the skip connection and final ReLU, and lines 35 to 37 the output transfer. Tensor d_r1 is not deallocated until line 30 because the skip connection (line 29) requires it after the second convolution.

Table 4.1: Device-tensor liveness table after DNN dialect conversion. Begin and End columns indicate the line indices at which each device tensor is allocated and deallocated, respectively. All sizes assume FP32 (4 bytes per element).

Tensor	Begin	End	Size (B)
d_x	1	11	192
d_w1	4	12	864
d_b1	7	13	32
d_c1	9	16	512
d_r1	14	29	512
d_w2	18	25	2,304
d_b2	21	26	32
d_c2	23	30	512
d_a1	27	33	512
d_r2	31	36	512

tion H2D (host-to-device) is inserted for parameter tensors (weights, biases) and input data to copy them from host to GPU memory. Fourth, a `DNN.Host-malloc` is inserted to allocate a host-side buffer for the final output, followed by a `DNN.Memcpy` with direction D2H (device-to-host) to return the inference result. All DNN operations produce temporary result variables (`t0`, `t1`, ...), and all intermediate computation results remain on the GPU to reduce host-device communication overhead.

Simultaneously, each ONNX dialect operation is converted to its DNN dialect counterpart. For example, `onnx.Conv` becomes `DNN.Conv`, `onnx.Relu` becomes `DNN.Relu`, and `onnx.Add` becomes `DNN.Add`.

Each DNN operation explicitly references device-resident tensors (prefixed with `d_`) rather than host-memory tensors. Figure 4.2 shows the resulting DNN dialect (denoted IR 1) for the running example. Lines 1 to 17 handle the first convolution-ReLU layer. Lines 18 to 28 handle the second convolution, and lines 29 to 34 implement the skip

connection and final ReLU. Notably, `d_r1` is not deallocated after the second convolution at line 25, because the skip connection requires it for the `DNN.Add` at line 29. It is only freed at line 30 after both consumers have completed. The final output is allocated on the host at line 35 and transferred at line 36. The liveness information is updated to reflect GPU tensor lifetimes, producing the updated device-tensor liveness table (Table 4.1).

Progressive Tensor Extension. Autoregressive language models introduce a tensor class whose size changes during one inference request. The key-value cache stores one key tensor and one value tensor for each transformer layer, and each decoding step appends one position to these tensors. The computation graph for a single decoding step is static, but the storage required by the cache is proportional to the number of positions already generated. This property conflicts with a single static pool size. If the compiler reserves the maximum context length, short requests occupy memory that remains unused. If the compiler reserves only an expected context length, longer requests can exhaust the pool during decoding.

The PAGRUS compiler represents this class as a progressive tensor. A progressive tensor has a fixed element stride and a capacity-tier upper bound. The stride determines the byte distance between consecutive token positions, and the tier bound determines the maximum number of positions that can be addressed in a compiled pool layout. For a transformer layer with cache width d, a tier with capacity c reserves $c \times d$ elements for the key tensor and $c \times d$ elements for the value tensor. The compiler therefore keeps static addressability inside each tier while avoiding a single allocation sized for the largest possible sequence.

The tiered layout preserves the address structure of the rest of the model. First, the compiler selects an ordered set of token capacities, such as 128, 256, 512, and 1024 tokens. Second, it computes the byte extent of each progressive tensor under each capacity. Third, it builds a pool layout for each tier in which non-progressive tensors keep stable offsets whenever their liveness and size are unchanged. Fourth, it emits tier metadata that records the progressive tensor base offset, stride, current logical length, and tier capacity. The generated code can then derive the address of position i by adding i strides to the progressive tensor base address, provided that i is smaller than the current tier capacity.

Runtime execution uses the tier metadata to advance the cache without changing the allocation discipline. During decoding, the runtime writes each newly produced key tensor and value tensor into the next logical position of the current progressive tensor. When the logical length reaches the capacity of the current tier, the runtime promotes the request to the next tier. Promotion preserves the live prefix of the cache, instantiates the larger tier layout, and resumes decoding with the same logical token order. Only live cache entries are carried across the promotion boundary, while tensors whose liveness has ended remain absent from the next tier. This rule keeps the method compatible with static liveness analysis because every tier has a fixed worst-case pool size, and every position written within a tier has a deterministic address.

The peak memory graph incorporates the tier upper bound rather than the instantaneous cache length for models containing progressive tensors (Section 4.3.1). This conservative representation lets the scheduler evaluate `Tenant` admission with the worst-case memory demand of the active tier. At the same time, tier promotion prevents the

compiler from committing every request to the largest context length at launch time. The resulting method supports LLM decoding in a static pool size scheme by combining fixed-offset access inside a tier with bounded promotion across tiers.

4.1.2 Hidden Backend Allocation Tracking

The visible liveness analysis described in Section 4.1.1 captures only the tensors that appear explicitly in the ONNX computation graph. However, GPU backend libraries such as cuDNN and cuBLAS frequently allocate additional temporary buffers at runtime to execute their kernels. These hidden allocations are invisible in the graph-level IR because they are created and destroyed entirely within a single backend API call. If these allocations are ignored, the compiler underestimates the true memory demand, and even a brief exceedance of the GPU memory capacity can trigger an immediate out-of-memory error.

Algorithm-Dependent Variability. The size of hidden workspace buffers varies dramatically depending on the algorithm selected by the backend library. Table 4.2 illustrates this variability for representative cuDNN convolution algorithms under different input configurations. For a $1 \times 64 \times 224 \times 224$ input with a 3×3 filter, the implicit precomputed GEMM (IPG) algorithm requires 24.64 MB of workspace, while the FFT-based algorithm (F) requires over 2,096 MB. The Winograd non-fused variant (WN) consumes 882.56 MB. Algorithm support and workspace sizes also change with the input shape and batch size. For a batch size of 8, the im2col GEMM workspace grows to 882 MB, and the FFT workspace increases to 2,322 MB. Because the backend library selects algo-

Table 4.2: Representative workspace sizes (MB) of cuDNN convolution algorithms for 32-bit floating-point data, measured with cuDNN 9.14 on an NVIDIA RTX 4090. (Each algorithm abbreviation represents IPG = implicit precomputed GEMM, G = im2col GEMM, F = FFT, FT = FFT tiling, W = Winograd, WN = non-fused Winograd, respectively.)

Input Shape				Filter		Pad		Algorithm	Support	WS (MB)
N	C	H	W	H	W	H	W			
1	64	224	224	3	3	1	1	IPG	YES	24.64
1	64	224	224	3	3	1	1	G	YES	110.25
1	64	224	224	3	3	1	1	F	YES	2,096.39
1	64	224	224	3	3	1	1	FT	YES	18.06
1	64	224	224	3	3	1	1	W	YES	0.39
1	64	224	224	3	3	1	1	WN	YES	882.56
1	64	224	224	5	5	2	2	IPG	YES	24.89
1	64	224	224	5	5	2	2	G	YES	306.25
1	64	224	224	5	5	2	2	F	YES	2,096.64
1	64	224	224	5	5	2	2	FT	YES	18.06
1	64	224	224	5	5	2	2	W	NO	—
1	64	224	224	5	5	2	2	WN	YES	827.84
8	64	224	224	3	3	1	1	IPG	YES	0.29
8	64	224	224	3	3	1	1	G	YES	882.00
8	64	224	224	3	3	1	1	F	YES	2,322.14
8	64	224	224	3	3	1	1	FT	YES	25.50
8	64	224	224	3	3	1	1	W	YES	0.39
8	64	224	224	3	3	1	1	WN	YES	882.56

rithms dynamically at runtime based on operator arguments (input dimensions, filter size, data type), predicting the workspace size requires knowledge of these arguments at compile time.

Profiling-Based Workspace Estimation. The PAGRUS compiler determines hidden allocation sizes through two complementary methods. First, when backend APIs explicitly expose workspace query functions, the compiler directly invokes them to obtain the exact workspace requirement for each algorithm and tensor configuration. For example, cuDNN provides `cudnnGetConvolutionForwardWorkspaceSize`, which returns the workspace size for a given convolution descriptor, algorithm choice, and ten-

sor dimensions. The compiler calls this function for every convolution operator in the model, using the statically known tensor shapes from the ONNX dialect.

Second, for operators or kernels that do not expose workspace query interfaces, the compiler employs a profiling-based approach. A dummy tensor matching the operator's argument shapes is constructed, and the operator is executed on the GPU while monitoring actual memory consumption. The difference between the observed peak allocation and the sum of input, output, and parameter tensor sizes yields the hidden workspace size. This profiling-based inference captures workspace behaviors that are not documented or queryable through public APIs.

Integration into the Liveness Table. Each identified hidden allocation is modeled as an ephemeral tensor with a liveness interval equal to the single operator that triggers it. For instance, if line 11 of the running example (`DNN.Conv` in Figure 4.2) requires a 256-byte cuDNN workspace, an ephemeral tensor entry is inserted immediately before and after the convolution operator. Figure 4.3 shows the resulting DNN dialect (denoted IR 2), which extends IR 1 by inserting workspace allocation and deallocation operations around each convolution. The workspace tensor `d_ws1` is allocated at line 10 and freed at line 12, and `d_ws2` is allocated at line 26 and freed at line 28. Table 4.3 shows the unified liveness table for IR 2, which includes all visible and hidden tensor entries. This unified table serves as the input to the peak memory graph construction (Section 4.3.1), which is performed after operation-level optimization.

```
0  x      = INPUT
1  d_x    = DNN.Malloc(192)
2  t0     = DNN.Memcpy(d_x, x) {H2D}
3  w1     = onnx.Constant()
4  d_w1   = DNN.Malloc(864)
5  t1     = DNN.Memcpy(d_w1, w1) {H2D}
6  b1     = onnx.Constant()
7  d_b1   = DNN.Malloc(32)
8  t2     = DNN.Memcpy(d_b1, b1) {H2D}
9  d_c1   = DNN.Malloc(512)
10 d_ws1  = DNN.Malloc(256)                // cuDNN workspace
11 t3     = DNN.Conv(d_x, d_w1, d_b1, d_c1, d_ws1)
12 t3a    = DNN.Dealloc(d_ws1)
13 t4     = DNN.Dealloc(d_x)
14 t5     = DNN.Dealloc(d_w1)
15 t6     = DNN.Dealloc(d_b1)
16 d_r1   = DNN.Malloc(512)
17 t7     = DNN.Relu(d_c1, d_r1)
18 t8     = DNN.Dealloc(d_c1)
19 w2     = onnx.Constant()
20 d_w2   = DNN.Malloc(2304)
21 t9     = DNN.Memcpy(d_w2, w2) {H2D}
22 b2     = onnx.Constant()
23 d_b2   = DNN.Malloc(32)
24 t10    = DNN.Memcpy(d_b2, b2) {H2D}
25 d_c2   = DNN.Malloc(512)
26 d_ws2  = DNN.Malloc(256)                // cuDNN workspace
27 t11    = DNN.Conv(d_r1, d_w2, d_b2, d_c2, d_ws2)
28 t11a   = DNN.Dealloc(d_ws2)
29 t12    = DNN.Dealloc(d_w2)
30 t13    = DNN.Dealloc(d_b2)
31 d_a1   = DNN.Malloc(512)
32 t14    = DNN.Add(d_r1, d_c2, d_a1)      // skip
33 t15    = DNN.Dealloc(d_r1)              // freed after Add
34 t16    = DNN.Dealloc(d_c2)
35 d_r2   = DNN.Malloc(512)
36 t17    = DNN.Relu(d_a1, d_r2)
37 t18    = DNN.Dealloc(d_a1)
38 h_out  = DNN.Host-malloc(512)
39 t19    = DNN.Memcpy(h_out, d_r2) {D2H}
40 t20    = DNN.Dealloc(d_r2)
```

Figure 4.3: DNN dialect (IR 2) for the running example after hidden backend allocation tracking. `d_ws1` (lines 10 to 12) supports the first convolution, and `d_ws2` (lines 26 to 28) supports the second.

Table 4.3: Unified liveness table for the running example, derived from Figure 4.3. Shaded rows indicate ephemeral workspace tensors. Begin and End columns reference IR 2 line indices.

Tensor	Begin	End	Size (B)
d_x	1	13	192
d_w1	4	14	864
d_b1	7	15	32
d_c1	9	18	512
d_ws1	10	12	256
d_r1	16	33	512
d_w2	20	29	2,304
d_b2	23	30	32
d_c2	25	34	512
d_ws2	26	28	256
d_a1	31	37	512
d_r2	35	40	512

4.2 Operation-Level Optimization

The second stage of the PAGRUS compiler applies domain-specific optimizations to the DNN dialect after tensor memory profiling. Two complementary techniques are employed. Layer fusion (Section 4.2.1) merges consecutive operators into a single fused operator, eliminating intermediate tensors entirely. Tensor coalescing (Section 4.2.2) enables elementwise operators to reuse input memory for their output, avoiding a separate allocation. After both passes complete, the compiler updates the unified liveness table to reflect the reduced tensor set and shortened lifetimes.

4.2.1 Layer Fusion

Layer fusion investigates consecutive DNN operators and substitutes matched patterns with a single fused operator. When two operators are fused, the intermediate ten-

sor that previously connected them is eliminated, removing one DNN.Malloc and one DNN.Dealloc from the IR. For example, a DNN.Conv followed by a DNN.Relu can be fused into a single DNN.Conv-relu operator. The convolution kernel writes its output through the activation function directly, and the intermediate activation tensor is never materialized in GPU memory. In the running example, this fusion is applied to the first convolution-ReLU pair. The DNN.Conv and DNN.Relu from IR 2 (Figure 4.3) are replaced by a single DNN.Conv-relu at line 11 in Figure 4.4, eliminating the intermediate tensor d_c1 and its associated allocation and deallocation.

The PAGRUS compiler supports the fusion patterns listed in Table 4.4.

Table 4.4: Layer fusion patterns supported by the PAGRUS compiler. Each pattern of consecutive operators is replaced by a single fused operator, eliminating the intermediate tensors that connect them.

Component operators	Fused operator
DNN.Conv + DNN.Relu	DNN.Conv-relu
DNN.Conv + DNN.BatchNorm + DNN.Relu	DNN.Conv-bn-relu
DNN.Conv + DNN.Add	DNN.Conv-add

An applicability constraint governs every fusion. The intermediate tensor must not be live-out at the fused operator, meaning no other operator in the program uses the intermediate tensor after the fusion point. If the intermediate tensor appears as an operand of a downstream operator, the fusion is not applied.

4.2.2 Tensor Coalescing

Elementwise operations such as DNN.Add, DNN.Relu, DNN.Sigmoid, and DNN.Mul process each tensor element independently. On a GPU, each CUDA thread accesses exactly

one element, performs the computation, and writes the result back to the same position. Because no inter-element dependencies exist, the input memory buffer can safely serve as the output buffer. The PAGRUS compiler exploits this property by coalescing the input and output tensors of eligible elementwise operators, allowing them to share the same memory space. For example, the `DNN.Add(d_r1, d_c2, d_a1)` in IR 2 is rewritten as `DNN.Add(d_r1, d_c2, d_c2)` (line 28 in Figure 4.4), where the output overwrites `d_c2` in place. This transformation eliminates the allocation of `d_a1` entirely, reducing both peak memory demand and the number of memory management operations. As with layer fusion, the overwritten input tensor must not be live-out at the operator, meaning no subsequent operator reads from that tensor. Similarly, the `DNN.Relu` at line 30 overwrites `d_c2` in place, eliminating `d_r2`. Figure 4.4 shows the complete optimized IR for the running example. Compared to IR 2 (Figure 4.3), three tensors (`d_c1`, `d_a1`, `d_r2`) and six memory management operations are removed, reducing the device tensor count from twelve to nine.

4.3 Peak-Memory-Aware Model Partitioning

For multi-tenant GPU execution, a monolithic DNN model must be divided into independently schedulable units. Model-level scheduling treats the entire model as a single unit, resulting in a high peak-to-average memory ratio (PAMR) because the GPU reserves memory for the worst-case peak demand throughout the entire inference. Layer-level scheduling treats each operator as a separate scheduling unit, achieving fine-grained memory control but incurring prohibitive synchronization overhead (e.g., 176 signal exchanges per inference for ResNet50).

```
0  x      = INPUT
1  d_x    = DNN.Malloc(192)
2  t0     = DNN.Memcpy(d_x, x) {H2D}
3  w1     = onnx.Constant()
4  d_w1   = DNN.Malloc(864)
5  t1     = DNN.Memcpy(d_w1, w1) {H2D}
6  b1     = onnx.Constant()
7  d_b1   = DNN.Malloc(32)
8  t2     = DNN.Memcpy(d_b1, b1) {H2D}
9  d_r1   = DNN.Malloc(512)                  // fused output (d_c1)
10 d_ws1  = DNN.Malloc(256)                  // cuDNN workspace
11 t3     = DNN.Conv-relu(d_x, d_w1, d_b1, d_r1, d_ws1) // fused
12 t3a    = DNN.Dealloc(d_ws1)
13 t4     = DNN.Dealloc(d_x)
14 t5     = DNN.Dealloc(d_w1)
15 t6     = DNN.Dealloc(d_b1)
16 w2     = onnx.Constant()
17 d_w2   = DNN.Malloc(2304)
18 t9     = DNN.Memcpy(d_w2, w2) {H2D}
19 b2     = onnx.Constant()
20 d_b2   = DNN.Malloc(32)
21 t10    = DNN.Memcpy(d_b2, b2) {H2D}
22 d_c2   = DNN.Malloc(512)
23 d_ws2  = DNN.Malloc(256)                  // cuDNN workspace
24 t11    = DNN.Conv(d_r1, d_w2, d_b2, d_c2, d_ws2)
25 t11a   = DNN.Dealloc(d_ws2)
26 t12    = DNN.Dealloc(d_w2)
27 t13    = DNN.Dealloc(d_b2)
28 t14    = DNN.Add(d_r1, d_c2, d_c2)       // coalesced (d_a1)
29 t15    = DNN.Dealloc(d_r1)
30 t17    = DNN.Relu(d_c2, d_c2)            // coalesced (d_r2)
31 h_out  = DNN.Host-malloc(512)
32 t19    = DNN.Memcpy(h_out, d_c2) {D2H}
33 t20    = DNN.Dealloc(d_c2)
```

Figure 4.4: DNN dialect (IR 3) for the running example after operation-level optimization. Layer fusion replaces the `DNN.Conv` and `DNN.Relu` pair with a single `DNN.Conv-relu` at line 11, eliminating the intermediate tensor `d_c1` and its allocation. Tensor coalescing rewrites `DNN.Add` at line 28 to overwrite `d_c2` in place, eliminating `d_a1`, and rewrites `DNN.Relu` at line 30 to overwrite `d_c2` similarly, eliminating `d_r2`. Compared to IR 2 (Figure 4.3), three tensors and six memory operations are removed.

The PAGRUS compiler adopts an intermediate granularity called `Task`-level scheduling. A `Task` groups consecutive operators that exhibit similar memory usage patterns, producing scheduling units that balance memory efficiency against scheduling overhead. The partition boundaries are determined by the peak memory graph constructed in Section 4.3.1, and the partitioning process consists of two phases. First, the Tailor algorithm (Section 4.3.2) identifies points of significant memory usage change in the peak memory graph and uses them as initial partition boundaries, producing fine-grained units called `Tasklets`. Second, the Stitch algorithm (Section 4.3.3) merges excessively short `Tasklets` with their neighbors based on estimated execution time, producing the final `Task` set. For single-tenant inference, this section produces a single `Task` spanning the entire model, and the partitioning step is effectively a no-op.

4.3.1 Peak Memory Graph Construction

After operation-level optimization completes, the unified liveness table reflects the final set of tensors and their lifetimes. The next step is to convert this table into a peak memory graph, a time-series representation of the total GPU memory demand at each point during model execution. This graph serves as the primary input for model partitioning and memory pool sizing (Section 4.4).

Best-Fit Memory Allocation Simulation. To construct the peak memory graph, the PAGRUS compiler simulates the execution timeline using a best-fit memory allocation policy. The simulator processes the unified liveness table in operator-index order. At each operator's definition point, the corresponding tensor's byte size is added to the cur-

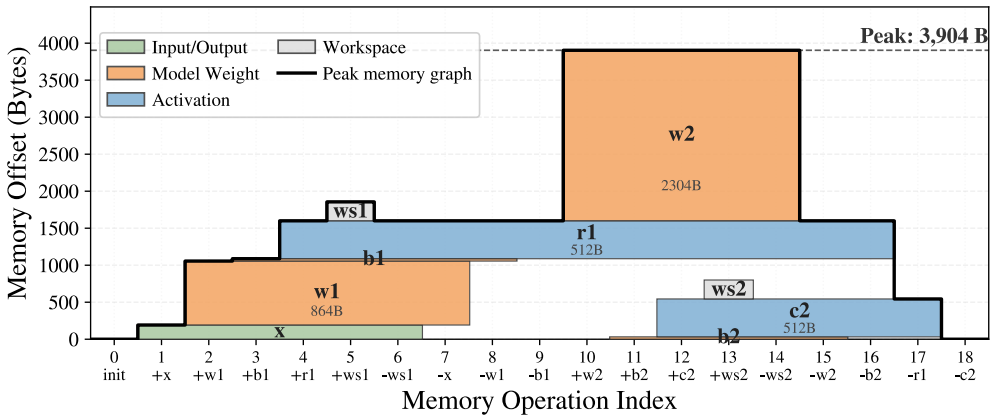


Figure 4.5: Best-fit memory allocation plan and peak memory graph for the running example. The x-axis enumerates only allocation and deallocation events (memory operation index), abstracting away non-memory lines. Each colored rectangle represents a device tensor, positioned vertically at its allocated memory offset and spanning horizontally across its lifetime. The bold black step-line traces the peak memory graph, the total GPU memory in use at each memory operation index. The dashed horizontal line marks the overall peak of 3,904 B, which occurs when the Conv1-relu fused output tensor `d_r1` (at offset 1,088) coexists with the second convolution's weight tensor `d_w2` (at offset 1,600, 2,304 B). The remaining tensors (`d_b2`, `d_c2`, `d_ws2`) fit within the memory gap below `d_r1`.

rent memory usage. At each operator's last-use point, the tensor's byte size is subtracted. When a new tensor must be allocated, the simulator searches for the smallest previously freed block that is large enough to hold the tensor, and reuses that block if one exists. This best-fit strategy mirrors the behavior of practical GPU memory allocators and produces a realistic estimate of peak memory consumption.

The output of the simulation is an array indexed by operator index, where each entry records the maximum GPU memory in use at that point. Figure 4.5 illustrates the resulting memory allocation plan for the running example. The x-axis enumerates only allocation and deallocation events (memory operation indices), abstracting away non-

memory lines so that memory state transitions are clearly visible. Each tensor is drawn as a colored rectangle whose vertical position indicates its offset in the GPU memory address space and whose horizontal span indicates its lifetime. The bold black step-line overlaid on the figure traces the peak memory graph, showing the total GPU memory in use at each memory operation index. The peak memory usage of 3,904 B occurs when the Conv1-relu fused output tensor `d_r1` (at offset 1,088, 512 B) coexists with the second convolution's weight tensor `d_w2` (at offset 1,600, 2,304 B). Because layer fusion causes `d_r1` to be allocated while the first convolution's inputs are still live, it occupies a higher offset than in the unoptimized plan. The remaining tensors `d_b2`, `d_c2`, and `d_ws2` fit within the gap at $[0, 1{,}088)$.

Moving Maximum Filter. In large-scale models containing hundreds or thousands of operators, the raw peak memory graph exhibits high-frequency fluctuations as tensors are rapidly allocated and deallocated. These local variations obscure the structural memory patterns that are relevant for partitioning. To mitigate this noise, the PAGRUS compiler applies a moving maximum filter to the peak memory graph. The filter slides a window of size $w = \lceil |peakGraph| \,/\, winNum \rceil$ across the graph, replacing each entry with the maximum value within the window. The parameter *winNum* is empirically set to 20 based on experimental evaluation. If *winNum* is too large (small window), the filter cannot suppress jitter. If *winNum* is too small (large window), the filter over-smooths the graph and obscures genuine memory usage transitions. The filtered graph retains the major peaks and valleys that correspond to meaningful memory demand changes across model layers.

Algorithm 4.2: Tailor model partitioning algorithm

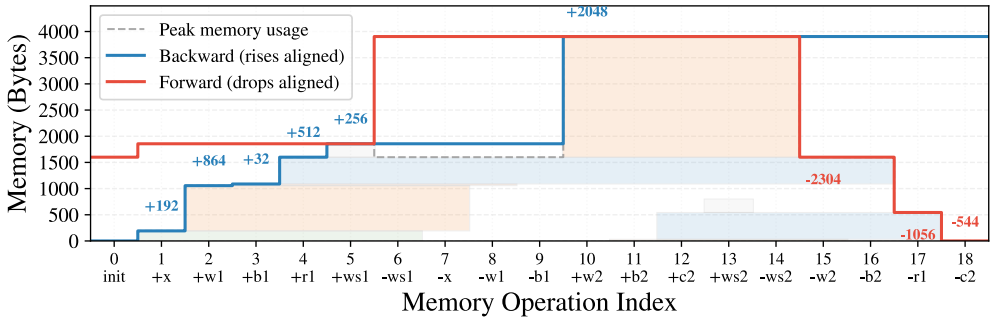
```
   Input: peakGraph(): Array of peak memory values
   Output: partIndices(): Array of partition point indices
 1 Function TailorPartition (peakGraph):
 2     winSize ← len (peakGraph) / winNum
 3     alterMap (mallocIdx, diffVal) ← diff (movingMax (peakGraph, winSize))
 4     partIndices ← [ ]
 5     maxClusters ← KMeans (alterMap.diffVals).selectTopTwo
 6     foreach cluster ∈ maxClusters do
 7         foreach value ∈ cluster do
 8             Idxs ← alterMap.findIndices (value)
 9             partIndices.append (Idxs)
10         end
11     end
12     return partIndices
13 end
```

For models containing progressive tensors (Section 4.1.1), the peak memory graph incorporates the capacity-tier upper bound rather than the actual runtime size. This ensures that the graph represents the worst-case memory demand for the current tier, enabling the scheduler to make conservative admission decisions.

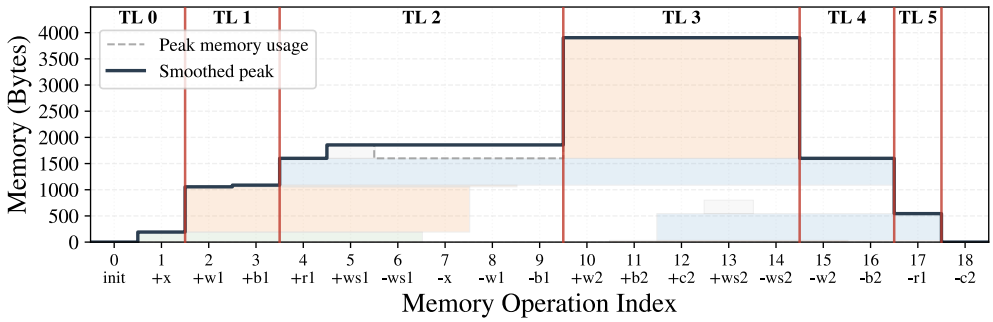
4.3.2 The Tailor Algorithm

The Tailor algorithm identifies natural partition boundaries in the peak memory graph by detecting points where memory usage changes most drastically. The intuition is that a sharp increase or decrease in memory demand indicates a transition between model regions with fundamentally different memory profiles, making such points effective partition boundaries.

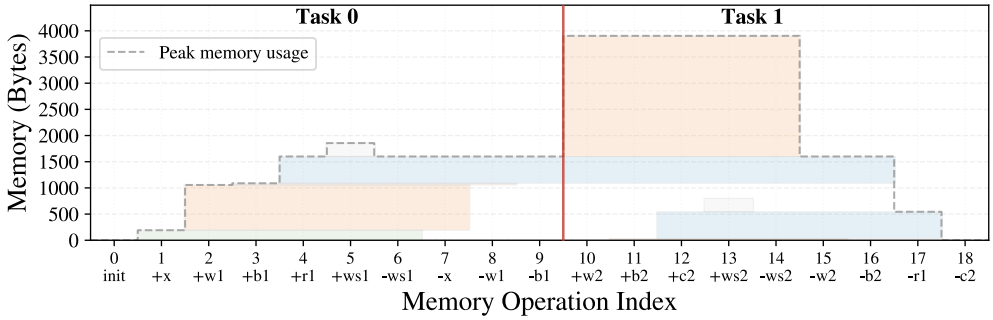
Algorithm 4.2 presents the algorithm. The input is the filtered peak memory graph



(a) Moving maximum filter applied to the raw peak memory graph. Blue and red lines show backward-looking and forward-looking maxima, respectively. Annotated values indicate the first-order difference magnitudes at each transition.



(b) Tailor partitioning. The smoothed peak (dark line) retains structural transitions from the filtered graph. Red vertical lines mark detected boundaries, producing six `Tasklets` (TL 0–TL 5).



(c) Stitch merging. The red vertical line marks the final `Task` boundary after short `Tasklets` are merged into two `Tasks`.

Figure 4.6: Tailor-and-Stitch partitioning pipeline applied to the running example.

produced by the moving maximum filter described in Section 4.3.1. Figure 4.6a illustrates this filtering on the running example, where the backward-looking maximum (blue) preserves rising transitions and the forward-looking maximum (red) preserves falling transitions. The element-wise maximum of these two directional filters produces the smoothed peak memory graph shown in Figure 4.6b, which retains structural memory transitions while suppressing transient fluctuations. The algorithm proceeds in three steps. First, it computes the first-order differences of the filtered graph, producing a sequence of values that capture the magnitude and direction of memory usage changes between consecutive operator indices. Second, it applies KMeans clustering to these difference values. The number of clusters is determined automatically by combining the silhouette score and the elbow method [99], where the silhouette score measures how well each data point fits within its assigned cluster relative to neighboring clusters and the elbow method locates the point of diminishing return in within-cluster variance. Third, the algorithm selects the top two clusters containing the differences with the largest magnitudes. The operator indices corresponding to these extreme differences are extracted as `Tasklet` boundaries. Figure 4.6b shows the resulting `Tasklets` for the running example, where six boundaries partition the peak memory graph into six `Tasklets` (TL 0 through TL 5). The sensitivity of `winNum` to partitioning quality is evaluated in Section 6.3.1.

4.3.3 The Stitch Algorithm

The Tailor algorithm may produce `Tasklets` that are too short in execution time, resulting in excessive scheduling overhead if each `Tasklet` were treated as an independent

Task. The Stitch algorithm addresses this by merging short Tasklets with their neighbors to produce the final Task set.

To estimate the execution time of each Tasklet, the PAGRUS compiler uses a layer-wise profiling lookup table. This table is populated ahead of time by profiling representative operators with varying input sizes and layer parameters (e.g., input channels, output channels, kernel size). During compilation, the Stitch algorithm queries the closest matching entry in the table for each operator in a Tasklet and sums the individual estimates to obtain the total estimated execution time of the Tasklet.

The merging threshold is empirically set to one-tenth of the total estimated model execution time. Tasklets whose estimated execution time falls below this threshold are merged with their neighboring Tasklet that has the most similar memory usage profile. Figure 4.6c shows the final result for the running example, where the six Tasklets are merged into two Tasks. The sensitivity of the merging threshold to the final Task count is evaluated in Section 6.3.1.

Once the final Task boundaries are determined, the compiler inserts partitioning markers (DNN.Model-partition) into the IR at the corresponding program points. Figure 4.7 shows the resulting IR for the running example, where the marker at line 17 separates the program into two Tasks. The partition introduces a critical distinction among tensors. Tensor d_r1, defined before the boundary (line 10) and consumed after it via the skip connection (line 33), must persist across the Task boundary and is therefore classified as a shared tensor. All other tensors have lifetimes contained within a single Task and are classified as intra-Task. This includes the ephemeral cuDNN workspace tensors d_ws1 and d_ws2, each confined to a single convolution, which are therefore

placed in the intra-Task pool of the Task that contains them. This classification drives the subsequent memory pool generation (Section 4.4), which assigns shared and intra-Task tensors to separate pools. At this stage, the DNN.Malloc and DNN.Dealloc operations remain unchanged.

4.4 Memory Pool Generation

After model partitioning, the PAGRUS compiler classifies every tensor as either intra-Task or shared. A tensor is classified as shared if a Task boundary exists between its definition point and its last-use point, meaning the tensor must remain accessible across consecutive Tasks. A tensor whose entire lifetime is contained within a single Task is classified as intra-Task. Although shared tensors account for only a small fraction of the total tensor count (typically 3 to 13 per model), their cumulative size can reach several gigabytes in large-scale models, motivating a dedicated GPU-resident memory pool separate from the intra-Task pools.

Because the partitioning step alters the grouping of operators, the liveness information from the profiling stage is no longer directly applicable. The compiler therefore re-analyzes the liveness of all tensors within each Task using the same method described in Section 4.1, producing per-Task liveness tables.

4.4.1 Optimal Intra-Task Pool Layout via CP-SAT

For each Task, the compiler must assign every intra-Task tensor a non-overlapping byte offset within a contiguous memory pool. This is an instance of the Dynamic Storage Allocation (DSA) problem, which is NP-hard in the general case (Section 1.1.1). The goal

```
0  // ====== Task 0: Conv1-relu ======
1  x      = INPUT
2  d_x    = DNN.Malloc(192)
3  t0     = DNN.Memcpy(d_x, x) {H2D}
4  w1     = onnx.Constant()
5  d_w1   = DNN.Malloc(864)
6  t1     = DNN.Memcpy(d_w1, w1) {H2D}
7  b1     = onnx.Constant()
8  d_b1   = DNN.Malloc(32)
9  t2     = DNN.Memcpy(d_b1, b1) {H2D}
10 d_r1   = DNN.Malloc(512)           // shared tensor (defined)
11 d_ws1  = DNN.Malloc(256)           // cuDNN workspace
12 t3     = DNN.Conv-relu(d_x, d_w1, d_b1, d_r1, d_ws1)
13 t3a    = DNN.Dealloc(d_ws1)
14 t4     = DNN.Dealloc(d_x)
15 t5     = DNN.Dealloc(d_w1)
16 t6     = DNN.Dealloc(d_b1)
17 ──────────────────────────────────────────────────────────

18 DNN.Model-partition()              // Task boundary
19 ──────────────────────────────────────────────────────────
20 // ====== Task 1: Conv2 + Skip + ReLU ======
21 w2     = onnx.Constant()
22 d_w2   = DNN.Malloc(2304)
23 t9     = DNN.Memcpy(d_w2, w2) {H2D}
24 b2     = onnx.Constant()
25 d_b2   = DNN.Malloc(32)
26 t10    = DNN.Memcpy(d_b2, b2) {H2D}
27 d_c2   = DNN.Malloc(512)
28 d_ws2  = DNN.Malloc(256)           // cuDNN workspace
29 t11    = DNN.Conv(d_r1, d_w2, d_b2, d_c2, d_ws2)
30 t11a   = DNN.Dealloc(d_ws2)
31 t12    = DNN.Dealloc(d_w2)
32 t13    = DNN.Dealloc(d_b2)
33 t14    = DNN.Add(d_r1, d_c2, d_c2)  // skip connection
34 t15    = DNN.Dealloc(d_r1)          // shared tensor freed
35 t17    = DNN.Relu(d_c2, d_c2)
36 ...
```

Figure 4.7: DNN dialect (IR 4) for the running example after model partitioning. The `DNN.Model-partition` marker at line 17 splits the program into two `Tasks`. Tensor `d_r1`, defined at line 10 and consumed at line 33, crosses the `Task` boundary and is classified as a shared tensor. All other tensors have lifetimes contained within a single `Task`.

Algorithm 4.3: Optimal intra-`Task` pool layout via CP-SAT

Input: $\mathcal{T} = \{(s_i, e_i, sz_i)\}_{i=1}^{n}$: tensors with liveness interval $[s_i, e_i)$ and byte size sz_i
Output: $\{o_i\}_{i=1}^{n}$: byte offset of each tensor; *peak*: optimal pool size

// Step 1: Compute lower bound via sweep-line
$LB \leftarrow \max_t \sum_{i \in \text{live}(t)} sz_i$ // $O(n \log n)$
// Step 2: Enumerate conflict pairs
$\mathscr{C} \leftarrow \{(i,j) \mid [s_i, e_i) \cap [s_j, e_j) \neq \emptyset\}$ // sweep-line with active set
// Step 3: Compute upper bound via first-fit heuristic
$UB \leftarrow \textsc{FirstFit}(\mathcal{T})$
// Step 4: Build CP-SAT model
foreach $i \in [1, n]$ **do**
  $o_i \leftarrow \textsc{IntVar}(0, UB)$ // byte offset variable
end
$peak \leftarrow \textsc{IntVar}(LB,\ UB)$
foreach $i \in [1, n]$ **do**
  $o_i + sz_i \leq peak$ // containment constraint
end
foreach $(i,j) \in \mathscr{C}$ **do**
  $b_{ij} \leftarrow \textsc{BoolVar}$
  $b_{ij} = 1 \implies o_i + sz_i \leq o_j$ // disjunctive
  $b_{ij} = 0 \implies o_j + sz_j \leq o_i$ // non-overlap
end
$o_{\arg\max_i sz_i} \leftarrow 0$ // symmetry breaking
$\textsc{Minimize}(peak)$
// Step 5: Solve
$\{o_i\}, peak \leftarrow \textsc{Solve}(\text{workers}=8)$
return $\{o_i\}, peak$

is to minimize the total pool size, or equivalently, the peak of the offset-plus-size values across all tensors. Algorithm 4.3 presents the complete formulation.

The algorithm takes as input the per-`Task` liveness table, where each tensor i is characterized by its liveness interval $[s_i, e_i)$ and byte size sz_i.

Lower Bound and Conflict Graph. The algorithm first computes a tight lower bound LB on the minimum pool size (Line 1–Line 2). A sweep-line traverses the operator timeline, maintaining a running sum of the sizes of all tensors simultaneously live at each point. The lower bound is the maximum of this running sum, computable in $O(n \log n)$ time. If the solver later achieves a pool size equal to LB, the solution is a provably perfect packing with zero fragmentation.

The algorithm then constructs a conflict graph (Line 3–Line 4). Two tensors (i,j) conflict if their liveness intervals overlap, meaning they must occupy disjoint memory regions. A second sweep-line with an active set enumerates all such pairs in $O(n \cdot \bar{k})$ time, where $\bar{k}$ is the average number of simultaneously live tensors.

CP-SAT Formulation. The PAGRUS compiler formulates the DSA problem as a CP-SAT (Constraint Programming with Satisfiability) model (Line 7–Line 21). For each tensor i, an integer variable $o_i \in [0, UB]$ represents its byte offset within the pool. An auxiliary variable $peak \in [LB, UB]$ represents the total pool size, where UB is initialized from a first-fit heuristic to tighten the search space.

Three types of constraints define the feasible region. First, a containment constraint (Line 13) ensures that every tensor fits within the pool by requiring $o_i + sz_i \leq peak$.

Second, for each conflict pair $(i,j) \in \mathscr{C}$, a disjunctive non-overlap constraint (Line 17–Line 18) ensures that the two tensors occupy non-overlapping address ranges. A Boolean indicator b_{ij} selects whether tensor i is placed below tensor j or vice versa. Third, a symmetry-breaking constraint (Line 20) pins the largest tensor to offset 0, eliminating solutions that differ only by a constant shift, which reduces the search space by more than half. The objective is to minimize *peak*. The solver is configured with 8 parallel workers (Line 23) and typically finds the provably optimal solution in under 5 seconds for up to approximately 1,500 tensors. When the solver returns an OPTIMAL status, the resulting pool size is the minimum achievable for the given tensor set. When $OPT = LB$, the packing is perfect, and when $OPT > LB$, the gap reflects structurally unavoidable fragmentation from conflicting lifetime patterns.

4.4.2 Shared Tensor Pool and Memory Pool Code Generation

Shared Tensor Pool. The PAGRUS compiler constructs a dedicated GPU-resident memory pool for shared tensors. Because the number of shared tensors is small (as noted in Section 4.4), a best-fit layout is sufficient and optimal pool generation via CP-SAT is not necessary. The shared pool is allocated before the first `Task` begins execution and deallocated after the final `Task` completes. During execution, subsequent `Tasks` access shared tensors via offset pointers into this pool without any host-to-device transfers. This zero-move sharing eliminates the data migration overhead that would otherwise be incurred at every `Task` boundary, where intermediate tensors would need to be backed up to host memory and restored to device memory. In the running example, tensor `d_r1` is the only shared tensor, and it is placed in a dedicated 512-byte shared pool (line 10 in

Figure 4.8) that persists across the Task boundary for consumption by Task 1.

Memory Pool Code Generation. After both the intra-Task and shared pools are sized and laid out, the compiler performs a final IR transformation. All DNN.Malloc operations are substituted with DNN.Mem-offset operations that reference the pool base address, the tensor's assigned offset, and its size. A DNN.Pool-init operation is inserted at program start (for single-tenant) or at each Task start (for multi-tenant intra-Task pools) to create the pool with the predetermined size. All DNN.Dealloc operations are eliminated, because individual tensor deallocation is unnecessary when the entire pool is deallocated at once. The resulting IR performs exactly one pool initialization per Task, replacing hundreds of individual allocation and deallocation operations. Figure 4.8 shows the complete transformation for the running example. The intra-Task pool of 1,344 bytes for Task 0 is created by a single DNN.Pool-init (line 1), and every tensor, including the ephemeral cuDNN workspace d_ws1, receives a compile-time constant offset via DNN.Mem-offset. All individual DNN.Dealloc operations from IR 4 are eliminated, and only pool-level deallocation remains (lines 13 and 31). In Task 1, tensor coalescing eliminates d_a1 and d_r2 entirely, as DNN.Add and DNN.Relu write their outputs into d_c2 in place, reducing the number of tensors requiring pool placement.

4.5 Scheduler Interface Code Generation

Whereas memory pool code generation (Section 4.4) produces the self-contained memory access code that each compiled binary executes on its own, this final compiler pass generates the scheduler interface, the code through which a compiled Tenant commu-

```
0  // ====== Task 0 (intra-Task pool: 1344 B) ======
1  pool_0 = DNN.Pool-init(1344)                 // single GPU
       allocation
2  d_x    = DNN.Mem-offset(pool_0, 1120, 192) // [1120, 1312)
3  t0     = DNN.Memcpy(d_x, x) {H2D}
4  w1     = onnx.Constant()
5  d_w1   = DNN.Mem-offset(pool_0, 0, 864)    // [0, 864)
6  t1     = DNN.Memcpy(d_w1, w1) {H2D}
7  d_b1   = DNN.Mem-offset(pool_0, 1312, 32)  // [1312, 1344)
8  d_ws1  = DNN.Mem-offset(pool_0, 864, 256)  // [864, 1120) cuDNN
       workspace
9  // ====== Shared tensor pool: 512 B ======
10 shared = DNN.Pool-init(512)
11 d_r1   = DNN.Mem-offset(shared, 0, 512)
12 t3     = DNN.Conv-relu(d_x, d_w1, d_b1, d_r1, d_ws1)
13 DNN.Dealloc(pool_0)                          // Task 0 pool freed
14 // d_r1 persists in shared pool -> Task 1
15 // ====== Task 1 (intra-Task pool: 3104 B) ======
16 pool_1 = DNN.Pool-init(3104)
17 w2     = onnx.Constant()
18 d_w2   = DNN.Mem-offset(pool_1, 0, 2304)   // [0, 2304)
19 t9     = DNN.Memcpy(d_w2, w2) {H2D}
20 b2     = onnx.Constant()
21 d_b2   = DNN.Mem-offset(pool_1, 3072, 32)  // [3072, 3104)
22 t10    = DNN.Memcpy(d_b2, b2) {H2D}
23 d_c2   = DNN.Mem-offset(pool_1, 2304, 512) // [2304, 2816)
24 d_ws2  = DNN.Mem-offset(pool_1, 2816, 256) // [2816, 3072)
       cuDNN workspace
25 t11    = DNN.Conv(d_r1, d_w2, d_b2, d_c2, d_ws2)
26 t14    = DNN.Add(d_r1, d_c2, d_c2)          // skip connection
27 DNN.Dealloc(shared)                         // shared pool freed
28 t17    = DNN.Relu(d_c2, d_c2)
29 h_out  = DNN.Host-malloc(512)
30 t19    = DNN.Memcpy(h_out, d_c2) {D2H}
31 DNN.Dealloc(pool_1)                         // Task 1 pool freed
```

Figure 4.8: DNN dialect (IR 5) for the running example after memory pool generation. All `DNN.Malloc` operations are replaced by `DNN.Pool-init` (lines 1 and 16) and `DNN.Mem-offset` with compile-time constant offsets. All individual `DNN.Dealloc` operations from IR 4 are eliminated, and only pool-level deallocation remains. The shared tensor `d_r1` resides in a separate pool (line 10) that persists across the `Task` boundary.

nicates with the PAGRUS scheduler at execution time. The pass is applied only for multi-tenant deployment, since a single-tenant binary runs to completion without any scheduler. The scheduler interface consists of two components: signal operations that notify the scheduler of state transitions, and a data-sharing interface that transmits structured memory usage information.

Signal Operation Insertion. The compiler inserts signal operations at four points in the generated code:

1. when a `Tenant` is initially issued (`ISSUE`, line 0 in Figure 4.9),
2. when a `Task` completes execution (`TASK_DONE`, lines 15 and 31),
3. before the shared tensor pool is allocated (`SHARED_ALLOC`, line 10), and
4. after the shared tensor pool is deallocated (`SHARED_DEALLOC`, line 26).

Each signal operation invokes a lightweight callback that notifies the scheduler of a memory usage state transition. The placement of these operations is determined statically from the `Task` boundaries and pool allocation points established in Sections 4.3 and 4.4. Figure 4.9 shows the final compiled IR for the running example, where six signal operations bracket the two `Tasks` and the shared pool lifetime. For single-tenant deployment, all signal operations are omitted, and the compiled binary runs without any scheduler involvement.

Task Information Analysis. Signals alone mark state changes without quantifying their memory impact. To address this, the compiler additionally generates a data-sharing

interface that allows each `Tenant` to report structured memory usage data to the scheduler. At the `ISSUE` signal (line 0), the `Tenant` transmits its `Task` count and a memory graph describing the peak memory demand of each `Task`, enabling the scheduler to perform worst-case memory analysis before execution begins. At subsequent signal points, the `Tenant` reports updated memory usage corresponding to pool allocation and deallocation events. The scheduler uses this information to track GPU memory availability and make admission decisions, as detailed in Section 5.

```
0  DNN.Signal-send(ISSUE, mem_graph, n_tasks=2) // Tenant start
1  // ====== Task 0 (intra-Task pool: 1344 B) ======
2  pool_0 = DNN.Pool-init(1344)
3  d_x    = DNN.Mem-offset(pool_0, 1120, 192)
4  t0     = DNN.Memcpy(d_x, x) {H2D}
5  w1     = onnx.Constant()
6  d_w1   = DNN.Mem-offset(pool_0, 0, 864)
7  t1     = DNN.Memcpy(d_w1, w1) {H2D}
8  ...
9  // ====== Shared tensor pool: 512 B ======
10 DNN.Signal-send(SHARED_ALLOC, size=512)
11 shared = DNN.Pool-init(512)
12 d_r1   = DNN.Mem-offset(shared, 0, 512)
13 t3     = DNN.Conv-relu(d_x, d_w1, d_b1, d_r1, d_ws1)
14 DNN.Dealloc(pool_0)                          // Task 0 pool freed
15 DNN.Signal-send(TASK_DONE, task_id=0)
16 // ====== Task 1 (intra-Task pool: 3104 B) ======
17 pool_1 = DNN.Pool-init(3104)
18 w2     = onnx.Constant()
19 d_w2   = DNN.Mem-offset(pool_1, 0, 2304)
20 t9     = DNN.Memcpy(d_w2, w2) {H2D}
21 ...
22 d_c2   = DNN.Mem-offset(pool_1, 2304, 512)
23 t11    = DNN.Conv(d_r1, d_w2, d_b2, d_c2, d_ws2)
24 t14    = DNN.Add(d_r1, d_c2, d_c2)          // skip connection
25 DNN.Dealloc(shared)                          // shared pool freed
26 DNN.Signal-send(SHARED_DEALLOC)
27 t17    = DNN.Relu(d_c2, d_c2)
28 h_out  = DNN.Host-malloc(512)
29 t19    = DNN.Memcpy(h_out, d_c2) {D2H}
30 DNN.Dealloc(pool_1)                          // Task 1 pool freed
31 DNN.Signal-send(TASK_DONE, task_id=1)
32 DNN.Signal-send(DONE)                       // Tenant end
```

Figure 4.9: Final DNN dialect (IR 6) for the running example with scheduler interface signal operations. Four signal types are inserted at compile time: `ISSUE` (line 0) transmits the compile-time memory graph, `TASK_DONE` (lines 15 and 31) notifies `Task` completion, and `SHARED_ALLOC`/`SHARED_DEALLOC` (lines 10 and 26) track shared pool lifetime. For single-tenant deployment, all signal operations are omitted.

5. PAGRUS Scheduler

The PAGRUS runtime extends the single-tenant compiler pipeline to support multi-tenant DNN inference. In single-tenant mode, the compiler produces a standalone optimized binary that executes a single DNN model without any scheduling intervention. In multi-tenant mode, the compiler generates scheduler interfaces (Section 4.5) that allow each compiled `Tenant` to communicate structured memory information to a lightweight scheduler. The scheduler coordinates the execution of multiple `Tenants` on a shared GPU, preventing out-of-memory errors through compile-time metadata rather than runtime memory polling. The scheduler architecture (Section 5.1), the OOM-aware scheduling algorithm (Section 5.2), and the scheduling policies (Section 5.3) are presented in turn.

5.1 Scheduler Architecture

The PAGRUS scheduler is an event-driven coordinator that receives signal events from all active `Tenants` and makes dispatching decisions based on compile-time memory metadata. Figure 5.1 illustrates the scheduler architecture and its six-step scheduling pipeline. The scheduler consists of two main components: a runtime handler that processes incoming signals and extracts `Tenant` metadata (❶–❷), and a scheduling algorithm that manages the scoreboard and deploys `Tasks` to the GPU (❸–❻).

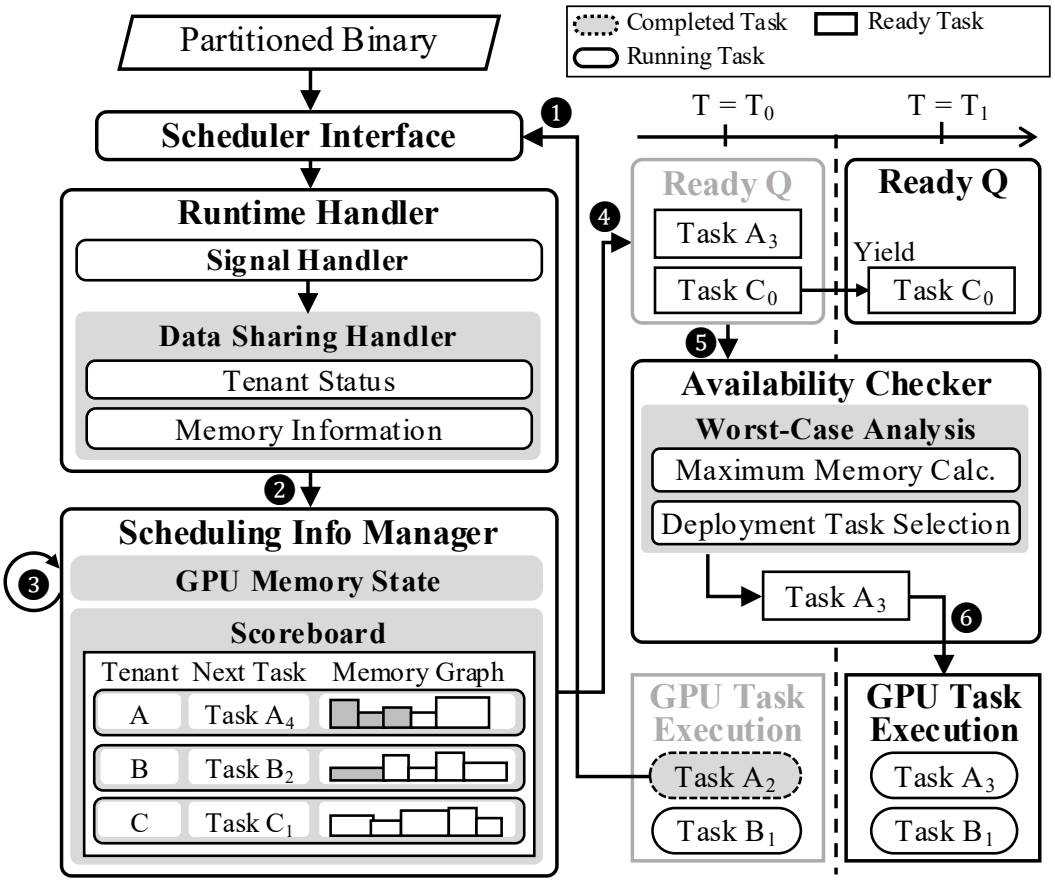


Figure 5.1: Scheduler architecture and scheduling walkthrough. A signal from `Tenant` A upon completion of `Task` A_2 triggers the runtime handler (❶), which passes the event metadata to the scheduling info manager (❷). The scoreboard is updated with `Tenant` A's next `Task` index and the current GPU memory state (❸). The ready queue is populated with the next pending `Tasks` (❹). The availability checker performs worst-case memory analysis to select deployable `Tasks` (❺). At $T{=}T_0$, `Task` A_3 is deployed while `Task` C_0 is yielded. At $T{=}T_1$, only C_0 remains in the ready queue, awaiting memory availability. The selected `Tasks` are dispatched to the GPU for execution (❻).

Runtime Handler. The runtime handler is the entry point for all scheduler interactions. When a `Tenant` sends a signal via the `DNN.Signal-send` operation inserted by the compiler (Section 4.5), the signal handler receives the event (❶) and invokes the data sharing handler to retrieve the associated metadata. As shown in Figure 5.1, the data sharing handler extracts two categories of information from the compiled binary: the `Tenant` status (e.g., new issue or `Task` completion, the index of the recently completed `Task`) and the memory information (an updated memory usage profile). If the signal represents the `Tenant`'s initial issue, the data sharing handler additionally retrieves the total number of `Tasks` and the corresponding peak memory graph, which encodes the worst-case memory demand of each `Task`. The retrieved information is then passed to the scheduling info manager (❷), which updates the internal scoreboard. If the `Tenant` is newly introduced, a new entry is created in the scoreboard.

Scoreboard. The scheduling info manager maintains a scoreboard that records the GPU memory state across all active `Tenants` (❸). As illustrated in Figure 5.1, each row in the scoreboard contains the `Tenant` identifier, the next `Task` to be executed, and the peak memory graph received at registration. The scoreboard is updated on every signal event. When a `Task` completes, the scheduling info manager identifies the corresponding row, updates the next `Task` index, and reflects the memory deallocation into the aggregate GPU memory state. When a new `Tenant` is issued, a new row is created with the complete metadata received from the data sharing handler. After the scheduling cycle completes, the scoreboard records that `Tenant` A is scheduled for `Task` A_4 next, `Tenant` B for `Task` B_2, and `Tenant` C for `Task` C_1.

Ready Queue. After updating the scoreboard, the scheduling info manager enqueues the next pending Tasks into the ready queue (❹). As shown at $T{=}T_0$ in Figure 5.1, the ready queue contains Task A_3 (the next Task from an already deployed Tenant A) and Task C_0 (the first Task from a newly registered Tenant C). The ready queue serves as the input to the availability checker, which processes each entry to determine deployment eligibility.

Availability Checker. After updating the scoreboard and ready queue, the availability checker is invoked to determine the next Tasks to deploy (❺). The availability checker begins from the head of the ready queue. If the Task is from an already deployed Tenant (e.g., Task A_3), it is deployed immediately because the worst-case memory demand of that Tenant was already accounted for during the initial admission. If the Task is from a new Tenant (e.g., Task C_0), identified by a Task index of zero, the availability checker performs worst-case memory analysis. It examines all previously deployed or currently running Tenants (e.g., A and B), together with the new Tenant (C), using Task indices and peak memory graphs to compute each Tenant's future peak memory demand. The sum of these demands, called the worst-case memory usage, represents the maximum total memory that the issued Tenants may consume in the future. If this cumulative demand fits within the GPU memory budget, the new Tenant is deployed. Otherwise, its execution is temporarily yielded, and the next new Tenant in the ready queue is analyzed. This process continues until no unexamined new Tenants remain. The detailed admission formulation is described in Section 5.2.1.

GPU Task Execution. The selected Tasks are dispatched to the GPU for execution (❻). In the scenario of Figure 5.1, Task C_0 is yielded while Task A_3 proceeds at timestamp $T{=}T_1$. At $T{=}T_0$, the GPU is executing Tasks A_2 and B_1, and at $T{=}T_1$, Task A_3 has replaced the completed A_2. Task C_0 remains in the ready queue, awaiting a future signal event that releases sufficient memory. Upon completion, each Task sends a signal operation back to the runtime handler (❶), restarting the scheduling cycle. Through this strategy, the scheduler maintains safe execution without triggering memory evictions, maximizing GPU utilization until all entries in the scoreboard are cleared.

5.2 OOM-Aware Scheduling

The core contribution of the PAGRUS runtime is its ability to prevent OOM errors deterministically, without runtime memory polling or tensor eviction. This is achieved through worst-case memory analysis using compile-time metadata, combined with a yield execution strategy that defers deployment when memory is insufficient.

5.2.1 Worst-Case Memory Analysis

The availability checker (❺ in Figure 5.1) determines whether deploying a new Tenant would eventually cause an OOM error. When a new Tenant's first Task (identified by a Task index of zero) reaches the ready queue, the checker computes the worst-case aggregate memory demand of all Tenants that would be simultaneously active if the candidate were admitted.

For each already deployed Tenant, the availability checker examines the peak memory graph stored in the scoreboard and the current Task index to determine the max-

imum memory that the Tenant may consume during its remaining execution. This future peak memory demand accounts for all subsequent Tasks, including their intra-Task pools and any shared tensor pools that have not yet been deallocated. For the candidate Tenant, the availability checker uses the complete peak memory graph received at registration.

The admission decision is formulated as follows:

$$\sum_{\forall\, t \in \text{deployed}} \text{futurePeak}(t) \;+\; \text{futurePeak}(\text{candidate}) \;\leq\; M_{\text{GPU}} \tag{5.1}$$

where M_{GPU} is the total GPU memory capacity and $\text{futurePeak}(t)$ is the maximum memory demand of Tenant t from its current Task index onward. If the inequality holds, the candidate Tenant is admitted and its first Task is dispatched to the GPU (❻). Otherwise, the candidate is deferred (yielded), and the scheduler evaluates the next new Tenant in the ready queue. This process continues until no unexamined new Tenants remain, so that the active deployment is conservatively protected from OOM errors under the worst-case demand assumption.

This approach provides a conservative guarantee. Because the peak memory graph captures the worst-case demand at every point in the execution timeline, admission is granted only when the summed worst-case demand of the admitted Tenants fits within the GPU memory capacity. This deterministic guarantee is fundamentally different from runtime polling approaches, where memory availability queries such as cudaMemGetInfo are subject to measurement latency and race conditions between polling and allocation. On embedded platforms such as the NVIDIA Jetson Orin Nano, the av-

erage latency of `cudaMalloc` or `cudaMallocManaged` is 271 microseconds per 1 MB allocation, creating inconsistency windows between the polled availability and the true state. By relying on compile-time metadata stored in the scoreboard, PAGRUS removes the polling-induced inconsistency window described above.

5.2.2 Yield Execution Strategy

When no candidate `Tenant` can be admitted under the worst-case memory analysis (Equation (5.1)), the scheduler enters a yield state. During yield, no new `Tenants` are deployed, and the GPU executes only the currently running `Tasks`. The scheduler resumes evaluation when a running `Task` completes and sends a signal (❶), releasing memory and potentially allowing a previously deferred candidate to be admitted.

The yield strategy stands in contrast to eviction-based approaches used in prior multi-tenant systems. To understand the difference, it is necessary to first examine how shared tensors create the conditions that lead to eviction.

Shared tensor retention and stale accumulation. When a model is partitioned into multiple `Tasks`, intermediate tensors shared between consecutive `Tasks` (e.g., between X_k and X_{k+1}) must remain accessible across `Task` boundaries. The conventional approach backs up these shared tensors to host memory after each `Task` completes and restores them to device memory before the subsequent `Task` begins. This backup-restore pattern incurs substantial host-device transfer overhead at every `Task` boundary.

To eliminate this overhead, PAGRUS allocates a dedicated GPU-resident memory pool for shared tensors (the shared tensor pool, described in Section 4.4.2). Each shared

tensor pool X_{kk+1} retains its contents in GPU memory even after Task X_k finishes, allowing Task X_{k+1} to directly access the data without any memory transfers. This zero-move strategy eliminates backup-restore costs entirely.

However, retaining shared tensor pools introduces a new problem. As multiple Tenants execute concurrently, their shared tensor pools accumulate in GPU memory. Pool A_{01} remains allocated while Task A_1 uses it and afterward, pool B_{01} similarly persists, and so on. These stale shared tensor pools occupy memory without active use, progressively reducing the free memory available for new Task deployments. On embedded platforms with unified memory, where backing up evicted data reaches flash-backed storage at limited bandwidth, the penalty is particularly severe.

Figure 5.2 provides a detailed comparison of how eviction-based and yield-based schedulers handle this stale tensor accumulation, using the three-Tenant scenario from Figure 1.3.

Eviction-based scheduling. In the eviction-based approach (Figure 5.2a), the scheduler deploys Tasks as soon as memory permits without anticipating future contention. At t=0, Tasks A_0, B_0, and C_0 along with their shared tensor pool allocations (A_{01}, B_{01}, C_{01}) are all placed in the ready queue. The scheduler admits A_0 and B_0 because their combined memory fits within the 8 GB capacity, and allocates shared tensor pools A_{01} and B_{01} alongside them. As execution proceeds, A_0 completes and A_1 begins, but pool A_{01} remains resident. Pool B_{01} similarly persists after B_0 completes. By t=11, the accumulated stale pools leave insufficient free memory to deploy any new Task, since all remaining candidates require at least 2 GB. The scheduler is forced to evict

shared tensor pools to host memory, incurring costly data transfers before C_0 can finally proceed. These eviction operations block ongoing computation, degrading overall throughput and increasing end-to-end latency.

Yield-based scheduling. In the yield-based approach (Figure 5.2b), the scheduler applies worst-case memory analysis before each deployment decision. At t=0, the same `Tasks` and pools enter the ready queue. The scheduler admits A_0 and B_0 as before, but when evaluating C_0, the worst-case analysis examines the peak memory graphs of all deployed `Tenants` (A, B) together with the candidate (C). The analysis determines that deploying C_0 at this point would eventually force eviction of shared tensor pools to accommodate future `Task` memory demands. Rather than proceeding and incurring eviction overhead later, the scheduler yields C_0, marking it as deferred in the ready queue. C_0 remains in the queue until sufficient memory becomes available through the natural completion of earlier `Tasks` and the release of their shared tensor pools. Although `Task` C_0 experiences a brief idle period, the absence of eviction transfers results in shorter overall end-to-end latency compared to the eviction-based approach.

Yield overhead analysis. The cost of yielding is bounded by the remaining execution time of the longest currently running `Task`, because once any `Task` completes, the released memory may satisfy the deferred candidate's admission requirement. In `Task`-level scheduling with host backup (without yield or shared pools), the host-to-device and device-to-host memory copies required for backup and restoration can increase worst-case end-to-end latency by more than 50% relative to the single-tenant baseline (Sec-

tion 6.4.1). By contrast, the complete proposed system (with shared tensor pooling and yield scheduling) reduces this overhead to single-digit percentages, as demonstrated in Section 6.4.1. On embedded platforms with flash-based storage, the advantage of yield is particularly pronounced due to the limited bandwidth between flash storage and unified memory.

The key insight enabling the yield strategy is the `Task`-level scheduling granularity produced by the Tailor-and-Stitch partitioning (Section 4.3). Because each `Task` represents a balanced segment of the model rather than the entire model, the waiting time during yield is proportional to the duration of a single `Task` rather than the entire inference. Fine-grained `Task` boundaries ensure that memory is released frequently, limiting the maximum yield duration.

5.2.3 Scheduling Walkthrough

Figure 5.1 illustrates the complete scheduling pipeline through a concrete scenario involving three `Tenants` (A, B, and C). The walkthrough traces the six steps that occur when `Task` A_2 completes while `Tenant` C is awaiting its first deployment.

At timestamp $T{=}T_0$, the GPU is executing `Tasks` A_2 and B_1. `Task` A_2 completes and its signal operation triggers the runtime handler (❶). The signal handler receives the completion event, and the data sharing handler extracts the `Tenant` status and updated memory information from `Tenant` A's compiled binary.

The scheduling info manager receives the metadata (❷) and updates the scoreboard (❸). `Tenant` A's row in the scoreboard is updated: the next `Task` index advances to A_3, and the memory released by `Task` A_2 is reflected in the aggregate GPU memory state.

Tenant C has also been registered, and its complete metadata (peak memory graph) has been recorded in the scoreboard.

The ready queue (❹) is then populated with Task A_3 (the next Task from the already deployed Tenant A) and Task C_0 (the first Task from Tenant C). The availability checker (❺) processes the ready queue starting from the head. Task A_3 is from an already deployed Tenant, so it is deployed immediately without additional analysis. For Task C_0, identified by Task index zero as a new Tenant, the checker performs worst-case memory analysis. It examines all deployed Tenants (A and B) together with the new Tenant (C), using Task indices and peak memory graphs to compute each Tenant's future peak memory demand. Because the cumulative demand exceeds M_{GPU}, Task C_0 is yielded.

At timestamp $T{=}T_1$, Task A_3 has been deployed and is executing alongside Task B_1 (❻). The ready queue now contains only Task C_0, which remains deferred until a future Task completion releases sufficient memory. When either A_3 or B_1 completes and sends a new signal (❶), the cycle restarts, and the availability checker re-evaluates whether C_0 can be admitted.

5.3 Scheduling Policies and Extensions

The PAGRUS scheduler decouples the scheduling policy from the OOM prevention mechanism. The worst-case memory analysis and yield strategy described in Section 5.2 ensure memory safety regardless of the ordering in which Tasks are dispatched. Different scheduling policies can therefore be applied on top of the OOM-safe foundation without compromising the memory guarantee.

First-Come, First-Served (FCFS). The default policy dispatches `Tasks` in the order they arrive in the ready queue. FCFS provides fairness across `Tenants` and requires no additional metadata beyond arrival timestamps. This policy is appropriate for deployment scenarios where all DNN models have comparable importance and latency requirements.

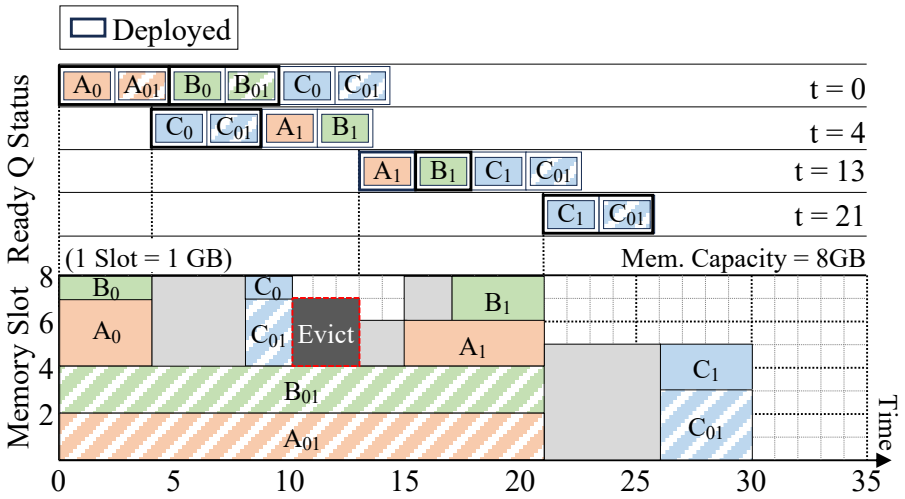
Shortest Job First (SJF). The SJF policy prioritizes `Tasks` with shorter estimated execution times, as provided by the compiler's compile-time profiling data. By dispatching shorter `Tasks` first, SJF reduces the average waiting time across all `Tenants`. However, this policy may starve long-running `Tenants` if short `Tasks` are continuously submitted, making it more suitable for batch processing scenarios.

Priority-Based Scheduling. In safety-critical deployments (e.g., autonomous vehicles), certain DNN models may require guaranteed low latency. The priority-based policy assigns static priorities to `Tenants`, and the scheduler dispatches higher-priority `Tasks` before lower-priority ones. This ensures that time-critical models (e.g., object detection for collision avoidance) are scheduled ahead of less critical models (e.g., comfort features).

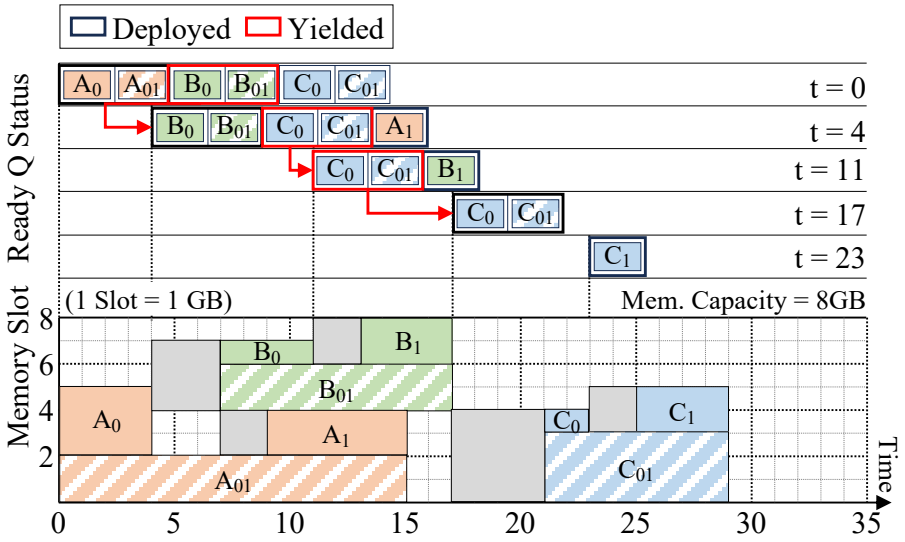
Aging for Starvation Avoidance. Both SJF and static priority scheduling can defer a long-running or low-priority `Tenant` for an unbounded time when shorter or higher-priority `Tasks` keep arriving. The decoupling between the scheduling policy and the OOM prevention mechanism makes this case directly addressable. The worst-case memory analysis only decides which candidates are admissible at a given instant, and the

ordering among admissible candidates is unconstrained by the memory guarantee. An aging policy raises the effective priority of a deferred candidate in proportion to its accumulated waiting time, so that the candidate is dispatched ahead of newly arrived `Tasks` once its waiting time crosses a threshold and memory permits its admission. This bounds the waiting time of any `Tenant` whose standalone memory demand is individually feasible, and it removes the indefinite deferral that pure SJF or static priority can otherwise produce, while preserving the worst-case memory safety enforced by the yield mechanism. Because aging operates entirely on the ordering of already admissible candidates, it requires no change to the worst-case analysis and adds only a per-`Tenant` waiting-time counter to the scoreboard.

Latency Stability. A notable property of the yield-based scheduling is its predictable latency behavior. Because yield simply defers execution without triggering costly data migration, the latency variance across inference iterations is substantially lower than in eviction-based systems. Eviction-based approaches exhibit high variance because the eviction cost depends on the volume of data to be transferred, which varies with the shared tensor sizes and the number of concurrent `Tenants`. The yield strategy produces bounded, predictable delays that do not depend on data transfer volumes, resulting in more stable end-to-end latency distributions.



(a) Without yield execution (Figure 1.3e)



(b) With yield execution (Figure 1.3f)

Figure 5.2: Detailed ready queue status and GPU memory timeline comparing eviction-based and yield-based scheduling for the three-Tenant scenario in Figure 1.3. a Without yield execution, stale shared tensor pools accumulate until free memory is exhausted, forcing eviction before Task C_0 can be deployed. b With yield execution, Task C_0 is deferred proactively based on worst-case memory analysis, avoiding all eviction transfers. Despite the brief idle period, the end-to-end latency is shorter because no eviction transfers occur.

6. Evaluation

Single-tenant evaluation (Section 6.2) measures memory usage reduction and inference latency for individual DNN models. Model partitioning evaluation (Section 6.3) analyzes the Tailor-and-Stitch algorithm and compares scheduling granularities. Multi-tenant evaluation (Section 6.4) demonstrates OOM-free concurrent execution with yield-based scheduling. Runtime overhead analysis (Section 6.5) quantifies the cost of scheduling and partitioning. Finally, case study evaluation (Section 6.6) examines workloads that stress the static pool scheme beyond conventional single-model inference.

6.1 Experimental Setup

6.1.1 Hardware Platforms

Two hardware platform sets are used throughout the evaluation, each pairing a desktop-class GPU with an embedded edge device, as summarized in Table 6.1.

Each platform set reflects a different hardware generation, and the two sets are not mixed within a single experiment. Single-tenant evaluation (Section 6.2) uses Set B, while model partitioning (Section 6.3) and multi-tenant evaluation (Section 6.4) use Set A, and each evaluation section states its platform explicitly. On the Jetson Orin Nano, flash storage is used for tensor eviction experiments because unified memory constraints pre-

Table 6.1: Hardware platform sets used in the evaluation. Each set pairs a desktop-class GPU with an embedded edge device, and the single-tenant, partitioning, and multi-tenant evaluations each report both a desktop and an edge result. The two sets are different hardware generations and are not compared within a single experiment. (Titan RTX is used only for allocation-overhead microbenchmarks. The MIG-constrained serving case study uses a separate GPU described in the text.)

Platform Set	Desktop GPU	Embedded edge device	Used by
A	RTX 4090 (24 GB)	Jetson Orin Nano (8 GB)	Model partitioning, Multi-tenant
B	RTX 3090 (24 GB), Titan RTX (24 GB)	Jetson AGX Xavier (16 GB)	Single-tenant

vent host-side backup. The case study evaluation (Section 6.6) reuses the Set A devices, with autoregressive decoding (Section 6.6.1) on the RTX 4090 and speculative decoding (Section 6.6.2) on the Jetson Orin Nano. The MIG-constrained serving case study (Section 6.6.3) instead runs on an NVIDIA RTX PRO 6000 Blackwell Max-Q GPU (96 GB), whose hardware-level Multi-Instance GPU (MIG) partitioning into isolated instances is not available on the other GPUs.

6.1.2 DNN Models and Baselines

The evaluation uses a total of eleven DNN inference models spanning diverse application domains. For single-tenant evaluation on Platform Set B, six models are evaluated with batch size 32 on the RTX 3090 (batch size 2 for SRGAN due to memory constraints) and batch size 16 on the Jetson AGX Xavier (batch size 1 for SRGAN). For multi-tenant evaluation on Platform Set A, four additional models are introduced for desktop GPU experiments, and three additional models are introduced for embedded GPU experiments. All models are converted from PyTorch to ONNX format and compiled through the PAGRUS pipeline. Table 6.2 summarizes the model characteristics.

Table 6.2: DNN model benchmark summary. (Each column abbreviation represents #Op = number of operators, #Var = number of variables (tensors) in the ONNX representation, respectively.)

Model	Application	#Op	#Var	Platform Set
ResNet50	Image classification	176	284	A, B
MobileNet	Mobile classification	153	259	B
EfficientNet	Image classification	457	689	A
SSD-ResNet50	Object detection	196	326	A, B
SRGAN	Super-resolution	113	154	B
YOLOv1	Object detection	83	135	B
BERT	NLP (transformer)	643	953	B
Vision Transformer	Image classification	619	870	A
GPT-2	Text generation	625	858	A
OpenPose	Pose estimation	64	106	A
DeepLab v3	Semantic segmentation	193	309	A

The following baseline configurations are compared against PAGRUS. For single-tenant evaluation, three compiler configurations are evaluated, all built on top of ONNX-MLIR. Eager allocation places all memory allocations at program start and deallocations at program end, resulting in a single memory management group (MMG) but leading to high peak memory. Lazy allocation allocates each tensor immediately before its first use and deallocates it after its last use, reducing peak memory but incurring frequent allocation overhead. PAGRUS uses a single pre-allocated memory pool with compile-time offset assignment, achieving both low peak memory and low allocation overhead. PyTorch (version 1.13) serves as an additional external baseline.

For multi-tenant evaluation, the following scheduling baselines are compared. Model-level scheduling with eviction (M+E) schedules entire models as indivisible units and evicts tensors to host memory when GPU memory is insufficient. Layer-level scheduling with eviction (L+E) schedules individual layers, enabling fine-grained concurrency but

Table 6.3: Single-tenant workload configuration on Platform Set B.

Model	Batch (RTX 3090)	Batch (Jetson Xavier)
ResNet50	32	16
MobileNet	32	16
SSD-ResNet50	32	16
BERT	32	16
YOLOv1	32	16
SRGAN	2	1

incurring high synchronization overhead. Layer-level scheduling with worst-case analysis (L+W) applies worst-case memory analysis at layer granularity, preventing evictions but suffering from excessive scheduling boundaries. NVIDIA Triton Inference Server is compared on the desktop GPU platform. Triton is excluded from the Jetson platform because the Jetson-specific Triton distribution does not support executing ONNX Runtime models, which are the input format used by the PAGRUS compiler. SwapNet, which targets OOM prevention on embedded GPUs through host-device tensor swapping, is compared on the Jetson Orin Nano platform.

For memory pool layout evaluation, two heuristic allocation policies are compared against the PAGRUS CP-SAT exact solver. These include first-fit and best-fit, each applied to the same set of tensors and liveness intervals.

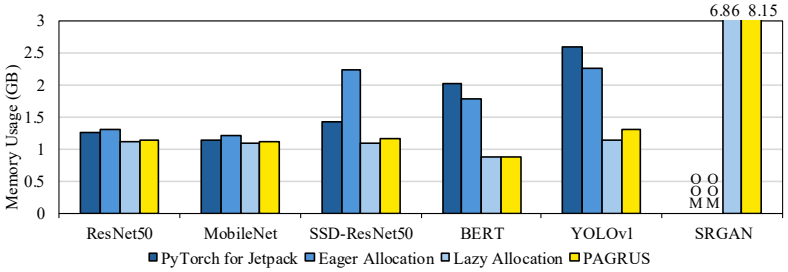
6.2 Single-Tenant Evaluation

Single-tenant evaluation measures the PAGRUS compiler on Platform Set B (RTX 3090 and Jetson AGX Xavier). Table 6.3 summarizes the batch sizes used for each model on each target device.

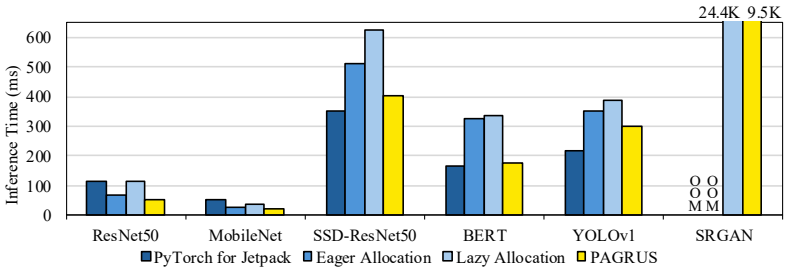
6.2.1 Memory Usage and Latency

Figure 6.1 shows that PAGRUS reduces memory usage with comparable inference time for most benchmarks on both platforms. On the Jetson AGX Xavier (Figures 6.1a and 6.1b), PAGRUS reduces memory usage by 31.2% compared to PyTorch while achieving 1.29× speedup. On the RTX 3090 (Figures 6.1c and 6.1d), PAGRUS reduces memory usage by 37.7% compared to PyTorch while achieving 1.21× speedup. Across both platforms, the geometric mean improvement is 34.6% in memory reduction and 1.25× in inference speedup compared to PyTorch. The memory reduction varies across models depending on the ratio of reusable tensors. BERT exhibits the largest reduction (56.3%) because a single large embedding layer bounds the memory pool size, and all other layers reuse that pool efficiently. YOLOv1 shows a similar pattern (50.1%) due to its large fully connected layer. In contrast, MobileNet shows only 3.7% reduction because MobileNet is a compact network where the CUDA context memory dominates the total GPU memory footprint, concealing the benefit of memory pooling. On the RTX 3090, BERT does not show significant memory saving because PyTorch does not support embedding operations on a GPU in this configuration, and instead runs the embedding table on the CPU. PAGRUS runs the embedding table on the GPU, allocating a memory pool of equivalent size. PAGRUS also prevents out-of-memory errors that occur under PyTorch. On the Jetson AGX Xavier (16 GB unified memory), PyTorch fails to execute SRGAN due to OOM, whereas the PAGRUS compiler successfully generates a memory pool and completes inference.

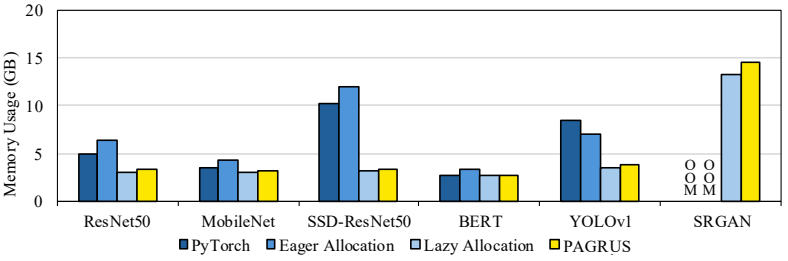
Figure 6.1 also illustrates the efficiency of the memory pool optimization compared



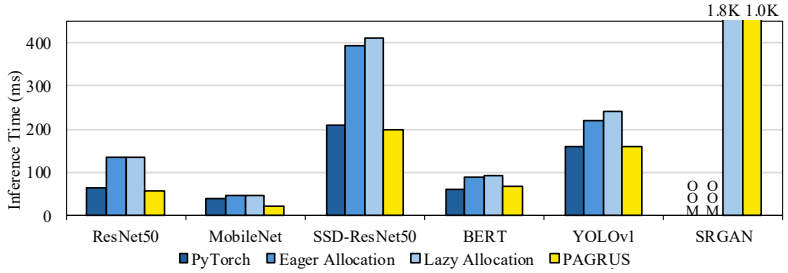
(a) Memory usage on Jetson AGX Xavier



(b) Inference time on Jetson AGX Xavier



(c) Memory usage on RTX 3090



(d) Inference time on RTX 3090

Figure 6.1: Memory usage and inference time of six DNN benchmarks on Platform Set B. Batch sizes are listed in Table 6.3. (OOM = out-of-memory, indicating that PyTorch fails to execute the model due to insufficient GPU memory.)

Table 6.4: Number of memory management groups (MMGs) and memory allocations for six benchmarks under three allocation strategies.

DNN Model	# Op	# Var	Eager		Lazy		PAGRUS	
			#MMG	#Alloc	#MMG	#Alloc	#MMG	#Alloc
ResNet50	176	284	1	284	123	284	1	1
MobileNet	153	259	1	259	107	259	1	1
SSD-ResNet50	196	326	1	326	133	326	1	1
BERT	643	953	1	953	556	953	1	1
YOLOv1	83	135	1	135	58	135	1	1
SRGAN	113	154	1	154	77	154	1	1

to eager and lazy allocation. PAGRUS uses only 7.3% more memory compared to lazy allocation while achieving 1.59× speedup. The eager allocation has one MMG because it allocates and deallocates every tensor at the beginning and end of inference, respectively. In contrast, lazy allocation causes more MMGs because it allocates and deallocates tensors just before their uses and after their liveness ends. As a result, lazy allocation achieves smaller peak memory usage but shows slower inference time because memory management operations interfere with computation. PAGRUS achieves only one MMG and one allocation with memory usage similar to lazy allocation, implying that the memory pool optimization captures the advantages of both allocation schemes. Table 6.4 summarizes the number of MMGs and allocations for each benchmark under the three strategies.

6.2.2 Memory Pool Optimality Analysis

To evaluate the quality of the PAGRUS compiler's memory pool layout, the CP-SAT exact solver is compared against two widely used heuristic allocation strategies (best-fit and first-fit) and the sweep-line lower bound. The sweep-line lower bound represents the

Table 6.5: Memory pool optimality comparison across allocation strategies. (Each abbreviation represents LB = sweep-line lower bound, CP-SAT = proposed exact solver, BF = best-fit heuristic, FF = first-fit heuristic, OPT/LB = ratio of CP-SAT solution to lower bound, T_{solve} = CP-SAT wall-clock solve time, respectively.)

Model	**Conflicts**	**LB (MB)**	**CP-SAT (MB)**	**BF (MB)**	**FF (MB)**	**OPT/LB**	T_{solve} **(s)**
ResNet50	745	73.5	73.5	85.8	85.8	1.000	0.1
MobileNet	606	73.5	73.5	85.8	85.8	1.000	0.1
SSD-ResNet50	1,336	135.4	135.4	180.5	180.5	1.000	0.5
BERT	3,311	90.9	90.9	90.9	90.9	1.000	0.3
YOLOv1	205	196.0	196.0	214.4	214.4	1.000	0.0
SRGAN	540	2,701.4	2,701.4	3,039.2	3,039.2	1.000	0.1

theoretical minimum memory footprint achievable under any valid non-overlapping placement. It is computed by sweeping through all allocation and deallocation events and recording the maximum sum of simultaneously live tensor sizes.

The CP-SAT formulation models memory layout as a constraint satisfaction problem. Each tensor is assigned an integer offset variable, and for every pair of tensors with overlapping liveness intervals (termed a conflict pair), a disjunctive constraint ensures that their allocated regions do not overlap. The number of conflict pairs reflects the density of the liveness overlap structure. As shown in Table 6.5, the conflict count ranges from 205 (YOLOv1) to 3,311 (BERT), indicating that the problem complexity varies significantly across models.

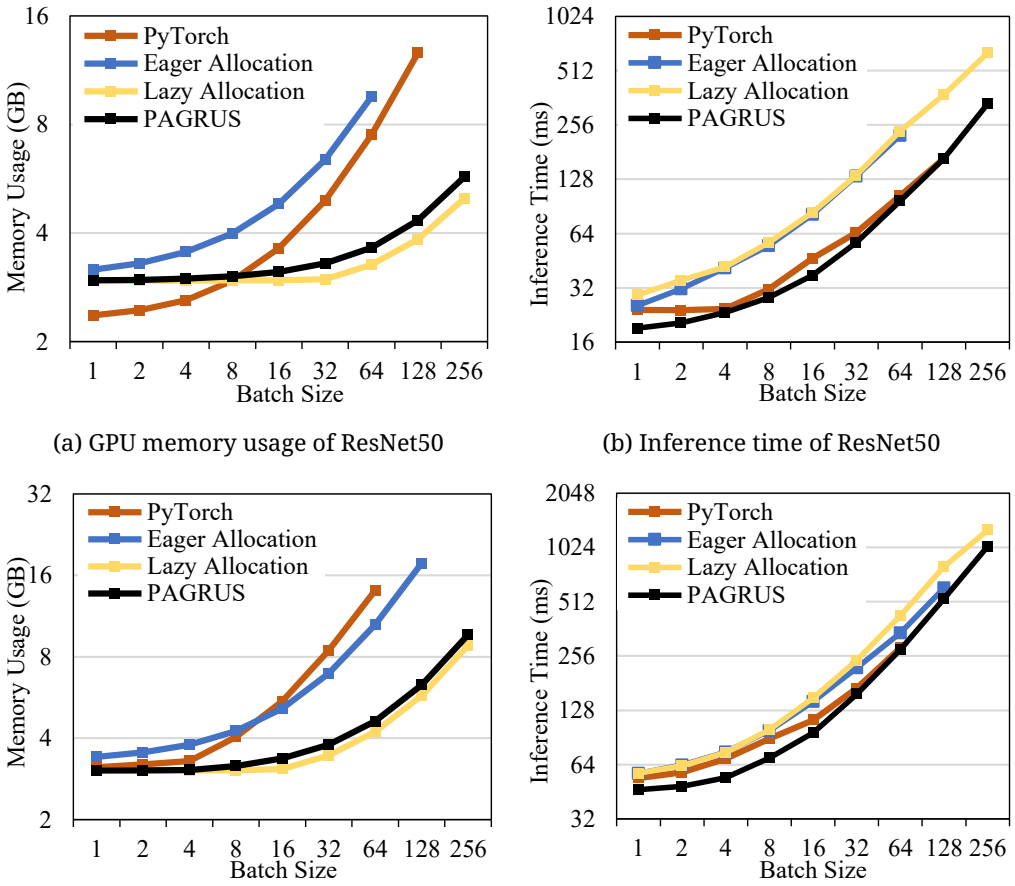
The CP-SAT solver achieves provably optimal pool sizes for all six benchmark models, matching the sweep-line lower bound exactly (OPT/LB = 1.000 in every case). This result shows that the PAGRUS compiler produces optimal intra-`Task` pool layouts for all six evaluated models, leaving zero memory waste beyond what the liveness constraints structurally require. In contrast, the best-fit and first-fit heuristics produce identical

results for all tested models, with overhead ranging from 0.0% (BERT) to 33.3% (SSD-ResNet50) above the lower bound. The fact that both heuristics coincide suggests that the allocation order, rather than the gap-selection policy, dominates the packing quality for these models.

The solve time of the CP-SAT solver remains at most 0.5 seconds for all models, including BERT with 3,311 conflict pairs. Since pool layout is determined at compile time and amortized over all subsequent inference executions, this one-time cost is negligible.

6.2.3 Batch Size Impact

Figure 6.2 shows the memory usage and inference time of ResNet50 and YOLOv1 with varying batch sizes on the RTX 3090 GPU. At small batch sizes (Figures 6.2a and 6.2c), PyTorch occasionally uses less memory than PAGRUS. This is because loading CUDA API functions in an imperative manner in PyTorch uses less driver-reserved memory than loading the complete kernel functions generated by the PAGRUS compiler. When CUDA API functions are invoked, the GPU driver pre-allocates library data (approximately 2 GB on the RTX 3090), and the amount varies depending on which API functions are used. Since PAGRUS and PyTorch use different GPU backends, the amount of CUDA context memory differs between the two systems. However, as the batch size increases, the model's memory footprint becomes dominant over the fixed CUDA context overhead, and PAGRUS achieves lower memory usage through memory pool reuse (Figures 6.2a and 6.2c). The inference time of PAGRUS is slightly lower than PyTorch at small batch sizes (Figures 6.2b and 6.2d), because PAGRUS eliminates memory allocation overhead. As the batch size increases, memory copy overheads dominate the inference time, caus-



(a) GPU memory usage of ResNet50

(b) Inference time of ResNet50

(c) GPU memory usage of YOLOv1

(d) Inference time of YOLOv1

Figure 6.2: Inference time and memory usage on RTX 3090 GPU with varying batch sizes. Missing data points indicate out-of-memory errors at the corresponding batch size.

ing both systems to converge.

6.3 Model Partitioning Evaluation

The Tailor-and-Stitch partitioning algorithm is evaluated on Platform Set A (RTX 4090 and Jetson Orin Nano).

6.3.1 Partitioning Results and Sensitivity

To validate the choice of hyperparameters, a sensitivity analysis was conducted on two key parameters of the Tailor-and-Stitch algorithm. The Peak-to-Average Memory Ratio (PAMR) is used as an evaluation metric, defined as the peak memory usage of a scheduling unit divided by its average memory usage. A PAMR value closer to 1 indicates high memory utilization, whereas a larger value indicates that the memory reserved for peak demand is significantly underutilized on average. The `winNum` parameter controls the granularity of the initial `Tasklet` partitioning by determining the window size of the moving maximum filter. As shown in Tables 6.6a and 6.6b, a larger `winNum` generally produces more `Tasklets` with improved memory efficiency (lower PAMR). This effect is most pronounced in GPT-2, where increasing `winNum` from 5 to 10 reduces the PAMR from 10.36 to 3.37. However, excessively high values lead to a superabundance of `Tasklets` (e.g., 93 for Vision Transformer at `winNum`=40), increasing subsequent analysis complexity without proportional PAMR improvement. The relationship between the number of `Tasklets` and PAMR is not always monotonic, because even if the initial Tailor candidates are the same, the actual `Tasklet` boundaries can vary after K-Means clustering, resulting in different `Tasklet` compositions. A value of `winNum`=20 provides

Table 6.6: Sensitivity analysis of key partitioning parameters. Sub-tables (a) and (b) show the impact of the Tailor algorithm's `winNum` on the initial `Tasklet` count and Peak-to-Average Memory Ratio (PAMR). Sub-tables (c) and (d) show the impact of the Stitch algorithm's `Threshold` on the final `Task` count and PAMR when `winNum` = 20.

winNum	1	5	10	20	40	80
ResNet50	8	8	9	9	13	21
EfficientNet	9	8	8	8	11	13
Vision Transformer	16	16	16	18	93	61
GPT-2	4	4	5	5	26	27
SSD-ResNet50	8	10	11	10	8	18
OpenPose	8	8	9	9	7	11
DeepLabV3	8	8	8	8	11	13

(a) Impact of `winNum` on `Tasklet` count.

winNum	1	5	10	20	40	80
ResNet50	1.85	1.85	1.70	1.69	1.59	1.51
EfficientNet	2.01	2.22	2.22	2.21	1.98	1.83
Vision Transformer	1.37	1.37	1.37	1.66	1.44	1.62
GPT-2	10.36	10.36	3.37	3.37	3.35	3.34
SSD-ResNet50	1.51	1.58	1.55	1.58	1.74	1.51
OpenPose	1.95	1.89	1.88	1.89	2.25	1.74
DeepLabV3	1.89	1.89	1.89	1.89	1.82	1.72

(b) Impact of `winNum` on Average PAMR.

Threshold	1/1	1/5	1/10	1/20	1/40	1/80
ResNet50	1	2	3	4	4	5
EfficientNet	1	2	4	6	6	6
Vision Transformer	1	3	5	5	8	8
GPT-2	1	2	3	5	5	5
SSD-ResNet50	1	2	4	5	5	6
OpenPose	1	2	4	5	7	9
DeepLabV3	1	3	5	5	6	7

(c) Impact of `Threshold` on `Task` count.

Threshold	1/1	1/5	1/10	1/20	1/40	1/80
ResNet50	3.63	3.06	2.65	2.50	2.50	2.28
EfficientNet	8.04	4.12	3.64	2.82	2.82	2.82
Vision Transformer	3.27	2.71	2.70	2.70	1.96	1.96
GPT-2	17.28	13.49	12.93	11.23	11.23	11.23
SSD-ResNet50	2.06	1.94	1.92	1.95	1.95	1.87
OpenPose	5.89	3.84	2.58	2.45	2.19	1.88
DeepLabV3	3.87	3.25	2.77	2.77	2.59	2.35

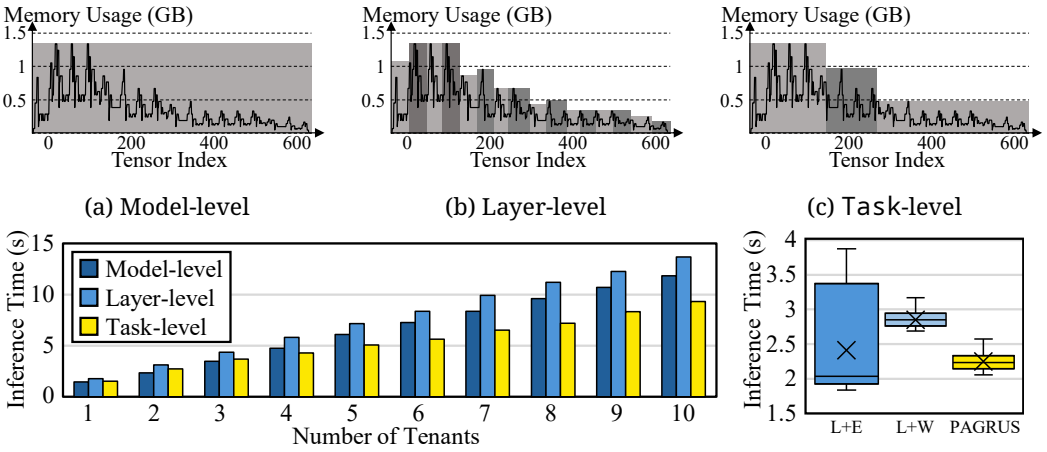
(d) Impact of `Threshold` on Average PAMR.

Table 6.7: Ratio of shared tensors and intra-Task tensors after model partitioning. (Each column abbreviation represents #Total = number of total tensors, #Shared = number of shared tensors, Shared Size = total size of shared tensors, #Intra = number of intra-Task tensors, Intra Size = total size of intra-Task tensors, respectively.)

Platform	Model	#Total	#Shared	Shared Size	#Intra	Intra Size
Desktop	ResNet50	350	3	0.29 GB	347	11.72 GB
	EfficientNet	871	6	1.17 GB	865	17.67 GB
	Vision Transformer	919	13	0.43 GB	906	15.94 GB
	GPT-2	908	10	0.06 GB	898	19.20 GB
Embedded	ResNet50	350	3	0.04 GB	347	1.49 GB
	EfficientNet	871	5	0.39 GB	866	2.43 GB
	SSD-ResNet50	405	11	0.14 GB	394	4.64 GB
	OpenPose	119	6	0.31 GB	113	8.33 GB
	DeepLabV3	380	7	0.23 GB	373	3.74 GB

an effective balance across all evaluated models. The Threshold determines how aggressively Tasklets are merged into final Tasks. As shown in Tables 6.6c and 6.6d, a smaller Threshold produces more Tasks with lower PAMR but incurs higher scheduling overhead due to more frequent Task boundaries. For example, the PAMR of EfficientNet improves from 3.64 to 2.82 as the Threshold is tightened from 1/10 to 1/20, but the number of Tasks increases from 4 to 6, requiring additional shared tensor pools. A Threshold of 1/10 effectively mitigates severe memory fragmentation while keeping the final Task count manageable.

Table 6.7 summarizes the tensor classification statistics after model partitioning. Although shared tensors account for only a small fraction of the total tensor count (typically fewer than 15 out of several hundred), their cumulative size can reach several gigabytes in large-scale models such as EfficientNet (1.17 GB on desktop) and Vision Transformer (0.43 GB on desktop). This observation validates the need for a dedicated GPU-



(a) Model-level (b) Layer-level (c) `Task`-level

(d) Impact of scheduling granularity (e) Latency stability

Figure 6.3: Scheduling granularity comparison. (a)–(c) show the scheduling units of ResNet50 under model-level, layer-level, and `Task`-level partitioning. (d) Average inference time of Vision Transformer with varying numbers of `Tenants`. (e) Latency distribution over 100 runs under memory-saturated conditions with Vision Transformer and GPT-2. (Each abbreviation represents L+E = layer-level with eviction, L+W = layer-level with worst-case analysis, respectively.)

resident memory pool to manage shared tensors, as transferring them between host and device memory at every `Task` boundary would introduce substantial overhead. On the embedded platform, shared tensor sizes are proportionally smaller due to reduced batch sizes, but the relative cost of host-device transfers is higher due to limited memory bandwidth, making GPU-resident shared pools even more critical.

6.3.2 Scheduling Granularity Comparison

Figures 6.3a to 6.3c show the scheduling units of ResNet50 under three partitioning schemes. Three scheduling granularities are compared for 1 to 10 homogeneous `Tenants` on the desktop GPU. In model-level partitioning (Figure 6.3a), memory pools are pro-

visioned for the entire model's peak usage, resulting in low GPU memory utilization when multiple `Tenants` share the device. Layer-level partitioning (Figure 6.3b) introduces many fine-grained partitions, increasing both potential parallelism and scheduling overhead. `Task`-level partitioning (Figure 6.3c), produced by the Tailor-and-Stitch algorithm, achieves 25.4% speedup over model-level and 37.7% speedup over layer-level scheduling on average, as shown in Figure 6.3d. Under memory-saturated conditions where the combined peak memory demand of concurrent models exceeds GPU capacity, scheduling stability becomes critical. To evaluate this, Vision Transformer (batch size 256, peak 11.5 GB) and GPT-2 (batch size 40, peak 13.6 GB) were executed concurrently on a 24 GB GPU over 100 runs. Figure 6.3e presents the latency distribution over 100 execution runs. Although layer-level scheduling with eviction (L+E) can potentially exploit fine-grained concurrency, it exhibits extreme latency variance, with peak memory layers overlapping in 27% of the runs and triggering memory evictions. Layer-level scheduling with worst-case analysis (L+W) prevents evictions but suffers from high median latency due to excessive synchronization. `Task`-level scheduling achieves the lowest median latency with deterministic stability, effectively balancing safety and efficiency.

6.4 Multi-Tenant Evaluation

Multi-tenant deployment puts the PAGRUS runtime scheduler under concurrent load on Platform Set A (RTX 4090 and Jetson Orin Nano). Table 6.8 summarizes the batch sizes, peak memory usage, partitioning results, and compile times for the evaluated models on each target device. The peak memory reported for each model is the liveness-aware

Table 6.8: Multi-tenant workload configuration on Platform Set A. Batch sizes are selected to produce realistic memory pressure on each device.

Device	Model	Batch	Peak Mem.	#Tasks	#Tasklets	Compile (s)
RTX 4090	ResNet50	256	4.6 GB	3	9	112.86
	EfficientNet	256	7.6 GB	4	8	62.43
	Vision Trans.	128	5.9 GB	5	18	363.79
	GPT-2	32/1024	11.1 GB	3	5	479.11
Jetson Orin Nano	ResNet50	32	0.7 GB	3	8	112.23
	EfficientNet	16	1.1 GB	3	8	59.49
	SSD-ResNet50	16	0.7 GB	4	10	104.20
	OpenPose	16	1.8 GB	4	9	35.72
	DeepLab v3	16	1.7 GB	5	8	185.41

peak from the compiler's peak memory graph rather than the eager footprint of PyTorch, so the multi-tenant comparison runs on the already-reduced footprint that the compiler produces.

Batch sizes are chosen to produce realistic memory pressure on each platform. On the RTX 4090 (24 GB), GPT-2 alone consumes 11.1 GB (46% of GPU memory), and running two concurrent instances of GPT-2 would exceed the GPU capacity. Even moderate workloads such as EfficientNet (7.6 GB) and Vision Transformer (5.9 GB) cannot run simultaneously without memory-aware scheduling. On the Jetson Orin Nano (8 GB unified memory shared with the OS), individual models consume 0.7–1.8 GB, but deploying three or more models concurrently can easily exhaust the available budget. The Tailor-and-Stitch partitioning produces 3–5 `Tasks` per model and 5–18 `Tasklets`, with compile times ranging from 36 to 479 seconds depending on model complexity.

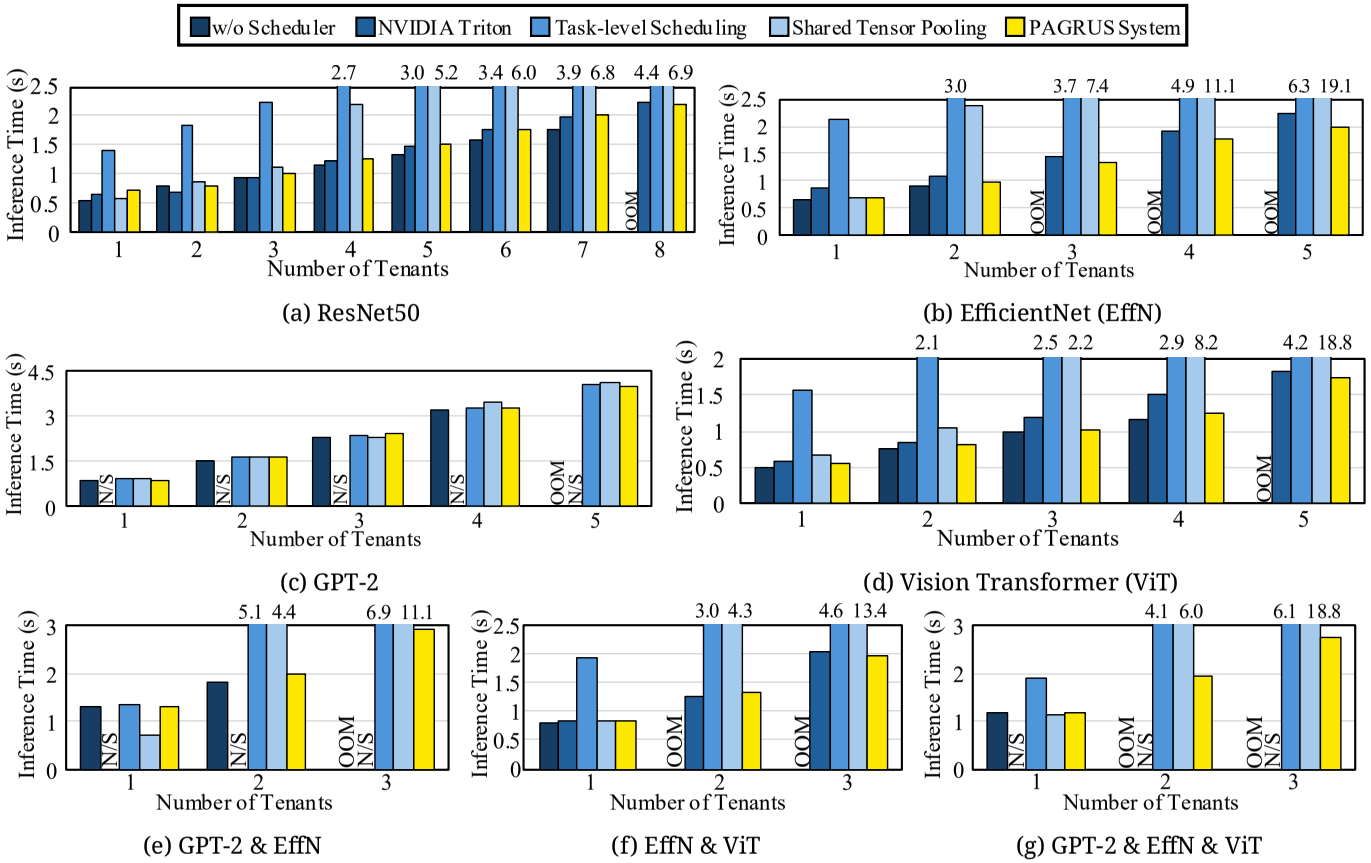


Figure 6.4: Multi-tenant DNN scheduling performance on the desktop GPU (NVIDIA RTX 4090). Top two rows show homogeneous workloads (1 to 8 instances of the same model, with the per-model maximum set by the onset of out-of-memory failure). Bottom row shows heterogeneous workloads with mixed model combinations.

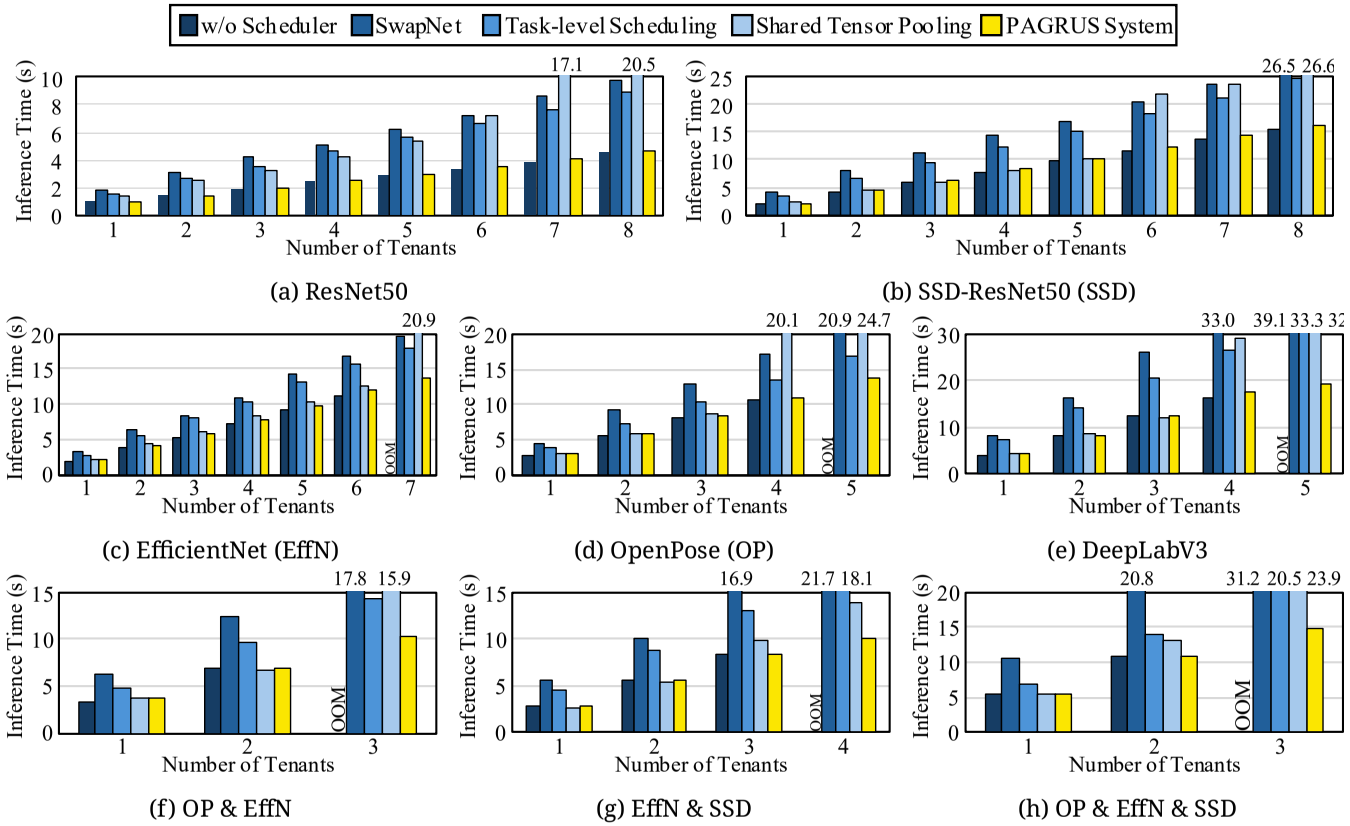


Figure 6.5: Multi-tenant DNN scheduling performance on NVIDIA Jetson Orin Nano. Top two rows show homogeneous workloads. Bottom row shows heterogeneous workloads with mixed model combinations.

6.4.1 Homogeneous and Heterogeneous Workloads

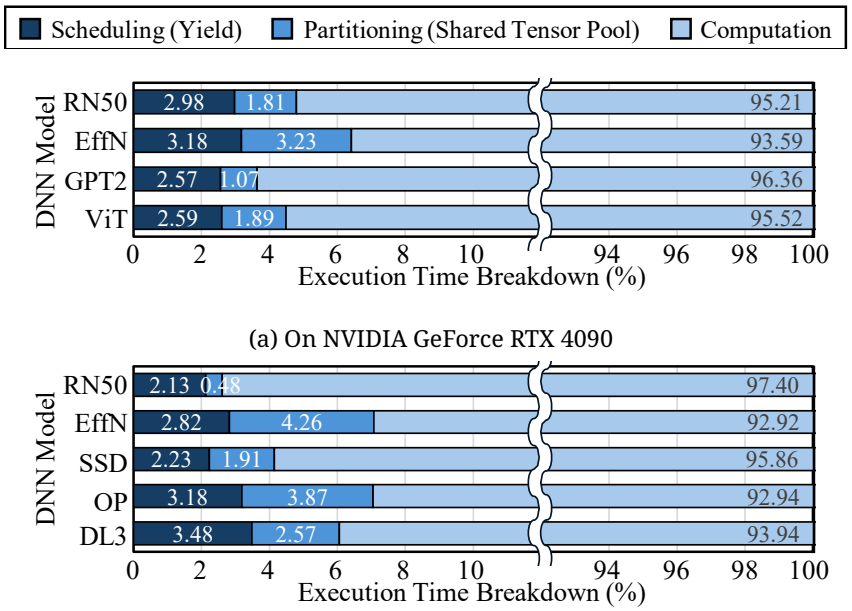
Figures 6.4 and 6.5 illustrate the effects of different scheduling policies on both desktop (Figure 6.4) and Jetson platforms (Figure 6.5). In homogeneous multi-tenant experiments, 1 to 8 instances of the same model are deployed concurrently, with the maximum tenant count per model bounded by the point at which unscheduled execution exhausts GPU memory. Without scheduling, inference fails due to OOM errors as the number of `Tenants` increases. `Task`-level scheduling with host backup (on desktop) or flash backup (on Jetson) avoids OOM but incurs up to 50.54% slowdown on the desktop GPU and 54.55% on the Jetson platform relative to the single-tenant baseline, due to frequent tensor data transfers between GPU and host or flash memory. Shared tensor pooling addresses this problem by retaining intermediate data in GPU memory across `Task` boundaries, eliminating host-device transfers for shared tensors. However, as the number of `Tenants` increases, evictions of shared tensors become more frequent. The PAGRUS yield-based scheduling prevents such evictions by deferring `Tenant` deployment when GPU memory is insufficient, rather than evicting existing tensors. Zero OOM events are observed across all experiments. On the Jetson platform, PAGRUS occasionally outperforms naive (unscheduled) single-tenant execution due to better cache reuse and reduced memory fragmentation that result from memory pool management.

Figures 6.4e to 6.4g and Figures 6.5f to 6.5h show the results for heterogeneous workloads. On the desktop GPU, two image classification networks (EfficientNet and Vision Transformer) and one language model (GPT-2) are used as mixed workloads. On the Jetson, three image networks (OpenPose, EfficientNet, and SSD-ResNet50) are evalu-

ated. PAGRUS effectively prevents OOM errors that occur when running multiple different models without a scheduler, while introducing low overhead for these workloads. Compile-time metadata enables the scheduler to make informed decisions about `Tasks` of different sizes for the evaluated heterogeneous workloads.

6.4.2 Comparison with Triton and SwapNet

On the desktop GPU, PAGRUS achieves a geometric mean improvement of 6.61% over NVIDIA Triton Inference Server across all evaluated configurations (Figure 6.4). Triton employs model-level scheduling with a static batching mechanism that groups identical requests within a fixed batching window. In heterogeneous multi-model scenarios, batching opportunities are limited because requests target different models, resulting in frequent underutilization of GPU resources. Additionally, fixed memory buffers allocated per model instance restrict the number of concurrently active models, exacerbating memory fragmentation. For heterogeneous workloads (Figures 6.4e to 6.4g), Triton does not support GPT-2, limiting the comparison to EfficientNet and Vision Transformer. Triton is 1.93% faster on average for two-model configurations, but when running three sets of `Tenants`, PAGRUS outperforms Triton by 3.56%, demonstrating better scalability under increasing `Tenant` diversity. On the Jetson Orin Nano (Figure 6.5), PAGRUS achieves 43.81% lower latency on average compared to SwapNet. SwapNet targets OOM prevention for embedded GPUs by partitioning models into multiple stages and swapping intermediate tensors between device and host memory. While this strategy prevents OOM failures, frequent device-host transfers and non-optimized partitioning boundaries introduce substantial execution overhead. SwapNet's static partitioning



(a) On NVIDIA GeForce RTX 4090

(b) On NVIDIA Jetson Orin Nano

Figure 6.6: Breakdown of total runtime overhead across representative DNN models on both platforms. (Each abbreviation represents RN50 = ResNet50, EffN = EfficientNet, ViT = Vision Transformer, SSD = SSD-ResNet50, OP = OpenPose, DL3 = DeepLabV3, respectively.)

policy divides models such that each partition consumes equal memory without considering the model's actual memory usage pattern. As a result, SwapNet is on average 17.46% slower than `Task`-level scheduling alone, as uniform partitioning leads to unnecessary data movement. In contrast, PAGRUS determines partition boundaries based on the profiled peak memory usage pattern of each model, yielding fewer and more balanced `Tasks` with higher throughput.

6.5 Runtime Overhead Analysis

To quantify the runtime cost of the PAGRUS scheduler, the ratio of scheduling and partitioning overheads relative to the total execution time was measured (Figure 6.6). The average overheads are 4.73% on the RTX 4090 (Figure 6.6a) and 5.04% on the Jetson Orin Nano (Figure 6.6b). Each optimization feature was selectively disabled to isolate its contribution. Since runtime behavior in multi-tenant settings depends on the characteristics of each model, the ablation evaluation was performed using two concurrent `Tenants` of each model. The measured runtime overhead is categorized into two components. Scheduling overhead includes the communication cost between each `Task` and the scheduler, the time consumed for yield-based scheduling decisions, and the actual delay incurred when a `Task` deployment is deferred. This overhead increases for models partitioned into a larger number of `Tasks`. Partitioning overhead is related to the allocation latency of shared tensor pools. Since the allocation is performed synchronously, computation must be temporarily halted until the allocation completes. Models with larger shared tensor pools exhibit greater partitioning overhead. Scheduling overhead dominates in models with many fine-grained `Tasks`, whereas partitioning overhead becomes significant when large shared tensor pools are required. An exceptional case occurs in models whose final layers contain fully connected (FC) operations, such as EfficientNet. In these models, yield scheduling can defer the FC `Task` deployment even with a small number of `Tasks`, resulting in slightly higher scheduling overhead compared to other networks. On the embedded GPU, unified memory constraints make storage-backed evictions substantially slower than on desktop platforms, making the

yield strategy even more beneficial. The 5.04% average overhead on the Jetson Orin Nano is a small price for preventing the 50%+ slowdown that would result from tensor eviction.

6.6 Case Study Evaluation

Three deployment case studies extend the evaluation beyond what the single-tenant and multi-tenant benchmarks of the preceding sections cover. The first case study extends the static pool scheme to autoregressive LLM decoding, where the cache grows with the generated sequence. The second case study places a draft model and a verifier model under a single fixed memory budget, which holds two model bodies and two cache streams at once. The third case study evaluates the PAGRUS framework in server-scale serving on real NVIDIA MIG partitions, where a router distributes requests across instances of fixed memory capacity and admits each request based on the smaller resident footprint that PAGRUS reports. The first two case studies test whether the compile-time peak-memory model that PAGRUS builds for static single-shot inference extends to sequence growth and to multi-model residency, and the third measures what the framework and its compile-time metadata add to server-scale serving across hardware-partitioned instances.

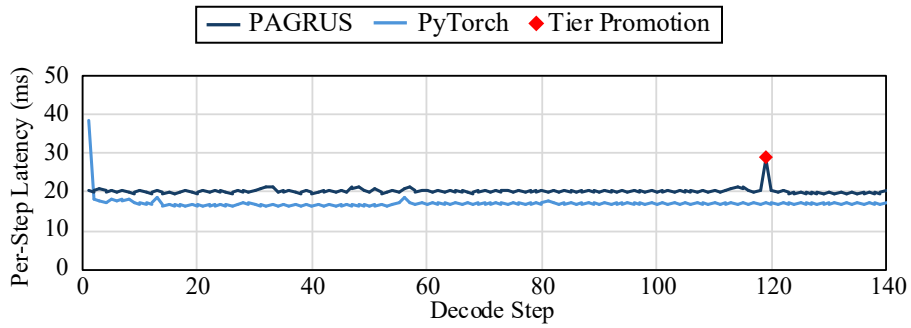
6.6.1 Autoregressive LLM Decoding

Autoregressive decoding grows its key-value (KV) cache with every generated token, so its memory footprint changes over the course of the inference. The static pool scheme extends to that growing cache, representing the KV state as a tiered pool that the runtime

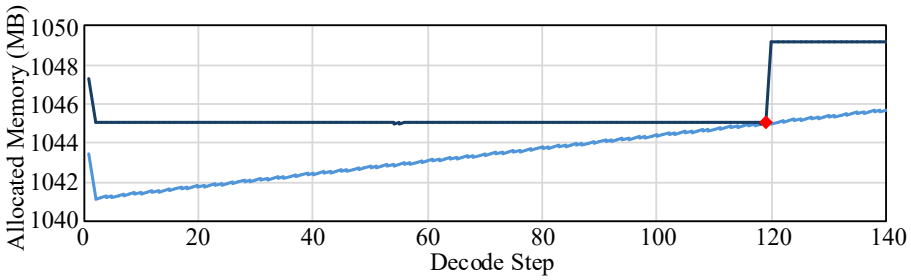
promotes to a larger tier as the sequence lengthens, and it keeps a bounded per-tier footprint while preserving the decoding output. The experiment runs a Llama 3.2 1B Instruct AWQ decoder on an NVIDIA GeForce RTX 4090, with a 10-token prompt and greedy decoding of 140 new tokens. The PyTorch baseline uses the HuggingFace model with its default KV cache. The PAGRUS configuration replaces this cache with a pool-backed progressive KV cache organized into 128, 256, 512, and 1024 token tiers, and keeps the model weights, prompt, decoding rule, and generated length identical to the baseline. Because the two configurations differ only in the cache, the generated token sequence is a direct correctness check.

The progressive KV cache separates logical sequence growth from pool addressability. Prefill writes the prompt keys and values into the active tier and returns prefix views to the attention operator, and each decode step appends one key and one value at the next position inside that tier. When the sequence crosses the current tier boundary, the runtime allocates the next tier, copies the live prefix into the corresponding offsets, and resumes fixed-offset access in the larger tier. The active tier sets the resident footprint, so the static pool holds a finite peak bound for each tier as the logical cache length grows.

The progressive cache reaches 47.50 tokens per second over the full run, against 45.62 for the native cache (Figure 6.7a). This margin comes from the first step, where prefill and one-time initialization take 591.1 ms in the native run and 53.8 ms in the already-warm progressive run, and that single gap dominates the 140-token run. In steady state the progressive cache is slower, with a median decode latency of 20.2 ms against 17.1 ms for the native cache, an overhead of about 3 ms per step from the pooled access path. The single promotion from the 128-token tier to the 256-token tier raises



(a) Per-step decode latency



(b) Allocated GPU memory

Figure 6.7: Autoregressive LLM decoding case study on the RTX 4090. (a) Per-step decode latency of the native dynamic cache and the PAGRUS progressive cache across generated tokens, with the tier promotion marked. (b) Allocated GPU memory for both caches. The marker indicates the promotion from the 128-token tier to the 256-token tier.

one step to 28.6 ms, of which 7.8 ms is the prefix copy and pool rebuild, and it occurs once over the run.

Memory usage stays bounded within each tier (Figure 6.7b). The native allocation rises from 1043.48 MB to 1045.73 MB as KV entries accumulate, while the progressive allocation holds near 1045.04 MB inside the first tier and steps to 1049.24 MB after promotion. The active KV pool doubles from 4.19 MB to 8.39 MB when the tier changes from 128 to 256 tokens. At the end of the run the progressive cache holds 3.51 MB more than the native baseline, the cost of reserving tier capacity and rebuilding the pool at the

boundary. Over 140 tokens the KV cache stays small relative to the model weights, so the two footprints are close, and the progressive cache caps its peak at the active tier capacity as the logical length grows.

The generated token sequence matches the native baseline, so the substitution preserves greedy decoding output. The static pool therefore extends to a growing KV cache, holding a per-tier memory bound across promotions at a one-time promotion cost.

6.6.2 Speculative Decoding

Speculative decoding keeps two models resident at once, a draft model and a verifier model alternating within a single request, and their weights, KV cache state, logits, sampling buffers, and decode scratch all share the same memory budget. The experiment runs a Llama 3.2 1B Instruct AWQ draft model and a Llama 3.1 8B Instruct AWQ verifier model on an NVIDIA Jetson Orin Nano under an 8 GB memory budget, executing 128 speculative steps with a draft width of five tokens. The aim is to show that one static pool holds both model bodies and both cache streams within this budget. Table 6.9 summarizes the configuration.

The two configurations differ only in the memory management mechanism. The native baseline does not use a memory pool, so it keeps only one model on the GPU at a time and migrates whole models on demand. Each speculative step therefore loads the draft model, runs the draft phase, and evicts it, then loads the verifier model, runs verification, and evicts it. The PAGRUS configuration instead allocates one model-and-decode pool, places the draft and verifier regions at fixed offsets, and copies only the required parameter blocks into reusable pool regions before each module executes. This replaces

Table 6.9: Speculative decoding workload configuration. The draft width is the number of tokens proposed before each verifier check.

Item	Configuration
Draft model	Llama 3.2 1B Instruct AWQ
Verifier model	Llama 3.1 8B Instruct AWQ
Batch size	1
Prompt length	9 tokens
Speculative steps	128
Draft width	5 tokens
Seed	0
GPU budget	8 GB
Verification path	Prefilled verifier prefix KV with suffix verification

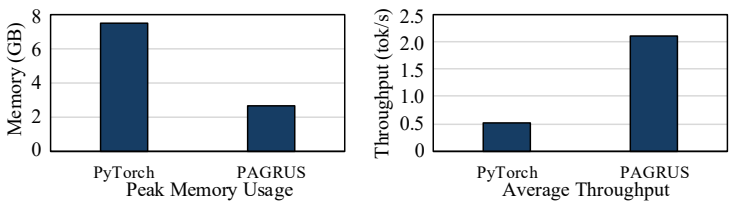


Figure 6.8: Peak memory and new-token throughput for native model-on-demand execution and PAGRUS pooled residency, on the Jetson Orin Nano under an 8 GB budget.

repeated whole-model migration between GPU and host with bounded on-demand parameter residency inside the static pool.

Figure 6.8 shows that pooled residency fits the two-model workload into the budget at a much lower peak. Native model migration reaches a peak of 7.50 GB and produces 0.525 tokens per second, while the PAGRUS configuration reaches 2.65 GB and produces 2.113 tokens per second, a 64.71% lower peak and a 4.03× higher throughput. The resident pool sets this footprint. Disabling weight streaming in an ablation leaves the peak nearly unchanged and reaches 2.136 tokens per second. Of the 2.65 GB, the draft and verifier weights and decode scratch occupy 2.61 GB, and the draft and verifier KV pools

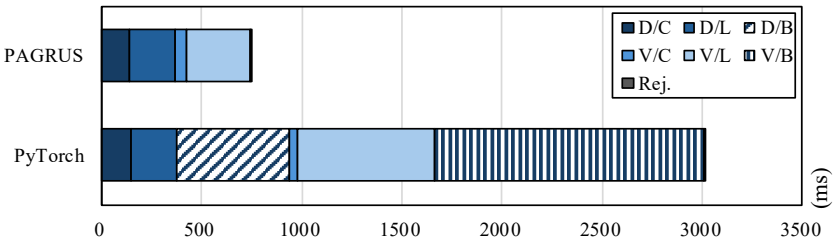


Figure 6.9: Mean per-step latency decomposed into model movement, weight loading, draft and verifier compute, and rejection. (`D/C` = draft compute, `D/L` = draft model load, `D/B` = draft backup-to-host, `V/C` = verifier compute, `V/L` = verifier model load, `V/B` = verifier backup-to-host, `Rej.` = token rejection.)

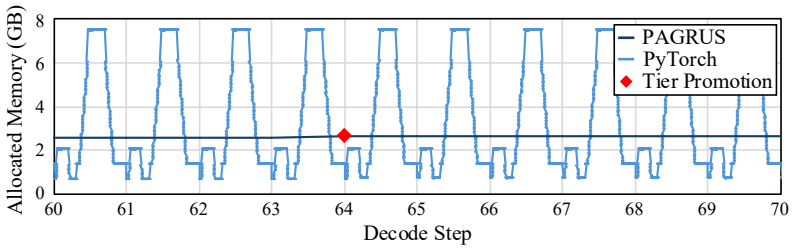


Figure 6.10: Native and PAGRUS memory usage over the decode steps, with the PAGRUS tier promotion marked.

add 41.94 MB. Reporting the KV pools separately keeps the verifier-side cache state inside the measured footprint, which a weight-only figure would omit.

Figure 6.9 shows where time goes once whole-model movement is removed. The native baseline has a mean speculative-step latency of 3007.8 ms, of which model movement accounts for 2811.6 ms across draft load, draft eviction, verifier load, and verifier eviction. The PAGRUS configuration reduces mean step latency to 744.3 ms, with on-demand weight loading inside the pool accounting for 542.9 ms, draft computation for 141.9 ms, and verifier computation for 58.6 ms. Parameter loading is now the dominant cost. The five-token autoregressive draft loop streams draft parameters once per

drafted token, so cumulative draft loading reaches the same order as verifier loading even though the draft model is smaller.

Figure 6.10 shows the per-step memory usage, with the PAGRUS tier promotion marked. The KV pools follow the same 128, 256, 512, and 1024 token tier policy as the previous case study, and the runtime promotes once at step 64 from the 128-token tier to the 256-token tier with 0.1 ms of promotion latency. The KV residency splits into 8.39 MB for the draft and 33.55 MB for the verifier. The verifier prefills the accepted prefix once, verifies only the drafted suffix against it, and rolls the cache length back to the accepted prefix after rejection sampling, which keeps verifier KV residency inside the measured footprint.

The pooled path also passes a correctness check. The PAGRUS run matches the native token sequence exactly for this seed, with a 211-token common prefix, and the mean acceptance rate is 0.316 for both native and pooled execution. The pooled path therefore preserves the decoding outcome while changing only the memory mechanism. The static pool runs the two-model workload under the fixed 8 GB budget that native execution meets only through repeated whole-model migration, and the latency breakdown attributes the remaining cost to weight streaming.

6.6.3 MIG-Constrained Serving

On real NVIDIA MIG partitions of an RTX PRO 6000 Blackwell Max-Q GPU, this case study evaluates how the PAGRUS framework benefits server-scale serving. A router distributes incoming requests across the MIG instances with a simple scoring function, and the experiment measures how much the PAGRUS framework and its compile-time meta-

Table 6.10: Mixed-8 serving workload. Each request uses the default catalog batch size. The two memory columns are the worst-case reservations the router uses for native and PAGRUS-aware admission, and Base ms is the base single-request latency that the routing score uses for each model.

Model	Application	Batch	PyTorch peak MB	PAGRUS peak MB	Base ms
MobileNetV2	Image classification	64	900	600	12
ResNet50	Image classification	32	2800	1800	42
EfficientNet-B0	Image classification	16	1900	1200	26
Vision Transformer B16	Image classification	8	3600	2300	64
DeepLabV3 ResNet50	Semantic segmentation	4	5200	3300	96
SSD-ResNet50	Object detection	2	6200	3900	110
BERT	NLP encoder	4	1200	800	38
GPT-2	Text generation	2	1600	1000	56

data improve serving over PyTorch. The workload is a Mixed-8 periodic serving stream in which each request belongs to one of the eight models in Table 6.10, spanning image classification, segmentation, detection, and language models. The experiment compares three MIG geometries, a four-way 1g layout of four 1g instances, an asymmetric layout of one 2g and two 1g instances, and a single 96 GB instance. The experiment runs both PyTorch native and PAGRUS-aware execution under identical request arrivals, and all latency numbers in this case study are route-to-done latency.

The router uses PAGRUS metadata in an admission gate that runs before scoring. For each request, the gate reserves a worst-case memory amount on every MIG instance and rejects any instance whose projected reservation would exceed 92% of its memory capacity. The native mode uses the model's measured PyTorch peak memory for this reservation, and the PAGRUS-aware mode uses the measured PAGRUS peak memory from Table 6.10, which is smaller for every model. This PAGRUS peak is the full-run resident footprint of an actual execution, including model weights, backend workspace, and shared library context, and it differs from the CP-SAT intra-`Task` tensor pool size

Table 6.11: Moderate-pressure Mixed-8 serving results under the four-way 1g and single 96 GB geometries. All 24 requests are admitted, and the latency columns report route-to-done latency.

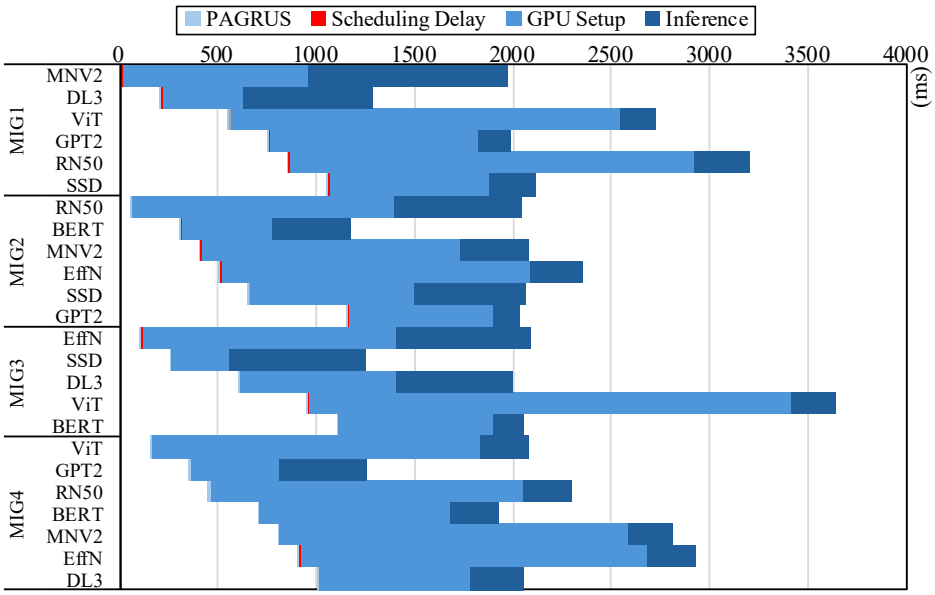
Geometry	Runtime	Makespan (ms)	Throughput (item/s)	Mean (ms)	p95 (ms)
Four-way 1g	PyTorch	3628.5	109.1	1557.3	2346.3
Four-way 1g	PAGRUS	2855.8	138.7	1626.0	2178.7
Single 96 GB	PAGRUS	3532.6	112.1	997.0	2241.8

reported in Section 6.2.2. Each feasible instance then receives a routing score, and the router dispatches the request to the instance with the highest score. The score is a simple fixed heuristic that combines a memory-headroom reward r_h with penalties for queue length p_q, service-time estimate t_r, weight-class mismatch p_s, and recent failures p_f, and a small bonus for a long-idle instance b_a,

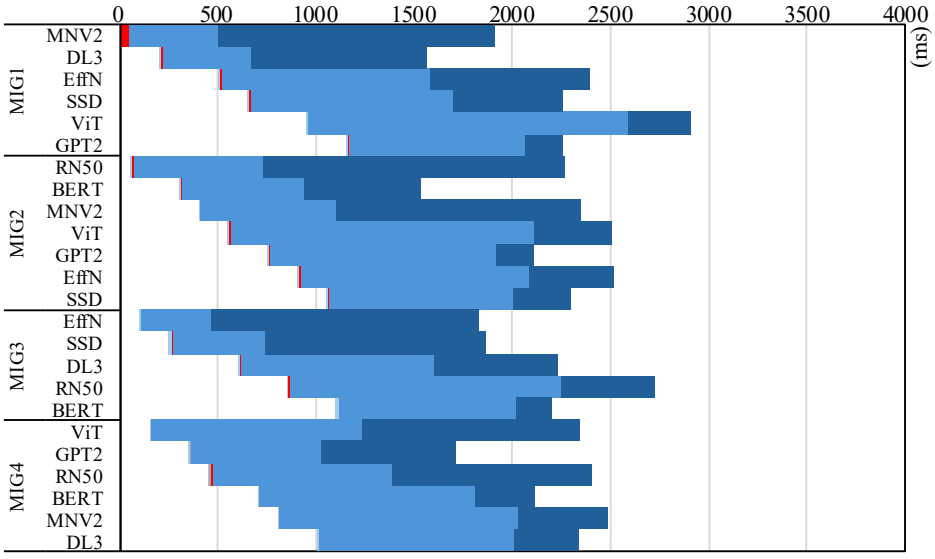
$$\text{score} = r_h - p_q - t_r - p_s - p_f + b_a, \tag{6.1}$$

where each term carries a fixed weight, tuned for these experiments and listed in Appendix B. The reward and the queue, failure, and idle terms depend on the instance state, and the service-time and weight-class terms depend on the request, so the score is both hardware-aware and workload-aware. The same scoring function and weights apply to the native and PAGRUS-aware runs, so the comparison isolates the admission footprint while holding the routing policy fixed. The gate does not drop a request under transient pressure. When every rejection reason is temporary, the request stays pending and is rescored after later completion events release reserved memory.

The moderate-pressure scenario issues three requests per model, 24 requests at 50 ms intervals, and every geometry admits all 24 (Table 6.11). The first comparison



(a) PyTorch native execution



(b) PAGRUS-aware execution

Figure 6.11: Per-request route-to-done timeline under the four-way 1g geometry, for (a) PyTorch native and (b) PAGRUS-aware execution. (MNV2 = MobileNetV2, RN50 = ResNet50, EffN = EfficientNet-B0, ViT = ViT-B/16, DL3 = DeepLabV3, SSD = SSD-ResNet50, GPT2 = GPT-2.)

Table 6.12: High-pressure Mixed-8 serving results. The four-way 1g geometry is reported for both PyTorch native and PAGRUS-aware execution, and the asymmetric 2g+1g+1g and single 96 GB geometries use PAGRUS-aware execution. All 32 requests are admitted, and the latency columns report route-to-done latency.

Geometry	Runtime	Makespan (ms)	Throughput (item/s)	Mean (ms)	p95 (ms)
Four-way 1g	PyTorch	4845.8	109.0	2458.0	3348.2
Four-way 1g	PAGRUS	3706.6	142.4	2064.0	2618.5
Asymmetric 2g+1g+1g	PAGRUS	3120.1	169.2	1649.7	2170.2
Single 96 GB	PAGRUS	4368.0	120.9	1000.2	2262.5

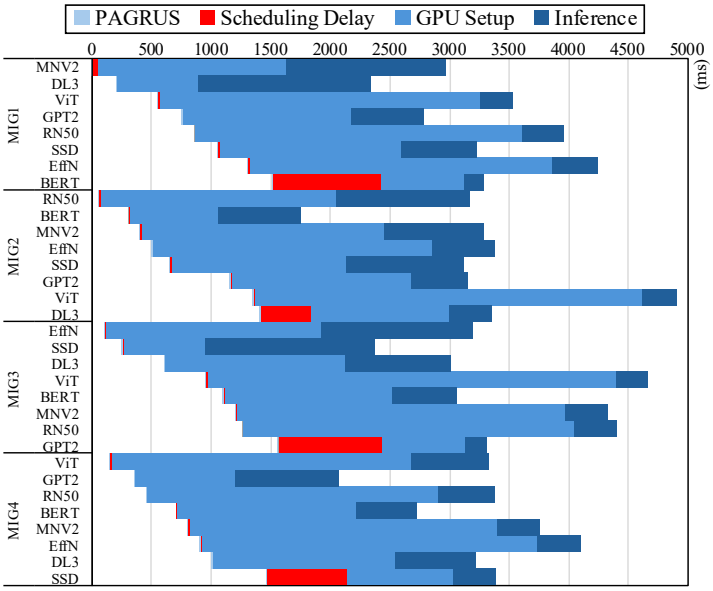
holds the runtime at PAGRUS-aware and varies the geometry. Splitting the workload across four 1g instances drains it faster than the single large instance, so the four-way layout completes in 2855.8 ms against 3532.6 ms for the single 96 GB layout, a 19.2% shorter makespan, and raises throughput from 112.1 to 138.7 items per second, a 23.7% gain, while p95 route-to-done latency drops from 2241.8 to 2178.7 ms. The single 96 GB layout keeps a lower mean per-request latency, since every request runs on one large instance, so routed serving helps at the workload level and in the tail.

The second comparison holds the four-way 1g geometry and varies the runtime, which isolates the effect of PAGRUS-aware admission (Figure 6.11). PyTorch native execution completes the same workload in 3628.5 ms at 109.1 items per second, against 2855.8 ms at 138.7 items per second for PAGRUS-aware execution. Each request reserves the smaller PAGRUS footprint, so more candidates pass admission and the headroom score improves, which shortens makespan by 21.3%, raises throughput by 27.1%, and lowers p95 latency by 7.1%. The timeline shows the cause, as native execution spends a larger share of each request on GPU setup and leaves the four lanes intermittently idle, while PAGRUS-aware execution shortens setup and packs the lanes more densely.

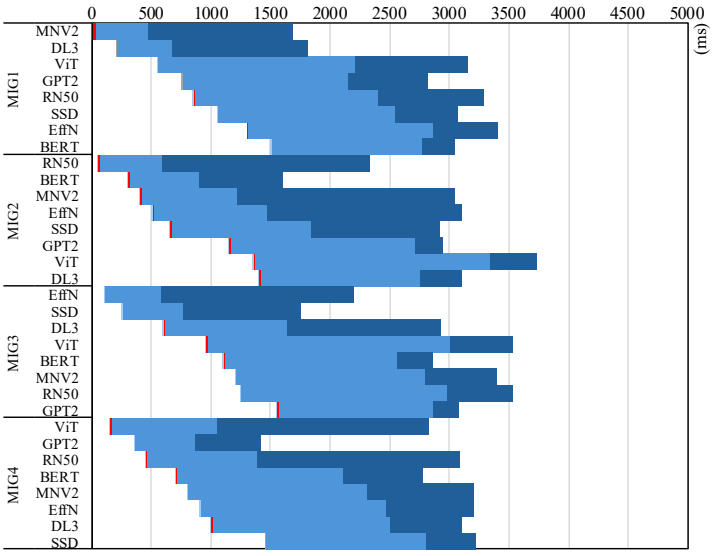
The high-pressure scenario issues four requests per model, for 32 requests at the same 50 ms interval, and all three PAGRUS-aware geometries admit every request (Table 6.12).

At this pressure the gap between native and PAGRUS-aware execution on the four-way 1g geometry is at its widest. Native execution reserves the larger PyTorch peak footprint for every request, so the four 1g instances saturate sooner and the router defers admission for several requests until earlier ones release memory. Four requests, on the BERT, GPT-2, SSD-ResNet50, and DeepLabV3 models, are deferred and wait between 0.42 and 0.90 seconds in admission before a slot frees, which serializes part of the workload. Native execution therefore completes in 4845.8 ms at 109.0 items per second, against 3706.6 ms at 142.4 items per second for PAGRUS-aware execution on the same geometry, a 23.5% shorter makespan and a 30.6% throughput gain, with mean and p95 route-to-done latency lower by 16.0% and 21.8%. Figure 6.12 shows the mechanism, as the native lanes carry long admission-yield gaps before several requests while the PAGRUS-aware lanes pack the same 32 requests more densely.

The best geometry shifts as pressure rises. The asymmetric 2g+1g+1g layout reaches 169.2 items per second with a 3120.1 ms makespan and 2170.2 ms p95 latency, against 142.4 items per second, a 3706.6 ms makespan, and 2618.5 ms p95 latency for the four-way 1g layout. Relative to the four-way layout, the asymmetric layout improves throughput by 18.8%, shortens makespan by 15.8%, and lowers p95 latency by 17.1%. Relative to the single 96 GB layout, it improves throughput by 40.0%, shortens makespan by 28.6%, and lowers p95 latency by 4.1%. The best partition therefore depends on workload pressure and model mix. The four-way layout splits memory into four equal 1g slices, which



(a) PyTorch native execution



(b) PAGRUS-aware execution

Figure 6.12: Per-request route-to-done timeline under the four-way 1g geometry at high pressure (four requests per model, 32 requests), for (a) PyTorch native and (b) PAGRUS-aware execution. (MNV2 = MobileNetV2, RN50 = ResNet50, EffN = EfficientNet-B0, ViT = ViT-B/16, DL3 = DeepLabV3, SSD = SSD-ResNet50, GPT2 = GPT-2.)

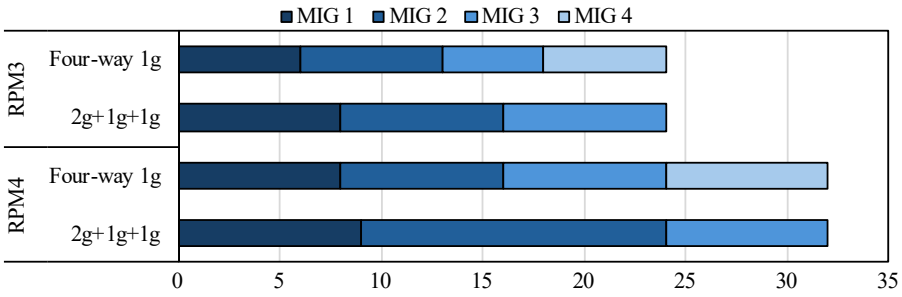


Figure 6.13: Per-instance request distribution under the four-way 1g and asymmetric 2g+1g+1g geometries at moderate and high pressure. Moderate and high pressure are labeled RPM3 (three requests per model, 24 requests) and RPM4 (four requests per model, 32 requests) in the figure. The four-way layout spreads requests evenly across its instances, whereas the asymmetric layout concentrates load on the 2g slice as pressure rises.

suits the lighter models under moderate pressure. Under high pressure the heavier models, whose base duration is at least 42 ms, need more room than a single 1g slice gives, so the right-sizing penalty routes them to the 2g slice of the asymmetric layout while the lighter models stay on its 1g slices.

Figure 6.13 shows how the routing score spreads requests across the available instances. The four-way layout distributes requests evenly under both pressures, as 6, 7, 5, and 6 at moderate pressure and 8 per instance at high pressure. The asymmetric layout routes almost evenly at moderate pressure, where ample headroom gives the score little reason to concentrate load, but under high pressure it sends a disproportionate share to the 2g slice, placing 9, 15, and 8 requests across its three instances through the right-sizing term. This pressure-dependent skew is why the asymmetric layout improves workload-level metrics only under high pressure.

7. Conclusion and Future Work

This study presented PAGRUS, an end-to-end DNN inference compiler and scheduler that shifts memory management decisions from runtime to compile time. The central thesis is that the static analyzability of DNN inference models, with their fixed computation graphs and deterministic tensor shapes, enables a compile-time approach to memory management that improves memory efficiency and, in most cases, execution performance relative to runtime approaches. The system makes six contributions that span the full compilation and execution pipeline.

First, a comprehensive tensor liveness analysis captures both visible tensors in the computation graph and hidden temporary workspace buffers allocated internally by backend GPU libraries, producing a unified liveness table that prevents unexpected OOM errors from unaccounted allocations. Second, two operation-level optimizations, layer fusion and tensor coalescing, reduce the number of live tensors and eliminate redundant memory operations. Third, a constraint programming formulation solves the Dynamic Storage Allocation problem exactly for each intra-`Task` memory pool, producing provably optimal memory layouts that replace heuristic algorithms with an optimality guarantee. Fourth, the Tailor-and-Stitch algorithm partitions DNN models into memory-balanced `Tasks` based on compile-time peak memory analysis, enabling fine-grained multi-tenant scheduling without layer-level overhead. Fifth, a dual memory

pool architecture retains shared tensors in GPU-resident memory across `Task` boundaries, eliminating the host-device transfer overhead that conventional backup-and-restore approaches incur. Sixth, a compiler-generated scheduler interface and yield-based scheduling policy prevent OOM errors through proactive worst-case analysis, deferring `Task` deployment when memory exhaustion is anticipated rather than evicting tensors after the fact.

The evaluation validates these contributions across single-tenant and multi-tenant deployment, complemented by three deployment case studies. In single-tenant evaluation on the NVIDIA RTX 3090 and Jetson AGX Xavier, the proposed system reduces memory usage by an average of 34.6% compared to PyTorch while achieving a geometric mean speedup of 1.25×. The CP-SAT exact solver produces provably optimal pool layouts for all evaluated models, with solve times under 5 seconds for models with approximately 1,500 tensors. The memory reduction is sufficient to enable SRGAN execution on the Jetson AGX Xavier, where PyTorch fails with an OOM error. In multi-tenant evaluation on the NVIDIA RTX 4090 and Jetson Orin Nano, the system achieves OOM-free concurrent execution with an average performance overhead of 3.20% on the desktop GPU and 5.04% on the embedded GPU, relative to single-tenant baselines. Without the yield strategy and shared tensor pool, `Task`-level scheduling incurs up to 50.54% worst-case latency degradation from eviction overhead. Compared to existing deployment solutions, the system provides up to 10.8% lower latency than the NVIDIA Triton Inference Server and 43.81% lower latency than SwapNet on embedded platforms. Three deployment case studies further extend these results to practical settings, covering autoregressive LLM decoding with a progressively growing activation tensor, two-model

speculative decoding under a single embedded memory budget, and memory-aware request routing across hardware-partitioned MIG instances. The unified compilation pipeline serves both single-tenant and multi-tenant deployment, with single-tenant as the degenerate case of one `Task` per model and no scheduler.

Several limitations of the current system suggest directions for future work. The compile-time approach assumes static input shapes, as tensor sizes must be known at compile time to perform liveness analysis and pool layout optimization. Supporting dynamic input shapes would require a tier-based precompilation strategy, where the compiler generates multiple pool layouts for representative input shape ranges and the runtime selects the appropriate layout at deployment. The current implementation targets NVIDIA GPUs exclusively, relying on CUDA APIs and the cuDNN and cuBLAS backend libraries. Extending the system to other accelerator platforms would require backend-specific profiling passes for hidden allocation tracking, though the compiler infrastructure (liveness analysis, CP-SAT solver, partitioning) is platform-independent. The hidden allocation profiling is specific to the installed library versions, requiring re-profiling when backend libraries are updated. However, this profiling is a one-time cost per library version and can be automated as part of the build process.

Future work can extend this system in several directions. Integration with model compression techniques such as INT8 quantization and FP16 mixed-precision inference would further reduce memory footprint while preserving the benefits of compile-time pool optimization. Extending the partitioning and scheduling framework to distributed multi-GPU inference would enable memory-aware workload distribution across multiple devices, combining the Tailor-and-Stitch algorithm with inter-device communica-

tion cost models. An adaptive tier management system could automatically generate and select pool layouts for different input shape configurations, bridging the gap between static compilation and dynamic deployment requirements. Formal verification of the OOM-freedom property would provide mathematical guarantees that the yield-based scheduler, combined with compile-time worst-case analysis, prevents memory exhaustion under all possible scheduling interleavings. Finally, hardware-aware partitioning for heterogeneous accelerator platforms with mixed GPU and NPU resources would enable the compiler to exploit the distinct memory hierarchies and compute capabilities of each accelerator type.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, Jun. 2016.

[2] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint*, 2017.

[3] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Sep. 2019, pp. 6105–6114.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot MultiBox detector," in *Proceedings of the European Conference on Computer Vision*, Springer International Publishing, 2016, pp. 21–37.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 779–788.

[6] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, “Photo-realistic single image super-resolution using a generative adversarial network,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, Jul. 2017.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*, 2019, pp. 4171–4186.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “PyTorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems*. Curran Associates Inc., 2019.

[9] NVIDIA Corporation, *NVIDIA Jetson Orin*, [Online], Available: `https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/`, 2025.

[10] K. Bhardwaj, C.-Y. Lin, A. Sartor, and R. Marculescu, “Memory- and communication-aware model compression for distributed deep learning inference on iot,” *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 5s, Oct. 2019.

[11] J. Lee, J. M. Lee, H. Jeong, H. Kwon, Y. Kim, Y. Park, and H. Kim, “Peak-memory-aware partitioning and scheduling for multi-tenant DNN model inference,” *Journal of Systems Architecture*, vol. 173, p. 103 696, 2026.

[12] NVIDIA Corporation, *NVIDIA cuDNN documentation*, [Online], Available: `https://docs.nvidia.com/deeplearning/cudnn/backend/latest/api/cudnn-cnn-library.html#cudnngetconvolutionforwardworkspacesize`, 2025.

[13] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, R. Rhodes, T. Wang, and P. Warden, “TensorFlow Lite Micro: Embedded machine learning on TinyML systems,” in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 800–811.

[14] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, “vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[15] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian, “Capuchin: Tensor-based GPU memory management for deep learning,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne, Switzerland, 2020, pp. 891–905, ISBN: 9781450371025.

[16] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, “SuperNeurons: Dynamic GPU memory management for training deep neural networks,” in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vienna, Austria: Association for Computing Machinery, 2018, pp. 41–53, ISBN: 9781450349826.

[17] Y. Wen, A. Anderson, V. Radu, M. F. O’Boyle, and D. Gregg, “TASO: Time and space optimization for memory-constrained DNN inference,” in *2020 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 199–208.

[18] The XLA Authors, *XLA: Optimizing compiler for machine learning*, `https://openxla.org/xla`, 2025.

[19] I. Gim and J. Ko, “Memory-efficient DNN training on mobile devices,” in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022, pp. 464–476.

[20] Z. Zhao, N. Ling, N. Guan, and G. Xing, “Miriam: Exploiting elastic kernels for real-time multi-DNN inference on edge GPU,” in *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*, 2023, pp. 97–110.

[21] L. Han, Z. Zhou, and Z. Li, “Pantheon: Preemptible multi-DNN inference on mobile edge GPUs,” in *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, 2024, pp. 465–478.

[22] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, “Autoware on Board: Enabling autonomous vehicles with embedded systems,” in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, IEEE, 2018, pp. 287–296.

[23] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu, “Co-optimizing performance and memory footprint via integrated CPU/GPU memory management, an implementation on autonomous driving platform,” in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 310–323.

[24] A. Wilson, A. Kumar, A. Jha, and L. R. Cenkeramaddi, “Embedded sensors, communication technologies, computing platforms and machine learning for uavs: A review,” *IEEE Sensors Journal*, vol. 22, no. 3, pp. 1807–1826, 2022.

[25] M. A. Khan, H. Menouar, A. Eldeeb, A. Abu-Dayya, and F. D. Salim, "On the detection of unauthorized drones—techniques and future perspectives: A review," *IEEE Sensors Journal*, vol. 22, no. 12, pp. 11 439–11 455, 2022.

[26] NVIDIA Corporation, *NVIDIA Triton Inference Server*, [Online], Available: `https://developer.nvidia.com/triton-inference-server`, 2024.

[27] G. Alexopoulos and D. Mitropoulos, "nvshare: Practical GPU sharing without memory size constraints," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 16–20, ISBN: 9798400705021.

[28] X. Wu, J. Rao, W. Chen, H. Huang, C. Ding, and H. Huang, "SwitchFlow: Preemptive multitasking for deep learning," in *Proceedings of the 22nd International Middleware Conference*, Québec city, Canada: Association for Computing Machinery, 2021, pp. 146–158, ISBN: 9781450385343.

[29] G. Yeung, D. Borowiec, A. Friday, R. Harper, and P. Garraghan, "Towards GPU utilization prediction for cloud deep learning," in *12th USENIX Workshop on Hot Topics in Cloud Computing*, USENIX Association, Jul. 2020.

[30] F. Yu, S. Bray, D. Wang, L. Shangguan, X. Tang, C. Liu, and X. Chen, "Automated runtime-aware scheduling for multi-tenant DNN inference on GPU," in *2021*

IEEE/ACM International Conference On Computer Aided Design (ICCAD), Munich, Germany: IEEE Press, 2021, pp. 1–9.

[31] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun, "Nimble: Lightweight and parallel GPU task scheduling for deep learning," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Vancouver, BC, Canada: Curran Associates Inc., 2020, ISBN: 9781713829546.

[32] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 392–405.

[33] Y. Choi and M. Rhu, "PREMA: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 220–233.

[34] Y. H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D. U. Kim, T. J. Ham, and J. W. Lee, "LayerWeaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 584–597.

[35] K. Wang, J. Cao, Z. Zhou, and Z. Li, “SwapNet: Efficient swapping for DNN inference on edge AI devices beyond the memory budget,” *IEEE Transactions on Mobile Computing*, 2024.

[36] J. Yi and Y. Lee, “Heimdall: Mobile GPU coordination platform for augmented reality applications,” in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, London, United Kingdom: Association for Computing Machinery, 2020, ISBN: 9781450370851.

[37] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, “CoEdge: Cooperative DNN inference with adaptive workload partitioning over heterogeneous edge devices,” *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2021.

[38] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems,” *arXiv preprint*, 2015.

[39] A. J. Calderón, L. Kosmidis, C.-F. Nicolás, and F. J. Cazorla, “XeroZerox: Analysis and optimization of GPU memory management for high-integrity autonomous systems,” *IEEE Access*, vol. 12, pp. 77 141–77 155, 2024.

[40] H. Mccoy and P. Pandey, “Gallatin: A general-purpose GPU memory manager,” in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Prac-*

tice of Parallel Programming, Edinburgh, United Kingdom: Association for Computing Machinery, 2024, pp. 364–376, ISBN: 9798400704352.

[41] C. Ji, F. Wu, Z. Zhu, L.-P. Chang, H. Liu, and W. Zhai, "Memory-efficient deep learning inference with incremental weight loading and data layout reorganization on edge systems," *Journal of Systems Architecture*, vol. 118, p. 102 183, 2021.

[42] X. Nie, X. Miao, Z. Yang, and B. Cui, "TSplit: Fine-grained GPU memory management for efficient DNN training via tensor splitting," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 2615–2628.

[43] D. Yang and D. Cheng, "Efficient GPU memory management for nonlinear DNNs," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, Stockholm, Sweden: Association for Computing Machinery, 2020, pp. 185–196, ISBN: 9781450370523.

[44] M. Winter, M. Parger, D. Mlakar, and M. Steinberger, "Are dynamic memory managers on GPUs slow? A survey and benchmarks," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 219–233, ISBN: 9781450382946.

[45] I. Gelado and M. Garland, “Throughput-oriented GPU memory allocation,” in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 27–37, ISBN: 9781450362252.

[46] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, “Scheduling page table walks for irregular GPU applications,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Los Angeles, California: IEEE Press, 2018, pp. 180–192, ISBN: 9781538659847.

[47] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, “An asymmetric distributed shared memory model for heterogeneous parallel systems,” in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2010, pp. 347–358, ISBN: 9781605588391.

[48] J. Jung, D. Park, Y. Do, J. Park, and J. Lee, “Overlapping host-to-device copy and computation using hidden unified memory,” in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California: Association for Computing Machinery, 2020, pp. 321–335, ISBN: 9781450368186.

[49] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: A performance and predictability perspective," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 213–225.

[50] A. A. Awan, C.-H. Chu, H. Subramoni, X. Lu, and D. K. Panda, "Oc-DNN: Exploiting advanced unified memory capabilities in CUDA 9 and Volta GPUs for out-of-core DNN training," in *2018 IEEE 25th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2018, pp. 143–152.

[51] C. Guo, R. Zhang, J. Xu, J. Leng, Z. Liu, Z. Huang, M. Guo, H. Wu, S. Zhao, J. Zhao, and K. Zhang, "GMLake: Efficient and transparent GPU memory defragmentation for large-scale DNN training with virtual memory stitching," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 450–466.

[52] B. Steiner, M. Elhoushi, J. Kahn, and J. Hegarty, "MODeL: Memory optimizations for deep learning," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 202, PMLR, 2023, pp. 32 618–32 632.

[53] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End op-

timizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594, ISBN: 978-1-939133-08-3.

[54] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 47–62.

[55] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018. arXiv: 1805.00907 [cs.PL].

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

[57] T. Jin, G.-T. Bercea, T. D. Le, T. Chen, G. Su, H. Imai, Y. Negishi, A. Leu, K. O'Brien, K. Kawachiya, *et al.*, "Compiling ONNX neural network models using MLIR," *arXiv preprint arXiv:2008.08272*, 2020.

[58] J. Lee, S. Jeong, S. Song, K. Kim, H. Choi, Y. Kim, and H. Kim, “Occamy: Memory-efficient GPU compiler for DNN inference,” in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[59] C. Reaño, F. Silla, D. S. Nikolopoulos, and B. Varghese, “Intra-node memory safe GPU co-scheduling,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 1089–1102, 2018.

[60] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, “MISO: Exploiting multi-instance GPU capability on multi-tenant GPU clusters,” in *Proceedings of the 13th Symposium on Cloud Computing*, San Francisco, California, 2022, pp. 173–189, ISBN: 9781450394147.

[61] S. Kim, H. Genc, V. V. Nikiforov, K. Asanović, B. Nikolić, and Y. S. Shao, “MoCA: Memory-centric, adaptive execution for multi-tenant deep neural networks,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2023, pp. 828–841.

[62] N. Ling, K. Wang, Y. He, G. Xing, and D. Xie, “RT-mDL: Supporting real-time mixed deep learning tasks on edge platforms,” in *Proceedings of the 19th ACM conference on embedded networked sensor systems*, 2021, pp. 1–14.

[63] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "MASK: Redesigning the GPU memory hierarchy to support multi-application concurrency," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2018, pp. 503–518.

[64] E. Baek, E. Lee, T. Kang, and J. Kim, "STFusion: Fast and flexible multi-NN execution using spatio-temporal block fusion and memory management," *IEEE Transactions on Computers*, vol. 72, no. 4, pp. 1194–1207, 2023.

[65] F. Strati, X. Ma, and A. Klimovic, "Orion: Interference-aware, fine-grained GPU sharing for ML applications," in *Proceedings of the Nineteenth European Conference on Computer Systems*, Association for Computing Machinery, 2024, pp. 1075–1092.

[66] P. Yu and M. Chowdhury, "Fine-grained GPU sharing primitives for deep learning applications," in *Proceedings of Machine Learning and Systems*, 2020.

[67] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 119–130.

[68] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 193–204.

[69] X. Wu, H. Xu, and Y. Wang, "Irina: Accelerating DNN inference with efficient online scheduling," in *Proceedings of the 4th Asia-Pacific Workshop on Networking*, 2020, pp. 36–43.

[70] Y. Yao, S. Liu, S. Wu, J. Wang, J. Ni, G. Yang, and Y. Zhang, "WAMP^{2}S: Workload-aware GPU performance model based pseudo-preemptive real-time scheduling for the airborne embedded system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2767–2780, 2022.

[71] B. Wu, X. Liu, X. Zhou, and C. Jiang, "FLEP: Enabling flexible and efficient preemption on GPUs," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, 2017, pp. 483–496.

[72] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, USENIX Association, 2022, pp. 199–216.

[73] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, 2017, pp. 615–629.

[74] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 1–15, ISBN: 9781450368735.

[75] J. Park, H. Kwon, S. Kim, J. Lee, M. Ha, E. Lim, M. Imani, and Y. Kim, "QuiltNet: Efficient deep learning inference on multi-chip accelerators using model partitioning," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, San Francisco, California: Association for Computing Machinery, 2022, pp. 1159–1164, ISBN: 9781450391429.

[76] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Association for Computing Machinery, 2011, pp. 233–248.

[77] Z. Huai, B. Ding, H. Wang, M. Geng, and L. Zhang, “Towards deep learning on resource-constrained robots: A crowdsourcing approach with model partition,” in *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, IEEE, 2019, pp. 989–994.

[78] A. K. Kakolyris, M. Katsaragakis, D. Masouros, and D. Soudris, “RoaD-RuNNer: Collaborative DNN partitioning and offloading on heterogeneous edge systems,” in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[79] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, “LaLaRAND: Flexible layer-by-layer CPU/GPU scheduling for real-time DNN tasks,” in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 329–341.

[80] M. Gao, R. Shen, L. Shi, W. Qi, J. Li, and Y. Li, “Task partitioning and offloading in DNN-task enabled mobile edge computing networks,” *IEEE Transactions on Mobile Computing*, vol. 22, no. 4, pp. 2435–2445, 2023.

[81] Q. Wang, M. Xu, C. Jin, X. Dong, J. Yuan, X. Jin, G. Huang, Y. Liu, and X. Liu, “Melon: Breaking the memory wall for resource-efficient on-device machine learning,” in

Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services, 2022, pp. 450–463.

[82] M. Ji, S. Yi, C. Koo, S. Ahn, D. Seo, N. Dutt, and J.-C. Kim, "Demand layering for real-time DNN inference with minimized memory usage," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2022, pp. 291–304.

[83] R. Chen, Z. Ding, S. Zheng, C. Zhang, J. Leng, X. Liu, and Y. Liang, "MAGIS: Memory optimization via coordinated graph transformation and scheduling for DNN," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, La Jolla, CA, USA, 2024.

[84] X. Li, Y. Li, Y. Li, T. Cao, and Y. Liu, "FlexNN: Efficient and adaptive DNN inference on memory-constrained edge devices," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 709–723.

[85] J. S. Jeong, J. Lee, D. Kim, C. Jeon, C. Jeong, Y. Lee, and B.-G. Chun, "Band: Coordinated multi-DNN inference on heterogeneous mobile processors," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, Portland, Oregon, 2022, pp. 235–247, ISBN: 9781450391856.

[86] M. Han and W. Baek, "HERTI: A reinforcement learning-augmented system for efficient real-time inference on heterogeneous embedded systems," in *2021 30th*

International Conference on Parallel Architectures and Compilation Techniques (PACT), 2021, pp. 90–102.

[87] M. Han, J. Hyun, S. Park, J. Park, and W. Baek, "MOSAIC: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 165–177.

[88] K. Y. Shin, H.-J. Jeong, and S.-M. Moon, "Enhanced partitioning of DNN layers for uploading from mobile devices to edge servers," in *The 3rd International Workshop on Deep Learning for Mobile Systems and Applications*, Seoul, Republic of Korea: Association for Computing Machinery, 2019, pp. 35–40, ISBN: 9781450367714.

[89] J. Jung, J. Kim, and J. Lee, "DeepUM: Tensor migration and prefetching in unified memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 207–221.

[90] H. Zhang, Y. Zhou, Y. Xue, Y. Liu, and J. Huang, "G10: Enabling an efficient unified GPU memory and storage architecture with smart tensor migrations," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 395–410.

[91] C.-C. Huang, G. Jin, and J. Li, "SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 1341–1355, ISBN: 9781450371025.

[92] W. Kang, J. Lee, Y. Lee, S. Oh, K. Lee, and H. S. Chwa, "RT-Swap: Addressing GPU memory bottlenecks for real-time multi-DNN inference," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE Computer Society, 2024, pp. 373–385.

[93] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, and X. Liu, "DeepWear: Adaptive local offloading for on-wearable deep learning," *IEEE Transactions on Mobile Computing*, vol. 19, no. 2, pp. 314–330, 2020.

[94] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 611–626.

[95] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "FlashAttention: Fast and memory-efficient exact attention with IO-awareness," in *Advances in Neural Information Processing Systems*, vol. 35, Curran Associates, Inc., 2022, pp. 16 344–16 359.

[96] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration," in *Proceedings of Machine Learning and Systems*, vol. 6, 2024, pp. 87–100.

[97] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "SmoothQuant: Accurate and efficient post-training quantization for large language models," in *Proceedings of the 40th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 202, PMLR, 2023, pp. 38 087–38 099.

[98] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMs," in *Advances in Neural Information Processing Systems*, vol. 36, Curran Associates, Inc., 2023, pp. 10 088–10 115.

[99] C. Shi, B. Wei, S. Wei, W. Wang, H. Liu, and J. Liu, "A quantitative discriminant method of elbow point for the optimal number of clusters in clustering algorithm," *EURASIP Journal on Wireless Communications and Networking*, vol. 2021, no. 1, Feb. 2021.

Appendices

DNN Dialect Operations

Table A.1 lists all operations defined in the PAGRUS DNN dialect, organized by functional category. The dialect is implemented as an MLIR dialect using TableGen definitions and provides a set of operations for DNN inference compilation targeting NVIDIA GPUs.

The memory management category contains the operations that the PAGRUS compiler inserts during the ONNX dialect to DNN dialect conversion pass (Section 4.1) and later transforms during pool generation (Section 4.4). In the initial DNN dialect, `dnn.malloc` / `dealloc` operations explicitly manage device memory for each tensor. After pool optimization, these are replaced by a single `dnn.mempool-init` followed by `dnn.mem-offset` operations that assign each tensor a fixed offset within the pre-allocated pool. The `dnn.memcpy` operation supports four CUDA transfer modes (H2D, D2H, D2D, and default unified addressing).

The computation operations fall into two groups. The cuDNN/cuBLAS operations (`dnn.convfwd`, `dnn.conv-bias-relufwd`, `dnn.activationfwd`, pooling, and matrix multiplication) delegate to vendor-optimized library routines and may require hidden workspace allocations tracked by the compiler's profiling pass (Section 4.1). The `dnn.conv-bias-relufwd` operation is the fused variant produced by the layer fusion optimization (Section 4.2.1). Element-wise operations (`dnn.add`, `dnn.mul`, `dnn.sub`, etc.) are implemented as custom CUDA kernels and are candidates for tensor coalescing (Section 4.2.2),

where the output tensor reuses the memory of an input tensor. The scheduling operations are inserted only during multi-tenant compilation. `dnn.model-PARTITIONING` marks the `Task` boundaries determined by the Tailor-and-Stitch algorithm (Section 4.3). `dnn.schedule_queue` registers each `Task` with the runtime scheduler, passing the compile time metadata (peak memory usage, estimated execution time, and per-`Task` memory array) that the scheduler uses for yield-based admission control (Section 5.2.2).

Listing notation and the scheduler interface. The running-example IR listings in Section 4 use a compact `DNN.*` notation chosen for readability, whereas the implemented dialect uses the lower-case `dnn.*` operation names of Table A.1. Table A.2 maps each notation used in the listings to its corresponding dialect operation. The scheduler-facing operations that the listings write as `DNN.Signal-send` and `DNN.Signal-recv` are not separate dialect operations. The implemented dialect expresses all scheduler communication through the single `dnn.schedule_queue` operation. When this operation is lowered, it emits a call to the runtime scheduling routine that reports the `Task` memory state to the scheduler and, once admission is granted, releases the waiting process through a per-process semaphore. The conceptual send and receive directions in the listings therefore correspond to the two halves of one `dnn.schedule_queue` exchange, the outbound metadata report and the inbound admission decision.

Table A.1: Complete list of DNN dialect operations for inference. Memory management operations control GPU memory lifecycle and pool addressing. Computation operations wrap cuDNN, cuBLAS, and custom CUDA kernels. Shape manipulation operations are zero-copy metadata transformations. Scheduling operations support multi-tenant deployment.

Category	Operation	Description
Memory Management	`dnn.malloc(size)`	Allocate device memory buffer
	`dnn.dealloc(devPtr)`	Release device memory buffer
	`dnn.host_malloc(size)`	Allocate host memory buffer
	`dnn.host_dealloc(hostPtr)`	Release host memory buffer
	`dnn.host_ptr_store(ptrA, ptrB)`	Store pointer A into pointer B
	`dnn.memcpy(dst, src, count, mode)`	Data transfer (H2D / D2H / D2D)
	`dnn.mempool-init(size)`	Create pre-allocated memory pool
	`dnn.mem-offset(base, offset, size)`	Constant-offset addressing within pool
	`dnn.handle()`	Create cuDNN library handle
cuDNN/cuBLAS Computation	`dnn.convfwd`	Forward convolution (cuDNN)
	`dnn.conv-bias-relufwd`	Fused convolution + bias + ReLU (cuDNN)
	`dnn.activationfwd`	Activation (sigmoid, ReLU, tanh, ELU)
	`dnn.maxpool`	Max pooling
	`dnn.averagepool`	Average pooling
	`dnn.softmax`	Softmax along specified axis
	`dnn.reduce`	Tensor reduction (sum, max, avg, norm)
	`dnn.matmul2d / matmulnd`	Matrix multiplication (2D via cuBLAS, N-D)
Element-wise Computation	`dnn.add`	Element-wise addition
	`dnn.sub`	Element-wise subtraction
	`dnn.mul`	Element-wise multiplication
	`dnn.div`	Element-wise division
	`dnn.negative`	Element-wise negation
	`dnn.reciprocal`	Element-wise reciprocal
	`dnn.sqrt`	Element-wise square root
	`dnn.pow`	Element-wise power
	`dnn.erf`	Gauss error function
	`dnn.prelu / leakyrelu`	Parametric and leaky ReLU
	`dnn.clip`	Clamp values to interval
	`dnn.cast`	Data type casting
Shape Manipulation	`dnn.reshape`	Reshape tensor dimensions
	`dnn.flatten`	Flatten to 2D
	`dnn.transpose`	Permute dimensions
	`dnn.squeeze / unsqueeze`	Remove or insert size-1 dimensions
	`dnn.expand`	Broadcast following shape rules
	`dnn.concat`	Concatenate along axis
	`dnn.split`	Split into N tensors along axis
	`dnn.gather / nonzero`	Index-based selection
Scheduling / Multi-Tenant	`dnn.model-PARTITIONING`	Mark `Task` partition boundary
	`dnn.schedule_queue`	Register `Task` with runtime scheduler

Table A.2: Mapping between the compact operation notation used in the running-example IR listings (Section 4) and the corresponding DNN dialect operations (Table A.1).

Listing notation (Section 4)	**DNN dialect operation**
`DNN.Malloc` / `DNN.Dealloc`	`dnn.malloc` / `dnn.dealloc`
`DNN.Host-malloc`	`dnn.host_malloc`
`DNN.Memcpy`	`dnn.memcpy`
`DNN.Pool-init`	`dnn.mempool-init`
`DNN.Mem-offset`	`dnn.mem-offset`
`DNN.Conv`	`dnn.convfwd`
`DNN.Conv-relu`	`dnn.conv-bias-relufwd`
`DNN.Relu`	`dnn.activationfwd`
`DNN.Add`	`dnn.add`
`DNN.Model-partition`	`dnn.model-PARTITIONING`
`DNN.Signal-send` / `DNN.Signal-recv`	`dnn.schedule_queue`

MIG Routing Score

The MIG-constrained serving case study (Section 6.6.3) routes each request with a simple fixed-weight score, score $= r_h - p_q - t_r - p_s - p_f + b_a$, where the reward and bonus enter with a plus sign and the penalties with a minus sign. Table B.1 gives the weight magnitude of each term and the quantity it captures. The same weights apply to the native and PAGRUS-aware runs, so they are held fixed across the comparison and are listed here for reproducibility.

Table B.1: Fixed weights of the MIG routing score and the quantity each term captures.

Symbol	Term	Weight	Captures
r_h	Headroom reward	500	free fraction of the 92% safety budget on the instance
p_q	Queue penalty	175	outstanding requests beyond the queue limit
t_r	Service-time penalty	0.75	model-dependent and batch-dependent latency estimate
p_s	Right-sizing penalty	160	mismatch between model weight class and instance capacity
p_f	Failure penalty	500	recent out-of-memory or timeout events on the instance
b_a	Idle bonus	50	long-idle instance, for fairness

The weights follow a simple rule. The two memory-safety terms carry the largest magnitude, so the router strongly prefers an instance with spare memory through r_h and strongly avoids an instance with a recent failure through p_f. The queue penalty is the next largest, so among safe instances the router balances load. The right-sizing penalty steers a heavy model toward a larger slice and a light model toward a smaller one. The service-time estimate carries a small weight and mainly breaks ties by expected

duration, and the idle bonus adds a mild fairness preference. Instance-side memory safety therefore dominates the decision, load balance comes next, and the request-side terms refine the choice.

Abstract in Korean

최대 사용 메모리 인식 다중 테넌트 DNN 모델 추론을 위한 모델 분할 및 스케줄링

GPU 기반 심층 신경망(DNN) 추론 시스템은 메모리 관리에 상당한 시간을 소모한다. CUDA API를 통한 동적 메모리 할당이 전체 추론 시간의 33%에서 50%를 차지하는 반면, 실제 연산은 20%에서 25%에 그친다. 이 문제는 다중 모델 동시 실행 환경에서 더욱 심각해진다. cuDNN, cuBLAS 등 백엔드 라이브러리가 내부적으로 할당하는 임시 작업 공간(workspace)이 그래프 수준 분석에 드러나지 않아 예기치 않은 메모리 부족(OOM) 오류를 일으키기 때문이다.

본 연구는 GPU 메모리를 실행 이전에 계획하는 DNN 추론 컴파일러 및 스케줄러를 제안한다. 추론 모델은 고정된 연산 그래프와 결정론적 텐서 크기를 가지므로 실행 이전에 메모리 수요를 정적으로 분석할 수 있으며, 이를 통해 텐서 단위 동적 할당을 소수의 풀 단위 할당으로 대체한다. 먼저 텐서 활성도 분석이 가시적 텐서와 숨겨진 백엔드 작업 공간을 하나의 메모리 수요 프로파일로 통합하고, 레이어 퓨전과 텐서 통합 같은 연산 수준 최적화가 중복 메모리 연산을 제거한다. 이어서 제약 프로그래밍 기법인 CP-SAT 솔버가 그 기반이 되는 동적 저장소 할당(Dynamic Storage Allocation) 문제를 정확히 풀어, 각 메모리 풀에 대해 증명 가능하게 최적인 레이아웃을 산출한다. 동시 실행을 위해 Tailor-and-Stitch 알고리즘이 컴파일 타임 피크 메모리 프로파일에 따라 모델을 메모리 균형이 맞춰진 스케줄링 단위인 Task로 분할한다. 이중 메모리 풀 구조는 Task 경계를 넘어 공유되는 텐서를 GPU 메모리에 상주시켜, 그렇지 않을 경우 발생할 백업 및 복원 전송을 없앤다. 컴파일러가 생성한 스케줄러 인터페이스는 각 Task를 최악의 경우 메모리 검사를 통과한 뒤에만 배치하는 양보(yield) 기반 스케줄러를 구동하여, 동시에 실행되는 여러 모델이 OOM 오류 없이 동작하도록 한다.

NVIDIA RTX 3090 및 Jetson AGX Xavier에서 단일 모델 추론은 PyTorch 대비 평균 34.6% 적은 메모리를 사용하며 기하 평균 1.25배 빠르게 동작한다. NVIDIA RTX 4090 및 Jetson Orin Nano에서는 데스크톱 GPU 기준 평균 3.20%의 성능 오버헤드로 여러 모델을 OOM 오류 없이 동시에 실행하며, 임베디드 플랫폼에서 NVIDIA Triton Inference Server 대비 최대 10.8%, SwapNet 대비 최대 43.81% 낮은 지연 시간으로 요청을 처리한다.

핵심되는 말: DNN 추론, 메모리 관리, 컴파일 타임 최적화, 메모리 풀, 모델 분할, 다중 모델 스케줄링, 메모리 부족 방지, MLIR