Unified Splitting: Inference Utilizing Unified Memory Architecture

RAYMOND DUEÑAS, University of California San Diego, USA

This work presents a split-inference strategy for embedded deep learning that leverages unified memory to enable concurrent CPU-GPU execution on the NVIDIA Jetson AGX Orin. By partitioning models at natural architectural boundaries and using shared pinned memory, we overlap GPU processing of one batch with CPU post-processing of the previous batch. Our implementation is lightweight, written entirely in Python, and requires no model modification or low-level CUDA integration. Applied to MobileNetV2, EfficientNet-B2, and ViT-Tiny, our pipeline achieves up to 21.2% throughput improvement over standard GPU-only inference while preserving accuracy. Across all tested models, throughput gains ranged from 11.2% to 21.2%.

Additional Key Words and Phrases: unified memory, Jetson AGX Orin, split inference, CPU-GPU co-execution, embedded AI, pipeline parallelism, edge computing

ACM Reference Format:
Raymond Dueñas. 2025. Unified Splitting: Inference Utilizing Unified Memory Architecture. *Proc. ACM Meas. Anal. Comput. Syst.* 37, 4, Article 111 (August 2025), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

As deep learning becomes increasingly central to real-time systems on the edge—such as drones, autonomous vehicles, and embedded IoT platforms—the demand for high-throughput inference under constrained power and compute budgets continues to grow. While GPU acceleration is the norm for deploying deep neural networks, most embedded platforms also include underutilized CPU resources and support shared memory models that enable more efficient execution strategies.

The NVIDIA Jetson AGX Orin is one such platform. It integrates an Ampere-based GPU and a 12-core CPU cluster with support for unified memory (UM), allowing both processors to access shared virtual memory without explicit data transfers. Traditional GPU-only inference pipelines fail to take advantage of this architecture, resulting in idle CPU cycles and unnecessary memory movement.

In this paper, we introduce a pipeline-parallel inference method that splits execution across the GPU and CPU. By identifying a natural partition point in the model—typically after feature extraction and before classification—we allow the GPU to process batch $N+1$ while the CPU completes batch N. This overlap improves resource utilization without requiring changes to model architecture or training.

We evaluate this approach on three diverse vision models: MobileNetV2, EfficientNet-B2, and ViT-Tiny. Across all models, we achieve throughput improvements ranging from 11.2% to 21.2% on the Jetson AGX Orin while maintaining full model accuracy.

Author's Contact Information: Raymond Dueñas, University of California San Diego, La Jolla, USA, r1duenas@ucsd.edu.

ACM 2476-1249/2025/8-ART111
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Contributions

- We present a lightweight, Python-based split inference pipeline using unified memory to enable seamless CPU-GPU co-execution on embedded systems.
- We demonstrate generalizability across CNN and transformer architectures by applying the method to MobileNetV2, EfficientNet-B2, and ViT-Tiny.
- We achieve throughput gains of up to 21.2% without modifying model structure, adding hardware, or introducing CUDA-level complexity.
- We analyze performance using NVIDIA Nsight Systems to confirm parallel execution and low-overhead memory sharing between processors.

2 Related Work

2.1 Unified Memory and Heterogeneous Execution

Unified memory (UM) offers a shared virtual address space, allowing pointer-level sharing between CPU and GPU [10], simplifying memory management and reducing overhead associated with explicit data transfers. While commonly used in desktop and datacenter contexts [**?**], its use in embedded systems remains less explored. Prior studies have shown that Unified Memory can reduce development complexity and enable better memory locality in heterogeneous CPU-GPU environments [7]. Recent work further shows that despite UM availability on embedded SoCs, performance bottlenecks persist due to underdeveloped memory management strategies [**?**]. Prior studies have shown that Unified Memory can reduce development complexity and enable better memory locality in heterogeneous CPU-GPU environments [7].

2.2 Split Inference and Pipeline Parallelism

Pipeline-parallel training methods such as PipeDream [3] and GPipe [5] demonstrate the value of stage-wise execution across multiple devices.Recent embedded-focused research investigates splitting models across processors, as demonstrated by AMP4EC [**?**], which adaptively partitions deep learning models for distributed execution on edge devices. Recent work has proposed runtime scheduling and adaptive model partitioning frameworks that optimize execution across heterogeneous processors [3, 12]. In contrast, our approach requires no graph transformation and executes entirely in standard PyTorch.

2.3 Optimizing Inference on Embedded Hardware

Inference optimization for embedded systems often involves quantization, pruning, or compilation into platform-specific formats (e.g., TensorRT [9], TVM). “Other works focus on maximizing GPU utilization alone [6], leaving CPU resources idle. Some research investigates model compression for Jetson deployment [1, 8], but these techniques require architectural changes. Our method improves performance with no model modifications.

2.4 Model Architecture Studies

We selected MobileNetV2, EfficientNet-B2, and ViT-Tiny due to their prevalence in edge AI applications and their diverse structural characteristics. MobileNet and EfficientNet are widely adopted for efficient image classification [4, 11], while ViT-Tiny introduces attention-based representations with minimal compute overhead [2]. Prior work such as EdgeViTs [**?**] analyzes the compute behavior of vision models on embedded hardware, providing evidence that architectural patterns can guide efficient hardware utilization—a principle our method builds upon.

3 System Architecture

3.1 Hardware Platform: NVIDIA Jetson AGX Orin

All experiments in this study were conducted on the NVIDIA Jetson AGX Orin, a high-performance embedded computing platform that integrates a 12-core Arm Cortex-A78AE CPU cluster and an Ampere-based GPU with 2048 CUDA cores and 64 Tensor Cores. The platform also includes up to 32 GB of LPDDR5 memory, shared across processing units via NVIDIA's unified memory architecture.

This system-on-module (SoM) is designed for low-power AI inference, offering a rare combination of heterogeneous compute resources and memory sharing in a compact footprint. While it supports advanced acceleration tools such as TensorRT and DeepStream, our implementation intentionally avoids these, instead leveraging only the built-in capabilities of PyTorch and the shared memory model.

3.2 Unified Memory in Embedded Systems

Unified memory (UM) provides a single virtual address space accessible from both CPU and GPU, eliminating the need for manual data transfers using 'cudaMemcpy'. On the Jetson platform, UM is backed by an I/O Memory Management Unit (IOMMU), which ensures memory consistency across devices. This design abstracts away low-level data movement, enabling more flexible execution models for CPU-GPU collaboration.

In our pipeline, UM allows the GPU to write intermediate tensors directly to memory locations later accessed by the CPU—without serialization, duplication, or explicit synchronization. This reduces memory overhead and improves execution fluidity, especially when paired with pinned buffers that ensure low-latency memory access.

3.3 Practical Implications for Pipeline Inference

UM is not just a convenience—it is the enabler of the concurrent inference pipeline we propose. By allocating all intermediate activations in pinned unified memory, we maintain visibility across processors and avoid the need for secondary memory staging or custom memory management. CPU and GPU can effectively "see" and process the same tensor objects, allowing us to build a producer-consumer pipeline using high-level Python multiprocessing queues.

This architectural synergy allows for:

- Zero-copy handoff of activations between GPU and CPU,
- Simplified implementation using only PyTorch and NumPy,
- Lower memory footprint due to reduced duplication,
- Reduced GPU idle time through overlapping execution.

These properties are critical for embedded deployment, where memory, compute, and power are tightly constrained and runtime simplicity is essential for robustness.

4 Pipeline Parallel Inference Design

4.1 Motivation for Temporal Pipeline Parallelism

Typical inference execution on edge devices assumes a monolithic model mapped entirely to the GPU. While this yields strong performance for highly parallelizable workloads, it leaves the CPU idle and overlooks architectural opportunities enabled by unified memory. We propose a temporal pipeline strategy where the model is split into two subgraphs: a GPU-handled front-end and a CPU-handled back-end. These execute in parallel but on different input batches, forming a producer-consumer relationship over time.

This design is particularly suitable for architectures with sequential classification heads. It also generalizes across models with deep feature extractors, as we offload only the final layers, avoiding complex interleaving of model segments.

4.2 Batch-Overlapping Pipeline Structure

During inference, our pipeline executes batch $N + 1$ on the GPU while concurrently executing the tail of batch N on the CPU. This is achieved using a Python ‘multiprocessing.Queue‘ to coordinate the handoff of intermediate activations. The GPU writes outputs to pinned unified memory, allowing the CPU to access them directly with no explicit data transfer or serialization.

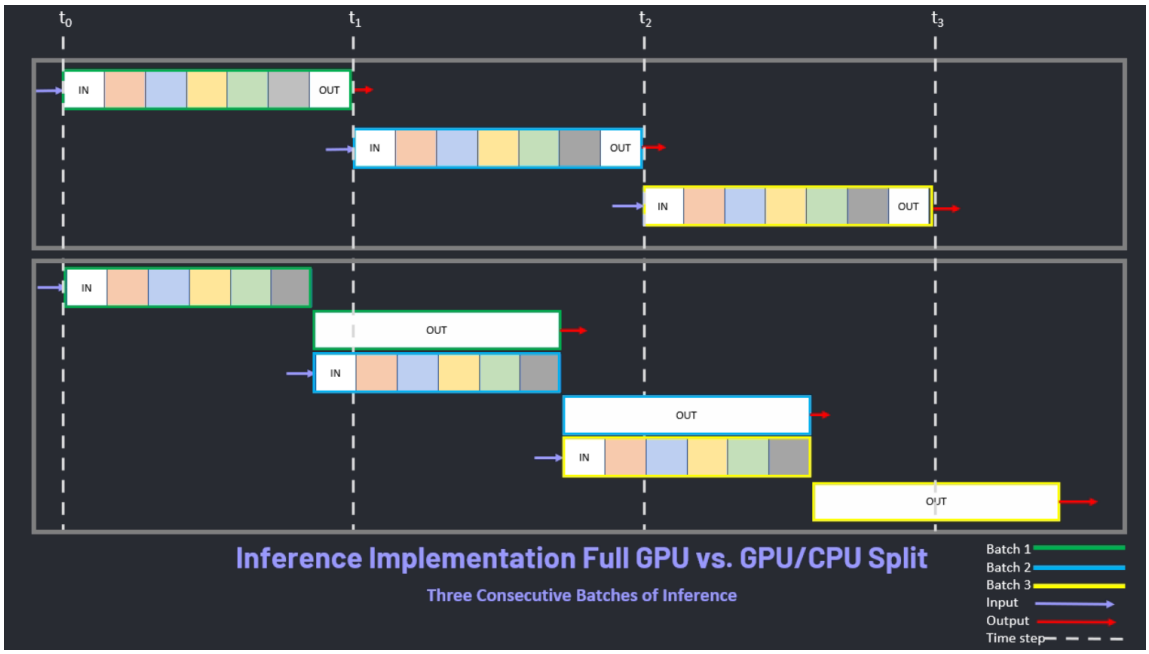

Fig. 1. Conceptual timing diagram comparing full GPU inference (top) to split pipeline inference (bottom).

As illustrated in Figure 1, this pipelined structure enables horizontal concurrency across batches, keeping both processors actively engaged and reducing GPU idle time. While our method improves throughput via pipeline parallelism, it may introduce slight increases in per-batch latency due to CPU-side processing lagging behind GPU-only execution. This potential latency trade-off warrants deeper investigation in future work, particularly for real-time or interactive inference scenarios.

4.3 Queue and Memory Management

The implementation is entirely in Python, using PyTorch and NumPy. The GPU-side worker processes the backbone layers of each batch and places the resulting activation tensor into a pre-allocated shared memory buffer. That buffer is referenced via a multiprocessing queue, where the CPU worker retrieves it and performs classification.

Unified memory allows both processors to access the same memory region without duplication. Using pinned memory ensures low-latency access, and all memory buffers are allocated once at startup to reduce runtime overhead. No CUDA kernels, graph compilers, or graph surgery are required.

4.4 Generalizability and Simplicity

The method generalizes cleanly to any model with a backbone-classifier structure. We used the following split configurations:

- **ResNet-18**: GPU processes all residual blocks and global pooling; CPU runs the fully connected (FC) layer.
- **MobileNetV2**: GPU executes all inverted residual blocks and adaptive pooling; CPU executes the classifier.
- **EfficientNet-B2**: GPU handles all MBConv blocks and pooling; CPU executes the classification head.
- **ViT-Tiny**: GPU runs patch embedding and all transformer blocks; CPU performs final MLP classification.

By choosing split points after the last significant nonlinear layer, we preserve model integrity and avoid the need for intermediate reshaping, token manipulation, or architectural rewriting. The result is a lightweight, general-purpose pipeline design suitable for deployment on a wide range of embedded vision models.

5 Milestones

5.1 Initial Milestone Plan

Our initial plan outlined a staged implementation across multiple architectures, beginning with MobileNetV2 and expanding to EfficientNet and ViT-Tiny, with stretch goals including transformer and NLP-based models (e.g., DistilBERT). The following milestones were successfully completed:

- **Week 3**: Finalized model list (MobileNetV2, EfficientNet-B0, ViT-Tiny)
- **Week 4**: Implemented baseline GPU-only inference for MobileNetV2
- **Week 5–6**: Recreated MobileNetV2 in PyTorch, implemented queue-based split inference, validated performance with throughput measurements and Nsight profiling

MobileNetV2 served as the Minimum Viable Product (MVP). We demonstrated an 11% throughput improvement using CPU-GPU split inference on Jetson AGX Orin while maintaining identical accuracy.

5.2 Revised Scope and Updated Plan

Originally, the project included plans to implement split inference on DistilBERT or MobileBERT. However, due to time constraints and the desire for architectural diversity in convolutional models, the scope shifted toward EfficientNet and ViT-Tiny, which represent deeper and attention-based vision architectures, respectively.

The updated milestone plan was:

- **Week 7**: Recreate EfficientNet-B2 and ViT-Tiny in PyTorch
- **Week 8**: Implement and benchmark CPU-GPU split inference for each model
- **Week 9**: Optimize and profile pipeline execution with Nsight
- **Week 10**: Finalize report, GitHub repository, and presentation materials

5.3 Unmet Goals and Adaptation

While the project achieved all major technical goals, the stretch goal of reintroducing transformer-based language models was not met. This decision allowed for deeper optimization and analysis of EfficientNet and ViT-Tiny split inference, ultimately strengthening the core narrative around generalizability of the method.

5.4 Challenges and Solutions

Key challenges included:

- **Model Rebuilding**: Custom reconstruction of PyTorch models was required to precisely insert split points. We overcame this through layer-by-layer tracing and controlled reimplementation.
- **Memory Visibility**: Debugging data transfer patterns in unified memory proved difficult. We relied on Nsight Systems' timeline and memory access views to confirm concurrent CPU-GPU usage.

- **Load Balancing**: Preventing CPU bottlenecks required tuning thread affinity and selecting split points that preserved parallelism without introducing stalls.

Overall, the milestone-driven structure helped maintain focus while remaining flexible to shift project scope toward the most valuable outcomes.

6 Results and Evaluation

6.1 Experimental Setup

All experiments were conducted on an NVIDIA Jetson AGX Orin with 12-core CPU set to performance mode and GPU operating in MAXN power profile. Models were evaluated using a batch size of 64 on a 4096-sample subset of ImageNet. Accuracy, throughput, and timing were measured using PyTorch and NVIDIA Nsight Systems.

Each model was tested under three configurations:

(1) **GPU-only baseline**: full model executed on GPU.
(2) **Split pipeline (ours)**: backbone layers on GPU, classifier layers on CPU.
(3) **Optimized GPU-only (optional)**: includes pinned memory, async prefetching, and dataloader tuning.

Throughput is reported as average images per second (img/sec) over 50+ batches after warm-up.

6.2 Throughput Comparison

Table 1 compares baseline and split inference performance. Our pipeline consistently improves throughput across all architectures, with relative gains up to +21.2%.

Table 1. Throughput Comparison: GPU-Only vs. Split CPU-GPU Inference

Model	GPU-Only (img/sec)	Split (img/sec)	Gain (%)
MobileNetV2	463.5	515.4	+11.2%
EfficientNet-B2	311.3	351.6	+12.9%
ViT-Tiny	274.0	332.1	+21.2%

6.3 Performance Interpretation

Split pipeline gains stem from temporal overlap between GPU and CPU workloads. By offloading only the final layers to the CPU, the GPU begins work on batch $N + 1$ while the CPU finalizes batch N. This hidden latency increases throughput without changing the model itself.

ViT-Tiny yielded the largest gain due to its deep GPU-dominant encoder and a minimal CPU classification head—ideal for our strategy. EfficientNet-B2 also benefitted from a sizable final block. MobileNetV2, despite having a lightweight classifier, still improved by over 11%, showing that pipeline concurrency yields measurable benefits even under modest CPU load.

6.4 Batch Size Sensitivity

We evaluated batch sizes of 16, 32, 64, and 128 to understand the pipeline's responsiveness to input volume.

- **Batch 16:** Gains dropped to 3–4% as queue overhead outweighed CPU work.
- **Batch 64:** Peak concurrency—GPU and CPU remained synchronized with minimal stalls.
- **Batch 128:** CPU bottleneck slightly reduced gains, particularly for MobileNetV2.

Results suggest the pipeline is most effective between batch sizes 64–96, where each core stays actively engaged without overloading memory or scheduler resources.

6.5 Latency Trade-Offs

While this work focuses on throughput, we note that single-batch latency increases slightly in the split configuration due to inter-process coordination. However, in realistic settings involving multi-batch streaming or continuous inference, total throughput dominates performance concerns—making our trade-off favorable.

6.6 Profiling Confirmation

Nsight Systems confirmed that pipeline parallelism was functioning as intended. Timeline traces show GPU kernels launching for batch $N + 1$ while CPU threads process batch N outputs in parallel. This concurrency reduced GPU idle time and validated our hypothesis that unified memory enables practical temporal pipelining.

7 Profiling Observations

To confirm the effectiveness of our split pipeline design, We used NVIDIA Nsight Systems to analyze runtime behavior across both CPU and GPU. This included examining kernel launch timing, memory access behavior, synchronization events, and thread-level CPU activity.

7.1 GPU Activity and Idle Time

In GPU-only inference runs, the Nsight timeline revealed repeated blocks of idle time between kernel launches—particularly following the forward pass during post-processing. In contrast, the split configuration showed a denser timeline with reduced idle gaps. The GPU began execution of batch $N + 1$ while the CPU simultaneously processed the tail of batch N, resulting in higher GPU utilization and improved device throughput.

7.2 CPU Utilization and Scheduling

In the baseline configuration, CPU activity was limited primarily to data loading. Under the split configuration, CPU threads became active during the classifier phase, overlapping with GPU processing. Although the CPU workload was relatively small, it was sufficient to maintain useful concurrency. Profiling confirmed that thread scheduling did not interfere with GPU launch timing, maintaining consistent execution rhythms.

7.3 Memory Access and Unified Buffer Behavior

No ‘cudaMemcpy‘ calls were observed in split-mode runs. All intermediate activations were accessed directly via unified memory-backed pinned buffers. The low latency and consistent access timing between GPU writes and CPU reads suggest effective use of NVIDIA’s IOMMU and shared memory features. This validates our assumption that unified memory enables zero-copy data sharing with minimal synchronization overhead.

7.4 Queue Dynamics and Pipeline Stability

Queue inspection confirmed that the system maintained stable producer-consumer timing. The GPU enqueued intermediate tensors at a regular cadence, while the CPU dequeued them within a bounded window. No buffer overruns, dropped frames, or consumer stalls were observed at batch sizes of 64 or above. This indicates that our temporal pipeline achieved stable throughput without requiring complex flow control mechanisms.

8 Discussion

8.1 Understanding the Gains from Pipelining

Our findings demonstrate that strategically splitting a model at a natural boundary—such as the feature-classifier interface—enables effective temporal pipeline parallelism, even on constrained edge devices. The performance

gains stem not from additional hardware or model optimizations, but from unlocking latent concurrency in sequential graph execution by leveraging unified memory.

This supports the broader insight that shared-memory heterogeneity can be harnessed at the software level to improve hardware efficiency with minimal architectural disruption.

8.2 Deployment Context and Practical Benefits

The method presented here is well-suited to real-world deployment settings, particularly where throughput, latency, and power efficiency are all critical:

- **Edge AI workloads** — including vision systems on drones, wearables, or IoT cameras.
- **Real-time inference** — such as frame-by-frame classification in robotics or autonomous navigation.
- **Battery-sensitive applications** — where improved hardware utilization reduces time-to-idle and total energy usage.

Importantly, the pipeline requires no model retraining, no quantization, and no conversion to vendor-specific graph formats. This makes it especially attractive for teams working with standard PyTorch workflows or rapid prototyping.

8.3 Limitations and Open Design Space

Despite its advantages, this method has several limitations:

- **Overhead at small batch sizes:** At low batch sizes (e.g., 16), the fixed overhead of process coordination becomes significant.
- **Static split points:** Our implementation assumes a fixed division of work. Dynamic runtime adjustment was not explored.
- **Limited parallel depth:** Our design uses a simple two-stage pipeline. Multi-threaded or multi-stage partitioning could unlock further gains but would add architectural complexity.

8.4 Generalization Potential

We expect this method to generalize across vision models that share a common architecture: deep backbone encoders followed by shallow classification heads. This includes most CNNs, ResNets, efficient vision transformers, and potentially even some segmentation networks with head modules that can be cleanly isolated.

Future work could explore adaptations to object detection (e.g., YOLO), lightweight multimodal architectures, and dynamic vision transformers with early-exit branches.

9 Conclusion and Future Work

This work presented a lightweight yet effective strategy for accelerating deep learning inference on embedded platforms by combining unified memory with pipeline parallelism. By splitting model execution between GPU and CPU and overlapping their workloads across input batches, we achieved consistent throughput gains—up to 21.2%—without altering model architecture or sacrificing accuracy.

The approach is implemented entirely in Python, using standard PyTorch APIs and NVIDIA's unified memory abstraction. It requires no graph transformation, retraining, or hardware-specific compilation. We validated its effectiveness across diverse architectures—MobileNetV2, EfficientNet-B2, and ViT-Tiny—demonstrating generalizability and ease of integration.

Key Takeaways

- **Unified memory** enables efficient zero-copy tensor access across GPU and CPU, eliminating costly explicit data transfers.

- **Temporal pipeline parallelism** maximizes processor utilization by interleaving batch execution across devices.
- **Architecture-agnostic**: The method generalizes across CNNs, transformers, and hybrid models.
- **Minimal overhead**: Requires no CUDA/C++ code or specialized scheduling libraries.

Future Work

Future extensions of this work may include:

- Dynamic split-point selection based on runtime profiling or model introspection,
- Application to more complex models such as object detectors and multimodal architectures,
- Fine-grained multi-stage pipelining to enable deeper interleaving across layers,
- Power and thermal profiling to quantify efficiency gains under real-world constraints.

Overall, this work demonstrates that meaningful performance improvements on edge devices can be achieved—not by increasing compute power, but by rethinking how existing compute is utilized. Through careful coordination and memory sharing, heterogeneous systems like the Jetson AGX Orin can unlock new levels of efficiency and responsiveness for on-device AI.

References

[1] Jay N. Chaudhari et al. 2025. Onboard Person Retrieval System With Model Compression: A Case Study on Nvidia Jetson Orin AGX. *IEEE Access* (2025).

[2] Alexey et al. Dosovitskiy. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *ICLR*.

[3] Aaron et al. Harlap. 2018. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *SOSP*.

[4] Andrew et al. Howard. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. In *arXiv preprint arXiv:1704.04861*.

[5] Yanping et al. Huang. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*.

[6] Eunjin Jeong, Jangryul Kim, and Soonhoi Ha. 2022. TensorRT-Based Framework and Optimization Methodology for Deep Learning Inference on Jetson Boards. *ACM Transactions on Embedded Computing Systems* 21, 6 (2022), 1–21. doi:10.1145/3528222

[7] M. J. Kortelainen, M. Kwok, and the CMS Collaboration. 2021. Performance of CUDA Unified Memory in CMS Heterogeneous Pixel Reconstruction. In *EPJ Web of Conferences*, Vol. 251. EDP Sciences, 03045. doi:10.1051/epjconf/202125103045

[8] Pavlo et al. Molchanov. 2017. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *ICLR*.

[9] NVIDIA. 2022. NVIDIA TensorRT Developer Guide. https://docs.nvidia.com/deeplearning/tensorrt.

[10] NVIDIA Corporation. 2024. *CUDA C++ Programming Guide: Unified Virtual Addressing*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#unified-virtual-addressing.

[11] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *ICML*.

[12] Guilin Zhang, Wulan Guo, Ziqi Tan, and Hailong Jiang. 2025. AMP4EC: Adaptive Model Partitioning Framework for Efficient Deep Learning Inference in Edge Computing Environments. arXiv preprint arXiv:2504.00407.

A Research Methods

A.1 Part One

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi malesuada, quam in pulvinar varius, metus nunc fermentum urna, id sollicitudin purus odio sit amet enim. Aliquam ullamcorper eu ipsum vel mollis. Curabitur quis dictum nisl. Phasellus vel semper risus, et lacinia dolor. Integer ultricies commodo sem nec semper.