Mälardalen University
Department of Computer Science and Engineering
Västerås, Sweden

Master of Science in Computer Science - 120 credits

AN EMPIRICAL ANALYSIS OF HETEROGENEOUS EXECUTION FOR YOLO-BASED OBJECT DETECTION IN CONTINUOUS MOBILE INFERENCE

Saad Abdullah
sah25006@student.mdu.se

Examiner: Shahina Begum
Mälardalen University, Västerås, Sweden

Supervisors: Mobyen Uddin Ahmed
Mälardalen University, Västerås, Sweden

Company supervisor: Matteo Allaix
Supersight Oy, Helsinki, Finland

June 12, 2026

This thesis was conducted within the Erasmus Mundus Joint Master Degree Programme on the Engineering of Data-intensive Intelligent Software Systems (EDISS) and Supersight Oy.

SUPERSIGHT

Acknowledgments

This thesis has been a long and demanding journey, and I am grateful to the people who supported me throughout it. I would first like to thank my company supervisor at Supersight Oy, Matteo Allaix, for giving me the opportunity to work on this thesis and for guiding me during the project. His technical feedback, practical suggestions, and encouragement helped me a lot during the work. I am also thankful to the team at Supersight for creating a welcoming and supportive work environment.

I would like to thank my academic supervisor, Mobyen Uddin Ahmed, for his guidance, feedback, and support throughout the thesis. I am also grateful to my examiner, Shahina Begum, for taking the time to review my work and provide valuable feedback. I would also like to thank the Erasmus Mundus Association and the EDISS programme for the scholarship and for the academic opportunity that made this study possible.

Finally, I am deeply thankful to my family and friends for their support and encouragement during my studies. Most of all, I want to thank my wife, Zainab, for her support, encouragement, and belief in me throughout this journey.

Saad Abdullah
Helsinki, June 2026

Abstract

Deep-learning inference is increasingly deployed directly on smartphones for computer-vision tasks such as object detection. This can reduce latency, limit dependence on cloud services, and keep visual data on the device. However, running these models continuously on a phone is difficult. Smartphones have limited power and cooling capacity, so performance that appears strong at the beginning may decrease as the device heats up and thermal throttling starts. This issue is relevant to the proof-of-concept deployment context of Supersight Oy, where Android smartphones are considered for long-running YOLO-based analytics.

This thesis studies sustained YOLO-based object detection on Android smartphones, with a focus on heterogeneous CPU–GPU execution. The main question is whether splitting inference across the CPU and GPU remains useful during long-running operation, once thermal effects, cut-point feasibility, data-transfer overhead, and runtime limitations are taken into account. To answer this, the study compares heterogeneous execution with CPU-only and GPU-only baselines under controlled live-camera workloads.

The evaluation follows an empirical systems approach. Controlled long-run experiments were conducted on representative Android smartphones using sequential and heterogeneous execution configurations. The study also examines camera-side workload pacing through frame-rate control, and re-evaluates selected configurations under capped input rates. The analysis focuses mainly on sustained throughput and thermal behaviour, while timing and profiling signals are used to interpret selected execution bottlenecks.

The results show that simple heterogeneous CPU–GPU execution does not provide a universal sustained advantage over strong sequential baselines. In several cases, it remains only approximately competitive. In selected device–model–cut combinations, it provides a modest benefit. The results also show that camera-side input-rate control can make selected heterogeneous configurations more competitive by reducing uncontrolled workload pressure. Overall, the findings provide practical guidance for evaluating long-running mobile object-detection systems within the studied NCNN-based Android pipeline, and suggest broader considerations for deployment under real device constraints.

Table of Contents

List of Abbreviations **v**

1. Introduction **1**

2. Background **3**
2.1. On-Device Inference on Mobile Platforms . . . 3
2.2. Heterogeneous Mobile Hardware and Execution Models . . . 3
2.3. YOLO-Based Object Detection on Mobile Devices . . . 5
2.4. Performance Metrics for Continuous Inference . . . 5
2.5. Thermal Constraints and Sustained Performance . . . 5

3. Related Work **6**
3.1. Mobile On-Device Inference and Performance Challenges . . . 6
3.2. Heterogeneous Execution on Mobile Devices . . . 7
3.3. Model Compression for Mobile Object Detection . . . 8
3.4. Thermal Effects and Long-Run Performance on Mobile Devices . . . 10
3.5. Summary of Related Work and Identified Research Gap . . . 11

4. Problem Formulation **13**
4.1. Problem Definition . . . 13
4.2. Purpose and Motivation . . . 13
4.3. Research Questions . . . 13
4.4. Goals, Assumptions, and Limitations . . . 14

5. Method **15**
5.1. Research Design . . . 15
5.2. Experimental Materials: Devices, Models, and Software Setup . . . 15
5.3. Workload and Experimental Setup . . . 16
5.4. Controlled Long-Run Execution Comparison . . . 17
5.5. Heterogeneous Execution Feasibility Investigation . . . 19
5.6. Input-Rate Control Experiments . . . 19
5.7. Practical Scope of the Evaluation . . . 20
5.8. Method–Research Question Alignment . . . 20
5.9. Methodological Limitations and Threats to Validity . . . 21

6. Ethical and Societal Considerations **22**

7. Experimental Development and System Realization **23**
7.1. Baseline Inference Pipeline . . . 23
7.2. Extending the Pipeline for Sustained Measurement . . . 24
7.3. Initial Design of Heterogeneous Execution . . . 25
7.4. Early Partitioning Attempt Using Split Model Files . . . 25
7.5. Final Heterogeneous Pipeline Realization . . . 25
7.6. Cut-Point Selection Problem . . . 27
7.7. Cut-Point Candidate Filtering . . . 28
7.8. Transfer Overhead and Zero-Copy Exploration . . . 29
7.9. Live Camera Workload Realization . . . 30
7.10. Camera Input-Rate Control Development . . . 31
7.11. Final Experimental Matrix . . . 32
7.12. Recorded Measurements . . . 33
7.12.1 Throughput and Latency . . . 33
7.12.2 Thermal Measurements and Android Thermal Labels . . . 34
7.12.3 Additional Measurements for Heterogeneous Runs . . . 34

8. Results **36**
8.1. Result Presentation Strategy . . . 36
8.2. Sequential Baseline Results . . . 37
8.3. Heterogeneous Execution Results: Main Comparison Against Sequential Baselines 39
8.3.1 Device A — YOLOv8 Medium . . . 40
8.3.2 Device A — YOLO-NAS Medium . . . 41
8.3.3 Device A — YOLO-NAS Small . . . 42
8.3.4 Device B — YOLO-NAS Medium . . . 44
8.3.5 Device B — YOLO-NAS Small . . . 45
8.4. Additional Heterogeneous Findings Beyond the Main One-Blob Runs . . . 47
8.4.1 Zero-Copy Refinement for YOLOv8 on Device A . . . 47
8.4.2 Exploratory Multi-Blob and Skip-Connected Cuts . . . 47
8.4.3 Failure Case Analysis: Why Some Splits Underperform . . . 49
8.5. Results Under Controlled Input Rate . . . 51
8.5.1 Sequential Execution Under Capped Input Rate . . . 51
8.5.2 Heterogeneous Execution Under Capped Input Rate . . . 52
8.6. Repeatability Check . . . 57
8.7. Energy Considerations and Thermal Coupling . . . 58

9. Discussion **59**
9.1. Overview of the Main Findings . . . 59
9.2. Discussion of RQ1: Sequential and Heterogeneous Execution Under Sustained Load 59
9.3. Discussion of RQ2: Practical Factors Governing Heterogeneous Execution . . . 60
9.4. Discussion of RQ3: The Role of Camera-Side Input-Rate Control . . . 60
9.5. Relation to Previous Work . . . 61
9.6. Significance and Practical Implications . . . 61
9.7. Limitations of the Study . . . 62
9.8. Generalizability and Broader Applicability . . . 63

10.Conclusions and Future Work **64**

11.Declaration of Generative AI and AI-Assisted Tools **66**

References **68**

Appendix A Literature Search Strategy **69**
A.1 Architecture-Oriented Search Queries . . . 69

List of Abbreviations

ACM	Association for Computing Machinery
AI	Artificial Intelligence
API	Application Programming Interface
ARM	Advanced RISC Machine
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSV	Comma-Separated Values
DNN	Deep Neural Network
DSP	Digital Signal Processor
EDISS	Engineering of Data-intensive Intelligent Software Systems
FPS	Frames Per Second
GB	Gigabyte
GPU	Graphics Processing Unit
IEEE	Institute of Electrical and Electronics Engineers
IoU	Intersection over Union
LPDDR	Low-Power Double Data Rate
MAC	Multiply–Accumulate Operation
mAP	Mean Average Precision
ML	Machine Learning
NCNN	Mobile neural network inference framework
NDK	Native Development Kit
NNAPI	Android Neural Networks API
NoC	Network-on-Chip
NPU	Neural Processing Unit
ONNX	Open Neural Network Exchange
OS	Operating System
p50	50th Percentile / Median
p95	95th Percentile
PICO	Population, Intervention, Comparison, Outcome
RGB	Red, Green, Blue
RQ	Research Question
SoC	System-on-Chip
TFLite	TensorFlow Lite
UX	User Experience
YOLO	You Only Look Once
YOLO-NAS	YOLO Neural Architecture Search
YOLOv8	Version 8 of the YOLO object detection family
YUV	Luma and Chroma Colour Format

1. Introduction

Deep learning is increasingly run directly on smartphones for computer-vision tasks such as object detection, activity recognition, and scene understanding. On-device inference can reduce cloud dependence, lower network traffic, and keep sensitive visual data on the device. This is important in privacy-sensitive settings and in deployments where connectivity is limited [1, 2]. At the same time, mobile devices have limited power and thermal headroom. During continuous workloads, processor temperature can rise until thermal management reduces clock frequencies. Performance may therefore change over time instead of staying at the level observed in a short initial run [3, 4, 5].

In this thesis, *sustained inference* means continuous inference that runs long enough to move beyond initial peak behaviour and expose performance under ongoing thermal and system constraints. This problem is relevant to the proof-of-concept deployment setting considered by Supersight Oy. In that setting, recycled Android smartphones are explored as fixed on-device vision sensors for long-running YOLO-based analytics in real environments, such as a top-down doorway-monitoring setup. For such use, it is not enough to know whether a model or execution strategy performs well in a short benchmark. It is also necessary to know whether performance remains usable and predictable during prolonged operation. This practical motivation also relates to sustainability. Reusing existing smartphones as edge-AI devices is meaningful only if the resulting systems remain effective under continuous load, not only in short demonstrations.

Prior work has explored several ways to improve mobile inference, including model compression, hardware acceleration, heterogeneous execution, and thermal-management techniques [6, 7, 8, 9, 10, 4, 5]. For YOLO-family detectors, compression studies have used pruning and quantization to support faster mobile deployment [6, 7]. However, these studies mainly focus on model size, throughput, or accuracy trade-offs. They do not focus on long-running behaviour under thermal constraints.

Research on heterogeneous execution has shown that CPU–GPU cooperation can reduce latency in some settings [8, 9, 11]. More recent systems have also explored broader heterogeneous scheduling across mobile processors [12, 8, 13, 9, 10, 14, 11]. However, this work often evaluates short-run execution or layer-level acceleration. It remains less clear whether heterogeneous CPU–GPU execution stays beneficial for long-running mobile object detection when cut-point feasibility, data-transfer overhead, runtime fallbacks, and framework limitations are considered together [8, 10].

Thermal-aware studies have shown that long-running mobile inference can cause substantial performance degradation. They also show that mitigation can be attempted through model shifting or processor scheduling [3, 4, 5]. These studies, however, do not resolve the specific question addressed here: whether heterogeneous CPU–GPU execution is practically worthwhile for sustained YOLO-based object detection on Android smartphones under realistic deployment constraints. The combination of sustained thermal behaviour, heterogeneous execution feasibility, and deployment-oriented YOLO inference therefore remains underexplored.

This work addresses that gap through an empirical study of sustained on-device inference for YOLO-based object detection on Android smartphones. The focus is on sequential and heterogeneous CPU–GPU execution under prolonged load. The scope is intentionally limited to evaluation on existing hardware and software stacks. The work does not propose new model architectures, pruning or quantization methods, scheduling algorithms, hardware modifications, operating-system changes, or cloud/edge offloading mechanisms. Its main interest is not how to redesign the model or platform. Instead, it studies how currently available execution options behave during sustained operation on real phones.

The experiments focus on Android smartphones and YOLO-based detectors within an NCNN-based pipeline. Even so, the findings may suggest useful evaluation principles for other continuous mobile vision workloads. Such generalization would still require validation across different platforms, frameworks, and model families. The intended audience includes researchers and practitioners in mobile systems, edge AI, and deployment-oriented on-device machine learning. It may also be useful for graduate students interested in sustained mobile inference under practical system constraints.

Methodologically, the study uses an empirical systems approach based on controlled long-run experiments on two Android smartphones with different mobile SoC characteristics. Sequential and heterogeneous CPU–GPU execution are compared under fixed conditions to examine how throughput changes over time under sustained load. Additional feasibility-oriented experiments help interpret when heterogeneous execution helps, fails, or becomes impractical. These experiments focus on partition feasibility, transfer overhead, and runtime/framework behaviour.

A further set of experiments evaluates camera-side workload pacing through frame-rate control. This makes it possible to examine whether an execution strategy behaves differently when the input rate is constrained rather than left uncontrolled. The methodological design is motivated by an industrial deployment context, but the study remains focused on controlled sustained-inference evaluation rather than formal field deployment validation. In this way, the method maps to the problem formulation: the main comparisons address execution strategy under sustained load, the feasibility experiments explain why heterogeneous execution may or may not help, and the pacing experiments address end-to-end workload effects. The formal problem formulation and research questions are presented in Chapter 4.

The main contributions of this thesis are:

- **Sustained-inference behaviour under real mobile constraints:** The work provides an empirical study of sustained YOLO-based inference on two Android smartphones under controlled 60-minute live-camera workloads. It compares CPU-only, GPU-only, and heterogeneous CPU–GPU execution using throughput, thermal-status, and temperature-binned analysis.
- **Practical conditions for useful heterogeneous execution:** The study identifies why selected CPU–GPU splits succeed or fail in practice. The analysis shows that heterogeneous execution is useful only when cut-point structure, one-blob or multi-blob handoff complexity, stage balance, transfer overhead, and runtime/framework behaviour align sufficiently well.
- **Camera-side input-rate control as part of sustained inference evaluation:** The experiments show that camera-side input-rate control can change the interpretation of sequential and heterogeneous execution results. Workload pacing should therefore be treated as part of the sustained inference problem, not as a separate implementation detail.

These findings matter for both research and practice. From a research perspective, they clarify an underexplored part of mobile inference. They show that sustained behaviour cannot be understood from peak short-run results alone, especially when thermal effects, execution feasibility, and end-to-end workload conditions interact. From an industrial perspective, the results provide practical guidance for evaluating and interpreting long-running mobile object-detection systems on existing smartphones. This is especially relevant when predictable behaviour over time is more important than isolated peak performance.

The remainder of this thesis is structured as follows. Chapter 2 presents the background needed to understand on-device inference, mobile hardware heterogeneity, YOLO-based object detection, performance metrics, and thermal constraints. Chapter 3 reviews related work and identifies the research gap addressed in this thesis. Chapter 4 formulates the problem and presents the research questions. Chapter 5 describes the methodology, including the experimental design and evaluation strategy. Chapter 6 discusses ethical and societal considerations. Chapter 7 presents the experimental development and system realization. Chapter 8 reports the results, which are then interpreted in Chapter 9. Finally, Chapter 10 concludes the thesis and outlines directions for future work.

2. Background

This chapter introduces the concepts needed to follow the thesis. It covers mobile on-device inference, heterogeneous mobile hardware, execution models, YOLO-based object detection, performance metrics for continuous inference, and thermal constraints in sustained mobile performance.

2.1. On-Device Inference on Mobile Platforms

On-device inference means running a trained machine-learning model directly on the target device, such as a smartphone, instead of sending the computation to a remote server. This deployment model is useful because it can reduce end-to-end latency, work better when network connectivity is limited, and avoid sending potentially sensitive user data to external infrastructure [1, 2, 15]. These properties make on-device inference attractive for mobile computer-vision applications that need fast responses and continuous operation.

Modern smartphones are suitable for these workloads because they combine several processor types within the same mobile system-on-chip (SoC) [5, 10]. In practice, applications usually access these processors through inference frameworks rather than directly. The framework therefore affects which execution strategies are possible on a given device.

TensorFlow Lite (TFLite) and NCNN illustrate two different styles of mobile inference framework. TFLite is a mobile-oriented runtime that can execute models on the CPU or delegate supported parts of the graph to hardware-specific backends, such as NNAPI or GPU delegates [1, 15]. In that case, the actual execution path depends on the model, the supported operators, and the vendor driver stack. Unsupported operations may fall back to another processor.

NCNN, by contrast, is a lightweight inference framework optimized for mobile deployment and is commonly used with Vulkan-based GPU execution rather than NNAPI delegation [16]. In simple terms, NCNN's Vulkan path runs supported neural-network operations on the mobile GPU through Vulkan compute. This is different from NNAPI-style delegation, where a framework may pass parts of the graph to system or vendor backends and may fall back when an operation is unsupported. The experimental pipeline in this thesis is NCNN-based, and the evaluated execution paths are NCNN CPU execution and NCNN Vulkan GPU execution. TFLite remains relevant because it is widely used in prior work and helps illustrate framework-level execution constraints in mobile inference.

2.2. Heterogeneous Mobile Hardware and Execution Models

Mobile SoCs are heterogeneous by design. Different processors inside the same chip are optimized for different kinds of work. CPUs handle general-purpose computation and control, and they may execute full or partial neural network inference. GPUs are commonly used for highly parallel computation. NPUs provide specialized acceleration for supported AI operations [12, 5, 10]. Figure 1 illustrates the typical roles of the CPU, GPU, and NPU in a heterogeneous mobile SoC.

In this thesis, *heterogeneous execution* refers to inference configurations where computation is distributed across more than one processor, mainly the CPU and GPU. When execution is divided at a selected boundary, it is referred to more specifically as *partitioned execution*. The boundary is called a *cut point*. When discussing prior work, the term *CPU–GPU co-execution* is used where it matches the terminology of the cited literature.

In deep learning, heterogeneous execution can take several forms. Some systems assign different layers or operators of a neural network to different processors. Others keep preprocessing and postprocessing on the CPU while the main network runs on the GPU. More loosely coupled approaches overlap work across frames. For example, one processor may prepare the next input while another processes the current one [12, 8, 13, 9, 11]. These approaches can improve resource utilization, but their benefit depends on the work assigned to each processor and the overhead needed to coordinate them.

In practice, mobile heterogeneous execution is controlled through inference frameworks, delegates, GPU runtimes, and vendor drivers. These software layers decide which operators are supported on each processor, how intermediate results are stored and transferred, and whether unsupported operations fall back to another processor [8, 10, 14]. As a result, the feasibility and

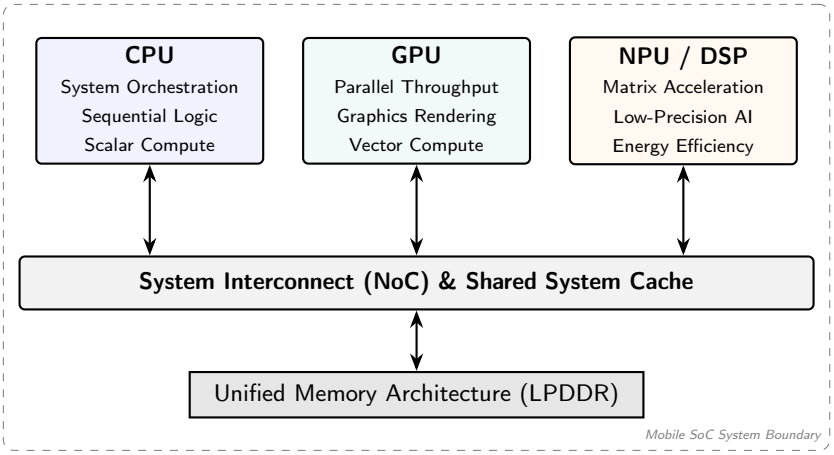

Figure 1: Heterogeneous mobile SoC architecture. CPUs provide general-purpose computation and control and may execute full or partial neural network inference, GPUs are commonly used for highly parallel workloads, and NPUs or DSPs provide specialized acceleration for supported AI operations. All units share system interconnect and memory resources.

effectiveness of heterogeneous execution can differ across software stacks and hardware platforms, even for the same model.

A cut point is the boundary in a neural network or inference pipeline where execution is divided between two processors, such as the CPU and GPU. Computation before the cut point runs on one processor, and the remaining part runs on the other. This requires transferring the intermediate output tensor produced at the boundary. Different cut points therefore lead to different communication costs, synchronization overheads, and implementation complexity. Cut-point selection is therefore a practical issue in heterogeneous mobile inference [8, 11].

This issue is especially relevant for the YOLO models studied in this thesis. In YOLOv8 and YOLO-NAS, early backbone stages process relatively large spatial feature maps. A cut placed close to the input can therefore require much larger intermediate activation transfers than a later cut placed after spatial downsampling. Later cut points may reduce transfer volume, but they also leave less computation available for partitioning. The usefulness of a cut point therefore depends not only on the nominal compute split, but also on the size and structure of the intermediate feature maps produced by the specific model variant. Figure 2 illustrates this concept by showing how spatial downsampling across model stages affects the amount of data transferred at the partition boundary.

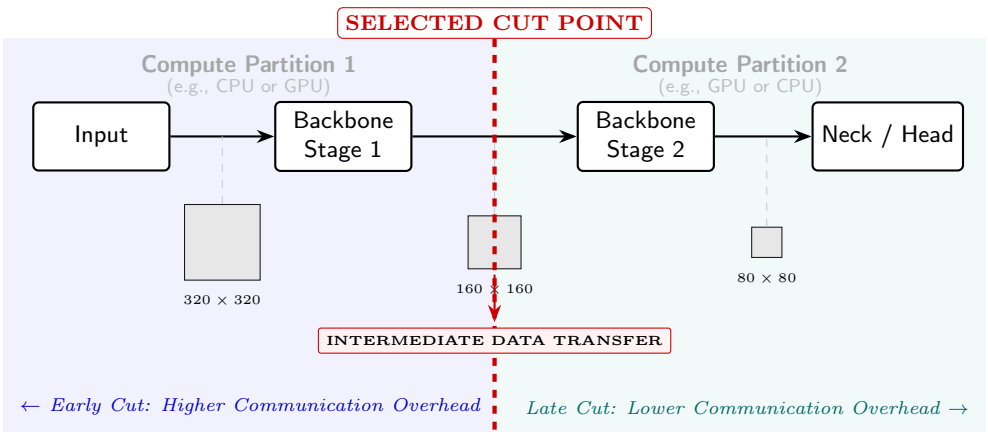

Figure 2: **Conceptual illustration of a cut point in partitioned inference.** The model is divided into a prefix and suffix executed on different processors. Earlier cuts often expose larger spatial activations, while later cuts usually occur after downsampling. The size and structure of the intermediate tensor influence the communication and synchronization overhead at the partition boundary.

2.3. YOLO-Based Object Detection on Mobile Devices

YOLO, or You Only Look Once, is a family of real-time object-detection models. It treats detection as a single-stage prediction task. Instead of first generating region proposals and then classifying them, YOLO predicts object locations and class probabilities directly from the input image in one forward pass. This design helps explain why YOLO models are common in real-time and near-real-time applications [6, 17]. YOLO-style pipelines are also useful for studying mobile inference because much of their computation consists of dense and regular operations.

Several YOLO variants have been developed for resource-constrained settings, including mobile and embedded devices. These variants balance accuracy, model size, and computational cost in different ways [6, 7]. This thesis uses both YOLOv8 and YOLO-NAS. YOLOv8 is a commonly used detector family with broad deployment support in current tooling [17]. YOLO-NAS is a newer detector family designed with neural architecture search and efficiency-oriented optimization in mind [18]. Both are real-time detectors, but they differ in network structure and deployment characteristics. This makes them useful complementary workloads for comparing mobile execution behaviour.

YOLO-based object detection is used in this thesis because it matches the company's target mobile-analytics pipeline. It is also a practical computer-vision task for live camera applications, and its single-stage design supports real-time use. In this study, it provides a workload for examining throughput, transfer cost, and thermal behaviour during long-running mobile inference.

2.4. Performance Metrics for Continuous Inference

The main performance metric in this thesis is *throughput*, expressed as processed frames per second (FPS). Throughput measures how much inference work the system actually completes under a given execution configuration. For long-running mobile object detection, this is more relevant than the nominal camera frame rate because the system may not process every frame produced by the camera [3, 4, 5].

This thesis distinguishes between *peak throughput* and *sustained throughput*. Peak throughput refers to the initial, or near-initial, processing rate before strong thermal effects appear. Sustained throughput refers to the rate that remains after the device has moved beyond the early transient phase and into a longer thermally constrained operating regime. For long-running mobile workloads, sustained throughput is the more meaningful deployment metric. It better reflects the level of service that can actually be maintained over time [3, 5].

2.5. Thermal Constraints and Sustained Performance

Thermal management is a basic part of mobile system design. When a heavy workload runs continuously, the device consumes power and heats up. To keep the hardware within safe limits, the mobile operating system may reduce processor frequencies, limit power draw, or change scheduling behaviour. This is commonly called *thermal throttling*. For inference workloads, throttling can reduce performance over time and turn an initially fast setup into a slower but more thermally stable one [3, 4, 5]. Figure 3 shows this behaviour conceptually.

In Android systems, thermal state may also be exposed through coarse-grained thermal status categories, often referred to in this thesis as *thermal labels*. These labels provide an application-visible indication of the device's current thermal condition, such as NONE, MODERATE, SEVERE, or CRITICAL thermal pressure [19]. Although they are not a substitute for raw temperature measurements, they are useful for interpreting when the device is moving from unconstrained behaviour toward thermally limited operation. Later chapters use such labels as a practical way to describe how throughput changes across different thermal regimes.

From a measurement perspective, sustained performance refers to the level of throughput that a device can maintain once thermal effects have taken hold and the system operates in a thermally constrained regime. This regime is distinct from the initial warm-up phase and better reflects the performance that can be expected in real long-running deployments [3, 5]. This distinction matters when evaluating on-device inference systems for continuous use. In such settings, the important question is not only how fast the system is at the beginning, but how predictably it performs over time.

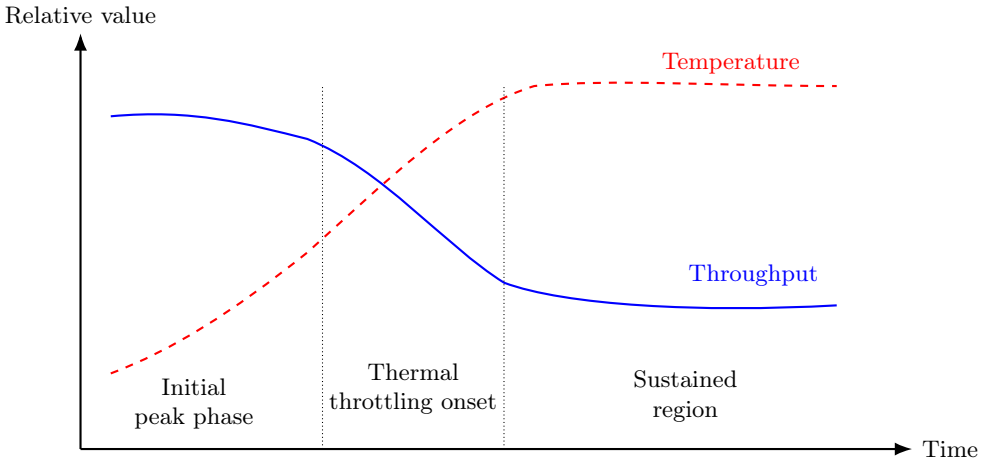

Figure 3: Conceptual illustration of thermal throttling during continuous mobile inference. Throughput may decline as temperature rises and the device enters a thermally constrained regime. The sustained region is more representative of long-running deployment behaviour than the initial peak phase. The curves are conceptual diagrams and do not represent empirical experimental data.

3. Related Work

This chapter reviews research on mobile on-device inference, heterogeneous execution, model compression, and sustained performance under thermal constraints. The review positions this thesis in relation to earlier work, summarizes the main findings from the literature, and identifies the limitations that motivate the research questions.

3.1. Mobile On-Device Inference and Performance Challenges

Running deep neural networks directly on mobile devices has been studied widely as smartphone hardware has become more capable. Prior work shows that on-device inference can support computer-vision tasks while reducing the latency, privacy, and connectivity issues associated with cloud-based processing [15, 1, 2]. Mobile inference is therefore no longer only a laboratory demonstration. It is now part of many practical systems.

Earlier continuous-vision systems such as DeepMon [20] showed that mobile GPUs can support camera-based on-device deep learning. This makes DeepMon especially relevant to deployment-oriented mobile vision. However, DeepMon mainly demonstrates the feasibility of continuous GPU-based inference. The present thesis instead studies how sequential and CPU–GPU heterogeneous execution behave over longer runs, where thermal progression and workload-delivery constraints become important.

Performance evaluation in this area is still often influenced by benchmark-oriented thinking. Standardized benchmarks, including MLPerf Tiny, are useful because they provide consistent workloads and common reporting conventions for comparing inference systems [21]. However, these benchmarks are usually built around fixed tasks, controlled inputs, and relatively short or bounded execution settings. They are useful for comparing nominal model or platform capability, but they say less about how performance changes during prolonged mobile operation under thermal and runtime constraints.

This limitation matters for sustained inference on smartphones. Mobile devices operate with limited power budgets and passive cooling. Continuous high-performance computation is therefore difficult to maintain. Studies of mobile inference in the wild show that behaviour seen in controlled or short-run evaluation can differ substantially from behaviour during extended operation on real devices [2]. Long-term inference studies also show that thermal throttling can gradually reduce processing capability over time [3]. Initial throughput is therefore not necessarily representative of what a deployment can maintain.

The concern here is not that benchmarking is inherently unhelpful. The issue is that many

benchmarks abstract away the factors that dominate sustained behaviour. These include thermal progression, runtime fallback behaviour, and end-to-end workload delivery over time.

For object-detection workloads, this concern is especially important. Models in the YOLO family are computationally demanding and are commonly used in continuous camera-based applications. In such settings, the main question is not only whether a device can reach a high initial frame rate. It is whether the device can maintain useful throughput during prolonged operation without unacceptable degradation. Long-run behaviour therefore becomes a first-order evaluation concern, not a secondary implementation detail. This view is consistent with work such as NestDNN, which treats continuous mobile vision as a resource-constrained deployment problem rather than as a single fixed inference task [22].

The present work builds on the broader mobile on-device inference literature, but focuses on a part of the problem that short-run evaluations capture only partially. It does not only ask how fast inference can run under controlled nominal conditions. It asks how YOLO-based object detection behaves on real Android smartphones during sustained operation, where thermal effects and system-level constraints become unavoidable.

3.2. Heterogeneous Execution on Mobile Devices

Heterogeneous execution uses more than one processing unit in a mobile system-on-chip during inference. Instead of running the full model on only the CPU or only the GPU, these approaches try to use both processors together. The aim is usually to reduce latency or increase throughput by using the strengths of each processor. In practice, however, this is not only a question of raw compute capability. The result also depends on where the model is split, how intermediate data is moved, and how the runtime coordinates work across processor boundaries.

A representative example is CoDL [8]. CoDL demonstrates CPU–GPU co-execution on Snapdragon smartphones across several models, including YOLO and RetinaFace. It reduces overhead by keeping tensors in GPU-friendly formats and by chaining operators to avoid repeated format conversions and synchronization. The study reports lower latency and lower energy consumption than CPU-only execution. This shows that mobile CPUs can contribute meaningfully to inference instead of acting only as a fallback.

At the same time, CoDL also highlights an important limitation. When operators are too small, synchronization and data-handling overhead can become larger than the useful computation. This limits the achievable speedup [8]. This point is important for the present thesis. CoDL proposes mechanisms for efficient CPU–GPU co-execution and evaluates them mainly from a latency perspective. The present work does not propose a new co-execution runtime. Instead, it examines whether heterogeneous execution remains worthwhile during sustained YOLO-based object detection on real Android smartphones, where thermal effects, transfer overhead, and framework constraints also matter. CoDL also reports energy benefits, while this thesis discusses energy only indirectly through thermal behaviour because direct power measurement was not included.

More recent work studies finer-grained CPU–GPU partitioning. Li et al. [11] split linear and convolution layers by output channels and select splits using learned latency predictors. They report speedups of up to 1.89× for linear layers and 1.75× for convolution layers on Pixel-class smartphones. A key contribution is the reduction of CPU–GPU synchronization overhead through fine-grained shared virtual memory and active polling. However, the same study also reports non-linear GPU latency caused by kernel-scheduling heuristics. This shows that accurate prediction remains difficult and that fine-grained approaches can be sensitive to device-specific runtime behaviour [11].

Reducing synchronization cost therefore does not remove the broader coordination problem. When computation is split more aggressively, performance becomes more exposed to runtime effects that are difficult to predict or control. This is consistent with nn-Meter, which shows that edge-device inference performance is hard to predict from model structure alone because hardware and runtime characteristics matter substantially [23].

NN-Stretch [9] takes a different view. It argues that fine-grained operator-level splitting may be poorly suited to mobile hardware. The study shows that, for many layers, communication cost between processors can exceed the layer computation time. NN-Stretch therefore branches the neural network graph into larger parallel subgraphs. This reduces the number of synchronization

points while still exploiting unified memory. Measurements on Snapdragon devices show that coarser-grained partitioning can outperform finer splits when layers are small [9]. This provides an important counterpoint to fine-grained splitting. Heterogeneous execution is not only about exposing more parallelism. It is also about whether the coordination cost of that parallelism is worth paying.

Related insights also appear in work on integrated CPU–GPU edge platforms. Band [24] studies coordinated inference across heterogeneous mobile processors, but focuses on scheduling multiple DNNs rather than sustained single-model YOLO object detection. It is relevant because it shows that coordination across mobile processors matters, although the workload structure and evaluation goal differ from this thesis. EdgeNN [13] evaluates hybrid execution across and within kernels on integrated CPU–GPU devices. It uses zero-copy unified memory and reports improvements in both execution time and energy efficiency compared with CPU-only and discrete-GPU baselines.

Although these systems differ from smartphones in form factor and thermal envelope, they reinforce a common theme: data movement and synchronization can be as important as compute balance [13]. Transfer overhead in heterogeneous execution is therefore not only explicit copy cost. It can also include format conversion, runtime coordination, waiting time, and shared-memory contention. These costs can reduce or remove the expected benefit of using two processors together.

Several empirical studies also warn that gains measured in controlled settings may not transfer directly to real mobile deployments. Liu et al. [10] conduct a multi-device empirical study and show that mobile inference frameworks often execute models on only one processor at a time. When an operator is not supported by the selected accelerator, the runtime may fall back to CPU execution. This can break the intended execution flow and reduce or remove speedups. Their measurements show high fallback rates on real Android phones, which means that framework and driver behaviour can strongly constrain heterogeneous execution in practice [10].

This point is central for the present thesis. Heterogeneous execution cannot be evaluated only as a theoretical partitioning problem. It also has to be understood as a framework- and device-dependent systems problem. Table 1 summarizes representative prior work on heterogeneous execution along these dimensions. It contrasts device classes, partition strategies, and reported performance outcomes.

Although this thesis uses NCNN rather than TFLite, the broader lesson remains relevant: the effective execution path depends on framework support, backend implementation, and device drivers, not only on the neural network graph. The NCNN-based results are therefore treated as framework-specific evidence and are not assumed to generalize automatically to other runtimes such as TFLite, ONNX Runtime, MediaPipe, or NNAPI-based execution.

Overall, the reviewed heterogeneous-execution studies show that CPU–GPU cooperation can reduce latency or increase throughput under certain conditions. They also show that the benefit depends strongly on execution granularity, transfer cost, synchronization overhead, and runtime support. Most studies emphasize short-run performance metrics and controlled evaluation settings. This leaves open the question of how heterogeneous execution behaves during continuous operation, when thermal constraints and framework limitations become more important. The study builds on these findings by examining heterogeneous execution under sustained workloads on real Android devices. The focus is long-run performance rather than peak benchmark gains, with explicit attention to execution boundaries, transfer overhead, and end-to-end pipeline behaviour.

3.3. Model Compression for Mobile Object Detection

Model compression is another common way to improve mobile inference performance. Its goal is to reduce computation and memory use while preserving as much accuracy as possible. Common techniques include quantization, pruning, and architectural simplification. For mobile object detection, compression is often used either on its own or together with hardware acceleration.

Several studies show that compression can improve inference speed for YOLO-based models on mobile hardware. YOLObile [6] evaluates YOLOv4 and lightweight YOLO variants on Android smartphones. It shows that reduced-precision arithmetic and smaller network configurations can reach real-time or near-real-time throughput on mobile GPUs. The study also shows the trade-off involved. Quantization and architectural simplification can reduce latency and energy consumption, but more compact models may lose accuracy, especially for smaller objects and crowded

Table 1: Representative strands of prior work relevant to sustained heterogeneous YOLO inference.

Strand or paper	**Device class**	**Partition strategy**	**Key result**
CoDL [8]	Smartphones, Snapdragon	Intra-operator split, GPU image tensors, operator chaining	Lower latency than CPU-only; near-ideal when overheads are small
Li et al.[11]	Smartphones, Pixel, OnePlus, Moto	Intra-operator split by output channels; kernel-aware predictors; shared virtual memory	Up to 1.89 and 1.75 times speedups; sync reduced from about 162 microseconds to around 7 microseconds
NN-Stretch, [9]	Smartphones	Coarse graph branching across processors	Avoids heavy per-operator communication; favours coarse splits
YOLObile, [6]	Smartphones	Compression with CPU–GPU collaboration	17.0 to 19.1 frames per second at comparable accuracy at 320 pixels; pruning aligned to vector width
Liu et al.[10]	Smartphones, multi-vendor	Survey and empirical pitfalls	High fallback rates can break parallelism
EdgeNN [13]	Integrated CPU–GPU edge	Inter and intra kernel hybrid execution with zero-copy	Time and energy efficiency improvements
Presto [25]	Servers, video inference	Hybrid CPU–GPU pipeline for preprocessing	Up to 4.37 times higher throughput; avoids GPU stalls

scenes [6].

A useful distinction is between pruning and quantization. Pruning reduces parameter count and computation by removing less important weights, channels, or structures from the model. Quantization reduces the numerical precision used for weights and activations. In YOLO-based detectors, pruning is attractive because it can reduce computational load while keeping the overall model structure. However, aggressive pruning may still harm detection quality if important features are removed. Quantization can improve memory efficiency and arithmetic speed more directly, especially when the hardware supports low-precision execution well. At the same time, it may make accuracy more sensitive and can depend strongly on runtime and hardware support. In practice, both techniques can improve deployability, but both require careful tuning to avoid unacceptable losses in detection quality [6, 7].

Recent work on YOLOv8 lightweighting [7] explores structured pruning and quantization for adapting modern YOLO variants to mobile deployment. The results show that pruning can reduce parameter count and computation with limited accuracy loss when it is carefully tuned. The study also notes a limitation. Aggressive pruning or low-bit quantization can make detection quality unstable across scenes. Retraining or fine-tuning is often needed to recover accuracy. This adds development effort and may limit compression in settings where model retraining is costly or undesirable [7].

Other work studies selective execution and partial approximation of model computation. μLayer [12] executes only a subset of model layers at high precision and approximates others, adapting computation to resource constraints. This can reduce computation, but it adds control logic and complexity. Its accuracy impact also depends strongly on the workload. Such techniques show the broader trade-off between computational efficiency and prediction quality, but they are less common in practical mobile object-detection pipelines [12].

Across these studies, the main pattern is clear: compression can improve performance, but it often introduces costs in accuracy, retraining effort, or engineering complexity. For continuous mobile analytics, detection quality matters because it affects the reliability of downstream analysis. A speed improvement is less useful if it makes detections unreliable. In addition, many compression techniques are evaluated mainly through short-run benchmarks. This leaves open the question of how compressed models behave during sustained operation under thermal constraints.

Model compression is therefore important background for this thesis, but it is not the main focus. The goal here is to understand sustained inference behaviour under sequential and heterogeneous execution on real Android devices. Compression is treated as a contextual factor and as an alternative route used in prior work to improve peak performance. This distinguishes the present work from studies that focus mainly on speed gains through aggressive model modification rather than sustained execution behaviour under realistic system constraints.

3.4. Thermal Effects and Long-Run Performance on Mobile Devices

Thermal constraints are a major limitation for mobile inference. Smartphones do not have the same cooling capacity as servers or desktop systems. They rely mostly on passive cooling and must operate within strict power limits. When heavy workloads run continuously, the device heats up. The operating system may then apply thermal management, such as dynamic voltage and frequency scaling, to avoid overheating. These mechanisms can directly reduce inference performance.

Several empirical studies show that short benchmark runs can be misleading when thermal effects are ignored. Smart at What Cost [2] studies deep neural network inference on smartphones and shows that sustained execution can lead to clear throughput degradation once throttling begins. Electronics 2020 [3] reports a similar pattern for long-term visual inference. Once the device enters a thermally constrained regime, performance can drop substantially. These results show that initial or peak performance is not enough to evaluate mobile inference under long-running conditions.

Other work studies the link between inference workloads and thermal behaviour more directly. Play It Cool [4] proposes a thermal-aware scheduling approach that adapts execution according to device temperature. The aim is to slow down heat buildup. This can delay throttling and improve energy efficiency, but it also introduces latency variation and does not remove performance degradation completely. The study is important because it evaluates performance over extended periods and shows that thermal management is part of sustained inference performance, not a separate concern [4].

Recent work has also studied thermal-aware execution on heterogeneous mobile hardware. Thermal-Aware Scheduling for Deep Learning on Mobile Devices with NPU [5] examines how distributing work across CPUs, GPUs, and NPUs can reduce thermal stress on individual components. The results show that spreading computation can delay throttling and improve energy efficiency. They also show that the benefit depends on workload characteristics and hardware support. Even with heterogeneous execution, thermal constraints can remain a dominant factor during long runs [5].

These studies make one point clear: thermal behaviour has to be part of mobile performance evaluation. At the same time, the literature has limits. Many mobile inference studies still use short experiments that capture mainly initial performance. Others reduce thermal effects through external cooling, which makes the results less representative of normal deployment. Thermal-aware techniques are also often studied separately from framework constraints, operator support, and workload delivery. This leaves open questions about how these factors interact during sustained mobile inference.

Evidence on YOLO-based object detection under sustained thermal load on smartphones is also limited. Existing sustained or thermal-aware studies mainly focus on classification workloads, model switching, or multi-accelerator scheduling. They do not fully address fixed-model YOLO object detection over long runs on real Android devices [4, 5, 3]. Prior work therefore shows that thermal effects matter, but it does not fully explain how thermal progression shapes heterogeneous YOLO inference in deployment-oriented mobile settings.

The present study addresses that gap by evaluating sustained YOLO-based inference on real Android devices. Thermal behaviour is treated as a central part of the evaluation, and heterogeneous execution is examined under practical system constraints. The contribution is therefore not only to confirm the importance of thermal effects. It is to study how thermal behaviour interacts with execution strategy, partition feasibility, framework behaviour, and long-run workload delivery in a mobile object-detection pipeline.

3.5. Summary of Related Work and Identified Research Gap

The reviewed literature shows three main ways of improving mobile inference performance: heterogeneous execution, model compression, and thermal-aware scheduling. Work on heterogeneous execution shows that CPU–GPU cooperation can reduce latency or increase throughput in some cases. It also shows that these gains depend heavily on execution granularity, synchronization overhead, transfer cost, and runtime support. Work on model compression shows that pruning and quantization can reduce the cost of YOLO-based object detection, but often with trade-offs in accuracy, retraining effort, or engineering complexity. Thermal-aware studies show that heat buildup strongly affects long-run mobile performance, and that short benchmark runs are not enough to characterize sustained behaviour.

The reviewed studies also differ in how long they run inference and how directly they treat thermal effects. Table 2 summarizes representative studies along these dimensions. The main pattern is clear. Much of the literature relies on short-run benchmarking or controlled profiling. Only a smaller subset studies long-run behaviour under thermal stress. Thermal effects are also often ignored, discussed only qualitatively, or reduced through external cooling to stabilize measurements. They are less often treated as a central experimental factor. Table 2 is not intended as a complete taxonomy of all mobile inference work. It is used as a positioning aid for the strands most relevant to this thesis.

This review reveals three main research gaps that motivate the present thesis:

- **Gap 1: Limited sustained evaluation under thermal progression.** Many studies use short-run benchmarks or bounded profiling experiments. These capture initial behaviour, but give limited evidence about what happens after the device enters a thermally constrained operating regime.
- **Gap 2: Limited evidence on the practical feasibility of heterogeneous execution.** Prior work shows that CPU–GPU cooperation can help under selected conditions. Less is known about how framework constraints, operator fallbacks, transfer overhead, synchronization cost, and cut-point feasibility affect its usefulness on Android smartphones.
- **Gap 3: Limited end-to-end perspective on sustained mobile inference.** Much of the literature evaluates model execution in relative isolation. In real deployments, sustained performance is also shaped by how work enters the inference pipeline over time, including workload delivery and camera-side input pacing.

In the reviewed literature, camera-side pacing and latest-frame workload delivery are usually treated as implementation details. They are rarely treated as primary evaluation variables. This leaves open whether controlling the rate at which frames enter the inference pipeline changes the interpretation of sustained execution results.

These gaps motivate the problem formulation in Chapter 4. They define the need to compare sustained execution strategies, examine the practical feasibility of heterogeneous execution, and assess the role of end-to-end workload delivery in long-run mobile inference.

Within the classification in Table 2, this thesis is positioned as an R4/T3 study. It explicitly targets sustained long-run behaviour and treats thermal effects as a central research variable.

This thesis addresses the identified gap by empirically evaluating sustained on-device inference for YOLO-based object detection on real Android smartphones. It does not propose a new algorithm or optimization method. Instead, it focuses on long-run behaviour, the practical benefits and limitations of heterogeneous execution, and comparison against strong sequential baselines under realistic system constraints. The aim is to provide clearer guidance on what governs sustained mobile inference performance in practice, and when heterogeneous execution offers meaningful benefits for long-running object-detection workloads.

Table 2: Summary of representative prior work by execution approach, run duration, thermal awareness, and key limitation.

Work	Main focus	Run duration	Thermal awareness	Key limitation
CoDL [8]	CPU–GPU co-execution on smartphones	R1	T0	Gains depend strongly on low sharing/synchronization overhead
NN-Stretch [9]	Branching-based parallel inference	R2	T2	Focuses mainly on classification models and controlled profiling
YOLObile [6]	Compression–compilation co-design for mobile YOLO	R1	T0	Short-run study; not a sustained thermal evaluation
EdgeNN [13]	CPU–GPU inference on integrated edge devices	R1	T1	Not focused on smartphone long-run thermal behaviour
Li et al. [11]	Fine-grained CPU–GPU co-execution	R2	T2	Focuses on layer-level latency, not sustained throughput
Play It Cool [4]	Thermal-aware dynamic model shifting	R4	T3	Improves stability by changing the model, not by fixed-model heterogeneous execution
Smart at What Cost? [2]	Measurement and characterization study	R3	T1	Observational; does not propose execution strategies
μLayer [12]	Single-layer cooperative CPU/GPU acceleration	R1	T0	Layer-level optimization only; no sustained end-to-end evaluation
Thermal-Aware Scheduling (NPU) [5]	Thermal-aware multi-accelerator scheduling	R4	T3	Focuses on NPU–GPU, not CPU–GPU execution
Optimizing Multi-DNN Inference [14]	Heterogeneous co-execution for multi-model workloads	R4	T3	Focuses on concurrent multi-model workloads rather than sustained single-model inference
This thesis	**Sustained sequential vs. heterogeneous YOLO inference on real Android phones**	**R4**	**T3**	**Limited to selected devices, models, and execution pipeline**

Table note / legend:
To make the table easier to interpret, prior work is grouped using two simple classification schemes:

Run duration classes:
R1 – short-run benchmarking **R2** – controlled profiling under load
R3 – extended execution with incidental thermal effects **R4** – sustained long-run evaluation targeting throttling or steady-state behaviour

Thermal-awareness classes:
T0 – thermal behaviour not considered **T1** – thermal effects mentioned but not evaluated directly
T2 – thermal effects mitigated to stabilize measurements **T3** – thermal behaviour explicitly measured and analysed using available system-level indicators

4. Problem Formulation

This chapter defines the problem addressed in the thesis and presents the purpose, research questions, goals, assumptions, and limitations of the study.

4.1. Problem Definition

The problem addressed in this thesis is the limited empirical understanding of sustained on-device inference on commodity Android smartphones under realistic operating constraints. Prior work shows that mobile CPUs and GPUs can run deep learning inference in short benchmark settings. However, short-run results do not reliably show what happens during continuous operation. In practice, long-run inference is affected by thermal throttling, framework constraints, data-transfer overhead, and device-specific runtime behaviour. These factors can change performance substantially over time.

This gap is important for YOLO-based object detection in continuous mobile analytics. Existing studies often examine heterogeneous execution, model compression, or thermal effects separately. Many also use short execution windows or assumptions that do not fully match deployment conditions on real smartphones. As a result, it is still not well understood how sustained inference behaves when sequential and heterogeneous execution strategies are used over extended periods.

There is especially limited empirical evidence on whether heterogeneous CPU–GPU execution provides a meaningful sustained benefit once practical system constraints are included. These constraints include cut-point feasibility, intermediate-tensor transfer cost, framework behaviour, and the way work enters the inference pipeline over time. For long-running object detection, these factors may matter as much as the nominal computational capability of the processors.

4.2. Purpose and Motivation

The purpose of this thesis is to empirically characterize sustained mobile inference for YOLO-based object detection on Android smartphones under realistic operating conditions. More specifically, the study examines how sequential and heterogeneous execution strategies behave over long durations. It also investigates which practical system-level factors influence whether heterogeneous execution is useful in sustained operation.

The motivation comes from the mismatch between short-run performance reported in much prior work and the needs of real continuous deployments. In practical mobile analytics systems, usable long-run throughput matters more than isolated peak performance. From a research perspective, this creates a need for evaluation that treats thermal behaviour, framework limitations, cut-point feasibility, and input-rate control as part of the inference problem itself.

4.3. Research Questions

Based on the defined problem and purpose, this thesis addresses the following research questions:

- RQ1: How do sequential CPU-only and GPU-only execution compare with heterogeneous CPU–GPU execution in terms of sustained throughput for YOLO-based object detection on Android smartphones?
- RQ2: What practical factors determine whether heterogeneous CPU–GPU execution is feasible and beneficial under sustained operation?
- RQ3: How does camera-side input-rate control influence sustained inference performance and the observed value of heterogeneous execution?

Together, these questions link execution strategy, implementation feasibility, and workload control to sustained mobile inference behaviour.

4.4. Goals, Assumptions, and Limitations

To answer the research questions, the study pursues the following goals:

1. Measure sustained YOLO-based object detection under continuous operation, focusing on throughput evolution, temperature progression, and long-run performance.
2. Compare sequential and heterogeneous execution under the same experimental conditions on representative Android smartphones.
3. Examine practical factors that affect heterogeneous execution, including cut-point feasibility, transfer overhead, and candidate partition boundaries.
4. Evaluate how camera-side input-rate control affects sustained performance, and re-assess execution strategies under controlled frame-delivery conditions.
5. Identify framework and device-specific runtime constraints that affect long-run inference performance.

The study makes three main assumptions. First, the experiments are conducted on unmodified consumer Android devices using standard mobile inference frameworks. Second, devices are allowed to cool between runs to reduce thermal carry-over effects. Third, the evaluated workloads are treated as representative of continuous mobile object detection.

The scope is limited to on-device execution. Cloud or edge offloading is not considered. The thesis also does not propose new model architectures, compression methods, scheduling algorithms, or hardware modifications. The findings are therefore specific to the evaluated hardware, software stack, and workload characteristics, although they may still be useful for understanding sustained mobile vision workloads more broadly.

5. Method

This study uses an empirical systems method to examine sustained on-device inference for YOLO-based object detection under mobile constraints. The research questions in Chapter 4 are addressed through controlled 60-minute continuous runs. These runs are treated as long-run experiments because they go beyond short benchmark behaviour. They capture the warm-up phase, thermal progression, and later sustained operation.

The workflow has four main steps: selecting representative devices and models, configuring the execution setup, running controlled 60-minute live-camera experiments, and analysing sustained performance across execution strategies. Figure 4 summarizes this workflow, from device and model selection to experimentation and analysis.

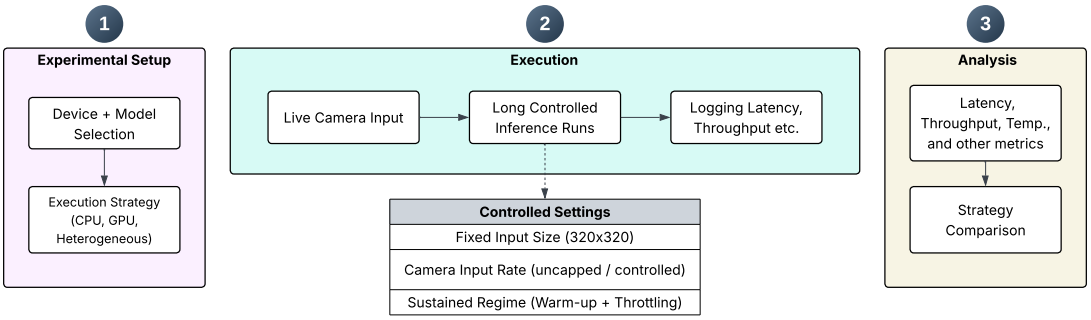

Figure 4: Main experimental workflow for sustained on-device inference, from device and model setup to long-run execution and comparative analysis.

5.1. Research Design

This work uses a controlled empirical systems design. In this thesis, research design refers to how the investigation is structured: which factors are varied, which conditions are fixed, and what evidence is collected to answer the research questions.

The comparison is organized around one main variable: the inference execution strategy. The evaluated strategies are CPU-only execution, GPU-only execution, and selected heterogeneous CPU–GPU execution. Within each comparison group, the device, model, input resolution, camera setup, software stack, and run procedure are kept the same.

This structure makes the results easier to interpret. When two runs use the same device, model, input resolution, camera setup, software stack, and run duration, differences in sustained throughput or thermal behaviour can be linked more clearly to the execution strategy being tested.

The design also matches the aim of the thesis. The work studies runtime behaviour in an existing mobile inference pipeline rather than introducing a new model architecture or scheduling algorithm. Real phones and a live camera workload keep the evaluation close to the deployment setting, while the controlled setup keeps the comparisons manageable.

5.2. Experimental Materials: Devices, Models, and Software Setup

The study evaluates two Android smartphones and multiple YOLO-based object-detection models. This provides a concrete hardware and workload basis for answering the research questions. The smartphones are referred to as Device A and Device B throughout the thesis. Detailed hardware identifiers, including commercial device names and specific chipset information, are omitted, focusing on the comparative performance analysis. The retained information is limited to what is needed for interpretation: broad processing class, SoC family, GPU family, memory/storage configuration, and software environment. The evaluated device–model combinations are shown in Table 3.

This selection gives diversity in mobile platforms, software stacks, and model characteristics while keeping the evaluation scope manageable. The chosen models are modern YOLO-style detec-

Table 3: Evaluated device–model combinations.

Device	Models evaluated
Device A	YOLOv8 Medium, YOLO-NAS Medium, YOLO-NAS Small
Device B	YOLO-NAS Medium, YOLO-NAS Small

tors relevant to continuous mobile analytics. They also allow comparison across different accuracy–complexity trade-offs on the selected devices.

The experiments build on an existing on-device inference pipeline from the company development environment. That pipeline was originally designed for sequential execution. It did not directly support the heterogeneous execution scenarios studied in this thesis. To enable the experiments, the surrounding execution pipeline was extended to support the required configurations and to log runtime behaviour during sustained execution.

Measurement and logging support were also added. The logs captured throughput, latency, thermal-status information, battery-temperature traces, and execution diagnostics during long-run experiments. The work does not modify the NCNN framework itself. Instead, it extends the surrounding execution pipeline to study how the available execution mechanisms behave under sustained operation on real devices.

The anonymised hardware and software characteristics of the evaluated smartphones are summarized in Table 4.

Table 4: Anonymised hardware and software characteristics of the evaluated smartphones.

Device	Processor class	CPU capability	GPU capability	Memory / Storage	OS / API
Device A	Mid-range Android mobile SoC	8-core ARM-based mobile CPU; Qualcomm SoC family	Qualcomm integrated mobile GPU; lower GPU compute capability among the evaluated devices	6 GB / 128 GB	Android 14 / API 34
Device B	Higher-performance Android mobile SoC	8-core ARM-based mobile CPU; Google Tensor SoC family	ARM Mali integrated mobile GPU; higher GPU compute capability among the evaluated devices	8 GB / 128 GB	Android 16 / API 36

The software stack consisted of an Android application using the Camera2 / NDK camera pipeline for image acquisition and NCNN for inference. The NCNN runtime used in the experiments was `ncnn-20250503-android-vulkan`. GPU configurations used Vulkan compute, whereas CPU configurations used NCNN's CPU execution path. In CPU-only execution, the runtime was configured to prefer the device's big CPU cores through `ncnn::set_cpu_powersave(2)`, and CPU-side parallelism was configured via `ncnn::set_omp_num_threads(ncnn::get_big_cpu_count())`. No custom NCNN framework modifications were introduced for the purpose of this study. NNAPI- or NPU-based execution was not evaluated; this scope decision is discussed in Section 5.7.

5.3. Workload and Experimental Setup

The primary experiments were conducted using a live camera stream rather than pre-recorded or annotated clips. This choice approximates the real-time characteristics of the deployment pipeline while still allowing controlled systems comparison. The purpose of the workload is not to benchmark scene diversity or application-level counting accuracy, but to evaluate sustained inference behaviour under realistic end-to-end input delivery.

The smartphone was placed on a desk with the rear camera facing downward toward the floor. The camera-to-scene distance was approximately 29–30 inches. The scene was intentionally simple and stable across runs. This made changes in measured behaviour more attributable to execution strategy and system behaviour than to scene variation. Lighting conditions corresponded to ordinary indoor room lighting and no intentional scene changes were introduced during the main long-run measurements.

Image acquisition used the Android Camera2 / NDK camera stack with a `TEMPLATE_PREVIEW` capture request. The request configuration explicitly set `ACAMERA_CONTROL_MODE=AUTO`, `ACAMERA_CONTROL_AF_MODE=CONTINUOUS_VIDEO`, `ACAMERA_CONTROL_AE_MODE=ON`, and `ACAMERA_CONTROL_AWB_MODE=AUTO`. Accordingly, autofocus was operated in continuous-video mode, while exposure and white balance remained under automatic camera control. ISO and exposure time were not manually fixed by the experiment code. Camera frames were acquired through an `AImageReader` in `YUV_420_888` format, then converted, cropped, rotated, and resized within the inference pipeline before model execution. Video stabilization was not explicitly configured in the capture request and was therefore left unspecified at the application level.

The camera stream was not passed to the model at the full sensor resolution. For all experiments, the model input was fixed at 320 × 320. This kept the runs comparable across devices, models, and execution modes. Mobile and edge inference studies use different input sizes depending on the model and evaluation goal, as shown in Table 5. The 320 × 320 setting is also used in prior mobile object-detection work such as YOLObile. It gives a workload that is heavy enough to expose sustained-performance effects, while still being realistic for mobile deployment.

Table 5: Representative input resolutions used in related literature.

Work / study	**Model focus**	**Input size**
YOLObile [6]	YOLOv4-derived	320 × 320
NN-Stretch [9]	ResNet, RegNet, EfficientNet	224 × 224
Smart at What Cost? [2]	InceptionV3, MobileNetV1	224 × 224
MLPerf Tiny [21]	MobileNetV1	96 × 96
Thermal impact studies [3]	SqueezeNet, MobileNet, ResNet	224 × 224
Play It Cool [4]	Slimmable ResNet50	224 × 224
YOLOv8 study [7]	YOLOv8n	N/A (Desktop)

A larger input size, such as 640 × 640, would process many more pixels per frame. This would increase computation, memory use, and energy demand. On smartphones, that usually means higher latency and faster heat buildup during continuous inference. The aim of this study is not to maximize input resolution, but to evaluate sustained execution under practical mobile conditions. For that reason, 320 × 320 was used as a stable workload for the long-run experiments. This resolution is also relevant to the intended deployment context, where it already gives satisfactory detection behaviour in practice.

Environmental and device conditions were kept as consistent as possible across runs. No external fans or directed airflow were used. The devices were operated in normal indoor room conditions and were allowed to cool between runs to reduce carry-over effects from previous experiments. During experimentation, the phones were connected to a PC for development and logging, and battery level was not allowed to fall below 20%. These measures do not eliminate all thermal and system-level variation, but they reduce uncontrolled differences between repeated experimental conditions.

5.4. Controlled Long-Run Execution Comparison

To address RQ1, the primary method is controlled long-run experimentation. Each experiment evaluates one execution configuration under fixed conditions on the same device and model. The compared configurations include:

- CPU-only baseline
- GPU-only baseline
- Heterogeneous execution, where the CPU and GPU are used together when supported by the implementation

The experiments run inference continuously so that both early and later behaviour can be observed. This includes the initial warm-up phase and the later period where thermal constraints start to matter. The main comparison metric is sustained throughput. Timing measurements and thermal-state observations are used as supporting evidence. During each run, the system records throughput summaries, execution-time statistics, Android thermal-status labels, and battery-temperature traces. This setup is needed because sustained behaviour cannot be judged reliably from short benchmark runs. It is shaped by thermal management, runtime behaviour, and execution overhead.

Each experiment runs for 60 minutes. This duration was chosen based on the exploratory runs. In those runs, visible changes in performance and thermal state appeared within the 60-minute window and continued to affect later behaviour. A 60-minute run was therefore long enough to capture the early transition period, the onset of thermally constrained behaviour, and the later sustained regime. This makes the evaluation a long-run study rather than a short benchmark snapshot. The choice is also consistent with prior work that studies thermal effects over extended execution windows, as summarized in Table 6.

Table 6: Representative run durations in related sustained mobile inference literature.

Work / study	**Run duration**	**Evaluation type**
YOLObile [6]	50 frames	Short-run benchmarking
NN-Stretch [9]	200 inferences	Controlled profiling
Smart at What Cost? [2]	1,000 inferences	Extended measurement
ADMS / Multi-DNN [14]	10–30 minutes	Thermal stress test
Play It Cool [4]	60–100 minutes	Sustained thermal evaluation
Thermal-Aware (NPU) [5]	∼6–30 minutes	Thermal-aware evaluation
This thesis	**60 minutes**	**Sustained long-run evaluation**

In this thesis, the *warm-up phase* refers to the early portion of a run in which performance and thermal state are still transitioning away from their initial conditions. The *throttling onset* is not identified from direct CPU/GPU frequency instrumentation, since such low-level signals were not logged as part of the experimental pipeline. Instead, it is interpreted from the combination of Android thermal-status progression, battery-temperature increase, and sustained changes in observed inference behaviour. Android thermal labels were therefore used as the primary system-level indicator of thermal progression. Battery temperature provided an additional coarse thermal proxy. Low-level thermal-daemon internals were not directly instrumented.

The purpose of these comparisons is to determine whether heterogeneous execution provides meaningful sustained benefits over strong sequential baselines, and whether such benefits remain once long-run thermal and runtime effects dominate the system's behaviour.

For the comparative analysis, the results are interpreted using three complementary views: median throughput by Android thermal status, temperature-binned throughput profiles, and a compact thermal-status-aware throughput score. The score is computed over the first 60 minutes of each run as the equal-weight average of per-status median throughput values. A thermal-status category is included only if it contains at least five samples and at least one minute of dwell time, so that very short thermal intervals do not dominate the score. Equal weighting is used to compare behaviour across thermal regimes, rather than to measure the time-weighted average throughput of a specific run. A time-weighted score would also reflect how long a device stayed in each thermal state, which is influenced by workload delivery, device thermal policy, and throttling behaviour. The score is therefore used as a compact diagnostic and ranking aid, not as the only evidence of sustained superiority. The final interpretation always considers the score together with the thermal-status curves and temperature-binned profiles.

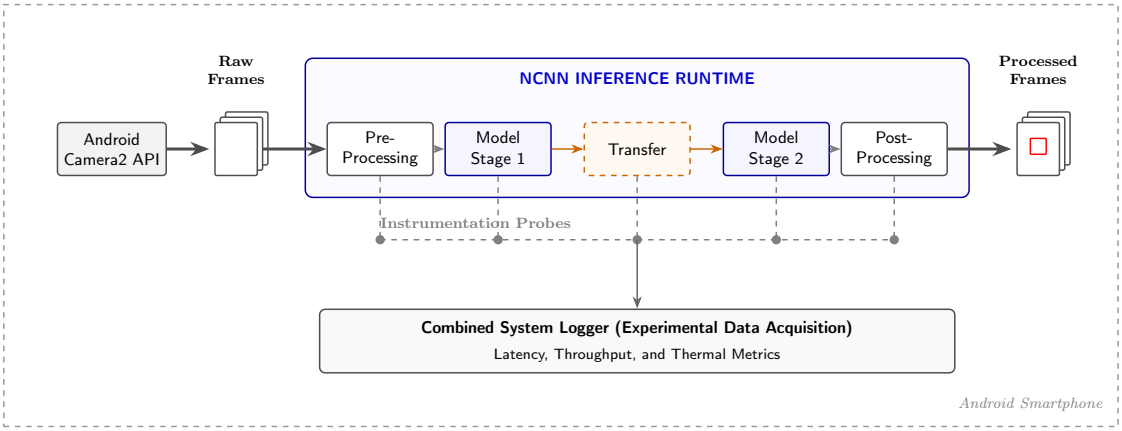

Figure 5: End-to-end experimental inference pipeline. Raw camera frames pass through the NCNN runtime to produce processed outputs, while a unified instrumentation layer logs performance and thermal metrics.

5.5. Heterogeneous Execution Feasibility Investigation

To address RQ2, the study extends the baseline comparison with a focused investigation of the practical factors that influence heterogeneous execution on real mobile systems. The work also examines whether heterogeneous execution is practically realizable. This depends on mobile inference stacks, framework behaviour, and transfer overhead.

This part of the methodology examines candidate cut points in the model execution pipeline. Particular attention is given to the number and structure of intermediate tensors transferred between processors. Preference is given to cut points with simpler transfer structure, especially boundaries where a single intermediate output is exchanged, as these are more practical to implement and compare. More complex cut points involving multiple outputs are also examined in order to understand how implementation complexity and communication overhead scale in less favourable cases.

The study also examines three practical issues: the cost of moving intermediate activations across the cut point, the balance of computation on the two sides of the split, and whether the runtime can keep the intended processor placement. A candidate cut point is therefore not judged only by the amount of computation it divides. It must also lead to execution that is implementable, measurable, and stable on the real devices.

Where feasible, the study also examines transfer overhead and more direct handoff mechanisms between execution stages. These experiments are not separate algorithmic contributions. They are included to help explain why heterogeneous execution succeeds in some cases, underperforms in others, or fails under sustained operation on real devices.

5.6. Input-Rate Control Experiments

To address RQ3, the methodology includes experiments on camera-side input-rate control. The reason is that sustained inference depends not only on how computation is executed. It also depends on how quickly new frames enter the inference pipeline. Figure 5 summarizes the end-to-end experimental pipeline. It shows the main processing stages, thread interaction points, logging hooks, and thermal sampling used in the study.

Two principal camera operation modes are evaluated. The first uses repeating camera requests through the Android Camera2 / NDK camera pipeline, implemented with `ACameraCaptureSession_setRepeatingRequest(...)`. In this mode, the camera continuously produces frames, and the target frame-rate range is constrained through `ACAMERA_CONTROL_AE_TARGET_FPS_RANGE`. The second uses scheduled single-capture operation, implemented with repeated calls to `ACameraCaptureSession_capture(...)` under application-side timing control. In this mode, a scheduler thread submits the next capture only after the previous capture has been processed and the configured pacing condition is satisfied.

The purpose of these experiments is to determine whether controlling input rate improves sustained system behaviour and changes the relative value of sequential and heterogeneous execution strategies. In repeating-request mode, the incoming workload is driven by the camera pipeline itself within the configured frame-rate range. In scheduled single-capture mode, pacing is enforced by the application through explicit timing between submissions, using the elapsed time since the previous capture to maintain a target interval. This allows the study to compare a more open-loop camera-driven pipeline against a more tightly controlled request-driven pipeline.

Another practical issue is frame handling while inference is busy. The experimental pipeline processes the most recent available frame instead of queueing all incoming frames. Frames that arrive while inference is occupied are therefore skipped rather than stored for later processing. These skipped frames were not logged as a separate measurement variable. In this thesis, they are treated as part of end-to-end workload delivery, not as a directly measured output metric.

After a suitable input-rate control method was established, selected sequential and heterogeneous experiments were repeated under capped frame-rate conditions. This part of the methodology treats input pacing as an end-to-end systems factor rather than as a separate camera-only issue. It is included because sustained inference cannot be fully understood when the incoming workload rate is left uncontrolled.

5.7. Practical Scope of the Evaluation

The evaluation in this thesis is carried out under controlled experimental conditions rather than as a formal field deployment study. This choice helps make the comparisons easier to interpret. It reduces the influence of uncontrolled variation in scene activity or deployment environment, so that observed differences can be linked more clearly to execution strategy, thermal progression, and workload-delivery conditions.

The evaluation is limited to CPU and GPU execution paths. NNAPI- and NPU-based execution are outside the scope of the study because the target devices and company pipeline did not provide a usable NPU path for these experiments. The implemented system therefore uses NCNN CPU execution and NCNN Vulkan-based GPU execution.

This scope also keeps the comparison within one inference framework. NPU access on Android usually depends on NNAPI or vendor-specific drivers, which can involve different operator support, fallback behaviour, and runtime policies. Prior work has shown that accelerator delegation on Android devices can be affected by unsupported operators and fallback behaviour. In such cases, it may be unclear whether a measured result reflects the intended accelerator path or a mixed execution path.

The CPU-only and GPU-only baselines should therefore be understood as the strongest sequential baselines within the evaluated NCNN CPU/Vulkan pipeline. They should not be read as claims about the best possible execution path on the device hardware.

Despite this controlled scope, the experiments retain key deployment-relevant elements. They use real Android smartphones, live camera input, and the same on-device inference pipeline used for the evaluated execution modes, rather than offline model-only benchmarking.

5.8. Method–Research Question Alignment

The methodology is linked directly to the three research questions. RQ1 is addressed through the controlled long-run comparison of sequential and heterogeneous execution. RQ2 is addressed through the feasibility analysis of heterogeneous execution, including cut-point structure, transfer overhead, and runtime constraints. RQ3 is addressed through the input-rate control experiments, which examine how camera-side pacing affects sustained inference behaviour and the interpretation of different execution strategies.

The evaluation scope is kept deliberately controlled. At the same time, the study remains close to the intended system context by using real devices, a live camera workload, and the evaluated on-device inference pipeline.

5.9. Methodological Limitations and Threats to Validity

The methodology has several limitations that should be considered when interpreting the results.

Internal validity. The experiments were conducted under controlled conditions, but some variation could not be removed completely. Possible sources include thermal noise across runs, background operating-system activity, vendor-specific runtime behaviour, and other device-level processes outside the application. The devices were allowed to cool between runs, and the setup was kept as consistent as possible. Even so, some residual variation may remain.

A fixed target initial battery temperature was not enforced before every run. Differences in starting thermal state may therefore have affected the early part of some runs. The devices also remained connected to a PC for logging. USB power and data transfer may have influenced charging state, background system behaviour, or thermal conditions. This condition was kept consistent across comparable runs.

External validity. The study evaluates two Android smartphones and a limited set of YOLO-based models. The findings are therefore most directly applicable to sustained mobile inference under similar hardware, software, and workload conditions. They may still be informative for a broader class of on-device vision workloads, but they should not be treated as representative of all mobile devices, frameworks, or model families.

Construct validity. The experimental workload uses a controlled live camera stream to preserve end-to-end capture and inference behaviour. It should be understood as a deployment-relevant approximation, not as a full reproduction of real-world visual diversity, motion, lighting variation, and scene complexity. The pipeline also uses a latest-frame handoff pattern. Configurations with longer or more variable inference latency may therefore skip more intermediate camera frames. Since skipped frames were not directly logged, two configurations with similar completed throughput may still have processed frames from different effective camera intervals. Future work should address this by logging per-frame camera timestamps and processed-frame identifiers. Android thermal-status labels are vendor-calibrated system indicators, not calibrated physical temperature measurements. For example, the same label, such as `THERMAL_STATUS_MODERATE`, may be triggered at different underlying temperatures or policy thresholds on different devices. These labels are useful for identifying relative system thermal pressure within a run. They should not be interpreted as precise or device-independent thermal states. Battery temperature was used as a secondary thermal signal. It gives a continuous trend across a run, but it is only a coarse proxy for device thermal progression. It is not a direct CPU, GPU, or SoC die-temperature measurement. The study also did not directly measure CPU/GPU utilization, clock frequencies, or operator-level runtime behaviour. Temperature-binned profiles should therefore be read as cooler-to-hotter trend views. Explanations involving underutilization, driver scheduling, Vulkan dispatch overhead, or operator-level bottlenecks remain interpretive rather than directly verified. For the evaluated YOLO models, most computation is performed by the fixed forward pass at the selected 320×320 input size. Scene content is therefore expected to affect throughput mainly through postprocessing and detection-output variability. Dynamic scenes may still change postprocessing cost and should be examined in future deployment-oriented studies.

Conclusion validity. Most configurations in the full experimental matrix were run once per experimental condition. The conclusions therefore rely mainly on consistent empirical patterns across configurations, not on large-sample statistical inference. To partially address this limitation, a targeted repeatability check was later conducted for selected capped-input configurations that were central to the interpretation of the results. This check provides evidence that the main capped-input findings were stable for those selected cases. It does not, however, amount to full statistical replication of every configuration in the experimental matrix. Score differences below approximately 1 FPS are therefore treated as approximate parity unless they are supported by consistent separation across the thermal-status and temperature-binned views. Larger differences, repeated trends across multiple views, and differences that persist across hotter thermal states provide stronger evidence than small score differences alone.

6. Ethical and Societal Considerations

This thesis does not involve human-subject experiments, interviews, or the collection of personal data for research. The study examines the sustained performance of on-device inference pipelines on Android smartphones under controlled experimental conditions. The live camera setup is used only to create a continuous real-time workload for systems evaluation. It is not used to analyse personal identity, behaviour, or other human-related attributes. For this reason, the work does not raise direct human-subjects research-ethics concerns in the conventional sense. Nevertheless, broader ethical aspects remain relevant.

Mobile object-detection systems may be deployed in settings where visual data includes people or activities. This work does not evaluate surveillance practices or decisions made about individuals. Still, such systems can raise privacy concerns depending on how they are deployed. One advantage of the on-device approach studied here is that inference can run locally on the smartphone. Visual data does not need to be transferred to external servers, which can reduce privacy exposure compared with cloud-based processing.

In a real deployment, especially in doorway areas or workplace-adjacent spaces, additional safeguards would be needed. People affected by the system should be informed that visual analytics are in use, and the deployment should be designed to avoid unnecessary collection, storage, or transmission of identifiable visual information. Any legal or organizational requirements would depend on the final deployment design and the applicable regulatory context. Data minimisation is also important: the system should collect and retain only what is necessary for the stated purpose. Access to outputs, logs, or stored data should be restricted and governed carefully. These safeguards matter even when processing is performed locally, because on-device inference can still support monitoring if used without proper controls. The doorway setting discussed in this thesis is therefore treated only as a motivating proof-of-concept context, not as a validated or approved deployment.

From a societal perspective, the work is relevant in two ways. First, it contributes to understanding long-running mobile AI systems, which are increasingly used in real-time analytics and edge-computing applications. Second, it has a sustainability dimension. The thesis is partly motivated by the reuse of existing Android smartphones as edge-AI devices. If such devices can sustain useful inference performance over long periods, their operational lifetime may be extended. This could help reduce electronic waste and improve resource efficiency.

Improved sustained inference performance is not automatically socially beneficial. More capable long-running mobile vision systems may also support more extensive monitoring in real environments. The technical findings should therefore be understood as performance-oriented. Their societal value depends on the broader deployment context, governance, and privacy safeguards.

7. Experimental Development and System Realization

This chapter describes how the experimental system was developed for the sustained mobile inference study. The experiments could not be treated as isolated benchmark runs. They required a practical inference pipeline that could run continuously on the evaluated Android smartphones, expose runtime behaviour over long periods, and support both sequential and heterogeneous execution modes.

The chapter first introduces the baseline inference pipeline that was available at the start of the work. It then explains how the pipeline was extended for sustained-performance measurement. Finally, it presents the initial heterogeneous execution design, which motivated the later feasibility and cut-point investigations. The focus is on implementation decisions that affected the experimental design and interpretation, not on a complete chronological development log.

7.1. Baseline Inference Pipeline

The experimental work in this thesis builds on an existing Android inference pipeline from the company development environment. This baseline pipeline was designed for real-time on-device object detection using NCNN and a live camera feed. In its original form, it supported sequential execution. A single model instance processed incoming frames end-to-end on the selected processor path. The pipeline could already run YOLO-based models on-device and display inference results, so it provided a suitable starting point for the sustained mobile inference experiments.

At a high level, the baseline flow had three main stages: camera capture, frame handoff, and inference. The Android camera continuously produced frames. The inference stage then acquired the most recent available frame whenever it was ready to process a new input. The system therefore did not try to process every frame generated by the camera. Frames arriving while inference was still busy were effectively skipped, and processing continued from the latest available frame. This behaviour is closer to a latest-frame producer–consumer pattern than to a managed queue that preserves all captured frames in order.

This design was reasonable for normal application use. In that setting, responsiveness to the newest scene content is more important than preserving every intermediate frame. However, the design was not sufficient for careful experimental control of sustained throughput and workload pacing. The camera continued to generate frames independently of inference speed. Therefore, the effective workload seen by the model depended not only on model execution time, but also on how frame acquisition interacted with the processing loop. Figure 6 illustrates this timing behaviour and the resulting dropped frames.

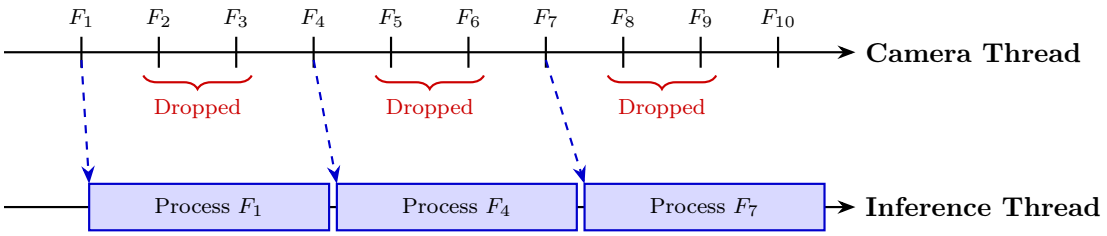

Figure 6: Timing diagram of the baseline inference pipeline, illustrating continuous frame generation and the effective dropping of intermediate frames (F_2, F_3, etc.) when the sequential inference stage is busy.

The dropped-frame behaviour in the baseline pipeline was not logged as an explicit per-frame counter in the original implementation. The uncapped baseline runs therefore do not include an exact measured drop ratio. However, the baseline camera path operated as a continuous producer, while throughput counted only frames that completed inference. In practical terms, when the completed inference rate was below the effective camera delivery rate, the difference corresponded to frames that were not processed. These frames were effectively overwritten by the latest-frame handoff.

The baseline pipeline should therefore be understood as a throughput-limited latest-frame system, not as a lossless frame-processing pipeline. In the reported measurements, latency was computed only for frames that entered and completed the processing path. Frames skipped at the latest-frame handoff did not contribute latency samples. Their effect appears indirectly through reduced completed throughput, rather than through inflated per-frame latency.

For the study, the baseline pipeline was important for two reasons. First, it provided a realistic deployment-oriented starting point instead of an isolated microbenchmark environment. Second, it showed early that the application already contained implicit workload-delivery behaviour. This behaviour could influence measured performance independently of the neural network execution itself. That observation later became important when the study expanded from model execution alone to the broader end-to-end inference pipeline.

7.2. Extending the Pipeline for Sustained Measurement

The baseline pipeline was a useful starting point, but it was not enough for the sustained empirical analysis required in this thesis. It did not directly support long-run experimental tracing, heterogeneous execution configurations, or systematic logging of performance over time. To run controlled sustained experiments, the surrounding pipeline therefore had to be extended.

The first set of changes focused on measurement and observability. Logging support was added to track throughput, latency, thermal-status transitions, battery-temperature readings, and split-pipeline timing during long-running experiments. Instrumentation points were added around the main camera and inference stages. These points recorded end-to-end frame timing, model inference timing, and selected heterogeneous stage timings over time.

Performance summaries were emitted every 60 *processed* frames, not every 60 camera-delivered frames. This distinction matters because the baseline latest-frame pipeline may skip intermediate camera frames while inference is busy. With this instrumentation, it became possible to analyse how inference behaviour changed as the device heated up and entered a thermally constrained operating regime. Without it, the system would have been suitable for normal application use, but not for scientific evaluation of sustained performance.

In the implemented pipeline, timing measurements used monotonic application-side timestamps. This avoided distortion from wall-clock changes during long runs. Performance summaries were written to CSV logs stored on the device. Diagnostic traces were also available through application logs during development and debugging. Thermal-status information came from the Android thermal interface, and battery temperature was recorded as an additional coarse thermal proxy. Logging overhead was not benchmarked as a separate experimental variable. However, the instrumentation was designed as lightweight runtime tracing rather than heavyweight offline profiling.

The second set of changes supported experimental control around the pipeline. The application did not include a fully automated experiment runner; runs were started and stopped manually, and settings were changed between runs. The key requirement was continuous execution with comparable logs under the selected configurations. Later, capped input-rate operation was added for the input-control experiments. These changes were made in the surrounding application code, not in the NCNN framework.

The third set of changes concerned execution flexibility. The original pipeline was built around sequential inference, where one processor path handled the full model. To study heterogeneous execution, the application had to support model execution across different processors. It also had to transfer intermediate outputs across a selected cut point. This was not only a configuration change. The execution flow had to be restructured around the existing inference components so that the system could support the intended CPU–GPU cooperation experiments.

These extensions transformed the baseline application into an experimental harness for sustained mobile inference. The resulting system kept the live on-device character of the original pipeline. It also added the instrumentation and execution flexibility needed for comparative long-run evaluation.

7.3. Initial Design of Heterogeneous Execution

The first heterogeneous design was based on a simple idea: use the GPU and CPU within the same inference path. Instead of running the complete model on one processor, the model would be stopped at a selected cut point. The part before the cut point would run on one processor, and the remaining part would run on the other. The intermediate tensor produced at the boundary would then be passed forward so that inference could continue to the final output. The intention was to make both processors contribute to the model computation, rather than using one processor only as a fallback.

In system terms, this produced a two-stage execution structure. A frame first entered the prefix stage, where the model was executed from the input to the selected cut point. The resulting intermediate tensor was then passed to the suffix stage, which completed the rest of the network. This structure could improve hardware utilization if both stages contained enough computation and if the handoff between them was inexpensive.

Figure 7 shows how this idea was realized in the experimental pipeline. The figure includes the separation between the camera capture and inference threads, the latest-frame handoff, the fork between sequential and heterogeneous NCNN execution paths, and the main synchronization and instrumentation points. It also makes the main implementation risks clear. The cut point had to be selected carefully. The boundary tensor had to be transferred efficiently. The two stages also had to be coordinated without losing the expected benefit through blocking, synchronization, or transfer overhead.

These issues shaped the later development work. The question was not only whether the model could be divided across the CPU and GPU. The more important question was whether that division remained useful on the evaluated smartphones during sustained operation.

7.4. Early Partitioning Attempt Using Split Model Files

One early approach was to physically split the NCNN model into two separate parts at a selected cut point. The idea was to create one partial model for the prefix of the network and another for the suffix. Each processor would then load and execute only the part assigned to it.

This approach was conceptually attractive because it matched the intended prefix–suffix execution structure. If the network could be separated into two valid NCNN submodels, heterogeneous execution would be straightforward. The first submodel would run up to the cut point, the intermediate tensor would be transferred, and the second submodel would continue from that tensor to the final output.

In practice, this approach did not work well for the target pipeline. NCNN models are graph-structured `param`/`bin` representations, not simple sequential layer lists. An internal cut therefore had to preserve valid blob names, graph continuity, and compatibility between the parameter and weight files on both sides of the split. This became especially difficult at non-trivial internal boundaries. In those cases, the suffix had to accept intermediate tensors with exactly the expected shape, name, and graph context. Boundaries with multiple outputs or skip-connected structure made the approach even more fragile. Given the available tooling and the application pipeline, the split-model-file approach was not a robust experimental path.

This failed attempt was still useful. It showed that heterogeneous partitioning was not only a model-design question. It was also a runtime and graph-validity problem. This led to a shift away from static file-level partitioning. Instead, the later implementation used a runtime realization that preserved valid model instances while still allowing execution to be divided at selected internal boundaries.

7.5. Final Heterogeneous Pipeline Realization

After the split-model-file approach proved unsuitable, the heterogeneous pipeline was implemented through runtime partitioning. The final design did not create two independent NCNN model files at arbitrary internal boundaries. Instead, it kept valid model instances intact and divided execution around a selected cut point during runtime. In practice, one execution stage produced the intermediate tensor at the cut point. A second stage then resumed from that tensor and completed the remaining computation.

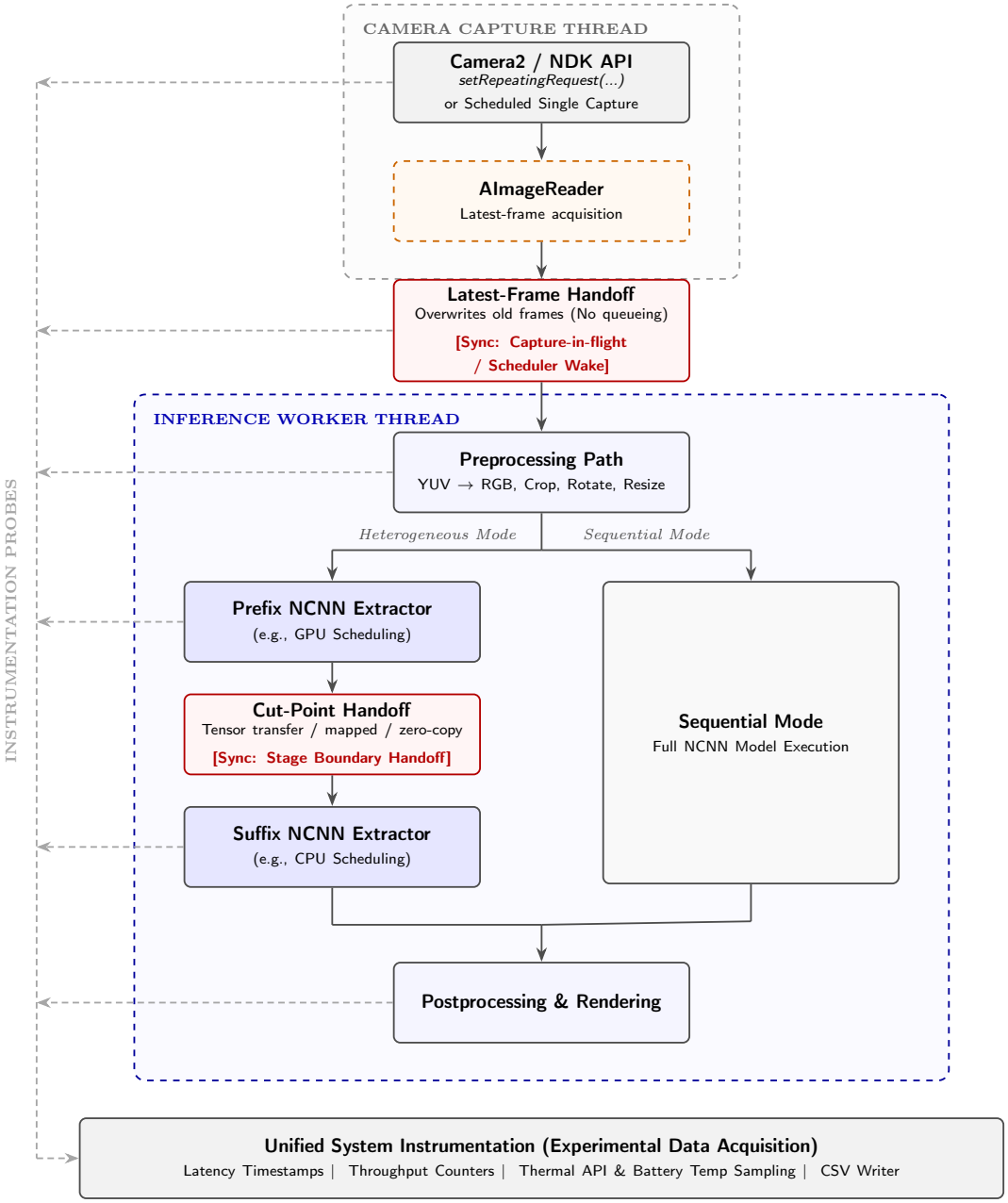

Figure 7: End-to-end experimental pipeline architecture. The diagram highlights the separation between camera capture and inference threads, key synchronization handoffs, and the fork between sequential and heterogeneous NCNN execution modes. A unified instrumentation backplane monitors latency, throughput, and thermal metrics across all stages.

This approach stayed compatible with the existing NCNN-based application structure. It also allowed different cut points to be tested without repeatedly rebuilding model files. The resulting pipeline therefore prioritized practical executability and experimental flexibility over a cleaner but more fragile static partitioning approach.

The realized heterogeneous flow consisted of three parts: a prefix stage, a boundary handoff, and a suffix stage. A live camera frame first entered the prefix stage. This stage executed the model from the input to the selected cut point on the assigned processor. The intermediate tensor produced at the boundary was then transferred or mapped across the handoff point. After that, the suffix stage completed the rest of the model on the other processor. Depending on the configuration, the prefix could run on the GPU and the suffix on the CPU, or the other way around. In practice, not all directions and cut locations were equally effective.

This realization changed the main practical question. Heterogeneous execution no longer depended only on whether a cut point existed in the model graph. It also depended on whether the boundary tensor could be handed off efficiently, whether the two stages could be coordinated without excessive blocking, and whether the split created a meaningful balance between the processors. These issues later helped explain why some heterogeneous configurations did not outperform strong sequential baselines.

The runtime realization also made later refinements possible. These included the study of transfer overhead and, where feasible, more direct handoff mechanisms. In this sense, it became the core execution structure for the later heterogeneous experiments.

7.6. Cut-Point Selection Problem

Chapter 2 introduced cut points as the boundaries where model execution can be divided between processors. Once heterogeneous execution was working inside the Android application, the next challenge was to choose suitable boundaries for sustained CPU–GPU experiments. This was not just a configuration choice. Each cut point changes how much computation is assigned to each processor. It also changes the size and number of intermediate tensors crossing the boundary, as well as the coordination cost between the two stages.

In NCNN, this problem is better viewed as a graph problem than as a simple split of a sequential layer list. An NCNN model can be represented as a directed layer–blob graph. In this graph, layers consume and produce named intermediate blobs. A cut point therefore divides the graph while preserving the blob names, tensor shapes, and dependencies needed by the suffix stage. Figure 8 illustrates this distinction using a simplified graph view.

A cut point that looks reasonable structurally may still perform poorly in practice. If the prefix stage is too small, most computation remains in the suffix and the split gives little opportunity for useful cooperation. If the prefix is too large, the suffix may be too short to justify the handoff overhead. The boundary structure also matters. Some cut points expose a single intermediate tensor that can be transferred relatively cleanly. Others require multiple blobs to cross the boundary. The latter case increases both implementation complexity and communication cost, making such cuts less attractive for sustained experimentation.

The cut-point problem is therefore treated as both a performance problem and an implementation problem. The goal is not only to find where the model can be divided. It is to find boundaries that remain practical and experimentally meaningful under real mobile constraints. In the main experimental scope, this led to a preference for cut points with simpler transfer structure, especially those involving a single exchanged tensor. A smaller number of multi-blob cases were kept only for exploratory feasibility testing.

This view also matches observations from prior mobile-systems research. Synchronization and data-movement overhead can offset the theoretical benefit of CPU–GPU cooperation, especially when the split is too fine-grained or the boundary activation is large. On mobile devices, this issue is particularly important because the CPU and GPU compete for limited shared memory resources. These considerations motivated a more systematic cut-point candidate-filtering strategy instead of ad hoc boundary selection.

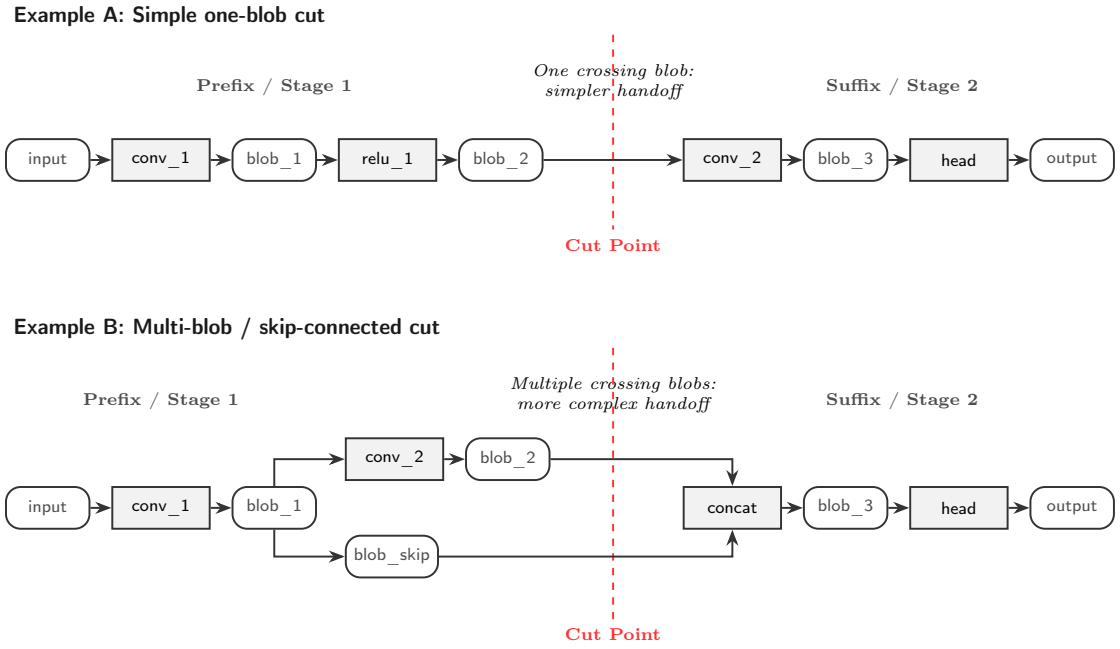

Figure 8: Simplified view of an NCNN model as a directed layer–blob graph. Layers consume and produce named blobs, and a cut point divides the graph into a prefix and suffix. A simple cut requires one intermediate blob to cross the boundary, whereas skip-connected or branching structures may require multiple crossing blobs, increasing implementation and coordination complexity.

7.7. Cut-Point Candidate Filtering

Because possible cut points vary in number and quality across a model, the study needed a systematic way to identify useful candidates. Choosing boundaries only by intuition would make the evaluation less reproducible and less informative. For this reason, a practical candidate-filtering procedure was used to rank cut-point candidates before heterogeneous configurations were implemented in the Android application.

The method was based on three observations. First, a useful split should leave non-trivial computation on both sides of the cut. Second, any benefit from dividing computation can be reduced or lost if the boundary requires too much intermediate data transfer. Third, cut-point suitability depends not only on the model graph, but also on the measured CPU-only and GPU-only performance of the target device.

To support this, the method estimates computational cost using MACs, or multiply–accumulate operations. MACs are used as an approximation of arithmetic workload at the layer and section level. The method also tracks *crossing blobs*, which are the intermediate tensors that must move from the prefix to the suffix when execution is divided at a cut point. The number and size of these blobs are used as a practical proxy for boundary overhead. In addition, measured whole-model CPU-only and GPU-only throughput values are used to give a device-aware indication of which processor is relatively better suited to each side of the split.

At a high level, the method takes an NCNN model, an input shape, and measured CPU-only and GPU-only throughput as input. It then parses the model graph, infers blob shapes, estimates per-layer MACs, and evaluates candidate cut points using both computation balance and boundary cost. Candidate cuts are filtered out if either side of the split is too small or if the crossing-blob structure is impractical for experimentation. For the remaining candidates, the method estimates a heuristic score for both execution directions, CPU→GPU and GPU→CPU. This score combines estimated section cost and boundary activation size. It is used to rank cut points, not to predict exact runtime.

The high-level filtering procedure is summarized in Algorithm 1.

The method is heuristic rather than strictly predictive. Its purpose is not to guarantee the

Algorithm 1 Candidate filtering for NCNN CPU/GPU cut points

Require: NCNN model M, input shape S, CPU FPS F_{cpu}, GPU FPS F_{gpu}
Ensure: Top-K cut candidates
Parse M and build the layer graph G
Infer the shape of every blob using S
Estimate MACs for each layer
Compute total MACs of the model
Initialize candidate set $C \leftarrow \emptyset$
for all valid layer l_i in G **do**
    Split G at l_i into Section 1 and Section 2
    Compute MAC share of Section 1 and Section 2
    if either section is too small **then**
        continue
    end if
    Find blobs crossing from Section 1 to Section 2
    if number of crossing blobs is invalid **then**
        continue
    end if
    Estimate boundary activation size from crossing blobs
    Estimate score for CPU→GPU execution
    Estimate score for GPU→CPU execution
    Keep the better direction and store this cut in C
end for
Rank all cuts in C by score (lower is better)
return Top-K cuts

optimal split. It is used to narrow the experimental search space to cut points that are more likely to be implementable and informative. This was important because implementing every possible internal boundary in the Android application would have been impractical. This limitation is consistent with prior work such as nn-Meter [23], which shows that edge-device inference performance is difficult to predict from model structure alone because hardware and runtime effects strongly influence latency. The candidate-filtering procedure was therefore used only to narrow the search space, not to replace empirical measurement on the target device.

In practice, the procedure favoured cuts that combined non-trivial computational division with relatively simple boundary structure. This matched later empirical observations. Cut points with a single exchanged tensor were generally easier to realize and reason about. Cut points with multiple crossing blobs quickly increased implementation effort and communication cost. The procedure should therefore be understood as a development aid for reducing the cut-point search space, not as a separately evaluated optimization algorithm. It identified feasible and informative candidates for the empirical study, while the actual usefulness of each selected cut was determined by the sustained experiments reported in Chapter 8.

7.8. Transfer Overhead and Zero-Copy Exploration

After candidate cut points had been identified and heterogeneous execution was working, the next challenge was the handoff of intermediate data between the two execution stages. In heterogeneous mobile inference, splitting computation across processors is only useful if the cut-point tensor can be transferred cheaply enough. If the transfer is expensive, it can reduce or remove the expected benefit. For this reason, transfer overhead was treated as a central part of the heterogeneous-execution investigation, not as a secondary implementation detail.

This issue was especially important on the evaluated smartphones. The CPU and GPU do not behave like separate desktop-class devices with independent high-bandwidth memory systems. Instead, they operate under shared mobile memory constraints. As a result, intermediate-tensor handoff can become a significant part of the total execution cost. There is also a pipeline-level concern. If the heterogeneous path behaves like a blocking relay rather than a meaningfully over-

lapped pipeline, total execution time becomes the combined cost of the prefix stage, the boundary handoff, and the suffix stage. In that case, heterogeneous execution can be worse than a strong sequential baseline, even when both processors are individually capable.

The study therefore compared the normal handoff of intermediate tensors with a zero-copy-oriented version where this was feasible. The zero-copy variant was introduced to reduce explicit copying at the cut point and to pass the boundary tensor more directly between the two stages. This type of handoff is most relevant when the boundary tensor is large, or when the second stage is not expected to save enough computation to compensate for a costly transfer.

Zero-copy was not assumed to solve the problem by itself. Removing an explicit copy can still leave other costs in the pipeline, including synchronization, memory visibility, runtime coordination, and imbalance between the two stages. For that reason, the zero-copy experiments were used to test whether a lower handoff cost actually improved sustained behaviour on the evaluated smartphones.

These experiments were included to support the interpretation of the heterogeneous results, not as a separate contribution of the thesis. They helped explain why some split configurations remained below the sequential baselines. Together with the cut-point analysis, they showed that the problem was not only where to divide the model. The split also had to leave enough useful work on both processors to justify the coordination cost.

7.9. Live Camera Workload Realization

The experiments used the phone camera directly, rather than replayed video clips or synthetic frame sequences. This was done because the application is intended to operate on a live camera stream. Using the camera during the experiments kept the input path closer to the real pipeline and made it possible to observe sustained inference under end-to-end operating conditions.

This setup was useful because it exposed behaviour that would be hidden in an offline benchmark. Frames were not injected at a fixed rate from a stored dataset. They were produced by the Android camera pipeline while inference was running. The workload seen by the model therefore depended on more than model execution time. It also depended on frame acquisition, the latest-frame handoff, and the timing of the processing loop.

The live setup also showed a limitation of the original pipeline. The inference system did not explicitly control how quickly new frames arrived. The camera could keep producing frames even when the model was still processing an earlier frame. The effective workload was therefore shaped by both inference speed and camera-side delivery behaviour. The physical setup used for this live workload is shown in Figure 9.

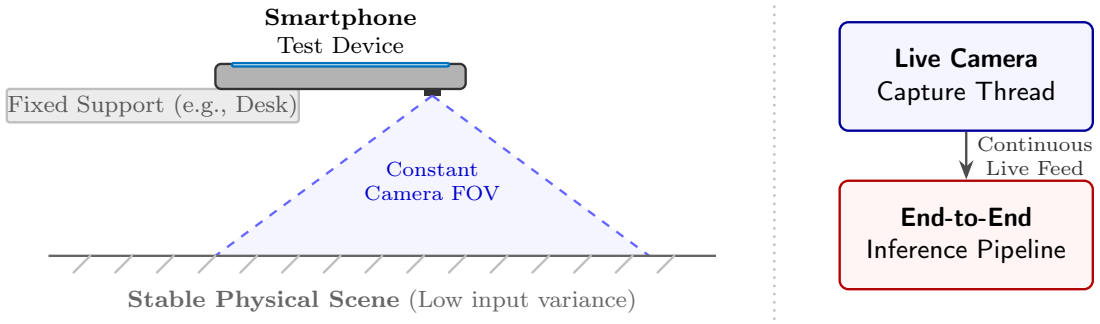

Figure 9: Realization of the live camera workload. The physical setup (left) utilizes a smartphone placed in a fixed position overlooking a stable scene. This reduces uncontrolled scene variance while preserving the continuous end-to-end behaviour of the software pipeline (right).

This became important later when camera-side input-rate control was added. The live camera workload was therefore not only an input source for the experiments. It was also the condition that made the limitations of the uncapped pipeline visible.

7.10. Camera Input-Rate Control Development

During the early sustained runs, it became clear that the CPU/GPU execution path was only part of the story. The camera pipeline also affected how much work reached the inference code. In the baseline setup, the camera kept producing frames continuously. The inference loop then picked up the newest available frame whenever it became free. This behaviour made sense for the application, because the displayed result stayed close to the current scene. For experiments, however, it made comparisons harder. The rate at which frames arrived was not directly tied to the rate at which the model could process them.

The first attempt to control this was single-capture operation. In this mode, frame capture was driven more directly from the inference loop. The aim was to avoid uncontrolled frame buildup and make input delivery follow the processing pace more closely. Although this gave more control in principle, it was not stable enough for the long 60-minute runs used in the main experiments. It also introduced practical limitations in the camera pipeline, so it was not used as the final setup.

A second approach explored burst-based capture. The idea was that grouped frame acquisition might help the pipeline maintain throughput while reducing some inefficiencies of the unconstrained baseline. In practice, burst-oriented operation was not the most suitable final solution for sustained evaluation. Workload pacing and long-run consistency still remained insufficiently controlled.

The most suitable approach was repeating-request execution with explicit frame-rate control through the Android camera interface. This allowed the camera to remain in continuous operation while limiting how quickly new frames entered the pipeline. Compared with the earlier approaches, it gave a better balance between deployment realism and experimental control. It preserved the live and continuously operating nature of the system, but reduced the mismatch between camera delivery rate and inference processing rate. The single-capture concept is illustrated in Figure 10, while the final capped repeating-request approach is illustrated in Figure 11.

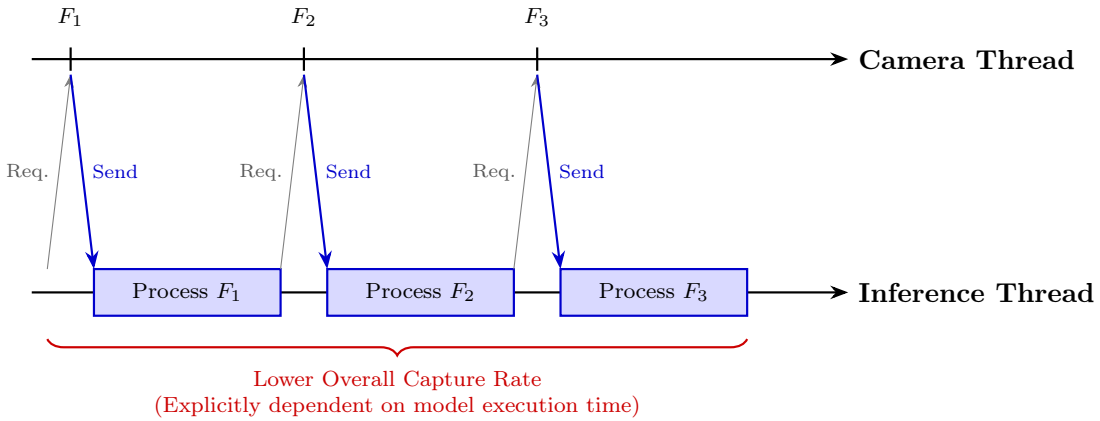

Figure 10: Single-capture input-rate control approach. The demand-driven model relies on an explicit request from the inference side to trigger one capture followed by processing, creating a tight dependency between execution speed and capture rate.

This development broadened the interpretation of the experiments. The results could no longer be understood only through model execution strategy; the surrounding input pipeline also had to be considered. Once camera-side pacing could be controlled more explicitly, selected sequential and heterogeneous configurations were re-evaluated under capped input-rate conditions.

In the capped-input experiments, each run label reports the configured camera FPS range in the form `min,max FPS`. For example, a label ending in `8,15 FPS` means that the Android camera interface was configured with a target range of 8–15 FPS. These ranges were selected manually based on earlier uncapped throughput observations and the practical processing range of each device–model configuration. The goal was not to impose one identical input rate across all experiments. Instead, the aim was to reduce uncontrolled workload pressure while keeping the input stream close to what each configuration could realistically process. This made it possible to distinguish execution-related effects from artefacts caused by uncontrolled workload delivery,

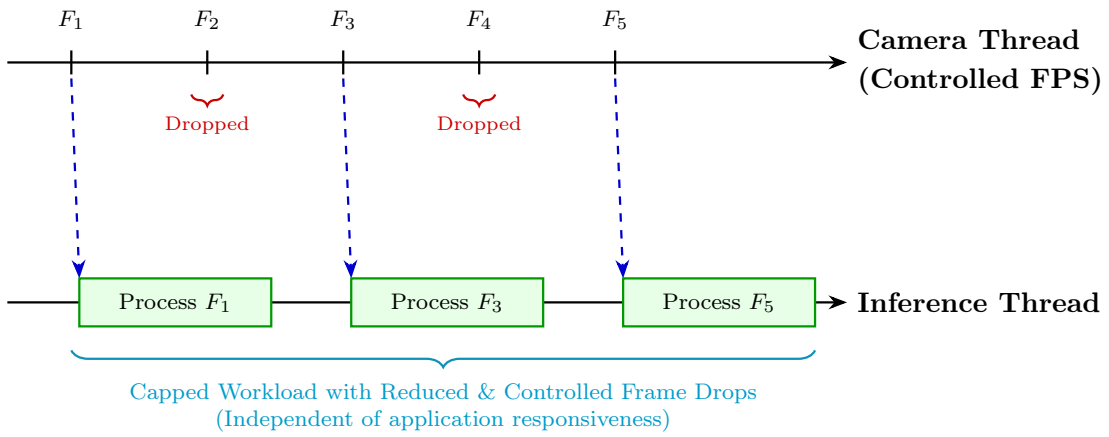

Figure 11: Repeating-request input-rate control approach with capped FPS. The input rate is explicitly paced. While intermediate frames may still be skipped when inference exceeds the capped interval, workload accumulation is constrained more tightly than in the uncapped baseline.

leading to a more realistic interpretation of sustained performance on the evaluated smartphones.

Input-rate control is therefore not treated here as a minor camera implementation detail. It is part of the core empirical finding that sustained mobile inference depends not only on how computation is divided between processors, but also on how the system regulates the flow of work over time.

7.11. Final Experimental Matrix

The development process in this chapter covered many experiments. These included sequential baseline runs, several heterogeneous realizations, exploratory cut-point trials, transfer-overhead checks, and later re-evaluation under controlled input-rate conditions. Not all of these configurations were suitable for formal comparison in the results chapter. The experimental space was therefore narrowed to a final matrix of configurations that were stable enough, representative, and methodologically comparable. Table 7 summarizes this matrix.

The final matrix includes the main device–model combinations, the sequential CPU-only and GPU-only baselines, selected heterogeneous configurations, and the subset later re-evaluated under capped input-rate conditions. It should be read as the set of configurations carried forward into the main comparative evaluation. It is not a complete record of every exploratory experiment performed during system development.

For the main heterogeneous evaluation, cut points with a single intermediate tensor crossing the processor boundary were preferred. This choice was made for practical reasons. One-blob cut points were easier to implement consistently, used simpler handoff logic, and gave a more reliable basis for 60-minute sustained runs. Cut points with multiple crossing tensors were harder to implement and made the execution path more fragile.

The study still included a small number of exploratory runs with two or three transferred tensors, including skip-connected boundaries. These runs were added to check whether the one-blob focus was too restrictive. In practice, the more complex cuts were much less robust. Some runs failed because of memory-related instability, including out-of-memory conditions. For this reason, these configurations were kept as exploratory feasibility cases rather than used as the main basis for sustained comparison.

The final matrix reflects a practical balance between coverage and reliability. It includes the configurations that were most suitable for sustained comparison across devices, models, execution strategies, and input-rate conditions. At the same time, it acknowledges the broader exploratory work that shaped the final evaluation design. The results in the next chapter are organized around this matrix.

Table 7: Final experimental matrix used for formal evaluation.

Device	Model	Execution mode	Representative cut-point type	Input-rate condition	Role in study
Device A	YOLOv8 Medium	CPU-only, GPU-only	None	Uncapped	Main sequential baselines
Device A	YOLOv8 Medium	Heterogeneous	1-blob cuts at early and late model boundaries (e.g., `cat_1`, `cat_16`)	Uncapped	Main heterogeneous evaluation
Device A	YOLOv8 Medium	Heterogeneous with persistent extractors / zero-copy-oriented transfer	1-blob cuts at early and late boundaries	Uncapped	Refined heterogeneous evaluation
Device B	YOLO-NAS Medium	CPU-only, GPU-only	None	Uncapped	Main sequential baselines
Device B	YOLO-NAS Medium	Heterogeneous	1-blob cuts at early and later boundaries (e.g., `convrelu_1`, `convrelu_8`)	Uncapped	Main heterogeneous evaluation
Device B	YOLO-NAS Medium	Heterogeneous	Multi-blob / skip-connected cut (e.g., `splitncnn_3`)	Uncapped	Exploratory feasibility only
Device A	YOLO-NAS Small	CPU-only, GPU-only	None	Uncapped	Main sequential baselines
Device A	YOLO-NAS Small	Heterogeneous	1-blob cut at early boundary (e.g., `convrelu_8`)	Uncapped	Main heterogeneous evaluation
Device A	YOLO-NAS Medium	CPU-only, GPU-only	None	Uncapped	Main sequential baselines
Device A	YOLO-NAS Medium	Heterogeneous	1-blob cuts at early boundaries (e.g., `convrelu_1`, `convrelu_8`)	Uncapped	Main heterogeneous evaluation
Device B	YOLO-NAS Small	CPU-only, GPU-only	None	Uncapped	Main sequential baselines
Device B	YOLO-NAS Small	Heterogeneous	1-blob cut at early boundary (e.g., `convrelu_8`)	Uncapped	Main heterogeneous evaluation
Device A	YOLO-NAS Small, YOLO-NAS Medium, YOLOv8 Medium	CPU-only, GPU-only, selected heterogeneous	Main 1-blob cuts only	Capped FPS	Re-evaluated under controlled input rate
Device B	YOLO-NAS Small, YOLO-NAS Medium	CPU-only, GPU-only, selected heterogeneous	Main 1-blob cuts only	Capped FPS	Re-evaluated under controlled input rate

7.12. Recorded Measurements

For each run, the application wrote a set of runtime measurements to the experiment logs. These logs were needed to follow three things over the 60-minute runs: how many frames completed inference, how long the processing steps took, and how the phone's thermal state changed over time. The main signals used in the analysis were completed throughput, latency, battery temperature, Android thermal status, and, for heterogeneous runs, stage-level timing information.

7.12.1 Throughput and Latency

The main performance measure in this thesis is *throughput*. Here, throughput means the number of frames that actually completed the inference pipeline per second. This is different from the nominal camera frame rate. The camera may deliver more frames than the model can process, and the latest-frame handoff can skip intermediate frames while inference is busy. For this reason, camera FPS alone would not show how much inference work was actually completed. Throughput is therefore used as the main performance metric.

Latency was recorded as well, but it is used only as a supporting diagnostic signal. The

thesis does not compare execution strategies mainly by latency. In the logs, *inference latency* refers to the time spent inside the model inference stage. *Total latency* refers to the broader frame-processing time measured by the application pipeline. These values help interpret pipeline behaviour, especially for heterogeneous runs, but the main results discussion remains based on throughput.

The analysis reports p50 and p95 summaries for both throughput and latency. The p50 value represents the median behaviour. For latency, p95 captures the slower tail of the distribution. For throughput-related timing intervals, it is used as an indicator of less favourable behaviour near the tail.

7.12.2 Thermal Measurements and Android Thermal Labels

Thermal behaviour was tracked using battery temperature and Android thermal status. Battery temperature gave a continuous trace during each run. Android thermal status gave a coarser system-level signal of how much thermal pressure the operating system reported. The official Android thermal-status meanings used in this thesis are summarized in Table 8.

Table 8: Android thermal status labels used in the analysis.

Thermal label	**Official description**
`THERMAL_STATUS_NONE`	Not under throttling
`THERMAL_STATUS_LIGHT`	Light throttling where UX is not impacted
`THERMAL_STATUS_MODERATE`	Moderate throttling where UX is not largely impacted
`THERMAL_STATUS_SEVERE`	Severe throttling where UX is largely impacted
`THERMAL_STATUS_CRITICAL`	Platform has done everything to reduce power
`THERMAL_STATUS_EMERGENCY`	Key components in platform are shutting down due to thermal condition; device functionalities will be limited

These labels are used as approximate indicators of increasing thermal stress. They should not be read as exact or device-independent performance states. In the analysis, Android thermal status is used as the main system-level thermal signal, while battery temperature is used to show the cooler-to-hotter trend within a run. For the temperature-binned plots, battery temperature was grouped into 1 °C bins, with each point plotted at the bin midpoint.

7.12.3 Additional Measurements for Heterogeneous Runs

For heterogeneous execution, the logs include additional stage-level timing measurements. These measurements are used to interpret split-pipeline behaviour. In this thesis, *Stage 1* denotes the execution stage before the cut point, and *Stage 2* denotes the stage after the cut point. These timings make it possible to see whether one side of the split dominates the total runtime.

The heterogeneous analysis also uses coordination-related quantities such as *boundary overhead*, *queue time*, and *gap time*. Boundary overhead refers to the cost of handing off the intermediate tensor at the cut point. Queue time and gap time represent waiting or coordination delays between the two stages. In the logs, queue time refers to time spent waiting before a stage can begin processing. Gap time refers to the idle interval between the completion of one stage and the start of the next stage. These measurements are used mainly to explain heterogeneous behaviour in the later profiling-based analysis, rather than as standalone headline metrics.

The main comparative results in Chapter 8 emphasize throughput-oriented outcomes. The additional profiling signals are used there to interpret representative heterogeneous bottlenecks and coordination failures. Table 9 summarizes which signals were directly instrumented in the experimental pipeline, how they were used in the thesis, and which limitations remain in their interpretation.

Table 9: Measurement coverage of the experimental instrumentation.

Signal	**Directly Instrumented?**	**Source / Implementation**	**Use in Thesis**	**Limitation**
Throughput	Yes	Completed result intervals from runtime logs / CSV	Main comparative metric	Reflects processed frames only
Inference latency	Yes	Application-side monotonic timing around inference	Supporting interpretation	Excludes skipped frames
Total latency	Yes	Application-side monotonic timing across frame path	Supporting interpretation	Not the main optimization target
Android thermal status	Yes	Android thermal API	Thermal regime interpretation	Coarse system-level label
Battery temperature	Yes	Android battery-temperature reading	Thermal proxy	Not die-level CPU/GPU temperature
Stage 1 / Stage 2 timings	Yes, heterogeneous runs only	Split-pipeline instrumentation	Bottleneck diagnosis	Available only for split runs
Boundary overhead / queue / gap time	Yes, heterogeneous runs only	Split-pipeline instrumentation	Coordination-cost interpretation	Used mainly in profiling analysis
Camera FPS (uncapped baseline)	No direct per-frame log	Inferred only conceptually from camera-driven pipeline	Structural interpretation only	No exact time series reported
Drop ratio	No direct per-frame log	Not explicitly instrumented	Not reported as a direct metric	Can only be inferred indirectly
CPU/GPU idle time	No	Not explicitly instrumented	Not a headline metric	Discussed indirectly via stage imbalance
Pipeline stall frequency	No explicit standalone counter	Not directly logged as a separate metric	Discussed through queue/gap behaviour	No dedicated stall-rate series

8. Results

This chapter presents the empirical results of the sustained mobile inference study. The focus is on two issues: how sequential and heterogeneous execution compare during long-running operation on the evaluated Android smartphones, and how thermal behaviour and input-rate control affect sustained performance. The results are organized around the final experimental matrix defined in Chapter 7, not around the full chronological sequence of development experiments.

The experimental work included many intermediate runs, exploratory trials, and implementation-oriented checks. The chapter therefore does not present every run individually. Instead, this chapter focuses on the configurations most relevant to the research questions. Selected exploratory cases are included only when they help explain practical limitations or unexpected behaviour. This keeps the results focused while preserving the engineering context needed to interpret them.

The chapter begins with the sequential baselines, which provide the reference point for later heterogeneous comparisons. It then examines heterogeneous execution across selected cut points and execution directions. This is followed by profiling-based analysis of split-pipeline bottlenecks and transfer-related optimizations. The chapter then presents the controlled input-rate results and uses them to revisit the main findings under a more stable workload-delivery setup. It concludes with a qualitative discussion of energy considerations and thermal coupling.

8.1. Result Presentation Strategy

To keep the results interpretable, the experiments are presented in groups rather than as a complete run-by-run log. For each comparison group, the aim is to show which configuration performed better and how its performance changed under increasing thermal pressure. The chapter therefore uses three complementary views of sustained performance.

The first view is median throughput p50 by Android thermal status. This gives a compact view of how each run behaves as the device moves through increasing levels of thermal stress. It is used as the main interpretive chart because it is easy to read and closely reflects the device's runtime thermal condition.

The second view is the temperature-binned throughput profile. This shows how throughput changes across the observed battery-temperature range. It is useful for identifying crossover behaviour, where one configuration performs better at lower temperatures but worse at higher temperatures.

The third view is the thermal-status-aware throughput score defined in Chapter 5. This score provides a compact ranking-oriented summary. However, it is not a time-weighted measure and should be interpreted together with the highest thermal status reached, especially when runs reach different thermal-state ranges.

These three views are used together because no single representation is sufficient. Thermal-status summaries are easier to interpret at a systems level. Temperature-binned profiles preserve more detailed thermal behaviour. The score provides a concise comparison aid, but it is simplified.

The sustained regime is interpreted operationally rather than by a fixed timestamp. A run is treated as more thermally constrained when it reaches higher Android thermal-status levels or shows clear throughput degradation at higher battery-temperature bins. For capped runs that only reached moderate thermal states within the 60-minute window, the device may not have entered a hot sustained regime comparable to the uncapped runs. Such configurations should therefore be interpreted as sustained 60-minute behaviour under controlled workload delivery, not as evidence of performance after severe thermal saturation. Using these views in combination keeps the results readable while still reflecting the sustained-performance behaviour observed in the experiments.

The chapter is also organized around the research questions. Section 8.2 establishes the sustained sequential baselines and addresses RQ1 from the sequential reference perspective. Section 8.3 compares the main heterogeneous configurations against those baselines, addressing RQ1 and contributing to RQ2. Section 8.4 adds profiling-based and exploratory heterogeneous results, mainly to answer RQ2 by explaining why some splits succeed, underperform, or fail. Section 8.5 addresses RQ3 by re-evaluating selected sequential and heterogeneous configurations under controlled input-rate conditions.

For heterogeneous runs, these views are complemented where necessary by stage-level profiling

information. In those cases, the goal is not only to compare throughput, but also to explain why a split succeeds or fails in practice. Stage imbalance, handoff overhead, and coordination effects are therefore introduced only where they are needed to interpret the observed behaviour.

A final point concerns uncertainty and variance. The reported p50 and p95 values summarize within-run behaviour over long executions. They provide a robust view of how a configuration behaved during a given run. However, most experimental conditions were executed once. Only a smaller number were repeated when reruns were needed after crashes or interruptions, as discussed earlier in the methodology. For that reason, most comparisons in this chapter do not use formal cross-run statistics such as standard deviation or confidence intervals.

A targeted repeatability check is reported separately in Section 8.6 for selected capped-input configurations. This check assesses the stability of the main capped-input findings, but it does not statistically replicate the full experimental matrix. Smaller differences between configurations should therefore be interpreted cautiously. Larger and repeated patterns provide stronger evidence, especially when they appear consistently across the thermal-status plots, temperature-binned profiles, and score summaries. In particular, score differences below approximately 1 FPS are treated as approximate parity unless they are also supported by clear separation in the thermal-status and temperature-binned views. This is a practical interpretation rule, not a formal statistical threshold. It is also consistent with the targeted repeatability check in Section 8.6, where the largest observed score range was 0.78 FPS.

8.2. Sequential Baseline Results

This section addresses RQ1 by establishing the sustained sequential reference behaviour. These baselines are later used to evaluate heterogeneous execution.

The results cover the evaluated device–model combinations. Their purpose is to define the reference point for later heterogeneous comparisons. Because the thesis focuses on sustained behaviour rather than short initial bursts, the baselines are examined from three perspectives: the thermal-status-aware score, median throughput by Android thermal status, and throughput across the measured battery-temperature range. Table 10 summarizes the overall rankings and the highest thermal states reached.

Table 10: Summary of sequential baseline results.

Rank	Device	Model	Execution Mode	Thermal-status-aware score (FPS)	Highest thermal status reached
1	Device A	YOLO-NAS-S	CPU	18.50	CRITICAL
2	Device B	YOLO-NAS-S	GPU	12.86	SEVERE
3	Device A	YOLOv8-M	CPU	12.31	CRITICAL
4	Device A	YOLO-NAS-M	CPU	10.89	CRITICAL
5	Device A	YOLO-NAS-S	GPU	9.64	SEVERE
6	Device A	YOLOv8-M	GPU	8.20	SEVERE
7	Device B	YOLO-NAS-M	GPU	8.11	SEVERE
8	Device B	YOLO-NAS-S	CPU	7.70	SEVERE
9	Device A	YOLO-NAS-M	GPU	6.82	SEVERE
10	Device B	YOLO-NAS-M	CPU	5.69	MODERATE

Note: Scores should be interpreted together with the highest thermal status reached.

Figure 12 presents the thermal-status-aware throughput score for all sequential runs. It gives a compact ranking-oriented summary of sustained performance over the first 60 minutes. The strongest sequential baseline is Device A with YOLO-NAS-S on CPU, which reaches 18.50 FPS and enters the CRITICAL thermal state. The next strongest configurations are Device B with YOLO-NAS-S on GPU at 12.86 FPS, Device A with YOLOv8-M on CPU at 12.31 FPS, and Device A with YOLO-NAS-M on CPU at 10.89 FPS. At the lower end, Device B with YOLO-NAS-M on CPU and Device B with YOLO-NAS-S on CPU perform clearly worse than their GPU counterparts. The ranking already shows a device-dependent pattern: on Device A, CPU execution is the stronger sequential baseline for all evaluated models, while on Device B, GPU execution is stronger for both YOLO-NAS-S and YOLO-NAS-M.

Figure 13 shows median throughput p50 by Android thermal status. This view makes the sustained behaviour easier to interpret than the score alone. Device A with YOLO-NAS-S on

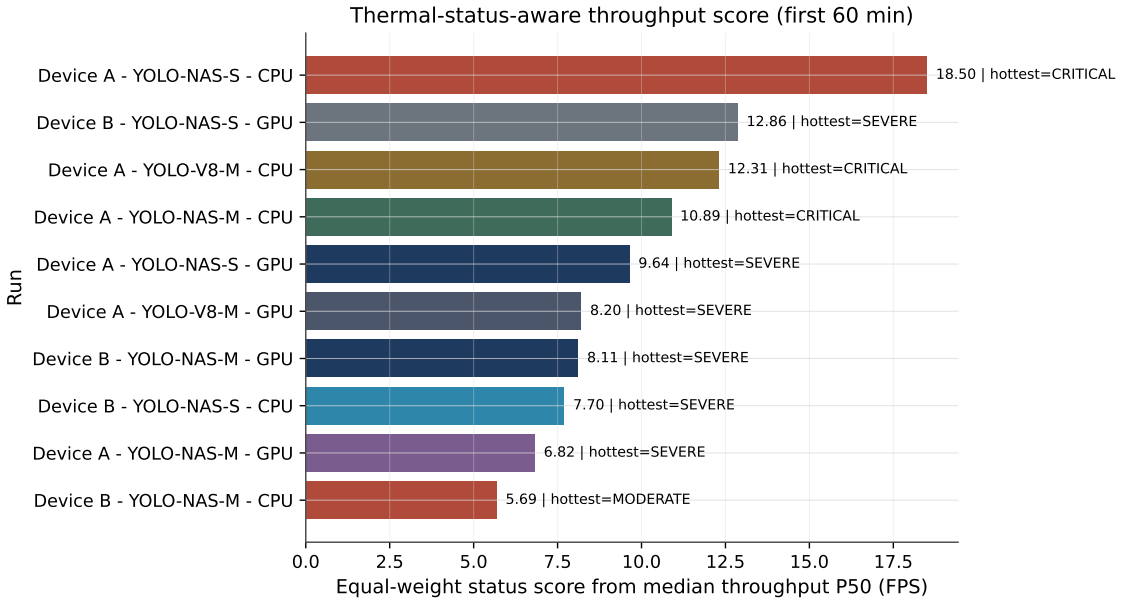

Figure 12: Thermal-status-aware throughput score for all sequential baselines.

CPU remains clearly ahead across the thermal progression. It starts above 20 FPS in the NONE state and still sustains approximately 15.5 FPS at CRITICAL. Device A with YOLOv8-M on CPU also degrades gradually but remains comparatively strong, while the corresponding GPU baseline stays consistently lower. A similar pattern appears for YOLO-NAS-M on Device A, where the CPU baseline remains above the GPU baseline across the reported thermal states. On Device B, the pattern is reversed. GPU execution is consistently stronger than CPU execution for both YOLO-NAS-S and YOLO-NAS-M. In particular, Device B with YOLO-NAS-S on GPU remains near 13 FPS from NONE to SEVERE, whereas the CPU version declines much more sharply.

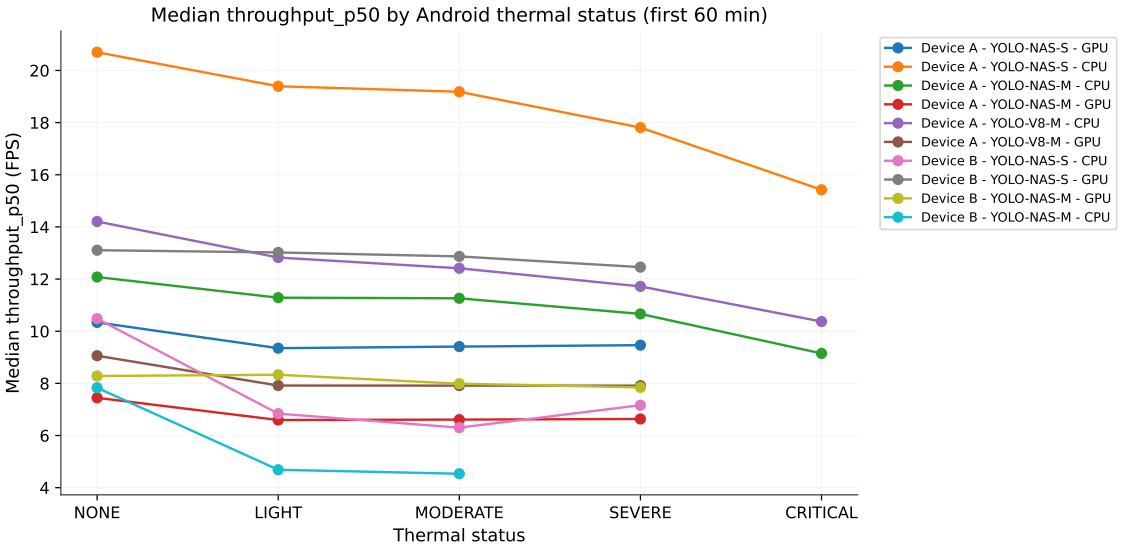

Figure 13: Median throughput p50 by Android thermal status.

Figure 14 complements the thermal-status view with battery-temperature-binned throughput profiles. The same overall ranking holds across most of the observed temperature range, but the temperature view shows more detail. Some runs degrade gradually and remain relatively flat over several bins. Others decline more steeply once the device enters the hotter part of the run.

This is especially visible for Device B CPU-based runs, which drop more sharply with increasing temperature than their GPU counterparts. Device A CPU baselines, particularly for YOLO-NAS-S, retain higher throughput over a wider temperature range. The GPU baselines tend to level off at lower values. The temperature-binned view therefore shows not only which configuration ranks higher overall, but also how that advantage changes as the device heats up.

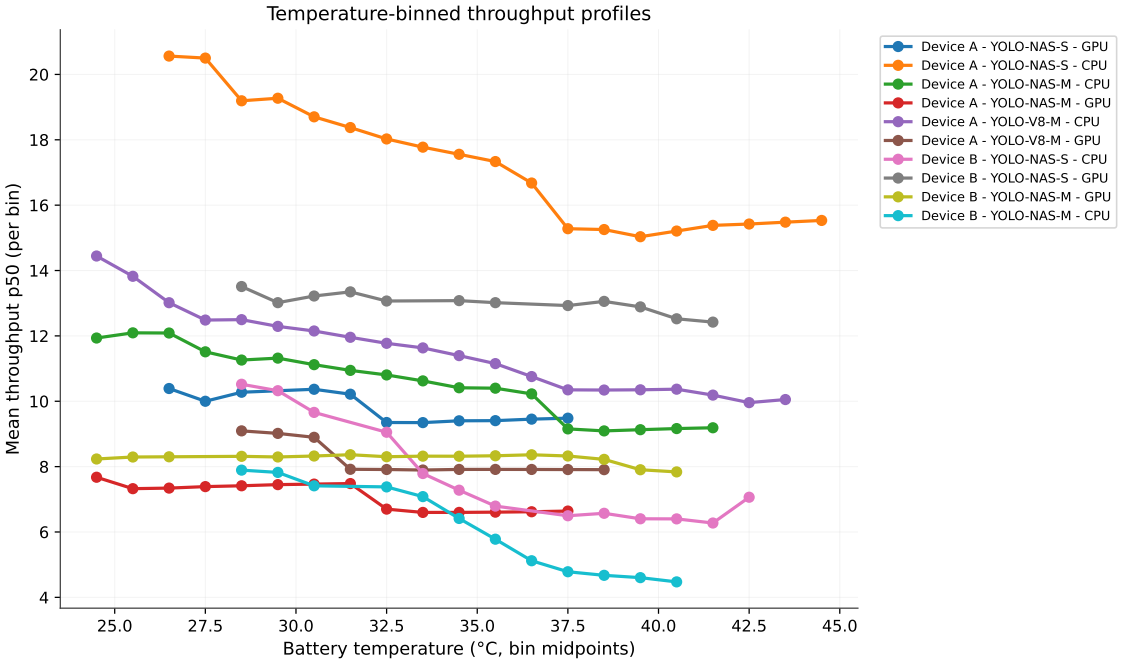

Figure 14: Temperature-binned throughput profiles.

The sequential baseline results establish two clear findings. First, the best sequential processor depends strongly on the device–model combination rather than on a single general rule. Second, sustained performance must be judged under thermal progression rather than by initial throughput alone. These baselines provide the reference point for evaluating whether heterogeneous execution delivers a practical benefit in the later sections.

8.3. Heterogeneous Execution Results: Main Comparison Against Sequential Baselines

This section addresses RQ1 by comparing sustained heterogeneous execution against the sequential baselines. It also contributes to RQ2 by showing how the outcome depends on cut-point location and execution direction. The best sequential baseline should be interpreted as a post-profiling reference. It represents the stronger of the available CPU-only and GPU-only results after both have been measured. It is therefore not necessarily the processor choice that would be known in advance for a new device or model. This makes the comparison strict for heterogeneous execution, because each split configuration must compete with the strongest observed sequential alternative. At the same time, this baseline should be understood as an optimistic sequential reference.

After establishing the strongest sequential results in Section 8.2, this section examines whether simple heterogeneous execution can provide sustained performance benefits over those references. The focus is on the main one-blob heterogeneous configurations introduced earlier in the thesis. In these configurations, the model is divided at a selected cut point and executed across the CPU and GPU in either direction. The purpose here is not yet to explain the internal causes of success or failure in detail. Instead, the aim is to determine which heterogeneous configurations are competitive with the best sequential alternatives under sustained operation.

To keep the presentation manageable, the results are grouped by device and model. Within each subgroup, the main simple heterogeneous runs are compared against the corresponding sequential

CPU-only and GPU-only baselines. The comparison uses the same three views introduced earlier: the thermal-status-aware throughput score, median throughput by Android thermal status, and, where useful, temperature-binned throughput profiles. This allows heterogeneous and sequential execution to be compared consistently while preserving the thermal context of the runs.

Across the following subsections, the key questions are whether heterogeneous execution improves sustained performance over the best sequential baseline, how strongly the outcome depends on cut-point location and execution direction, and whether any observed advantage persists once the device reaches its hotter sustained operating region.

8.3.1 Device A — YOLOv8 Medium

This subsection compares the main simple heterogeneous configurations for YOLOv8 Medium on Device A with the sequential CPU-only and GPU-only baselines. The heterogeneous cases use two cut points, `cat_1` and `cat_16`, in both directions: GPU→CPU and CPU→GPU. Table 11 summarizes these runs.

Table 11: Simple heterogeneous execution results for YOLOv8 Medium on Device A.

Configuration	Thermal-status score (FPS)	Highest thermal status	Diff vs best sequential
Sequential CPU	12.31	CRITICAL	0.00
Sequential GPU	8.20	SEVERE	−4.11
Split @ `cat_16` (GPU→CPU)	8.47	SEVERE	−3.84
Split @ `cat_16` (CPU→GPU)	12.73	CRITICAL	+0.42
Split @ `cat_1` (GPU→CPU)	12.15	CRITICAL	−0.16
Split @ `cat_1` (CPU→GPU)	9.79	CRITICAL	−2.52

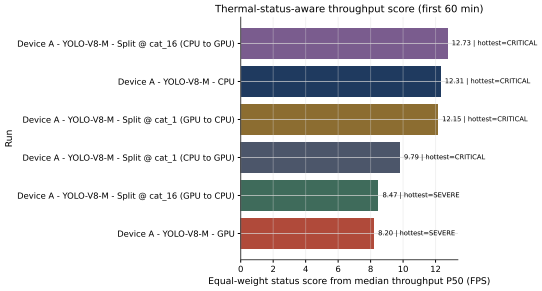

Figure 15: Thermal-status-aware throughput score.

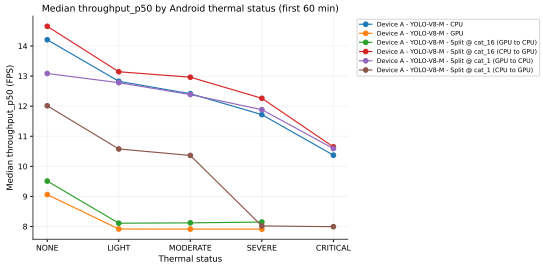

Figure 16: Median throughput by Android thermal status.

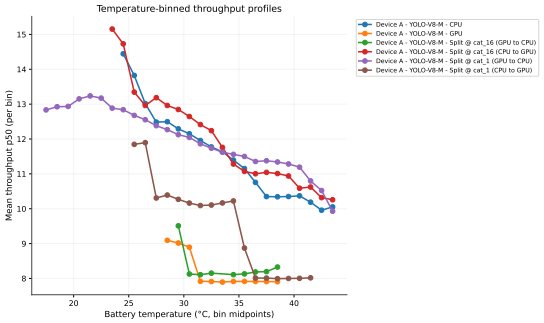

Figure 17: Temperature-binned throughput profiles.

Figure 15 shows the thermal-status-aware throughput score for the six configurations. The strongest score comes from the `cat_16` split in the CPU→GPU direction, at 12.73 FPS. It is

followed closely by the sequential CPU baseline at 12.31 FPS and the `cat_1` split in the GPU→CPU direction at 12.15 FPS. By contrast, the sequential GPU baseline reaches only 8.20 FPS. The `cat_16` split in the GPU→CPU direction is similarly weak, at 8.47 FPS. These scores show that simple heterogeneous execution can outperform the weaker sequential baseline. In the best case, it can also slightly exceed the strongest sequential baseline. However, the margin over the CPU-only run is small.

Figure 16 adds the thermal-status view. Across the thermal states, the sequential CPU baseline and the two stronger heterogeneous runs, `cat_16` CPU→GPU and `cat_1` GPU→CPU, remain close to each other in the upper part of the plot. The sequential GPU baseline and the weaker heterogeneous alternatives remain clearly lower. This means that the better heterogeneous configurations do not create a clearly different sustained regime from the CPU-only baseline. They mainly remain competitive with it as thermal pressure increases. The benefit of simple heterogeneous execution on this device is therefore limited and depends strongly on the chosen split.

Figure 17 shows the temperature-binned throughput profiles. In the lower-temperature region, some heterogeneous configurations have a visible advantage over the sequential baselines. The `cat_16` CPU→GPU split starts with the highest throughput in the plotted range, while the `cat_1` GPU→CPU split also remains strong through much of the early and middle temperature range. As temperature rises, however, these advantages become smaller. The stronger heterogeneous runs gradually move closer to the sequential CPU baseline, and their separation is much smaller in the hotter part of the range. This supports the interpretation that simple heterogeneous execution can provide an initial throughput boost on Device A, but that the gain is less clear once the device settles into a thermally constrained sustained regime.

The results also show that not all heterogeneous configurations are useful to the same extent. The `cat_16` GPU→CPU split and the sequential GPU baseline remain near the bottom of the comparison. The `cat_1` CPU→GPU split performs only moderately and stays below the best alternatives. This reinforces one of the central findings of the thesis: heterogeneous execution is highly sensitive to cut-point placement and execution direction. For this device and model, only some simple heterogeneous splits are competitive with the strongest sequential baseline. Even those offer only a limited sustained advantage.

For YOLOv8 Medium on Device A, the results suggest that simple heterogeneous execution can improve over weaker sequential alternatives and may slightly exceed the best sequential CPU baseline in the score summary. However, the best heterogeneous score is only 0.42 FPS above the sequential CPU score. Since most configurations were executed once, this difference should be interpreted as approximate parity rather than as a decisive advantage. The stronger heterogeneous splits therefore remain close to the CPU-only baseline. This motivates a closer examination of split-pipeline bottlenecks and later refinement strategies such as zero-copy execution.

8.3.2 Device A — YOLO-NAS Medium

This subsection compares the main simple heterogeneous configurations for YOLO-NAS Medium on Device A with the sequential CPU-only and GPU-only baselines. For this model, the feasible one-blob cut points were limited to `ConvRelu_1` and `ConvRelu_8`. Both are relatively early in the network. Since the sequential results in Section 8.2 showed that the CPU is stronger for this model on Device A, the heterogeneous evaluation focused on the GPU→CPU direction. In this setup, the smaller prefix runs on the GPU and the larger suffix runs on the CPU.

Table 12: Simple heterogeneous execution results for YOLO-NAS Medium on Device A.

Configuration	Thermal-status score (FPS)	Highest thermal status	Diff vs best sequential
Sequential CPU	10.89	CRITICAL	0.00
Sequential GPU	6.82	SEVERE	−4.07
Split @ `ConvRelu_1` (GPU→CPU)	8.85	CRITICAL	−2.04
Split @ `ConvRelu_8` (GPU→CPU)	11.15	CRITICAL	+0.26

Table 12 summarizes the results. In the thermal-status-aware score, shown in Figure 18, the `ConvRelu_8` GPU→CPU split achieves the highest value at 11.15 FPS. This is slightly above the

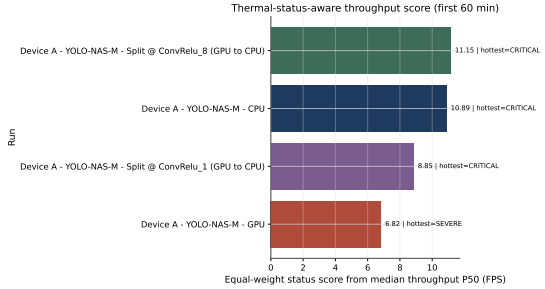

Figure 18: Thermal-status-aware throughput score.

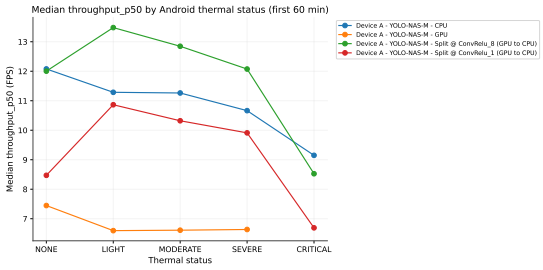

Figure 19: Median throughput by Android thermal status.

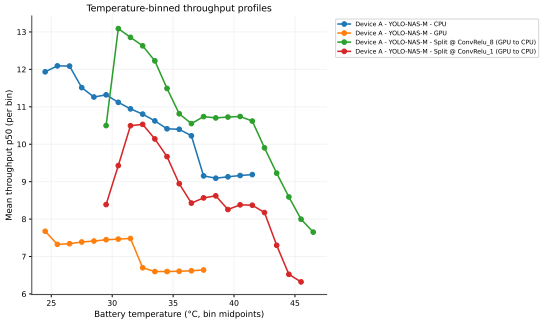

Figure 20: Temperature-binned throughput profiles.

sequential CPU baseline at 10.89 FPS. The `ConvRelu_1` split reaches 8.85 FPS, while the sequential GPU baseline is clearly weaker at 6.82 FPS. These values indicate that a carefully chosen simple heterogeneous split can be competitive with the strongest sequential baseline on this device and model. However, the margin over the sequential CPU baseline is only 0.26 FPS. This result should therefore be treated as approximate parity rather than as a decisive improvement.

The detailed thermal views make this advantage more cautious. In Figure 19, the `ConvRelu_8` split begins above the sequential CPU baseline and remains strong through the earlier thermal states. As thermal pressure increases, however, the gap narrows. Figure 20 shows the same pattern across temperature. The heterogeneous split performs best in the lower and middle parts of the observed range, but its lead becomes smaller in the hotter sustained region. By contrast, the `ConvRelu_1` split remains weaker throughout and does not become competitive with the sequential CPU baseline.

For YOLO-NAS Medium on Device A, the results suggest that simple heterogeneous execution can provide a short-term improvement when the split is chosen carefully. The sustained benefit, however, is modest. The stronger `ConvRelu_8` configuration is best understood as a case that looks promising early in the run, but does not establish a clearly stronger long-run regime than the sequential CPU baseline.

8.3.3 Device A — YOLO-NAS Small

For YOLO-NAS Small on Device A, the simple heterogeneous results are more encouraging than for YOLO-NAS Medium. Figure 21 shows that the `ConvRelu_8` GPU→CPU split achieves the highest thermal-status-aware throughput score, at 19.59 FPS. This is slightly above the sequential CPU baseline at 18.50 FPS. The `ConvRelu_1` GPU→CPU split reaches 18.00 FPS, while the sequential GPU baseline remains far behind at 9.64 FPS. Table 13 summarizes these configurations.

The median-throughput-by-thermal-status view in Figure 22 and the temperature-binned profile in Figure 23 support the same interpretation. The `ConvRelu_8` GPU→CPU split stays above the sequential CPU baseline across most of the observed range. It also shows the strongest sus-

Table 13: Simple heterogeneous execution results for YOLO-NAS Small on Device A.

Configuration	Thermal-status score (FPS)	Highest thermal status	Diff vs best sequential
Sequential CPU	18.50	CRITICAL	0.00
Sequential GPU	9.64	SEVERE	−8.86
Split @ `ConvRelu_1` (GPU→CPU)	18.00	CRITICAL	−0.50
Split @ `ConvRelu_8` (GPU→CPU)	19.59	CRITICAL	+1.09

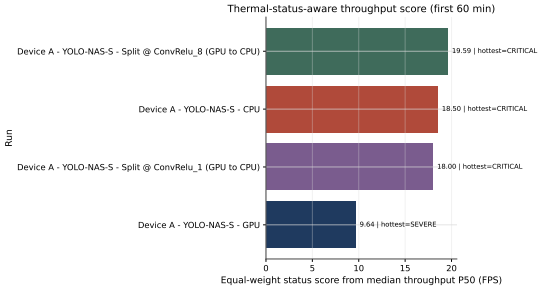

Figure 21: Thermal-status-aware throughput score.

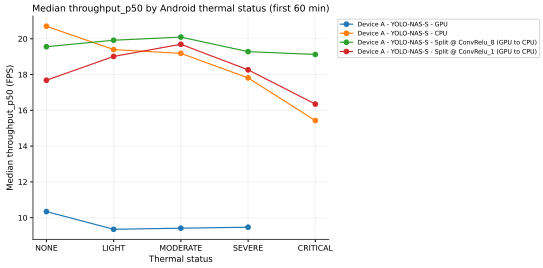

Figure 22: Median throughput by Android thermal status.

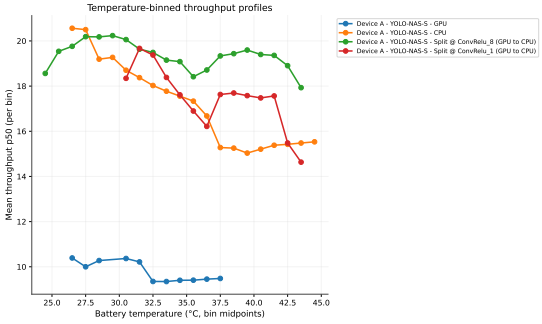

Figure 23: Temperature-binned throughput profiles.

tained behaviour among the compared configurations. The `ConvRelu_1` split is also competitive, although less consistently strong. The sequential GPU path is clearly uncompetitive for this model on this device.

The heterogeneous advantage is not only a short-lived early effect. Unlike the medium-sized case, the stronger split remains competitive as temperature rises. The gap to the sequential CPU baseline also does not collapse as clearly in the hotter region. This suggests that, for a smaller model such as YOLO-NAS Small, the overhead of simple heterogeneous execution may be easier to justify. A well-chosen split can therefore translate into a modest sustained benefit.

These results provide more encouraging evidence for simple heterogeneous execution on YOLO-NAS Small than on the medium-sized model. Unlike the medium-sized case, the better heterogeneous split does not only provide an early boost. It remains competitive across much of the thermal progression and exceeds the best sequential baseline by just over 1 FPS in the score summary. This should still be interpreted as a modest sustained advantage, not as a large or general heterogeneous speedup.

8.3.4 Device B — YOLO-NAS Medium

This subsection presents the simple heterogeneous results for YOLO-NAS Medium on Device B. In contrast to Device A, the sequential results in Section 8.2 showed that the GPU is the stronger sequential processor for this model on Device B. For simple heterogeneous execution, only two feasible one-blob cut points were available: `ConvRelu_1` and `ConvRelu_8`. Both were located near the top of the model. As a result, neither cut produced a balanced split of computation between the two processors. Table 14 summarizes these configurations.

Table 14: Simple heterogeneous execution results for YOLO-NAS Medium on Device B.

Configuration	Thermal-status score (FPS)	Highest thermal status	Diff vs best sequential
Sequential GPU	8.11	SEVERE	0.00
Sequential CPU	5.69	MODERATE	−2.42
Split @ `ConvRelu_1` (CPU→GPU)	7.58	SEVERE	−0.53
Split @ `ConvRelu_8` (CPU→GPU)	5.96	SEVERE	−2.15
Split @ `ConvRelu_8` (GPU→CPU)	3.88	SEVERE	−4.23
Split @ `ConvRelu_1` (GPU→CPU)	2.79	SEVERE	−5.32

Figure 24 shows the thermal-status-aware throughput score for the evaluated configurations. The sequential GPU baseline remains the best result at 8.11 FPS. All simple heterogeneous configurations fall below it. The strongest heterogeneous case is `ConvRelu_1` in the CPU→GPU direction, at 7.58 FPS. It is followed by `ConvRelu_8` CPU→GPU at 5.96 FPS. The GPU→CPU splits are substantially weaker, with 3.88 FPS for `ConvRelu_8` and only 2.79 FPS for `ConvRelu_1`. Even the sequential CPU baseline, at 5.69 FPS, outperforms some heterogeneous cases. These results already indicate that simple heterogeneous execution does not provide a sustained advantage over the strongest sequential baseline for this device and model.

The median-throughput-by-thermal-status view in Figure 25 makes the sustained behaviour clearer. The GPU baseline remains the strongest and most stable configuration across the thermal states. The heterogeneous runs decline more sharply as thermal pressure increases. In particular, the CPU→GPU runs can look competitive in the earlier part of execution, but they do not retain that advantage as the run progresses. The GPU→CPU runs remain clearly below the sequential baseline throughout. Any early heterogeneous improvement therefore does not translate into better sustained performance over the full run.

The temperature-binned throughput profiles in Figure 26 show the same pattern. The `ConvRelu_1` CPU→GPU split starts relatively high and briefly approaches the sequential GPU baseline in the cooler part of the run. It then declines steadily as temperature rises. The `ConvRelu_8` CPU→GPU case follows the same overall pattern at a lower level. By comparison, the sequential GPU baseline stays much flatter over the observed temperature range. This is the key result for this device and model: simple heterogeneous execution may appear promising in shorter-term behaviour, but in the hotter sustained region it falls well below the best sequential approach.

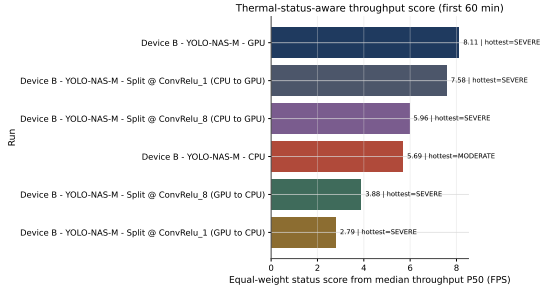

Figure 24: Thermal-status-aware throughput score.

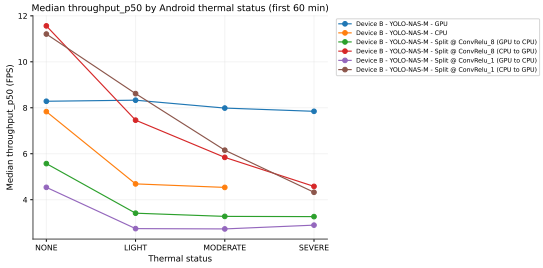

Figure 25: Median throughput by Android thermal status.

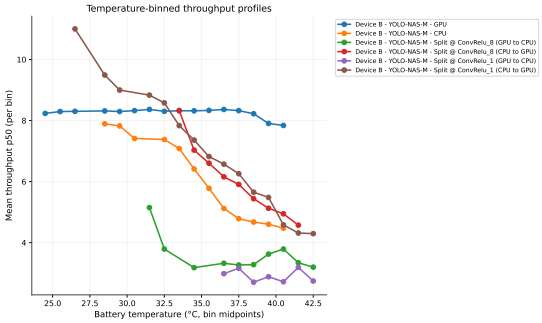

Figure 26: Temperature-binned throughput profiles.

For YOLO-NAS Medium on Device B, the feasible one-blob simple splits were not effective. Neither `ConvRelu_1` nor `ConvRelu_8` produced a heterogeneous configuration that surpassed the sequential GPU baseline in sustained operation. This makes Device B a useful counterexample in the thesis. Unlike the more encouraging Device A cases, simple heterogeneous execution fails here to provide a practical long-run benefit, despite showing some short-term promise in selected runs.

8.3.5 Device B — YOLO-NAS Small

For YOLO-NAS Small on Device B, the simple heterogeneous evaluation focused only on the `ConvRelu_8` cut point. The earlier `ConvRelu_1` boundary was considered too close to the start of the model to give a meaningful split, so it was not kept for the main comparison. This choice also matched the processor behaviour of Device B. Since the GPU was already the stronger sequential processor for this model, the more promising direction was expected to be CPU→GPU. In that direction, the larger remainder of the network stays on the GPU. Table 15 summarizes these configurations.

Table 15: Simple heterogeneous execution results for YOLO-NAS Small on Device B.

Configuration	Thermal-status score (FPS)	Highest thermal status	Diff vs best sequential
Sequential GPU	12.86	SEVERE	0.00
Sequential CPU	7.70	SEVERE	−5.16
Split @ `ConvRelu_8` (GPU→CPU)	7.57	MODERATE	−5.29
Split @ `ConvRelu_8` (CPU→GPU)	14.40	SEVERE	+1.54

Figure 27 confirms that this expectation was reasonable, at least in the score summary. The `ConvRelu_8` CPU→GPU split achieves the highest thermal-status-aware throughput score, at 14.40 FPS. This exceeds both the sequential GPU baseline at 12.86 FPS and the sequential CPU baseline at 7.70 FPS. By contrast, the `ConvRelu_8` GPU→CPU split remains weak at 7.57 FPS and does not outperform even the sequential CPU run.

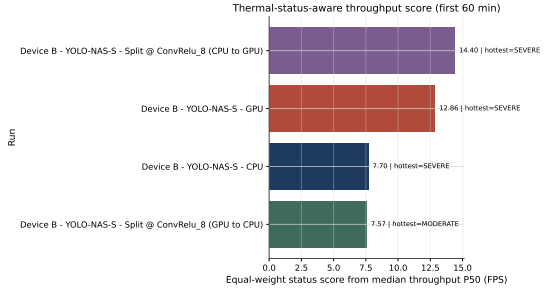

Figure 27: Thermal-status-aware throughput score.

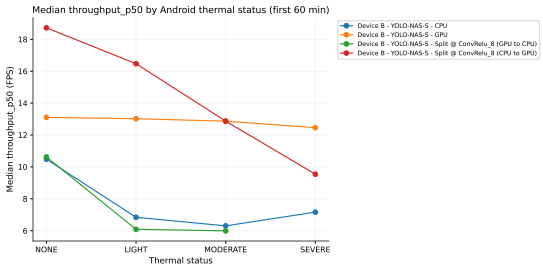

Figure 28: Median throughput by Android thermal status.

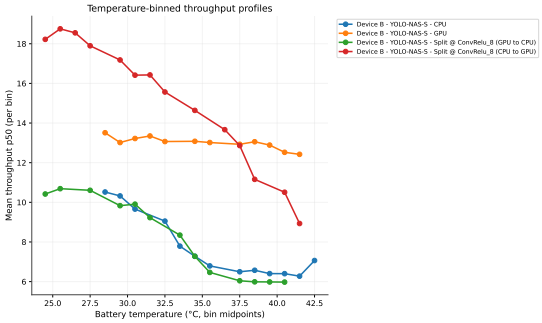

Figure 29: Temperature-binned throughput profiles.

The median-throughput-by-thermal-status view in Figure 28 helps explain why the score favours the CPU→GPU split. In the earlier thermal states, this heterogeneous configuration performs noticeably above both sequential baselines. It therefore outperforms them through a substantial part of the run. As thermal pressure increases, however, the gap narrows. The advantage is not preserved as clearly in the hotter sustained region.

The temperature-binned throughput profile in Figure 29 supports the same interpretation. The `ConvRelu_8` CPU→GPU split is strongest across much of the cooler and mid-temperature range, which is enough to produce the best overall score. At higher temperatures, its throughput drops more sharply. The sequential GPU baseline remains comparatively steadier. The heterogeneous result is therefore best understood as a configuration that performs very well through most of the run, but loses its advantage once Device B becomes sufficiently hot.

Taken together, these results make YOLO-NAS Small on Device B an important example of temperature-dependent heterogeneous behaviour. Simple heterogeneous execution achieves a measurable advantage in the overall score, and the direction that leaves more work on the stronger GPU is clearly preferable. At the same time, this case illustrates an important thesis theme: a heterogeneous strategy can achieve the best overall sustained score while still becoming weaker than the sequential counterpart in the hottest part of the run.

Across the main heterogeneous comparisons, the results answer RQ1 and also sharpen RQ2. Simple heterogeneous execution is not uniformly beneficial. Its value depends strongly on the device, model, cut point, and execution direction. On Device A, selected splits can remain competitive with or slightly exceed the strongest sequential baseline, especially for YOLO-NAS Small. On Device B, the results are more mixed. YOLO-NAS Medium does not benefit in sustained operation, whereas YOLO-NAS Small shows a heterogeneous advantage in overall score that weakens in the hottest part of the run. The main conclusion from this section is that heterogeneous execution can help, but only under constrained conditions. It should not be treated as a generally superior replacement for strong sequential baselines.

8.4. Additional Heterogeneous Findings Beyond the Main One-Blob Runs

The previous section focused on the main simple one-blob heterogeneous configurations and compared them with the sequential baselines. This section adds a smaller set of follow-up findings that help explain those results. It examines whether implementation refinements reduced the limitations of the simple split pipeline. It also considers whether more complex execution variants revealed additional practical constraints.

The purpose is not to repeat the full comparison from Section 8.3. Instead, this section highlights follow-up experiments that clarify why heterogeneous execution did or did not provide meaningful sustained benefits. The first case is the zero-copy refinement applied to YOLOv8 Medium on Device A.

8.4.1 Zero-Copy Refinement for YOLOv8 on Device A

To examine whether transfer overhead was a major limiting factor in the simple heterogeneous YOLOv8 runs on Device A, a refined implementation was evaluated. This version used persistent extractors and zero-copy-oriented Vulkan memory mapping. It was tested on the same main cut points and execution directions. The goal was to determine whether reducing explicit copy and allocation overhead would materially improve sustained throughput.

Table 16 summarizes the results. In some cases, the zero-copy refinement produced a small improvement. For example, the `cat_16` GPU→CPU split improved from 8.47 FPS to 8.72 FPS. The `cat_1` CPU→GPU split also improved, from 9.79 FPS to 10.12 FPS. The stronger configurations did not improve in the same way. The `cat_1` GPU→CPU split changed only slightly, from 12.15 FPS to 12.05 FPS. The `cat_16` CPU→GPU split changed from 12.73 FPS to 12.21 FPS. This is a sub-1-FPS difference and is treated as approximate parity under the interpretation rule used in this chapter. This small decrease shows that the zero-copy-oriented refinement did not improve all cases. It is therefore treated as approximate parity, not as a clear regression with an identifiable cause.

Table 16: Zero-copy refinement results for YOLOv8 Medium on Device A.

Configuration	Simple heterogeneous score (FPS)	Zero-copy score (FPS)	Change
Split @ `cat_16` (GPU→CPU)	8.47	8.72	+0.25
Split @ `cat_16` (CPU→GPU)	12.73	12.21	−0.52
Split @ `cat_1` (GPU→CPU)	12.15	12.05	−0.10
Split @ `cat_1` (CPU→GPU)	9.79	10.12	+0.33

The median-throughput-by-thermal-status view in Figure 31 and the temperature-binned view in Figure 32 show the same overall pattern. Some refined runs shift slightly upward, but the relative ordering and broader thermal trends remain largely unchanged. The stronger splits remain stronger, the weaker splits remain weak, and the more promising configurations still converge toward similar sustained behaviour in the hotter region. In other words, zero-copy refinement changes the magnitude of some results, but it does not fundamentally change which splits are effective.

The zero-copy results suggest that transfer overhead was not the only, or even the dominant, reason why simple heterogeneous execution failed to deliver a clearly stronger sustained regime for YOLOv8 on Device A. Zero-copy reduced some overhead and produced modest improvements in selected cases, but the overall trend remained the same. This indicates that the main limitations of heterogeneous execution on this device were more structural. They include split balance and processor-side workload distribution, rather than copy cost alone.

8.4.2 Exploratory Multi-Blob and Skip-Connected Cuts

In addition to the main one-blob heterogeneous configurations, a small number of exploratory runs were conducted on YOLO-NAS Medium for Device B using a more complex split. This split,

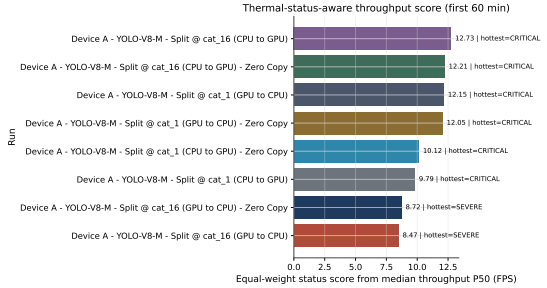

Figure 30: Thermal-status-aware throughput score (zero-copy).

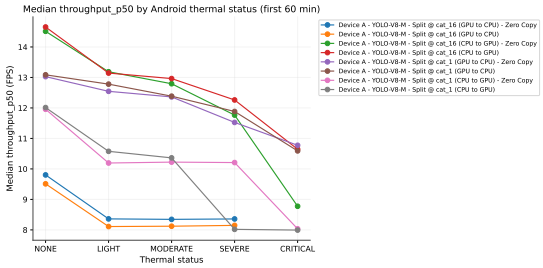

Figure 31: Median throughput by Android thermal status (zero-copy).

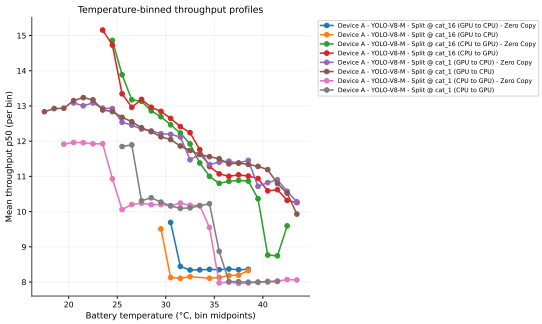

Figure 32: Temperature-binned throughput profiles (zero-copy).

referred to here as `Splitncnn_3`, involved a skip-connected boundary. It required transferring multiple intermediate outputs rather than a single tensor. These experiments were not expanded into a broader evaluation because such cut points were much more time-consuming to implement. Each multi-blob boundary required split-specific code changes, and the expected performance benefit was uncertain.

Table 17 compares this split with the strongest simple heterogeneous and sequential alternatives for the same model. The `Splitncnn_3` CPU→GPU run reached a thermal-status-aware score of 8.08 FPS. This is slightly higher than the best simple heterogeneous case, `ConvRelu_1` CPU→GPU at 7.58 FPS, and close to the sequential GPU baseline at 8.11 FPS. By contrast, the `Splitncnn_3` GPU→CPU run remained clearly weaker at 3.81 FPS.

Table 17: Exploratory multi-blob and skip-connected results for YOLO-NAS Medium on Device B.

Configuration	Thermal-status score (FPS)	Highest thermal status	Diff vs best sequential
Sequential GPU	8.11	SEVERE	0.00
Split @ `ConvRelu_1` (CPU→GPU)	7.58	SEVERE	−0.53
Split @ `ConvRelu_8` (CPU→GPU)	5.96	SEVERE	−2.15
Split @ `Splitncnn_3` (CPU→GPU)	8.08	SEVERE	−0.03
Split @ `Splitncnn_3` (GPU→CPU)	3.81	SEVERE	−4.30

The detailed thermal views in Figures 34 and 35 show why these results should be interpreted cautiously. The `Splitncnn_3` CPU→GPU run performs strongly in the cooler and mid-temperature region. For part of the run, it compares favourably with the simpler CPU→GPU cuts. However, this did not translate into a clearly better sustained outcome than the sequential GPU baseline. The GPU→CPU direction remained uncompetitive throughout.

Robustness was the larger practical concern. In one exploratory CPU→GPU run, memory usage continued to grow until Android restarted the application. Another run lasted longer without the same instability. This behaviour suggested that placing the heavier suffix of the network on the

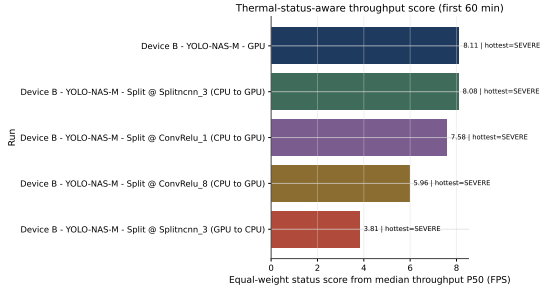

Figure 33: Thermal-status-aware throughput score (multi-blob).

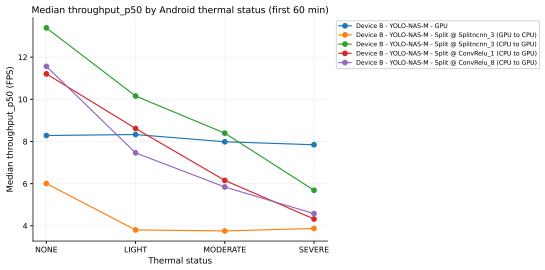

Figure 34: Median throughput by Android thermal status (multi-blob).

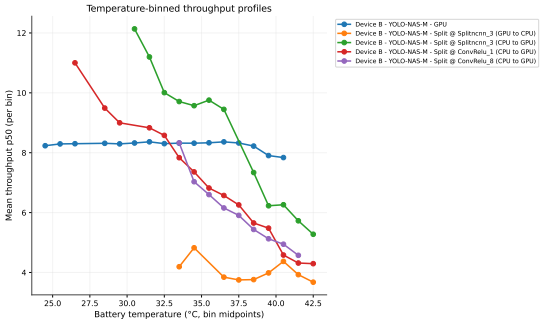

Figure 35: Temperature-binned throughput profiles (multi-blob).

GPU created sustained Vulkan memory pressure. The approach was therefore fragile, even when short-term throughput looked promising. For this reason, multi-blob and skip-connected cuts were not carried forward into the main evaluation scope.

These exploratory experiments show that more complex cut structures can occasionally approach the best simple or sequential results. However, they did not provide a sufficiently robust or clearly superior path for sustained evaluation. This supports the methodological choice made earlier in the thesis to focus the main comparison on one-blob cut points.

8.4.3 Failure Case Analysis: Why Some Splits Underperform

To better understand why some heterogeneous configurations underperformed, a small set of representative profiling runs was analysed in more detail. These were cases where the split looked reasonable from the model structure alone. Instead of comparing throughput only, the profiling examined how execution time was distributed between the two stages. It also measured how much time was lost in coordination. The aim is to explain the main failure modes behind the heterogeneous results reported earlier, especially stage imbalance, handoff overhead, and synchronization-related delay.

The profiling results revealed four recurring bottleneck patterns, summarized in Table 18: imbalance toward stage 2, imbalance toward stage 1, coordination-heavy execution, and stable but still imbalanced execution. The percentages in the table are approximate diagnostic ratios from representative profiling logs. They are not statistically averaged values across repeated runs. They should also not be read as a partition of total wall-clock time, and the values in a row are not expected to sum to 100%.

Stage 1 and Stage 2 timings describe the relative dominance of the two execution stages. Gap, queue, and boundary measurements capture coordination-related delays. These delays can overlap with, or be measured relative to, the split execution interval. The table is therefore diagnostic. It shows which signals are consistent with a given bottleneck interpretation, rather than proving a complete causal decomposition of runtime.

Device B YOLO-NAS Medium runs illustrate both imbalance failures clearly. In the GPU→CPU

Table 18: Representative failure patterns associated with selected heterogeneous splits underperform.

Representative case	**Dominant pattern**	**Approximate diagnostic ratios**	**Interpretation**
Device B, YOLO-NAS-M, `ConvRelu_1` (GPU→CPU)	Stage2-dominant	Stage 2 ≈87% of split time; stage 1 ≈6%	The cut is too early, so most computation remains after the boundary and stage 2 continues to dominate runtime
Device B, YOLO-NAS-M, `ConvRelu_8` (GPU→CPU)	Stage2-dominant but improved	Stage 2 ≈77%; stage 1 ≈13%	Stage 2 still dominates, but less severely than at `ConvRelu_1`, which improves stability and reduces total latency
Device B, YOLO-NAS-M, `ConvRelu_1` (CPU→GPU)	Stage1-dominant	Stage 1 ≈88%; stage 2 ≈4%	Too much work remains before the cut, so stage 1 on the CPU dominates while the GPU is underused
Device B, YOLO-NAS-M, `ConvRelu_8` (CPU→GPU)	Stage1-dominant but partially shifted	Stage 1 ≈72%; stage 2 ≈16%	Compared with `ConvRelu_1`, some work is moved to the GPU, but the CPU still remains the main bottleneck
Device A, YOLOv8-M, `cat_16` (GPU→CPU)	Stable but imbalanced	Stage 1 ≈93%; stage 2 ≈7%; coordination terms below about 1%	Most work remains in stage 1, but handoff overhead is almost negligible, so the run stays stable
Device A, YOLOv8-M, `cat_1` (GPU→CPU)	Coordination-heavy	Stage 1 ≈37%; stage 2 ≈42%; gap/queue ≈23% each; boundary ≈21%	The compute split is more balanced, but waiting, queueing, and boundary overhead become large enough to dominate the loss
Device A, YOLOv8-M, `cat_16` (CPU→GPU)	Stage2-dominant	Stage 2 ≈88%; stage 1 ≈11%; coordination terms below about 1%	The handoff is cheap, but too much work remains after the cut, so stage 2 becomes the limiter

direction, the early `ConvRelu_1` cut leaves almost the whole effective workload in stage 2. Stage 2 accounts for roughly 87% of split time. Moving the cut to `ConvRelu_8` improves the situation by reducing the stage 2 share to roughly 77%. However, stage 2 still remains the main limiter. In the CPU→GPU direction, the opposite problem appears. At `ConvRelu_1`, stage 1 accounts for roughly 88% of split time, so the CPU dominates while the GPU is underused. At `ConvRelu_8`, more work is shifted to the GPU, but stage 1 still accounts for roughly 72% of split time. These cases show that the cut can fail from either side: too much work may remain after the boundary, or too much may remain before it.

Device A YOLOv8 runs illustrate a different distinction. The `cat_16` GPU→CPU case is highly imbalanced, with stage 1 accounting for roughly 93% of split time. It remains stable, however, because the coordination terms stay below about 1%. By contrast, the `cat_1` GPU→CPU case appears more balanced in raw compute terms. Stage 1 and stage 2 contribute roughly 37% and 42%, respectively. Despite this, it performs worse because the coordination-related terms become large: gap and queue are each around 23%, while boundary overhead is around 21%. The `cat_16` CPU→GPU case shows a third pattern. The split itself is cheap, but stage 2 accounts for roughly 88% of split time. This means the configuration is limited by backend compute rather than by handoff cost. These cases support the interpretation that a heterogeneous split can underperform for two main reasons: poor workload balance or high coordination cost. This can happen even when the cut looks plausible at the graph level.

The profiling results suggest that heterogeneous underperformance was not associated with a single bottleneck. Some splits failed because the prefix and suffix were poorly balanced. Others failed because handoff and synchronization-related overhead became too expensive. In other cases, one processor remained underutilized even though heterogeneous execution was active. This also explains why later refinements such as zero-copy improved only selected cases modestly. When the dominant problem was workload balance or processor underutilization rather than copy cost alone, reducing transfer overhead was not enough to fundamentally change the outcome.

8.5. Results Under Controlled Input Rate

This section addresses RQ3 by examining camera-side input-rate control. The focus is on how controlled frame delivery changes sustained inference behaviour and the comparison between sequential and heterogeneous execution.

The previous sections used the original uncapped camera pipeline. In that setup, performance was affected not only by model execution, but also by how quickly frames entered the pipeline. To reduce this uncontrolled workload effect, selected experiments were repeated with the capped repeating-request setup introduced in Chapter 7.

The FPS labels in the capped-run figures refer to the configured Android camera range. For example, `8,15 FPS`, `10,15 FPS`, and `15,23 FPS` mean that the camera was configured to operate within those ranges. The ranges were chosen from earlier uncapped throughput observations and from the practical processing capacity of each device–model configuration. They were not intended to impose one identical input rate across all experiments, and results from different caps should not be read as direct fixed-FPS comparisons. Instead, the capped results should be interpreted as controlled workload-delivery comparisons.

The section first examines the capped sequential baselines. It then re-evaluates selected heterogeneous configurations under controlled camera-side pacing, to assess whether input-rate control changes the practical value of split execution.

8.5.1 Sequential Execution Under Capped Input Rate

This subsection compares the capped sequential runs with their uncapped counterparts. The main result is that input-rate control changes the thermal operating regime more clearly than it changes the overall ranking. The strongest sequential modes largely remain the same as in Section 8.2. However, the devices now run under noticeably lower thermal stress and, in several cases, achieve slightly better sustained throughput. Figure 36 summarizes the capped thermal-status-aware scores. Table 19 compares the capped and uncapped sequential baselines directly.

The uncapped values are taken from Table 10, and the capped values are taken from the capped sequential score summary.

Table 19: Sequential baselines under uncapped and capped input rate.

Device	Model	Mode	Uncapped score (FPS)	Uncapped hottest status	Capped score (FPS)	Capped hottest status	Main effect
Device A	YOLO-NAS-S	CPU	18.50	CRITICAL	18.84	CRITICAL	Slight throughput gain, still strongest overall
Device A	YOLO-NAS-S	GPU	9.64	SEVERE	9.70	MODERATE	Similar score, clearly lower thermal stress
Device A	YOLO-NAS-M	CPU	10.89	CRITICAL	11.41	CRITICAL	Small throughput gain, same top thermal class
Device A	YOLO-NAS-M	GPU	6.82	SEVERE	6.91	LIGHT	Slight gain, much lower thermal stress
Device A	YOLOv8-M	CPU	12.31	CRITICAL	12.30	CRITICAL	Essentially unchanged score, same strongest mode
Device A	YOLOv8-M	GPU	8.20	SEVERE	7.91	MODERATE	Slightly lower score, lower thermal stress
Device B	YOLO-NAS-S	CPU	7.70	SEVERE	10.12	MODERATE	Clear throughput gain and lower thermal stress
Device B	YOLO-NAS-S	GPU	12.86	SEVERE	12.79	MODERATE	Nearly same score, lower thermal stress
Device B	YOLO-NAS-M	CPU	5.69	MODERATE	7.82	LIGHT	Clear gain and lower thermal stress
Device B	YOLO-NAS-M	GPU	8.11	SEVERE	9.06	MODERATE	Moderate gain and lower thermal stress

The clearest difference is the highest thermal state reached by the runs. In the uncapped sequential baseline set, nine of the ten configurations reached at least SEVERE thermal status, and three reached CRITICAL. Under capped input rate, the thermal picture is milder for most configurations. Seven of the ten capped sequential runs remain in LIGHT or MODERATE, while three still reach CRITICAL. This indicates that camera-side pacing substantially reduces the thermal burden of long-run inference, even when the overall throughput ranking remains broadly similar.

The effect on throughput is more moderate, but still meaningful. The strongest capped result again comes from Device A with YOLO-NAS-S on CPU. Its score increases from 18.50 FPS in the uncapped case to 18.84 FPS with capped input rate. Device B with YOLO-NAS-S on GPU remains the strongest sequential baseline on Device B and changes only slightly, from 12.86 FPS to 12.79 FPS. At the same time, its highest thermal status shifts from SEVERE to MODERATE. Device A with YOLOv8-M on CPU remains effectively unchanged, at 12.30 FPS compared with 12.31 FPS uncapped. The thermal behaviour is still relevant, however, because the capped GPU counterpart decreases slightly from 8.20 FPS to 7.91 FPS while reaching only MODERATE rather than SEVERE. Across the full sequential set, the main pattern is not a dramatic score increase. It is a more favourable trade-off between sustained throughput and thermal stress.

The median-throughput-by-thermal-status view in Figure 37 supports the same interpretation. Under capped input rate, the runs occupy fewer and generally lower thermal states than before. This makes the sustained behaviour flatter and easier to interpret. The effect is especially visible in Device B and in weaker GPU cases, where capped operation prevents the runs from entering the more thermally stressed states that were common in the uncapped condition.

The temperature-binned profiles in Figure 38 show the same general effect. Capped operation limits thermal escalation and keeps the runs in a narrower temperature range. This helps explain why some configurations gain around 0–2 FPS even though the score ranking itself does not change dramatically.

The capped sequential results show that input-rate control does not mainly create a new best processor choice. Its main contribution is to reduce thermal escalation and produce a more controlled sustained operating regime. This matters because part of the long-run behaviour observed in the uncapped experiments was influenced by how aggressively work entered the inference pipeline. It was not determined only by the computational characteristics of CPU and GPU execution.

8.5.2 Heterogeneous Execution Under Capped Input Rate

This subsection re-evaluates selected heterogeneous configurations under capped input-rate conditions. It compares them with their uncapped heterogeneous counterparts and with the strongest capped sequential baselines. Unlike Section 8.3, the aim here is not to present every capped heterogeneous run with a full set of plots. That would add repetition without much interpretive

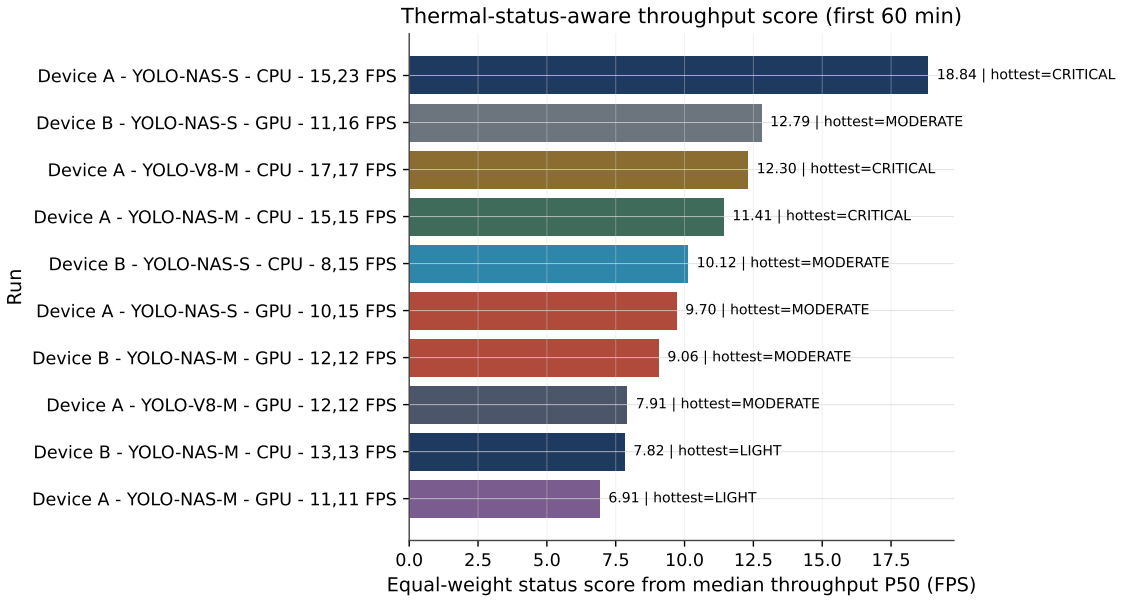

Figure 36: Thermal-status-aware throughput score under capped input rate.

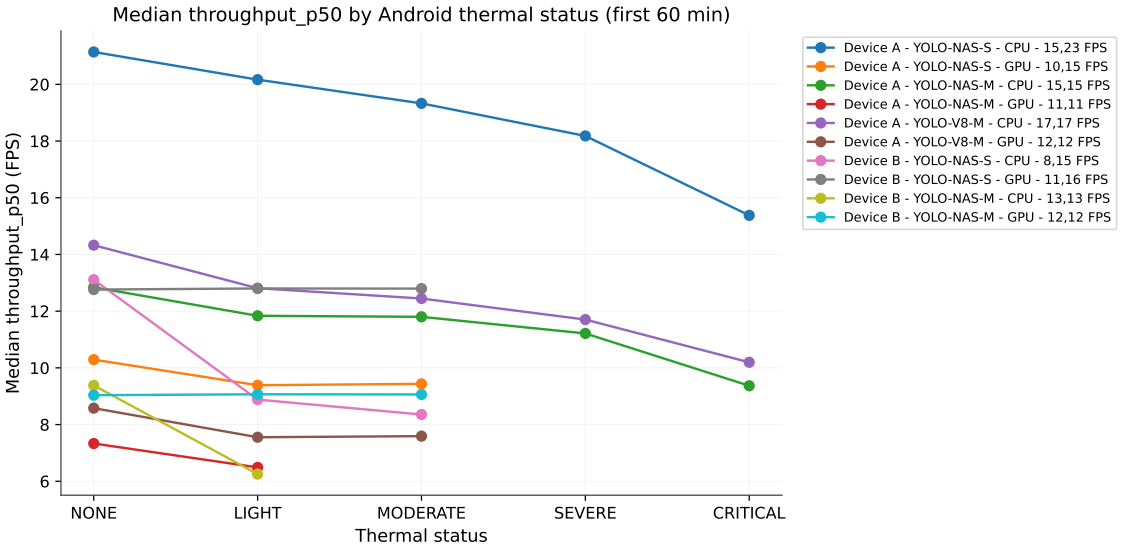

Figure 37: Median throughput by Android thermal status under capped input rate.

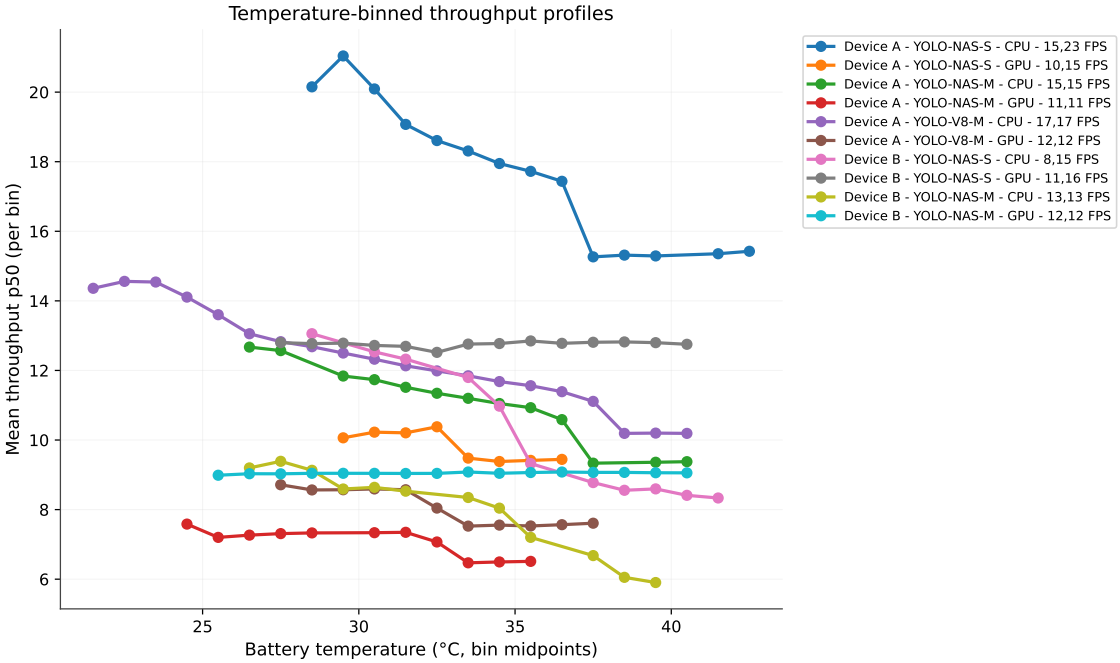

Figure 38: Temperature-binned throughput profiles under capped input rate.

value.

Instead, Table 20 summarizes the capped heterogeneous runs that have a clear uncapped counterpart in the main one-blob evaluation. The figures then focus on a smaller set of representative cases. These cases show how controlled frame delivery changed the earlier interpretation. They cover three outcomes: a previously weak heterogeneous case that became competitive under capped pacing, a promising case that became more convincing, and a central YOLOv8 case that connects the capped results to the earlier heterogeneous and zero-copy findings.

Table 20: Capped heterogeneous runs compared with uncapped counterparts and best capped sequential baseline.

Device	Model	Heterogeneous config	Uncap. score	Uncap. status	Capped score	Capped status	Best cap. seq score	Diff vs seq	Main interpretation
Device A	YOLO-NAS-M	Split @ `ConvRelu_8` (GPU→CPU)	11.15	CRITICAL	12.81	CRITICAL	11.41	+1.40	Clear improvement under capping; now more convincingly above capped CPU baseline
Device A	YOLO-NAS-M	Split @ `ConvRelu_1` (GPU→CPU)	8.85	CRITICAL	12.57	CRITICAL	11.41	+1.16	Strong gain under capping; suggests earlier uncapped weakness was partly workload-delivery related
Device A	YOLO-NAS-S	Split @ `ConvRelu_8` (GPU→CPU)	19.59	CRITICAL	20.87	CRITICAL	18.84	+2.03	Best capped heterogeneous result; strongest evidence that capped heterogeneity can help
Device A	YOLO-NAS-S	Split @ `ConvRelu_1` (GPU→CPU)	18.00	CRITICAL	20.00	CRITICAL	18.84	+1.16	Improved and clearly competitive under capped input
Device B	YOLO-NAS-M	Split @ `ConvRelu_1` (CPU→GPU)	7.58	SEVERE	10.31	SEVERE	9.06	+1.25	Marked gain under capping; now exceeds capped GPU baseline
Device B	YOLO-NAS-M	Split @ `ConvRelu_8` (CPU→GPU)	5.96	SEVERE	10.32	SEVERE	9.06	+1.26	Strong improvement; capped pacing changes earlier negative interpretation
Device A	YOLOv8-M	Split @ `cat_16` (CPU→GPU)	12.73	CRITICAL	13.27	CRITICAL	12.30	+0.97	Capping preserves approximate parity with the capped CPU baseline and slightly improves the score summary
Device A	YOLOv8-M	Split @ `cat_1` (GPU→CPU)	12.15	CRITICAL	13.13	CRITICAL	12.30	+0.83	Improves under capping; remains approximately competitive with the capped CPU baseline

Table 20 shows a consistent pattern. In all directly comparable cases, the capped heterogeneous score is higher than the uncapped one. This is especially visible for configurations that were previously weak or only marginally useful. For example, Device A YOLO-NAS-M at `ConvRelu_1` (GPU→CPU) rises from 8.85 FPS to 12.57 FPS. Device B YOLO-NAS-M at `ConvRelu_8` (CPU→GPU) rises from 5.96 FPS to 10.32 FPS. Already stronger heterogeneous

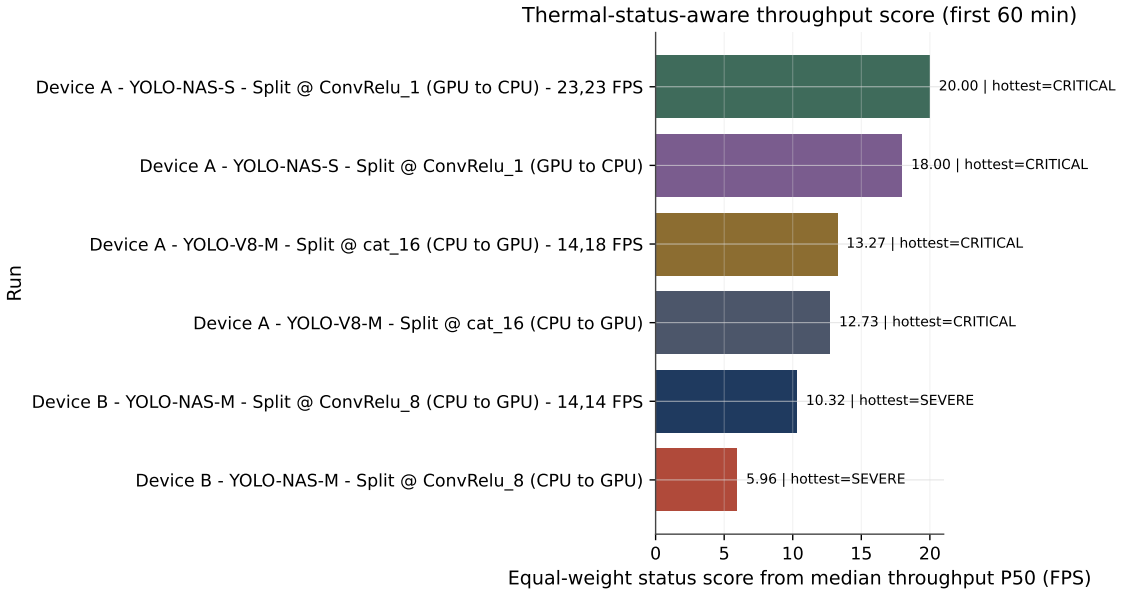

Figure 39: Thermal-status-aware throughput score for selected capped heterogeneous configurations.

cases also improve. Device A YOLO-NAS-S at `ConvRelu_8` (GPU→CPU), for example, reaches 20.87 FPS and exceeds the best capped sequential baseline by more than 2 FPS. This indicates that controlled frame delivery did not only reduce thermal load in the sequential pipeline. In the evaluated cases, it also changed how heterogeneous execution should be interpreted, especially for configurations that were weak or marginal under uncapped input delivery.

The first representative case is Device B with YOLO-NAS-M at `ConvRelu_8` in the CPU→GPU direction. It is visualized alongside other configurations in Figures 39–41. In the uncapped setting, this configuration scored only 5.96 FPS and remained clearly below the sequential GPU baseline. It was therefore one of the weaker heterogeneous results in Section 8.3. Under capped input rate, its score rises to 10.32 FPS. This places it above the best capped sequential baseline for the same device and model, which is 9.06 FPS on GPU. Figure 40 shows that the capped run remains consistently above the uncapped counterpart across the observed thermal states. Figure 41 also shows a clear upward shift across most of the temperature range. This is one of the clearest cases where capped pacing changes the earlier interpretation. A heterogeneous configuration that previously looked unconvincing becomes competitive once workload delivery is controlled.

The second representative case is Device A with YOLO-NAS-S at `ConvRelu_1` in the GPU→CPU direction. This configuration was already reasonably strong in the uncapped setting, at 18.00 FPS. Under capped pacing, it improves to 20.00 FPS. The best capped sequential baseline for the same device and model is 18.84 FPS on CPU. In the thermal-status figure, the capped run stays above the uncapped counterpart throughout the reported thermal progression. The temperature-binned profile shows the same relationship across most of the observed temperature range. Unlike the Device B case above, this result is not a reversal from failure to success. Instead, it shows that capping can strengthen a heterogeneous configuration that was already promising and make its sustained advantage clearer.

The third representative case is Device A with YOLOv8-M at `cat_16` in the CPU→GPU direction. This case is included because YOLOv8 was one of the central model families in the earlier heterogeneous study. It also appeared in the zero-copy refinement analysis. In the uncapped setting, this split already slightly exceeded the best sequential baseline at 12.73 FPS. Under capped input rate, it rises further to 13.27 FPS. The best capped sequential baseline for the same device and model is 12.30 FPS on CPU. The thermal-status and temperature-binned plots show that the capped run remains above the uncapped counterpart across the observed range. The improvement, however, is more moderate than in the Device B YOLO-NAS-M case. This suggests that capped

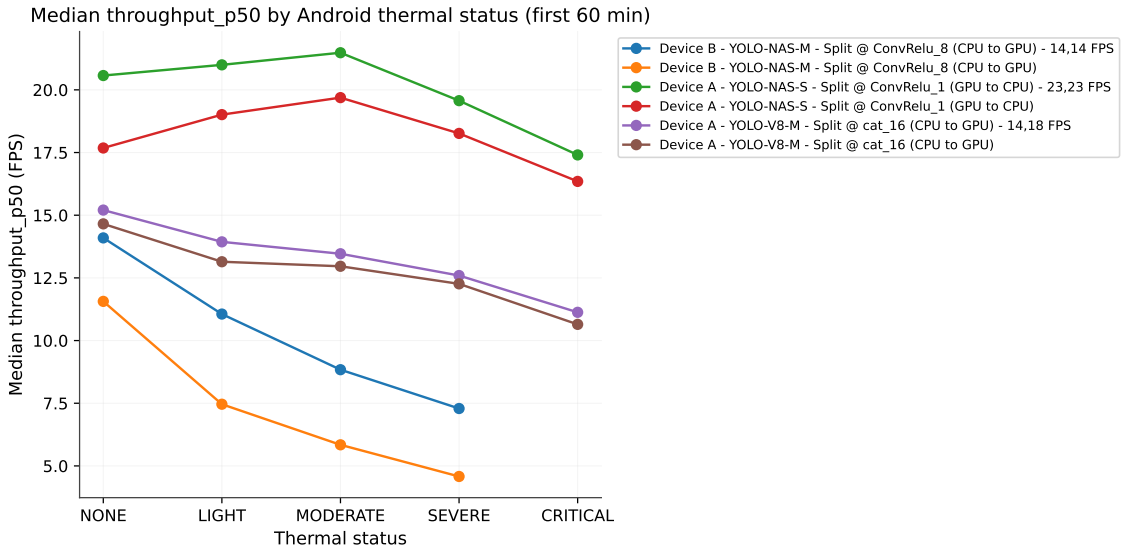

Figure 40: Median throughput by Android thermal status for selected capped heterogeneous configurations.

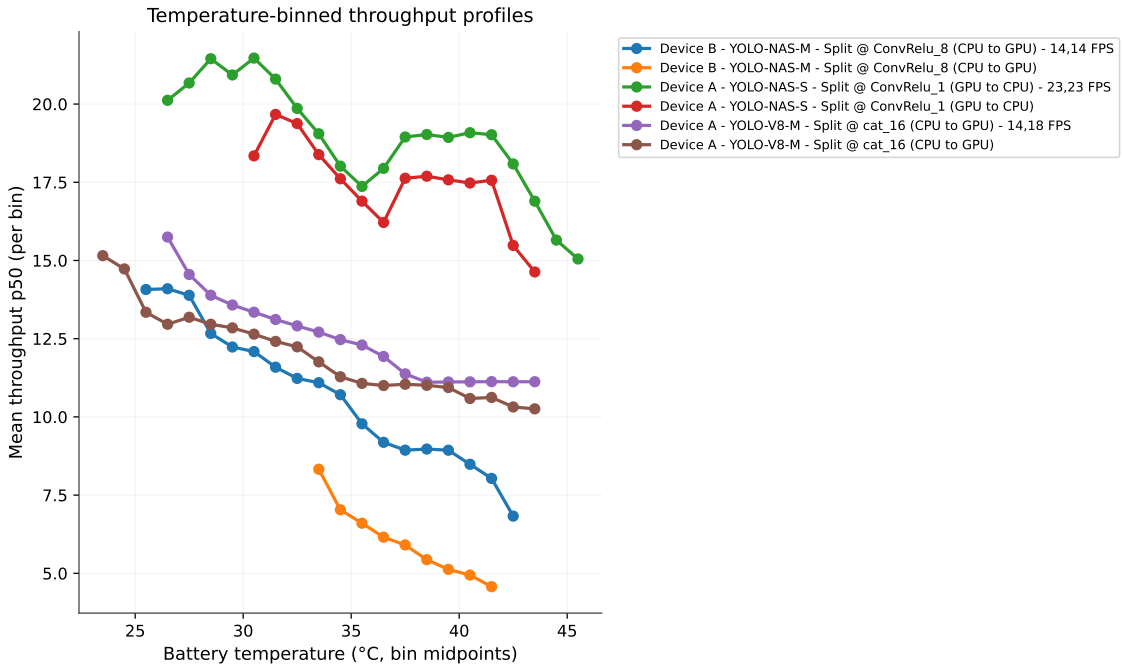

Figure 41: Temperature-binned throughput profiles for selected capped heterogeneous configurations.

pacing reinforces the earlier YOLOv8 result rather than changing it fundamentally. The split was already one of the stronger simple heterogeneous configurations, and input-rate control makes that sustained advantage somewhat clearer.

Taken together, the capped heterogeneous results show that controlled input rate changes the interpretation of heterogeneous execution in two ways. First, it can make previously weak configurations substantially more competitive by reducing workload-delivery pressure that amplifies thermal and pipeline effects. Second, it can strengthen already promising configurations and make their competitiveness relative to capped sequential baselines more evident. These findings show that the practical value of heterogeneous execution cannot be judged from uncapped runs alone.

This directly answers RQ3. Camera-side pacing materially affects sustained inference behaviour and can change how sequential and heterogeneous execution should be interpreted in practice. Under controlled input delivery, several heterogeneous configurations show lower workload pressure and become more competitive than they appeared in the uncapped setting. Workload regulation is therefore not a peripheral implementation detail, but a central part of sustained mobile inference performance.

8.6. Repeatability Check

Most configurations in the full experimental matrix were executed once. For this reason, a targeted repeatability check was conducted for selected capped-input configurations. The purpose was not to statistically replicate the entire experimental matrix. Instead, the aim was to assess whether the main capped-input findings remained stable under repeated execution.

The selected cases include the strongest capped sequential configuration on each evaluated device. They also include representative capped heterogeneous configurations that are central to the capped-input interpretation. Each selected configuration was evaluated three times in total: the original capped run and two additional repeated runs. Table 21 summarizes the thermal-status-aware throughput scores. The score range denotes the difference between the highest and lowest score across the three runs.

Table 21: Repeatability check for selected capped-input configurations.

Device	Model	Configuration	FPS cap	Run 1	Run 2	Run 3	Mean	Std. dev.	Score range	Highest thermal status
Device A	YOLO-NAS-S	CPU	15–23	18.84	18.87	18.56	18.76	0.17	0.31	CRITICAL in all runs
Device A	YOLO-NAS-S	Split @ `ConvRelu_1` GPU→CPU	23–23	20.00	20.48	19.70	20.06	0.39	0.78	CRITICAL in all runs
Device B	YOLO-NAS-S	GPU	11–16	12.79	12.81	12.82	12.81	0.02	0.03	LIGHT–MODERATE
Device B	YOLO-NAS-M	Split @ `ConvRelu_8` CPU→GPU	14–14	10.32	10.46	10.84	10.54	0.27	0.52	SEVERE in all runs

The repeated runs show limited variation. The standard deviations were small in all selected cases: 0.17 FPS for Device A YOLO-NAS-S CPU, 0.39 FPS for Device A YOLO-NAS-S heterogeneous execution, 0.02 FPS for Device B YOLO-NAS-S GPU, and 0.27 FPS for Device B YOLO-NAS-M heterogeneous execution. The score ranges were also limited, from 0.03 FPS to 0.78 FPS. For these selected cases, the variation is small compared with the main differences discussed earlier in the chapter.

The qualitative conclusions remained unchanged. On Device A with YOLO-NAS-S, the heterogeneous configuration stayed above the CPU baseline in all repeated capped runs. Device B YOLO-NAS-S GPU remained highly stable across all three runs. The selected Device B capped heterogeneous configuration also remained consistently above 10 FPS. The highest thermal status was consistent for the Device A runs and for the Device B heterogeneous runs.

Figure 42 provides a visual example of this repeatability for one representative heterogeneous case. The exact throughput values differ slightly between runs, but the three profiles follow the same overall pattern. Throughput remains near the capped input rate in the cooler region and then declines as battery temperature increases. This supports the interpretation that the observed behaviour is a repeatable thermal trend rather than an isolated one-run artifact.

Overall, this repeatability check supports the interpretation that the main capped-input findings are not artifacts of a single run. At the same time, it remains a targeted check rather than a full statistical replication of every configuration in the experimental matrix.

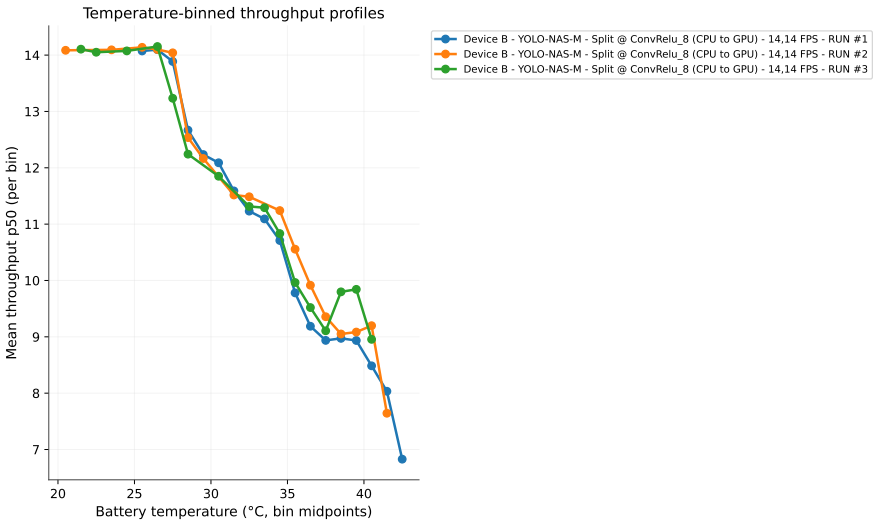

Figure 42: Temperature-binned throughput profiles for three repeated capped runs of Device B with YOLO-NAS-M using the `ConvRelu_8` CPU→GPU split at 14–14 FPS. The repeated runs follow a similar degradation pattern as battery temperature increases, supporting the repeatability of the observed thermal trend.

8.7. Energy Considerations and Thermal Coupling

This section discusses energy considerations qualitatively. The experimental pipeline did not include direct power or current measurement. The observations below are therefore interpretive. They are based on thermal behaviour and sustained throughput patterns, not on direct energy evidence.

Although direct energy use was not measured, the results are still relevant to energy discussion. Sustained heat buildup reflects continued power dissipation under passive cooling. In this sense, thermal behaviour can be treated as an indirect systems-level signal of workload pressure. It should not be treated as a substitute for power measurement [2, 4, 5].

The sequential and heterogeneous results show that higher initial throughput is not necessarily the best sustained operating point. Configurations that push the device more aggressively may perform well early in the run. However, they can also accelerate thermal escalation and lose part of that advantage after throttling. Conversely, configurations that keep the device under lower thermal pressure may preserve more stable sustained throughput.

The capped input-rate experiments reinforce this point. When camera-side pacing reduced workload pressure, several runs reached lower thermal states while maintaining similar or better sustained throughput. This suggests that thermal stress was influenced not only by model choice or processor selection. It was also influenced by the rate at which work was delivered to the system. Input-rate control therefore affects the energy–thermal trade-off indirectly, even though this thesis does not quantify energy directly.

The same caution applies to heterogeneous execution. Distributing work across the CPU and GPU may improve utilization, but it can also introduce handoff, synchronization, and memory-traffic overheads. Heterogeneous execution may therefore not always be more energy-efficient. This cannot be confirmed without direct power measurement. Its practical value depends on whether the benefit of sharing computation is larger than the added coordination cost [8, 5].

Overall, the results support a cautious interpretation. Sustained mobile inference is thermally coupled to underlying energy demand, but the exact energy-performance relationship was not measured in this thesis. Direct power measurement would be needed to quantify energy per frame and to compare sequential, heterogeneous, and capped-input configurations precisely.

9. Discussion

This chapter interprets the empirical results from Chapter 8. The aim is not to repeat the detailed comparisons, but to explain what the results mean for sustained mobile inference in practice. The discussion covers the main findings, the answers to the research questions, and the broader implications under realistic mobile constraints. It also relates the results to prior work and considers what they suggest for long-running on-device object detection systems.

9.1. Overview of the Main Findings

Three main findings emerge from the study. First, sustained performance depends strongly on the device–model combination. The strongest sequential baseline was not the same on both platforms. On Device A, CPU execution was consistently stronger for the evaluated models. On Device B, the GPU was generally stronger for the YOLO-NAS models.

The strong YOLO-NAS-S CPU result on Device A is especially important. It should be read as a platform-specific empirical result, not as a general claim that CPU execution is better for smaller YOLO-NAS models. The CPU and GPU runs used the same model variant, 320×320 input resolution, live-camera pipeline, and 60-minute procedure. The GPU run also reached a high thermal state. This makes a workload or setup difference unlikely as the main explanation.

A more plausible interpretation is that, for this model–device–framework combination, the NCNN CPU path was more efficient than the Vulkan GPU path. One possible reason is that the smaller model did not provide enough GPU-side work to offset dispatch, memory, and runtime overheads. However, GPU utilization, operator-level timings, and Vulkan driver behaviour were not directly profiled. This explanation should therefore be treated as an interpretation, not as a confirmed causal mechanism. Overall, the results show that sustained mobile inference cannot be explained by a single processor preference rule.

Second, simple heterogeneous execution did not provide a universal benefit. Some heterogeneous configurations matched or modestly exceeded the strongest sequential baseline in the score summary. Others were competitive only in the cooler part of the run or failed clearly under sustained operation. The outcome depended on cut-point location, execution direction, and the relative CPU/GPU strengths on each device. Heterogeneous execution was therefore not useful simply because two processors were used. It helped only when the split structure and processor assignment were well matched.

Third, the practical value of heterogeneous execution was shaped by system-level factors beyond model partitioning. Zero-copy refinement improved only some cases, and usually by a modest amount. This suggests that transfer cost alone was not the main limitation in many runs. More complex multi-blob cuts sometimes approached good performance, but they were fragile and difficult to realize robustly.

Camera-side input-rate control had a clearer effect. Once workload delivery was controlled, several capped runs became more competitive and the thermal behaviour became easier to interpret. The targeted repeatability check also showed that the main capped-input trends were stable across repeated runs, although it was not a full statistical replication of the entire experimental matrix. These results support the view that sustained inference should be treated as an end-to-end systems problem, not only as a model-execution problem.

9.2. Discussion of RQ1: Sequential and Heterogeneous Execution Under Sustained Load

RQ1 asked how sequential CPU-only and GPU-only execution compare with heterogeneous CPU–GPU execution for sustained YOLO-based object detection on Android smartphones. The main answer is that heterogeneous execution can help, but only conditionally. It did not provide a consistent enough benefit to be treated as generally better than strong sequential execution.

The results also show why the sequential baselines are important. The best sequential path differed across the evaluated devices and models, and it was often difficult to beat by a clear margin. This means that heterogeneous execution should not be evaluated only against a weak

baseline. It should be compared with the strongest available sequential alternative on the same device and model.

A further point is that early-run behaviour was not always representative of sustained behaviour. Some heterogeneous runs looked promising while the device was cooler, but their advantage narrowed or disappeared as thermal pressure increased. This supports the central premise of the thesis: sustained mobile inference has to be evaluated over time, not inferred from short initial behaviour.

Within the evaluated NCNN CPU/GPU configurations, heterogeneous execution is therefore best understood as a conditional optimization. It can be useful when the split, execution direction, and device–model characteristics align. It should not be treated as the default strategy for sustained mobile object detection.

9.3. Discussion of RQ2: Practical Factors Governing Heterogeneous Execution

RQ2 asked what practical factors determine whether heterogeneous CPU–GPU execution is feasible and beneficial under sustained operation. The results show that the answer is not simply "use both processors." What matters is how the split is implemented and how well it matches the device, model, and runtime.

Partition feasibility was the first constraint. Although many internal boundaries may exist in a model, only some can be implemented cleanly in a mobile application. Boundaries with a single crossing tensor provided the most practical basis for systematic comparison. More complex multi-blob and skip-connected boundaries were useful for exploration, but they were harder to implement and less robust under sustained execution.

The cut-point suggestion method should therefore be understood as a way to reduce the experimental search space. It helped identify implementable candidates, but it did not predict sustained performance on its own. The later results showed that a structurally feasible cut could still perform poorly because of stage imbalance, runtime behaviour, or coordination overhead.

The profiling results highlight the importance of balance and coordination. A split can fail when too much work remains before the cut, when too much work remains after the cut, or when the handoff itself becomes too expensive. This means that heterogeneous execution is governed by the combined effect of partition structure, processor assignment, and runtime coordination.

Transfer overhead is also important, but it is not the whole explanation. The zero-copy experiments showed that reducing explicit copy cost can help in selected cases, but it cannot fix an unsuitable split or an unfavourable processor assignment. Overall, heterogeneous execution is feasible only when model structure, runtime support, tensor handoff, and processor balance align sufficiently well.

9.4. Discussion of RQ3: The Role of Camera-Side Input-Rate Control

RQ3 asked how camera-side input-rate control affects sustained inference performance and the observed value of heterogeneous execution. The main finding is that workload delivery is part of the sustained inference problem. It is not just a camera implementation detail.

Under the uncapped pipeline, the workload entering the model depended on the interaction between camera delivery and inference speed. This made the measured behaviour depend not only on the execution strategy, but also on how aggressively frames were supplied to the pipeline. Capped repeating-request execution reduced this uncontrolled workload pressure and made the sustained operating regime easier to interpret.

For the sequential baselines, the main effect of capping was lower thermal escalation rather than a complete change in ranking. This suggests that part of the degradation seen in uncapped runs was caused by workload pressure from the surrounding pipeline, not only by the CPU or GPU execution path itself.

For heterogeneous execution, input-rate control was especially important because split execution can add coordination work on top of model computation. Boundary handoff, synchronization, and simultaneous CPU–GPU activity may make some heterogeneous configurations more sensitive

to uncapped frame delivery. When the input rate was controlled, those configurations could be judged under a less aggressive workload.

The broader implication is that sustained heterogeneous execution should be evaluated together with workload pacing. Without controlling how work enters the system, the benefit of heterogeneity may be underestimated in some cases or exaggerated in others.

9.5. Relation to Previous Work

The results are consistent with the main themes identified in Chapter 3, but they also extend them in a practical direction. Prior work on heterogeneous mobile inference shows that CPU–GPU cooperation can help in selected cases, but only when execution granularity, synchronization overhead, and runtime support are favourable. The findings here support that view. Split quality, execution direction, and transfer structure all affected whether a heterogeneous configuration was useful.

A direct numerical comparison with systems such as CoDL or NN-Stretch would not be reliable because the devices, models, runtimes, partitioning methods, and evaluation durations differ. The more relevant comparison is qualitative. Like those systems, the results show that heterogeneous gains require suitable partitioning, low coordination cost, and reliable runtime support.

The results also support earlier findings on thermal effects and long-run mobile inference. Prior studies reviewed in Chapter 3 showed that short-run benchmark gains may not persist once thermal throttling becomes important. Several heterogeneous configurations followed the same pattern: they looked promising early, but their advantage narrowed or disappeared in the hotter sustained region.

The capped-input experiments add another point. Thermal behaviour is important, but it is not the only system factor that shapes long-run performance. The way frames enter the inference pipeline also affects the observed outcome.

The thesis also differs from some earlier work in its focus. Model compression is discussed in Chapter 3 as a common way to improve mobile performance, but it is not the main strategy studied here. The focus is sustained fixed-model execution under realistic mobile conditions. The contribution is therefore closer to an empirical systems characterization than to a model-optimization study.

9.6. Significance and Practical Implications

The study gives a more realistic view of what sustained mobile inference requires in deployment-oriented settings. One practical implication is that strong sequential baselines should not be treated as trivial references. On the evaluated smartphones, the best sequential processor differed by device and model. In several cases, the strongest sequential baseline was also difficult to beat decisively. For practitioners, this means that heterogeneous execution should not be adopted automatically. It should be used only when the split structure, processor balance, and transfer path are favourable enough to give a real sustained benefit.

A second implication is that heterogeneous execution on mobile devices is an engineering problem, not only a scheduling or hardware-utilization problem. The results show that feasibility, robustness, and sustained usefulness depend on practical details. These include cut-point structure, tensor-transfer simplicity, and runtime coordination. In applications similar to the Supersight deployment context, these details matter. The goal is not just to show that a split can run. The goal is to maintain predictable long-run performance in a live end-to-end system. Implementation constraints therefore become part of the decision on whether heterogeneous execution is worthwhile.

A third implication is that workload pacing should be considered during deployment design. The capped-input experiments showed that camera-side input-rate control can reduce thermal escalation and change the apparent value of heterogeneous execution. For practical mobile analytics systems, sustained performance cannot be judged only by choosing a processor or a cut point. It also depends on how the surrounding pipeline controls the rate at which work enters the model. In deployment terms, execution strategy and input-rate control should be considered together.

A final implication concerns the industrial context motivating this thesis. In the company setting, smartphone-based sensors may be mounted in top-down doorway positions for on-device

Figure 43: Proof-of-concept deployment context motivating the thesis. A smartphone is mounted in a top-down doorway position to illustrate the type of practical setting considered by the company. The figure is included for contextual motivation only and was not part of the controlled experimental evaluation or a formal field deployment validation.

visual analytics, as illustrated in Figure 43. This setup is included as a proof-of-concept deployment context. It is not presented as a validated use case or a formal field evaluation. The thesis does not evaluate full mounted on-site operation, counting accuracy, robustness to real-world scene variation, or long-term deployment reliability in that environment. The figure is used only to show the practical setting that motivates the controlled sustained-inference experiments. The results should therefore be read as guidance for evaluating long-running mobile edge-AI pipelines under controlled conditions, not as proof that the complete deployment scenario has been validated.

9.7. Limitations of the Study

This study has several limitations that should be considered when interpreting the results. First, the evaluation covers a small number of device–model combinations. Device A and Device B provide useful diversity in hardware and software characteristics, but they do not represent the full range of Android smartphones or mobile accelerators. The evaluated models are also limited to selected YOLO-based detectors, not the broader space of mobile vision architectures. The devices also ran different Android versions, which may affect thermal labels and runtime behaviour.

A further limitation is that the evaluated smartphones are reported using anonymised device labels rather than exact commercial hardware identifiers. This limits exact external reproduction of the experiments. The results should therefore be interpreted as evidence for the evaluated device classes and the studied NCNN CPU/Vulkan pipeline, rather than as fully reproducible device-specific benchmark results.

Given these limitations, the exact throughput values and processor preferences reported in this

thesis should not be generalized directly to all devices or models. Some pipeline-level quantities were also not directly instrumented as standalone metrics. These include exact drop ratio, per-frame camera FPS in the uncapped baseline, CPU/GPU idle time, and stall frequency. They are therefore interpreted indirectly rather than measured explicitly.

Second, the heterogeneous evaluation is intentionally limited to configurations that could be realized within the available pipeline. The main comparison focuses on one-blob cut points. Multi-blob and skip-connected cases are treated only as exploratory evidence. This was appropriate for a robust sustained comparison, but it also means that the study does not cover the full design space of possible internal partitions. Similarly, the zero-copy investigation studies a practical refinement within the existing experimental structure, not a fully redesigned mobile runtime.

Third, the capped input-rate evaluation is based on selected reruns. It is not a complete re-execution of every configuration under every cap setting. The targeted repeatability check increases confidence in the selected capped-input findings. However, it does not remove the broader limitation that the full experimental matrix was not statistically replicated. The capped study was sufficient to show the importance of workload pacing, but it should be read as a focused re-evaluation rather than a full second experimental campaign.

Finally, the thesis does not propose a new scheduling algorithm, automated runtime controller, or adaptive deployment policy. Its contribution is empirical and interpretive. It characterizes sustained behaviour and identifies the factors that govern it under realistic conditions. Although the thesis is motivated by an industrial doorway-deployment context, it does not include a formal field evaluation of mounted on-site operation. The reported findings should therefore be interpreted as controlled sustained-inference evidence, not as full deployment validation.

9.8. Generalizability and Broader Applicability

The numerical results are specific to the evaluated devices, YOLO-based models, and NCNN CPU/Vulkan execution pipeline. The reported throughput values, processor preferences, cut-point behaviour, and transfer-overhead effects should not be generalized directly to other smartphones, model families, or inference frameworks. In particular, cut-point feasibility, operator support, memory-management behaviour, and GPU execution characteristics are framework-, driver-, and SoC-dependent. They may differ under runtimes such as TensorFlow Lite, ONNX Runtime, MediaPipe, or NNAPI-based execution.

At the same time, the study suggests broader practical principles that may be relevant beyond the specific runs reported here. These principles still need validation on the target device and software stack. The results indicate that sustained mobile inference is shaped not only by nominal compute capability. It also depends on the interaction between thermal behaviour, partition feasibility, transfer overhead, runtime support, and workload delivery. These factors are relevant to many continuous on-device vision systems, even though their quantitative impact may differ across platforms.

The findings are also relevant beyond YOLO-based object detection at the level of evaluation method. The specific cut points and processor balances studied here are model-specific. Other streaming vision workloads, such as tracking, activity recognition, and continuous scene analysis, may still face similar sustained-performance constraints on smartphones or comparable edge devices. The transferable lesson is not that a particular CPU, GPU, or split strategy will generalize. It is that sustained inference should be evaluated as an end-to-end mobile systems problem, with execution strategy, thermal behaviour, framework constraints, and input pacing considered together.

Generalization should therefore remain conditional. The study does not support a universal claim that heterogeneous execution is either good or bad for mobile inference. It supports a more limited conclusion: heterogeneous execution can be useful only under favourable model, device, runtime, and workload-delivery conditions. This conditionality is one of the broader lessons of the thesis and motivates the future research directions presented in the next chapter.

10. Conclusions and Future Work

This thesis studied sustained YOLO-based object detection on Android smartphones under realistic operating constraints. It compared CPU-only, GPU-only, and heterogeneous CPU–GPU execution during 60-minute live-camera runs, while also examining cut-point feasibility, transfer overhead, thermal progression, and camera-side input-rate control. The main contribution is an empirical characterization showing that heterogeneous execution is useful only under favourable conditions, when device–model fit, partition structure, coordination overhead, thermal behaviour, and workload pacing align.

The results show that sustained performance depends strongly on the device–model combination. No processor type was best in all evaluated cases. On Device A, CPU execution was the stronger sequential baseline for the evaluated models. On Device B, the GPU was generally stronger for the YOLO-NAS models. Strong sequential baselines therefore remained important throughout the study. In several cases, they were difficult to beat decisively.

For RQ1, heterogeneous CPU–GPU execution provided a sustained benefit in selected cases, but not consistently enough to be treated as a generally superior strategy. Some simple one-blob heterogeneous configurations slightly outperformed the strongest sequential baseline, especially on Device A. Other configurations performed well only in the cooler part of the run, or failed once the device entered a hotter sustained region. The main conclusion is that heterogeneous execution is conditional. It can help, but only for suitable cut points, execution directions, and device–model combinations.

For RQ2, the usefulness of heterogeneous execution was shaped by practical implementation factors, not only by the availability of two processors. Important factors included cut-point feasibility, intermediate-tensor transfer, stage balance, and coordination overhead. One-blob cut points were the most practical basis for sustained comparison. Multi-blob and skip-connected boundaries were harder to implement robustly and could become unstable during long runs. The zero-copy refinement improved only selected cases modestly. This suggests that transfer cost alone did not explain the limitations observed in many configurations.

For RQ3, camera-side input-rate control materially influenced sustained inference behaviour. Under capped repeating-request operation, the devices generally reached lower thermal states than in the uncapped setup, and several heterogeneous configurations became more competitive. This indicates that workload pacing is part of sustained mobile inference evaluation, not merely a camera implementation detail. The targeted repeatability check supports this interpretation, as the main capped-input trends remained stable for the selected repeated configurations.

Overall, this work contributes an empirical analysis of sustained sequential and heterogeneous YOLO inference on Android smartphones under practical system constraints. It shows that partition feasibility, stage balance, transfer overhead, runtime coordination, thermal progression, and input pacing all affect whether heterogeneous execution is useful in practice. By including camera-side input-rate control, the study treats sustained mobile inference as an end-to-end systems problem rather than only a processor-assignment problem.

Several directions remain for future work:

- Extend the evaluation to more smartphones, model families, and execution backends. In particular, future work should evaluate NPU and other accelerator execution paths where runtime support is mature enough for sustained long-run comparison.
- Develop cut-point selection into a more automated device-aware method that uses runtime profiling more directly when ranking candidate partitions.
- Improve the experimental pipeline with more robust shared-memory or zero-copy mechanisms, so that transfer overhead can be isolated more precisely under sustained load.
- Include direct power and energy measurement. This would allow configurations to be compared by energy per processed frame as well as sustained throughput.
- Extend instrumentation to measure exact drop ratio, camera-side delivery rate, CPU/GPU idle time, and pipeline stall frequency more directly.

- Explore adaptive input-rate control. Instead of using manually selected capped FPS ranges, future systems could adjust the camera input rate dynamically based on thermal state, throughput drift, or runtime stability. This could also be combined with execution-mode or cut-point selection.
- Conduct a larger deployment-oriented study in real environments to examine how the findings translate across hardware platforms, runtime stacks, and deployment conditions.

11. Declaration of Generative AI and AI-Assisted Tools

During the preparation of this thesis, the author used generative AI and AI-assisted tools, including ChatGPT and Google Gemini. These tools were used to improve language clarity and readability, assist with LaTeX formatting and troubleshooting, and support the refinement of diagrams. These tools were not used to generate experimental results, fabricate references, or introduce scientific claims without review and verification. All AI-assisted material was reviewed, revised, and validated by the author, who takes full responsibility for the final content of the thesis.

References

[1] S. S. Ogden and T. Guo, "Characterizing the deep neural networks inference performance of mobile applications," arXiv preprint, Sep. 2019.

[2] M. Almeida, S. Laskaridis, A. Mehrotra, L. Dudziak, I. Leontiadis, and N. D. Lane, "Smart at what cost?: Characterising mobile deep neural networks in the wild," in *Proceedings of the 21st ACM Internet Measurement Conference (IMC)*. ACM, Nov. 2021, pp. 658–672.

[3] T. Benoit-Cattin, D. Velasco-Montero, and J. Fernández-Berni, "Impact of thermal throttling on long-term visual inference in a cpu-based edge device," *Electronics*, vol. 9, no. 12, p. 2106, Dec. 2020.

[4] Y. Zhou, F. Liang, T.-W. Chin, and D. Marculescu, "Play it cool: Dynamic shifting prevents thermal throttling," arXiv preprint, Jun. 2022.

[5] T. Tan and G. Cao, "Thermal-aware scheduling for deep learning on mobile devices with npu," *IEEE Transactions on Mobile Computing*, vol. 23, no. 12, pp. 10 706–10 719, Dec. 2024.

[6] Y. Cai, H. Li, G. Yuan, W. Niu, Y. Li, X. Tang, B. Ren, and Y. Wang, "Yolobile: Real-time object detection on mobile devices via compression-compilation co-design," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 2, 2021, pp. 955–963.

[7] Y. Cheng, "Yolo v8 network lightweight method based on pruning and quantization," *Mathematical Modeling and Algorithm Application*, vol. 3, no. 2, pp. 44–51, Nov. 2024.

[8] F. Jia, D. Zhang, T. Cao, S. Jiang, Y. Liu, J. Ren, and Y. Zhang, "Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*, ser. MobiSys '22. Association for Computing Machinery, Jun. 2022, pp. 209–221.

[9] J. Wei, T. Cao, S. Cao, S. Jiang, S. Fu, M. Yang, Y. Zhang, and Y. Liu, "Nn-stretch: Automatic neural network branching for parallel inference on heterogeneous multi-processors," in *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*. Association for Computing Machinery, Jun. 2023, pp. 70–83.

[10] S. Liu, W. Zhou, Z. Zhou, B. Guo, M. Wang, C. Fang, Z. Lin, and Z. Yu, "Deep learning inference on heterogeneous mobile processors: Potentials and pitfalls," in *Proceedings of the 2024 Workshop on Adaptive AIoT Systems*. Association for Computing Machinery, Jun. 2024, pp. 1–6.

[11] Z. Li, M. Paolieri, and L. Golubchik, "Accelerating mobile inference through fine-grained cpu-gpu co-execution," arXiv preprint, Oct. 2025, university of Southern California, Los Angeles, CA, USA.

[12] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)*. ACM, Mar. 2019, pp. 1–15.

[13] C. Zhang, F. Zhang, K. Chen, M. Chen, B. He, and X. Du, "Edgenn: Efficient neural network inference for cpu-gpu integrated edge devices," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2023, pp. 1193–1207.

[14] Y. Gao, Z. Zhang, P. K. Donta, C. K. Dehury, X. Wang, and D. Niyato, "Optimizing multi-dnn inference on mobile devices through heterogeneous processor co-execution," *IEEE Transactions on Mobile Computing*, pp. 1–16, 2025.

[15] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 2016, pp. 1–12.

[16] Tencent and the ncnn contributors, "ncnn: High-performance neural network inference framework optimized for mobile platforms," GitHub repository, 2017, accessed: 2026-04-27. [Online]. Available: https://github.com/tencent/NCNN

[17] G. Jocher, A. Chaurasia, and J. Qiu, "Ultralytics yolov8," GitHub repository, 2023, official implementation and documentation. Accessed: 2026-04-24. [Online]. Available: https://github.com/ultralytics/ultralytics

[18] Deci AI Research Team, "Yolo-nas: A new state-of-the-art model for object detection using neural architecture search," Technical blog / official model release, 2023, accessed: 2026-04-27. [Online]. Available: https://github.com/Deci-AI/super-gradients/blob/master/YOLONAS.md

[19] Android Developers, "Thermal api," Android Developers documentation, 2026, accessed: 2026-04-27. [Online]. Available: https://developer.android.com/games/optimize/adpf/thermal

[20] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2017, pp. 82–95.

[21] C. R. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Király, P. Montino, D. Kanter, S. Ahmed, D. Pau, U. Thakker, A. Torrini, P. Warden, J. Cordaro, G. D. Guglielmo, J. M. Duarte, S. Gibellini, V. Parekh, H. Tran, N. Tran, W. Niu, and X. Xu, "Mlperf tiny benchmark," 2021. [Online]. Available: https://arxiv.org/abs/2106.07597

[22] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2018, pp. 115–127.

[23] L.-L. Zhang, Y. Duan, Y. Zhang, Y. Yang, and Y. Liu, "nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2021, pp. 81–93.

[24] 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 (MobiSys)*. ACM, 2022, pp. 235–247.

[25] J. Lee, D. Han, and J. Kim, "Presto: Hybrid cpu-gpu preprocessing framework for video-based ai inference system," in *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services (MobiSys)*. ACM, Jun. 2025, pp. 735–740.

A Literature Search Strategy

To ensure a systematic and transparent selection of literature, relevant studies were identified using a structured search strategy based on the PICO framework. Table 22 summarizes the main elements, associated keywords, and exemplar Boolean search strings used to query scientific databases (such as IEEE Xplore, ACM Digital Library, arXiv, and Google Scholar).

Table 22: PICO elements, keywords, and exemplar Boolean strings.

PICO Element	**Keywords & Synonyms**	**Example Boolean String**
Population	mobile inference, edge AI, Android, smartphone	("mobile" OR "Android") AND ("inference" OR "on-device" OR "edge")
Intervention	heterogeneous execution, co-execution, CPU GPU NPU, collaborative inference	("heterogeneous" OR "collaborative" OR "CPU GPU" OR "co-execution")
Comparison	GPU-only inference, quantization, pruning	("GPU-only" OR "quantization" OR "pruning")
Outcome	FPS, latency, energy efficiency, thermal stability, sustained performance	("FPS" OR "latency" OR "thermal" OR "energy" OR "sustained")
Model Family	YOLO, object detection	("YOLO" OR "object detection")

A.1 Architecture-Oriented Search Queries

In addition to the primary PICO-based searches, targeted exploratory queries were conducted to identify literature related to mobile hardware architectures, Android smartphone inference, heterogeneous CPU–GPU execution, and sustained thermal behaviour. Exact commercial device names, chipset model identifiers, and architecture-specific search terms are not listed here because they are treated as company-sensitive information. Instead, the queries below report the anonymised and generalized form of the search strategy used for Device A and Device B.

Primary Architecture and Device-Class Queries:

- "Device A YOLO inference"
- "Device A deep learning benchmark"
- "Device A object detection mobile"
- "Device B YOLO inference"
- "Device B object detection benchmark"
- "Device B deep learning inference performance"
- "Android smartphone sustained inference YOLO"
- "thermal throttling smartphone CNN inference"

Broadened Exploratory Queries:

- "YOLO mobile inference"
- "Android smartphone object detection benchmark"
- "Android heterogeneous CPU GPU deep learning"
- "continuous inference smartphone thermal"
- "mobile GPU inference thermal throttling"
- "sustained on-device inference Android"