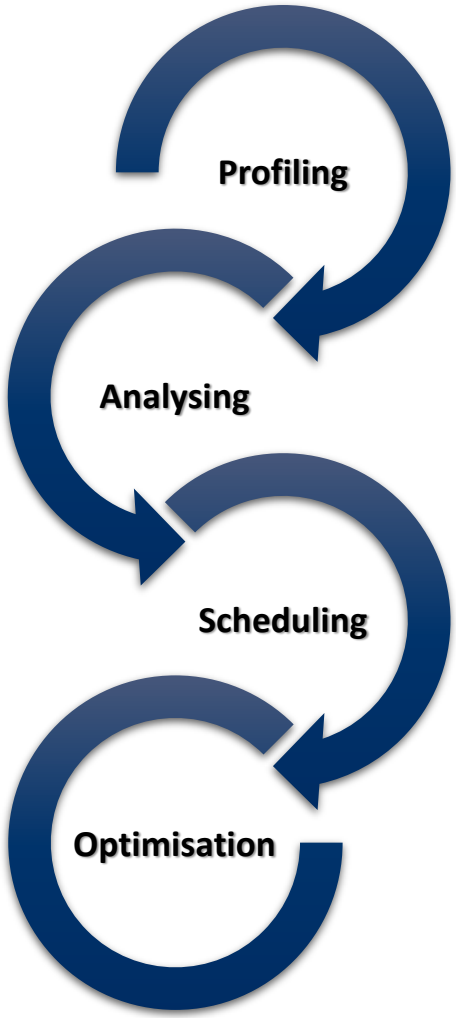
Deployment on Edge Devices

Profiling, Analysing, Scheduling & Optimisation (PASO)

Motivation - The Need for Optimisation

Aspect	Model-Centric Optimisation	System-Level Optimisation
Focus Area	Neural network architecture and parameters	Hardware, OS, memory, and data movement
Objective	Reduce model size, complexity, and computation	Maximize hardware utilization and system efficiency
Techniques	Pruning, Quantization, Knowledge Distillation, NAS	I/O Optimization, Memory Management, Scheduling
Benefits	Lower memory and computation costs	Faster and more efficient inference
Challenges	Maintaining accuracy after optimization	Balancing resource contention and system overhead

Deployment Optimisation Process



Aspect	Techniques	Benefits
Advanced Profiling	Fine-grained performance analysis, Memory access patterns	Identifies bottlenecks and guides optimizations
Analysing Bottlenecks	CPU/GPU utilization, Memory/I/O, OS-level	Maximizes hardware utilization
Dynamic Scheduling	Task scheduling, Model partitioning, Adaptive precision	Reduces latency and improves resource efficiency
Advanced Optimization	Hardware acceleration, Code optimization, Graph optimization	Accelerates computation and minimizes power usage

Advanced Profiling Techniques

SINGAPORE INSTITUTE OF TECHNOLOGY

Technique	Description	Tools/Examples	Benefits
Model Performance Analysis	Analyzes execution time of individual layers, operations, and functions. Identifies bottlenecks like convolutions or activation functions.	- **TensorBoard Profiler**: Layer-wise execution time and memory usage. - **NVIDIA Nsight Systems**: GPU-specific profiling for CUDA operations.	Pinpoints the most time-consuming layers for targeted optimizations.
Memory Analysis	Examines memory access patterns to detect cache misses and inefficient data movement.	- **Valgrind's Cachegrind**: Analyses cache performance. - **ARM Streamline**: Memory usage profiling on ARM devices.	Optimizes data locality and cache usage, reducing memory bottlenecks.
Power Profiling	Measures power consumption to identify energy-intensive operations. Essential for battery-powered edge devices.	- **ARM Energy Probe** - **Intel Power Gadget**	Optimizes power usage and scheduling strategies for energy efficiency.
System-Level Profiling	Analyzes low-level metrics such as cycles, instructions, cache misses and branch mispredictions.	- **perf**: Profiling tool for Linux systems, e.g. *perf stat -e cycles, instructions my_program*	Provides deep insights into CPU and hardware behavior (system-level bottlenecks) beyond model-specific issues.
Network Profiling	Captures and analyzes network traffic to detect latency, bandwidth limitations, and packet loss.	- **tcpdump**, **Wireshark** - Network latency and bandwidth usage diagrams.	Identifies network-related performance issues in edge-cloud communication.

Case Study: CPU & Memory Bottleneck Analysis *(perf)*

SINGAPORE INSTITUTE OF TECHNOLOGY

- ***perf*** is a powerful profiling tool for Linux systems that provides detailed insights into CPU and system performance. It tracks low-level metrics such as *CPU cycles*, *instructions executed*, *cache misses* and *branch mispredictions*.
- **Scenario**: Profiling an object detection model running on a Raspberry Pi 4 to analyze CPU utilization, cache misses, and branch mispredictions.
- **Objective**: Identify CPU and memory bottlenecks to optimize model inference speed.

sudo taskset -c 2 python3 object_detection.py

sudo perf stat -C 2 -e cycles,instructions,cache-references,cache-misses,branch-misses -t <process_id>

*sudo perf stat **-e** cycles,instructions,cache-references,cache-misses,branch-misses **python3 object_detection.py***

```
Performance counter stats for 'python3 object_detection.py':
    2,345,678,901      cycles                    #  1.50 GHz
    3,456,789,012      instructions              #  1.47  insns per cycle
      234,567,890      cache-references
       12,345,678       cache-misses             #  5.27% of all cache refs
        1,234,567       branch-misses            #  1.02% of all branches
```

Key Insights:

- **Low IPC:** Indicates a CPU bottleneck or poor instruction parallelism.
- **Cache Misses:** Suggests memory bottlenecks due to poor data locality.
- **Branch Misses:** Indicates control flow changes that can affect pipeline efficiency.

Optimization Actions:

- **Improve Cache Usage:** Optimize data structures and memory access patterns for better cache locality. Use smaller model weights or quantisation techniques to fit in the cache.
- **Reduce Branch Mispredictions:** Minimize complex conditional statements in the critical path. Refactor loops and branches for more predictable control flow.

Case Study: CPU & Memory Bottleneck Analysis *(perf)*

SINGAPORE INSTITUTE OF TECHNOLOGY

sudo perf record –F 99 –g python3 object_detection.py

Explanation of perf Report:
- **Overhead:** % of total CPU time spent on a specific function or library.
- **Command:** The executable that triggered the event.
- **Shared Object:** The shared library or binary file where the event occurred.
- **Symbol:** The specific function or symbol being executed

Overhead	Command	Shared Object	Symbol
30.15%	**python3**	**/usr/lib/arm-linux-gnueabihf/libc-2.28.so**	**[.] memcpy**
20.23%	python3	/usr/lib/python3.7/site-packages/numpy/core/umath.cpython-37m-arm-linux-gnueabihf.so	[.] ufunc_generic_call
15.10%	**python3**	**/usr/lib/arm-linux-gnueabihf/libc-2.28.so**	**[.] __memcpy_arm**
10.12%	python3	/usr/lib/python3.7/site-packages/tensorflow/python/_pywrap_tensorflow_internal.so	[.] _tensorflow::CopyData
8.05%	python3	/usr/lib/arm-linux-gnueabihf/libpthread-2.28.so	[.] __pthread_mutex_lock
7.02%	**python3**	**/usr/lib/arm-linux-gnueabihf/libc-2.28.so**	**[.] malloc**
4.18%	python3	/usr/lib/python3.7/site-packages/tensorflow/python/_pywrap_tensorflow_internal.so	[.] _tensorflow::Conv2DCompute
3.15%	python3	/usr/lib/python3.7/site-packages/numpy/core/umath.cpython-37m-arm-linux-gnueabihf.so	[.] reduce_sum
2.00%	**python3**	**/usr/lib/arm-linux-gnueabihf/libc-2.28.so**	**[.] free**

Some Insights:

Event	Description	Possible Optimization Actions
30.15% memcpy	Frequent data copying between memory buffers.	- **Reduce Memory Copy Operations:** Optimize tensor data structures to minimize copying. - **Use In-Place Operations:** Modify data in place to avoid unnecessary copies.
15.10% __memcpy_arm	ARM-specific memory copy operation, confirming memory-bound bottlenecks.	- **Optimize Data Layout:** Reorder data structures to improve cache locality. - **Use NumPy Broadcasting:** Leverage broadcasting to minimize memory operations.
7.02% malloc	frequent memory allocations due to dynamic creation of temporary buffers in NumPy and TensorFlow operations.	Pre-allocating memory outside loops and reusing memory buffers to minimise heap allocations.

Analysing System-Level Bottlenecks

SINGAPORE INSTITUTE OF TECHNOLOGY

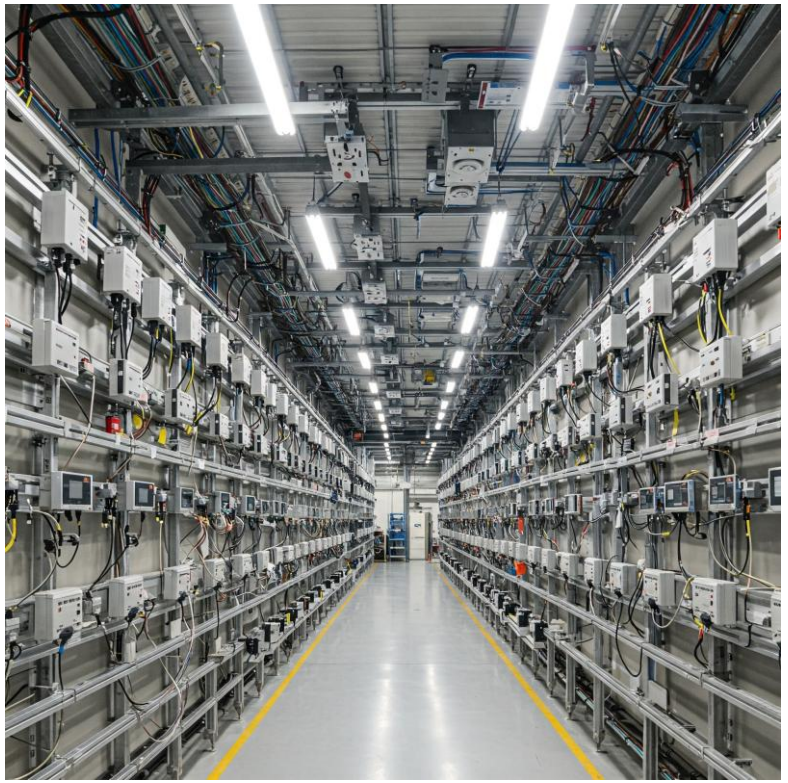
Aspect	Description	Tools/Examples	Optimization Approach
CPU/GPU Utilization	Identifies under-utilization or overloading of processing units.	- **Perf** (Linux Performance Profiler) - **NVIDIA Visual Profiler (nvvp)**	• Load balancing between CPU and GPU. • Utilize heterogeneous computing (e.g., NPUs).
Memory and I/O	Analyzes latency due to memory access or data transfer operations.	- **gprof**, **Valgrind** - I/O solutions: Faster storage (e.g., SSD), buffering data	• Memory Pooling: Reuse memory buffers. • Layer Fusion: Reduce intermediate data movement. • Optimize data access patterns.
OS-Level	Examines the impact of OS scheduling, resource allocation, and context switching.	- **Zephyr RTOS** for real-time constraints - Lightweight runtime (e.g., TensorFlow Lite Micro)	• Real-Time Scheduling: Prioritize ML tasks. • Minimize OS overhead with lightweight runtimes.
Concurrency Issues	Addresses thread contention, locking, and synchronization challenges.	- **Thread Pools** - **Asynchronous Operations**	• Use thread pools and asynchronous operations. • Implement lock-free data structures.

Case Study: Smart Camera for Object Detection

- **Application:** An edge-based smart camera performing real-time object detection & alerting.
- **Bottleneck:** Camera transmits results with high latency to the cloud for processing.
- **Tools:** *tcpdump* and *Wireshark*, network traffic analysis revealed high transmission delays due to limited bandwidth and intermittent connectivity.
- **Possible Solution:**
  - *Edge-Cloud Partitioning:* Critical detection logic was moved to the edge device, sending only summarised alerts to the cloud, reducing data.
  - *Protocol Optimisation:* Switching from HTTP to MQTT for more efficient, lightweight communication.

Case Study: Industrial Sensor Network

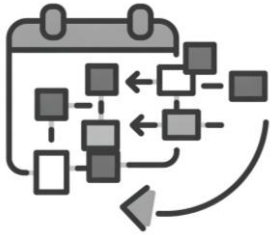
- **Application:** A large-scale sensor network monitoring environmental parameters in a manufacturing facility.
- **Bottleneck:** The gateway device experienced delays in aggregating and processing data from hundreds of sensors.
- **Tools:** *perf* and *Valgrind* profiling showed CPU under usage due to inefficient data handling and I/O operations.
- **Possible Solution:**
  - *Memory Pooling:* Implemented buffer reuse to reduce memory allocation overhead.
  - *Layer Fusion:* Combined data pre-processing layers to minimise intermediate data movement.
  - *Concurrency Optimization:* Introduced asynchronous data processing and lock-free data structures.



Case Study: Autonomous Drone Navigation

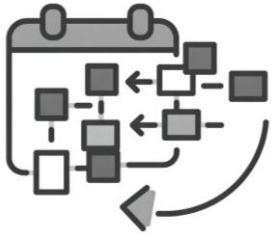
- **Application:** An edge-based drone using deep learning for obstacle detection and navigation.
- **Bottleneck:** The GPU was overloaded while the CPU was underutilised, leading to navigation lag.
- **Tools:** Profiling with *NVIDIA Nsight Systems* showed GPU saturation during inference while CPU resources were idle.
- **Possible Solution:**
  - *Load Balancing:* Offloaded pre-processing and sensor fusion tasks to the CPU, optimising GPU utilisation for model inference only.
  - *Heterogeneous Computing:* Integrated an NPU to accelerate lightweight tasks, balancing the workload across CPU, GPU, and NPU.

Dynamic and Adaptive Scheduling



Technique	Description	Approach	Benefits	Examples
Dynamic Task Scheduling	Dynamically assigns tasks to processing units (CPU, GPU, NPU) based on resource availability.	- **Work Stealing Algorithms:** Idle processors take over tasks from busy ones. - **Priority-Based Scheduling:** Assigns priority levels to tasks (e.g., high, medium, low).	- Maximizes hardware utilization. - Reduces latency by prioritizing critical tasks.	- **TVM Runtime**: Dynamic scheduling for heterogeneous devices. - **Preemptive Scheduling**: Interrupts lower-priority tasks for high-priority ones.
Adaptive Model Partitioning	Splits model execution across multiple devices (e.g., edge and cloud) and adapts partitioning based on runtime conditions like network latency.	- **Edge-Cloud Offloading:** Executes latency-critical layers on the edge and offloads other layers to the cloud.	- Balances latency, energy, and computational load. - Improves performance under varying network conditions.	- **Neurosurgeon Framework:** Real-time model partitioning between edge and cloud.
Adaptive Resource Allocation	Dynamically allocates system resources (CPU, memory) based on current system load and task requirements.	- Allocates more resources to high-priority or critical tasks during peak load. - Minimizes resource allocation for background tasks during idle periods.	- Enhances system responsiveness and efficiency. - Ensures optimal resource usage without overloading.	- **Real-Time Operating Systems (RTOS):** Adaptive scheduling in time-critical applications.

Honourable mentions



Technique	Description	Approach	Benefits	Examples
Adaptive Precision and Quantization	Dynamically adjusts model precision (e.g., FP32 to INT8) based on available resources and accuracy requirements.	- **Hardware-Aware Quantization:** Adapts precision to match hardware constraints, optimizing for both accuracy and efficiency.	- Maintains a balance between accuracy and computational efficiency. - Reduces power consumption and memory usage.	- **HAQ (Hardware-Aware Quantization):** Adjusts precision based on hardware limitations.
Containerization for Edge	Packages applications and dependencies into containers for portability and isolated resource management on edge devices.	- **Docker and Containers:** Ensures consistent environment across edge devices. - Simplifies deployment and scaling of ML workloads.	- Enhances portability and isolation. - Efficient resource management and scheduling.	- **Docker Containers:** Edge deployment with consistent environments and efficient scaling.

Case Study: Smart Security Camera with Motion-Activated Object Detection

Imagine: You have a security camera to detect people or objects only when there's movement, saving power and resources when nothing is happening.

Smart Scheduling:

- **Movement Detected**: The camera immediately runs object detection to see what's happening.
- **No Movement**: Object detection pauses or runs in the background at a lower priority, freeing it to do other things or saving power.

What You Need:

- **Motion Sensor:** A sensor that detects movement.
- **Camera:** To capture video.
- **Software:**
  - **OpenCV:** Helps the camera understand the video stream.
  - **Multiprocessing:** The MCU can perform object detection as a separate task.
  - **Object Detection Model:** A pre-trained model like MobileNet SSD, which is small and fast.

Think of it like a guard dog that only wakes up when it hears something!

Why This is Cool:

- **Saves Power:** The camera doesn't need to run object detection all the time, so it uses less power.
- **Faster Response:** When something happens, the camera reacts instantly.
- **Efficient:** The Raspberry Pi can do other things when it's not busy detecting objects.

Case Study: Smart Home Camera Performing Real-time Face Recognition

Imagine: You have a home security camera that can recognise faces. It can tell if a family member is at the door or a stranger is approaching. The camera can quickly alert you if an unknown person is at your door, even if the internet is down.

Challenge: High latency and power consumption when running complex face recognition models entirely on the edge device.

Adaptive Model Partitioning:

- The camera's face recognition "brain" is split into two parts.
- **Local**: The camera uses its own "brain" to quickly analyse faces and see if they're familiar.
- **Cloud**: If the camera needs extra brainpower to recognise someone, it sends the video to a powerful computer in the cloud for help. This happens when the internet connection is strong, and the camera isn't too busy.

Why This is Cool:

- **Fast Response:** Critical layers run on the edge, ensuring quick response.
- **Saves Energy:** Heavy layers offloaded to the cloud, reducing power consumption on the edge device.
- **Reduces Cost:** Minimizes cloud usage by running lightweight layers locally.

Think of it like a security guard with a walkie-talkie. They can handle most situations themselves, but they can call for backup from the central office when needed!

Case Study: Smart Farm with Adaptive Crop Monitoring

Imagine: A network of cameras and sensors in a greenhouse monitors the health and growth of crops. The system needs to detect pests, diseases, and nutrient deficiencies, while also optimizing water and fertilizer usage.

How it Works:

- **Smart Priorities:** The system's "brain" shifts focus based on the plants' needs.
  - **Young Plants:** More attention to pest and disease detection.
  - **Mature Plants:** Focus on optimising water and nutrients.
- **Quick Response:** If a problem is found, the system focuses more resources on that area.

Challenge: Continuously running all analysis tasks at full power would consume excessive energy and processing capacity.

Adaptive Resource Allocation: The system dynamically adjusts how much processing power and sensor sampling frequency is dedicated to each task based on the current needs of the crops and the presence of any issues.

Think of it like a farmer tending to their crops. They pay close attention to young seedlings, providing extra care and attention. As the plants mature, the farmer adjusts their focus to ensure optimal growing conditions and address any specific needs!

Why This is Cool:

- **Healthy Crops:** By focusing on the most critical tasks at each growth stage, the system helps ensure healthy crop development and maximises yield.
- **Resource Efficiency:** The system conserves energy and processing power by allocating resources only where and when they are most needed.
- **Early Problem Detection:** By dynamically increasing resource allocation to potential problem areas, the system enables early intervention, preventing widespread issues.

Advanced Optimisation Techniques

SINGAPORE INSTITUTE OF TECHNOLOGY

Focus Area	Description	Techniques	Examples	Challenges
Hardware Acceleration	Utilizes specialized hardware components (e.g., NPUs, TPUs, FPGAs) to accelerate ML computations and reduce latency.	- **Operator Fusion:** Combines multiple operations into one kernel to minimize memory access. - **Domain-Specific Accelerators:** Custom accelerators for specific tasks (e.g., convolution, matrix multiplication).	- **Google Edge TPU:** Low-latency inference on edge devices. - **NVIDIA TensorRT:** GPU acceleration for deep learning models.	- **Hardware-Specific Tuning:** Requires optimization for specific hardware. - **Limited Portability:** Custom accelerators may not be compatible across devices.
Code Optimization	Refines code for faster execution, reduced memory usage, and lower power consumption.	- **Kernel Optimization:** Improves cache locality and reduces memory access. - **Loop Unrolling and Tiling:** Enhances parallelism and data locality. - **Vectorization:** Utilizes SIMD (Single Instruction, Multiple Data) for faster computation.	- **LLVM and GCC Compilers:** Low-level code optimization. - **XLA (Accelerated Linear Algebra) Compiler:** Efficient kernel generation in TensorFlow.	- **Complexity vs. Maintainability:** Balancing performance gains with code maintainability. - **Deep Hardware Knowledge Required:** Effective optimization demands understanding of underlying hardware architecture.
Graph-Level Optimization	Rearranges computation graphs for improved memory and computation efficiency.	- **Common Subexpression Elimination:** Avoids redundant computations. - **Constant Folding:** Pre-computes constant expressions during compilation.	- **TensorFlow XLA and TVM:** Perform graph-level optimizations for efficient execution.	- **Graph Dependency Management:** Requires careful management of graph dependencies to avoid runtime errors.

Future Trends

Trend	Description	Benefits	Challenges	Example Use Cases
AI-Driven Optimization	ML algorithms dynamically optimize system parameters and model architectures.	- Automated tuning - Real-time adaptation - Hardware-aware optimization	- High complexity - Privacy concerns	- Google AutoML - Alibaba RL Cluster Optimizer
Dynamic and Adaptive Allocation	AI-driven scheduling adjusts resources based on real-time demands.	- Optimized utilization - Reduced latency - Cost efficiency	- Complex scheduling - Coordination overhead	- Alibaba Adaptive Resource Manager - TVM Runtime
Cross-Layer Optimization	Jointly optimizes hardware, runtime, compiler, and application layers.	- Holistic efficiency - Maximum performance gains - Scalability	- High complexity - Compatibility issues	- TVM Compiler - Google Edge TPU Compiler
Collaborative & Federated Opt.	Distributed optimization preserving data privacy and security.	- Data privacy - Personalized models - Scalable learning	- Non-IID data - Communication overhead	- Google Federated Learning - OpenMined PySyft

Summary

- **Advanced Profiling Techniques** provide detailed insights into model and system performance, guiding further optimizations.
- **System-Level Bottlenecks** address hardware, memory, and OS constraints, ensuring efficient resource usage.
- **Dynamic and Adaptive Scheduling** enables real-time adjustments, maximizing performance under changing conditions.
- **Advanced Optimization Techniques** exploit hardware accelerators and low-level code enhancements for peak efficiency.

These strategies collectively enhance the deployment of ML models on edge devices, achieving high performance and low latency while minimizing power consumption.

Some Relevant References

1. J. Zhang, C. Li, and Y. Wang, "**EdgeShard: Efficient LLM Inference via Collaborative Edge Computing**," arXiv preprint arXiv:2405.14371v1, 2024. [Online]. Available: https://arxiv.org/html/2405.14371v1.
2. H. Kim, S. Park, and J. Lee, "**Inferencing on Edge Devices: A Time- and Space-aware Co-scheduling Framework**," Proceedings of the ACM on Embedded Systems, vol. 2, no. 1, pp. 1-13, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3576197.
3. X. Liu, M. Zhang, and K. Chen, "**Boosting DNN Cold Inference on Edge Devices**," arXiv preprint arXiv:2206.07446, 2022. [Online]. Available: https://arxiv.org/abs/2206.07446.
4. Y. Sun, X. Wang, and Z. Li, "**BCEdge: SLO-Aware DNN Inference Services with Adaptive Batching on Edge Platforms**," arXiv preprint arXiv:2305.01519, 2023. [Online]. Available: https://arxiv.org/abs/2305.01519.